DISTRIBUTED CODE GENERATION

USING OBJECT LEVEL ANALYSIS

CHU YINGYI

(B.ENG, TSINGHUA)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2003

ACKNOWLEDGEMENT

I would like to express my deep gratitude and appreciation to my supervisor, Dr. Tay Teng Tiow for his patient guidance, instructive advice, continuous support and warm encouragement over the years of my study.

Sincere thanks for all the friends who have gone out of their way to help and support me.

I am grateful to my beloved family members for their selfless love, encouragement and support.

I also give thanks to the National University of Singapore for granting me research scholarship to support my study and research.

TABLE OF CONTENTS

ACKNOWLEDGEMENT……………………………………………………………….i

TABLE OF CONTENTS……………………………………………………………..ii

SUMMARY……………………………………………………………………………v

LIST OF FIGURES…………………………………………………………………vii

LIST OF TABLES…………………………………………………………………..viii

CHAPTER 1 INTRODUCTION………………………………………………………1

1.1 Background…………………………………………………………………..1

1.2 Motivation and Contributions……………………………………………..5

1.3 Related Work………………………………………………………………...7

1.4 Thesis Organization………………………………………………………...10

CHAPTER 2 AN OVERVIEW……………………………………………………13

2.1 The Decentralized System………………………………………………….13

2.2 The Components of JDGC…………………………………………………15

2.3 The Network Layer…………………………………………………………18

2.4 Summary……………………………………………………………………19

CHAPTER 3 SYSTEM METHODOLOGY…..…………………………………20

3.1 The Object Model of Applications……………………………………………20

3.2 The Methodology of Distributed Code Generation using Object Level Analysis…………………………………………………………………..21

3.3 Flow of Processing…………………………………………………………23

3.4 Summary……………………………………………………………………26

CHAPTER 4 THE METHOD OF OBJECT LEVEL ANALYSIS……………...27

4.1 Class-Level Analyzer and Three Dimensional Affinity Metrics…………..…27

4.1.1 The Three Dimensions…………………………………………………...28

4.1.2 Analyzing Procedure……………………………………………………..30

4.2 Object-Level Analyzer……………………………………………………...32

4.2.1 Visible Scope Analysis and Mapping of Affinity Metrics……………32

4.2.2 Analyzing Procedure…………………………………………………….35

4.3 Summary……………………………………………………………………36

CHAPTER 5 DISTRIBUTED CODE GENERATION………………………….38

5.1 Code Generation in Two Steps……………………………………………..38

5.2 Tree Transformation………………………………………………………….40

5.2.1 Changes for Object Placement and Communications…………………41

5.2.2 The Wrapper Class……………………………………………………….42

5.2.3 Addition and Substitution of Sub AST………………………………..44

5.3 Summary……………………………………………………………………46

CHAPTER 6 JDGC SYSTEM AND APPLICATIONS…………………………47

6.1 User Interface……………………………………………………………47

6.2 Runtime Application Manager………………………………………...50

6.3 A Discussion on Security Issues………………………………………54

6.4 Applications…………………………………………………………...58

CHAPTER 7 CONCLUSIONS…………………………………………………65

7.1 Contributions.…………………………………………………………65

7.2 Recommendations……………………………………………………..67

APPENDIX A KEY SOURCE CODE……………………………………………71

Code for Class-Level Analysis…………………..…………………………71

Code for Object-Level Analysis……………………………………………77

APPENDIX B EXAMPLES OF CODE TRANSFORMATION…………..……….86

Matrix Multiplication………..…………………..………………………86

Curve Fitting…………………………………………………………….95

REFERENCES……………………………………………………………...108

SUMMARY

This thesis proposes a Java Distributed code Generating and Computing (JDGC) platform. The key function of this system is a method of distributed code generation using object level analysis. This method analyzes an input application written in object-oriented language to extract communication affinity metrics between the runtime objects. Using these object level affinity metrics, the original application program is automatically transformed to run on the network machines. The generated distributed code incorporates the runtime object placement according to the affinities computed. The resultant code is able to run on the distributed environment, handling object manipulations and communications on network machines.

The object level analysis is a new analysis method for object-oriented applications, providing object level communication affinity metrics. Our approach uses a two level analyzer with an intermediate 3 dimensional communication metrics. The result of this analysis is a 2 dimensional metrics containing the communication affinities between runtime objects. In our system, this information serves as the guidance for the object placement in the subsequent distributed code generation. This analyzing method while motivated from our integrated system for object distribution, is nevertheless a general

approach for extracting object-to-object communication affinity metrics and can also be used in other contexts.

The distributed code generation is an automatic process that incorporates the object placement on network machines into the distributed code according to their communication requirements. This process is completely automatic and does not require special programming paradigm for the user's program. A normal program for single machine is transformed to a form that runs on the network. The distributed code generator performs the code transformation, traditionally performed by hand, in a structured mode.

The proposed JDGC platform is an integrated system with the above distributed code generation as a key component. As a fully functional system, JDGC also has a comprehensive user interface, and uses a runtime application manger to keep tracks of the dynamic status of the system.

LIST OF FIGURES

Figure 2.1 A Network Overview of the Computing Scheme……………………..14

Figure 2.2 Components of Java Distributed code Generating and Computing (JDGC) Platform……………………………………………………17

Figure 3.1 The Flow of Processing for the Object-Level Analysis and Distributed Code Generation ………..……………………………………………25

Figure 4.1 The Three Dimensions of Object-Oriented Program…………………30

Figure 4.2 Flow Chart of the Object-Level Analysis Procedure ………………...37

Figure 5.1 Template of Wrapper Class…………………………………………...43

Figure 6.1 User Interface: Transformation of Application………………………48

Figure 6.2 User Interface: Submission and Execution of Application……………49

Figure 6.3 The Data Structure in Runtime Application Manager………………...53

LIST OF TABLES

Table 6.1 The Average Execution Time with One Host……………………………59

Table 6.2 The Average Execution Time for Four Simultaneous Applications…..59

Table 6.3 The Average Execution Time When Num of Applications equals Num of Hosts……………………………………………………………..60

Table 6.4 The Average Execution Time of Multiple Threaded Applications…...61

Table 6.5 The Overheads in Creation and Communication……………………...61

Table 6.6 Comparison of Different Object Allocation Methods………………...62

Table 6.7 The Support for Multiple Applications………………………………..63

CHAPTER 1

INTRODUCTION

1.1 Background

Distributed computing is the emerging platform for many computational intensive applications. This development is enabled by two significant changes in the world of computing in the last ten years. Firstly, the advent of cheaper and more powerful processors greatly enhanced the computational capability of the ubiquitous workstations and personal computers. Today the processor that powers a personal computer is often as powerful as that which powers a big departmental server. Secondly, few computers work as standalone machines. They are either permanently networked or at least are equipped with the facility to be connected to a network as and when needed. The networked computers on a local area network or the Internet naturally form a system of loosely coupled multiple processors. These systems have the advantage of scalability and relatively lower investment when compared to the multiprocessor mainframes.

On the other hand, more and more applications in scientific and business computing require large amount of computational powers. Some examples of such applications

are human gene analysis, cryptographic computing, numerical solution for dynamic systems, processing for large remote-sensing images, modeling of long time series in equity market, etc. The distributed computing architecture is most suitable for these applications in a low cost. Such systems utilize a number of mostly idle processors connected by a network to collectively solve large problems.

Although the hardware for such an architecture has been in place, there must be a software platform on every participant computer to support the distributed execution of applications. A distributed computing platform should provide the following functions:

- Find the available computational resources among the whole system.
- Host and execute a piece of computation task.
- Transmit data between different computation tasks via the network.
- As a robust platform, be able to recover from the machine crashing.

Aside from the underlying platform, applications that run on networked machines are different from those for single machines. Since only concurrent applications can benefit from the multiple processors, we will only consider these applications in our distributed computing context.

Compared to a single machine concurrent application, a distributed application has to manage the following tasks in addition to the computation task:

- The application partitions its computation into parts.
- The application launches its parts to other hosts.
- The application handles interactions between different parts through local or network communication depending on their relative locations on the network.

All these considerations make the application development for distributed system a daunting task. Developers have to analyze the structure of application, partition it into components, and recognize those interactions that involve inter-machine communication. During programming, the handling of task distribution and communications must be incorporated into the code. The debugging of distributed applications is also much more complex than that of single machine applications.

Object-Oriented Programming Language (OOPL) is a relatively new paradigm of developing complex applications. Unlike traditional procedure-based method, OOPL uses a bottom-up composition model of development, and possesses the advantages of information hiding, encapsulation and modularity. The unique features of OOPL make it particularly suitable for the distributed computing applications for the following reasons:

- Objects are the unit of manipulation in object-oriented applications, making the partitioning task easier compared to the monotholic processes in procedure-based applications.
- Encapsulation makes the object a relatively separate entity. This makes it an ideal candidate of distribution.
- The structured interface of classes facilitates the identification of communications between objects.

Therefore, distributed object-oriented computing has been studied widely in recent years. Three important aspects in the research on distributed object-oriented computing are summarized as follows:

Firstly, the platforms must support the object-oriented applications. Most newly proposed platforms like [1,2] are designed for object-oriented applications. These platforms provides the basic functions of transferring object through data network, making method invocations remotely, etc. Techniques and protocols such as Object Serialization, RMI, and CORBA [3] have been developed to support distributed object-based computing.

Secondly, the mapping of object-oriented applications to networked machines is based on the analysis of communication relations between objects. In this aspect, program analysis methods in [4-6] provide the methods for analyzing class-to-class communication affinity of general object-oriented applications. While the class-based analysis is static and does not provide enough information for the runtime objects, the analysis method of [7] models the runtime behavior using the scenario diagram. However, for object-oriented application, the unit for distribution is the runtime objects. The existing analysis methods cannot explicitly give object-to-object communication information. They are then not directly usable in the distribution of objects. Some other proposals for object-oriented distributed computing uses alternate method for extracting the object level communication metrics. The method in [8] was based on previous tracing records, which are obtained by running the applications on an instrumented virtual machine beforehand. Local behaviors of individual objects are recorded, which include the time spent on computation, on sending messages and on receiving messages (all measured in the unit of clock tick). These trace records are used as the guidance in the object distribution algorithm. In [9], a programmer driven approach was used. While the initial object placement and the dynamic re-clustering are based on static weights set to the object graph, several classes of hints from an

informed programmer are injected into the algorithm. These hints help the co-locating of objects (e.g. explicit parallelism), migrating of objects, replicating of objects, etc. Proposal [10] facilitated the object distribution using a certain programming language for distributed object-oriented applications. The proposed Orca language is a dedicated language for implementing explicitly-parallel application. The language feature makes it possible to design a compiler that can determine the access patterns of the objects. These information is then passed to the runtime system for making proper object placement decisions.

Thirdly, programming paradigms and language supports were introduced to facilitate the production of distributed object-oriented applications. As discussed above, porting concurrent program for single machine to network machines is a difficult task for the programmers. This is also true for object-oriented applications. In proposals such as [11,12], new language features were introduced to facilitate the programming. Other proposals [13,14] made use of existing language like Java, but defined certain programming paradigms. In these systems, the production of distributed code is not automatic but is done by the programmers using system-specific programming rules.

1.2 Motivation and Contributions

A major issue for distributed computing systems is the partition of the application such that it can run correctly and efficiently on network machines. This thesis introduces an integrated distributed computing platform to do this efficiently. A new method of distributed code generation using object level analysis is proposed. The object level analysis is a general analysis method for object-oriented applications, providing

detailed communication relation information. The distributed code generation is an automatic process that incorporates the object placement on network machines into the distributed code according to their communication requirements. The major motivations of this research are twofold.

Firstly, as discussed in Section 1.1, the development of distributed code is a difficult task. The introduction of certain programming paradigms or language features still requires the developer to manually incorporate the changes for distributed code. The process is still far from automatic and easy. Moreover, they impose system-specific rules into the distributed code, which inevitably causes portability problem. In this thesis, we propose an automatic distributed code generation method, which only requires standard Java programs, but does not require special treatments of original code. The method aims to transform standard concurrent programs running on single machine to the form that runs on networked machines. The object placement is incorporated into the generated distributed code according to the communication relations of the objects. The whole process is automatic and does not require a programmer's intervention.

Secondly, the object placement in the distributed code generation is based on the analysis of communication relations between objects. As discussed in Section 1.1, the existing analysis methods are inadequate because they do not account for runtime instances. In this thesis, we address the problem in a new method for object level analysis. In this method we uses a two level analyzer with an intermediate 3 dimensional communication metrics. This analyzing method while motivated from our integrated system for object distribution, is nevertheless a general approach for

extracting object-to-object communication affinity metrics and can also be used in other contexts.

This thesis proposes a Java Distributed code Generating and Computing (JDGC) platform, which integrates the above object level analysis method and the distributed code generation. The JDGC is a fully functional distributed computing platform. The network layer basis is a set of network layer API functions, proposed in a previous work [15]. The key component of JDGC system and the emphasis of this thesis is the distributed code generation. The contribution of this research is summarized as follows:

- A new method of object level analysis for object-oriented applications. The output is detailed communication affinities between runtime objects. The method is general.
- An automatic distributed code generation method for standard concurrent programs. The input is standard program. The method is completely automatic.
- An integrated system to support distributed object-oriented applications. The system is based on pure Java. It supports multiple applications submitted simultaneously and has crash recovery function.

1.3 Related Work

There are several areas of work that influences our research. We will discuss the related work in the areas of distributed computing platform, object-oriented application analysis and compiler construction.

Distributed computing platforms are the runtime environment for the execution of distributed applications on the network. In such system, a group of computers connected to a network is viewed as a virtual supercomputer of multiple processors. Previous researches on this area influence the design and implementation of the proposed platform. Next we will briefly discuss the features of several previous works in this area.

Network of Workstations (NOW) [16]. This is one of the earliest works that build system support for using a network of workstations to act as a distributed supercomputer on a building-wide scale. This system covers most basic issues in a multi-computer distributed system, such as the fast communication implementation, distributed file systems, operation system, I/O, and programming environment.

Javelin/Javelin++ [13]. This is a Java-based infrastructure for global computing. The system employs Java-enabled web technology to address the problem of interoperability, security, and the ease of participation. A broker component in the architecture is the central process that coordinates the supply and demand for computing resources. The client in this system is the web browser. The proposed platform in this thesis is also written in Java, but it is not based on web browsing and the client-server mode.

POPCORN [2]. This project provides another Java-based distributed computing platform. An important feature of POPCORN is that it proposed a programming paradigm for writing distributed applications. A highly structured template is provided for developers for POPCORN system. Different from this solution, our research aims to automate the development of distributed code by the system, and eliminate the system-specific features in the programming.

In distributed object-oriented computing, the analysis of communication affinity between objects is necessary for the efficient distribution of objects to networked machines. We briefly summarize the existing methods of object-oriented program analysis.

The modeling tools [17,18]. There are many forward/reverse engineering tools for object-oriented applications. The forward process in these tools is to generate program from describing models such as UML. This function is mainly used in the design of complex software system. On the other hand, the reverse process takes an existing program as the input. Then the tool tries to analyze the code and obtains the structure of the program. Most these tools are able to model the classes of the program, as well as the interactions between classes. Although only class-level information is inadequate for the distribution of runtime objects, these information is nevertheless the basis for more detailed analysis.

SCED [19]. In SCED a method of modeling dynamic behavior of programs is proposed. This method uses the Scenario diagram. It decomposes the runtime behavior of a program into a series of scenarios. In each scenario, the involved objects are connected by the messages passed between them. This approach, while extracts the runtime information, does not retain the objects as an integrated entity in the resultant model.

The key function in our platform involves the preprocessing of computer program and code generation for distributed computing platform. These issues are closely related to compiler constructions. The development of our system uses several standard components of compilers or translators, such as parser and code generator.

The technique of compiler construction is well established, and many tools exist for building compiler components. There are compiler generator tools, which automatically generate compiler components for certain languages based on its grammar definition. For example, ANTLR [21] provides an easy way to construct Java parser. However, ANTLR currently does not provide the code generation for the syntax tree format being used. In the development of our system, we instead use the Pizza compiler [20] for Java language. It is an open source compiler for an extension of Java language. Its data structures for various syntax tree nodes are used as the building blocks of the syntax tree of our system. The hierarchical structure of the data structures facilitates the development of the tree transformation module. The Java program parser and bytecode generator used in our system are also adapted from this Java compiler.

1.4 Thesis organization

This thesis is organized as follows:

Chapter 2 gives an overview of whole distributed computing environment. The roles of client and server host in the decentralized system are first described. Then we introduce the architecture and components of the JDGC system software. The underlying network protocol is summarized at the end of the chapter.

Chapter 3 describes the key function of JDGC, namely the distributed code generation using object level analysis. As the basis of the whole method, the general features of object-oriented applications are first summarized. Then we introduce the distributed

code generation and the object level analysis method. The six parts of the procedure and the processing flow are given.

Chapter 4 details the method of object level analysis. The Class-Level Analyzer and Object-Level Analyzer are discussed in Section 4.1 and 4.2 respectively. For the Class-Level Analyzer, we firstly explain the concept behind it, namely the 3 dimensional affinity metrics. Based on it the details of the class-level analysis method are given. For the Object-Level Analyzer, the key concepts are the visible scope analysis and mapping from class-level affinity. We then present the details of the object-level analysis.

Chapter 5 details the automatic distributed code generation. After a discussion of the two steps in the distributed code generation, we examine the Abstract Syntax Tree (AST) transformation method in detail. First we discuss the necessary changes to the original program. Then the main techniques used to automate the transformation are introduced. Finally we give the details of the addition and substitution of sub AST.

Chapter 6 details the JDGC platform and presents sample applications. Firstly, the user interface is introduced briefly. Then we discuss the runtime application management system, which is the runtime environment of JDGC and provides crash recovery ability. Security issues, as another important aspect in an open distributed system like JDGC, is also discussed briefly. After the complete description of JDGC, two sample applications running on the platform are presented. We test the execution time for the two applications in different situations and discuss the result of the tests in detail.

Chapter 7 summarizes the contributions and concludes the thesis. We also give recommendations for possible extension of the proposed method.

CHAPTER 2

AN OVERVIEW

The proposed JDGC platform is a fully functional software system that supports the distributed execution of Java applications. In this chapter, we first introduce the decentralized distributed computing scheme in the system perspective. Within the scheme's context, we describe the software architecture of JDGC. The key function of JDGC is an object level analysis and automatic code transformation, which is the emphasis of this thesis. The system runs on the network layer protocols proposed in a previous work [15]. An overview of the protocol is summarized.

2.1 The Decentralized System

In this section, we introduce the distributed computing scheme in the perspective of the whole system. Unlike most existing schemes, the architecture of our scheme is fully decentralized, that is, there is no one dedicated central component that controls the working of the entire system. Every host participating in the scheme is identical in function, and each host can act as a requesting host and/or a contributing host, and possibly at the same time. Computing hosts can join or leave the scheme as and when desired without the need to register their presence or absence with any controller. In

such a scheme, the whole system can be viewed as a set of functionally identical machines/hosts connected to a network as depicted in Fig.2.1, although at a certain time every host has its own role, that is, it may be idle, be requesting resources or be contributing their own resources.

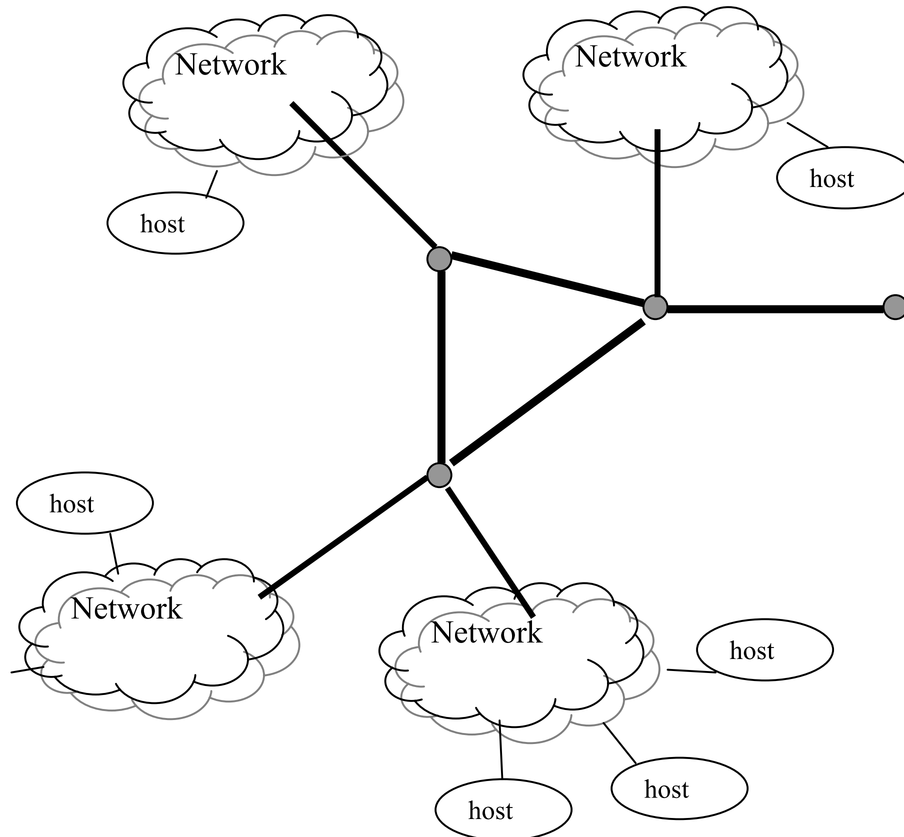

Fig.2.1. A Network Overview of the Computing Scheme

Any distributed application running on the scheme involves a group of coordinating hosts. Each host in the group may possibly take two roles; namely as a server and as a client. As a client, the host first recruits available hosts in the system using a group communication protocol. After it receives responses from enough contributing hosts, the client will submit its own application to run on those recruited hosts. The recruited hosts take the role of a server. A server host responses to the recruiting request from

the client host, and contributes its own computation resources to application from the client.

To participate this computing scheme, a host starts a supporting software, that is, the JDGC system proposed in this thesis. Because the hosts in the scheme work on a peer-to-peer mode, the supporting JDGC system on every participant host has the same functions. As discussed above, a host may act as a server or a client or both in different applications. Therefore JDGC integrates the functions of requesting resources and submitting applications as a client, and the functions of responding to recruiting request as a server. The components and architecture of JDGC is described in the next section.

2.2 The Components of JDGC

The JDGC software system running on every host provides the functionality of both a client and a server host. As a server, user can start the server daemon so that the host listens to recruitment requests. The function of the server daemon and the protocol involved in the network layer is detailed in [15], and will be summarized in Section 2.3. As a client, user can open one or more standard single machine oriented Java applications. The application will perform the analysis and transformation, so that the code generated integrates seamlessly into the network layer API and is able to run directly in the distributed environment. The JDGC software system consists of the following components:

- Graphical User Interface. This integrates both client and server functions.
- Object level analysis and distributed code generation. This is a key function of JDGC. The object level analysis is a general method to extract communication affinities in the object level. The distributed code generation consists of a syntax tree transformation, which makes use of the obtained affinities metrics and automatically transforms standard Java program to run on the network using the underlying network layer API. Every application before submitted to the network performs the analysis and transformation. The output of this preprocessing is the distributed code, which will then be submitted and managed by the runtime application manager.
- The runtime application manager: This manager supports multiple applications that are submitted simultaneously. All the information on the current applications, objects and hosts are maintained in two cross-linked lists. The runtime application manager also backs up the instances during the lifetime of an instance, and allows recovery from crash automatically without the user's intervention.
- Network layer protocol and API: This contains the recruit function for client, which uses multicast protocol to request for available machines in the group of hosts. The communication primitives are provided for the application to pass data among objects. The API functions are incorporated into the resultant distributed code.

The above components and their relationships are illustrated in Fig.2.2.

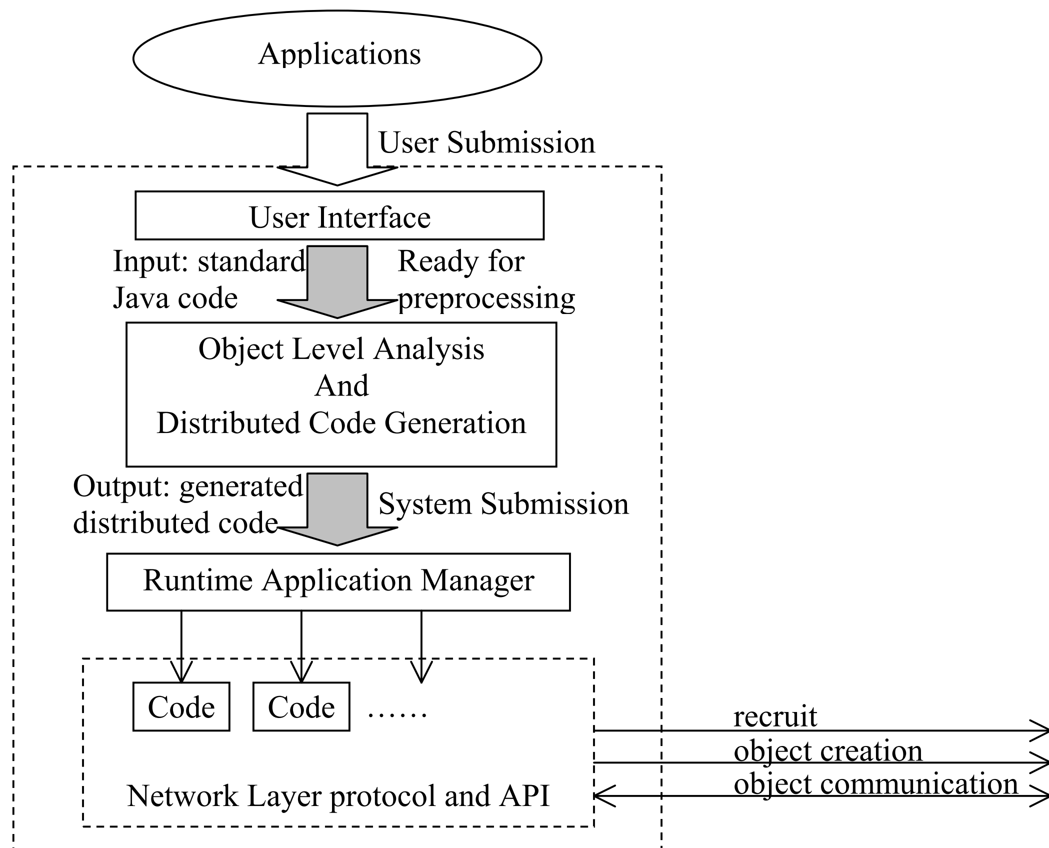

Fig.2.2 Components of Java Distributed code Generating and Computing (JDGC) Platform

The JDGC system is currently implemented in pure Java. The applications supposed to run on JDGC are also Java applications. This Java-based design is based on the following reasons:

- Portability of JDGC: In an open distributed computing scheme, the machine architectures of the participant hosts are different. They also may run different operating system. An important advantage of using Java in the system software is that the production is portable to almost all of these architectures and operating systems.
- Provide security solutions: Another advantage of using Java for system software is that it provides an intermediate level of security protection. The

bytecode verification is an effective solution to protect local system. Other primary security solutions such as sandbox and type safe feature are also developed based on Java. The security issues are further discussed in Section 6.3.

– Java is the most popular language for distributed object-oriented computing applications, because of its portability, security and the improvement of its performance. More and more applications are being developed using the Java language.

2.3 The Network Layer

The network layer protocol and API are not the emphasis of this thesis. They are detailed in a previous work. However the network layer is the basis on which the proposed JDGC platform and the distributed code generation method are built. The provided API functions are incorporated into the generated distributed code, as well as the runtime application manager. Therefore in this section we give a brief summary about the network layer. For further information, please refer to reference [15].

The network layer consists of two sets of protocols and their API functions. Firstly, the recruiting of an available server host among the system employs a group communication protocol. A server host in the scheme is required to start a server daemon in the network layer to monitor on a pre-assigned multicast address. A client, when requiring available server hosts, starts the process by invoking the recruit API function, which in turn sends a recruiting message on the multicast address.

Server hosts on receiving the recruiting message, reply to the client. Until the reply is acknowledged by the client, no action is taken on the server side. Once the reply is acknowledged, the server daemon creates a new Java virtual machine to receive and host the objects to be distributed from the client.

Secondly, the communications between the objects uses the ordinary uni-cast protocol. For Java applications, the communications are in the form of method invocation or field referencing. The network layer provides the functions for accessing the methods and fields of an object on a remote server host. The implementation of these functions is based on object serialization and RMI interface of Java [3]. Both the multicast and uni-cast protocols are implemented on the top of TCP/IP to ensure the reliable delivery.

2.4 Summary

This chapter presented the decentralized computing scheme and the software architecture of the JDGC platform. The key functions, the object level analysis and automatic code transformation, are overviewed, yet they are the emphasis of the following chapters. The network layer protocols, on which the whole system runs, were also summarized in this chapter.

CHAPTER 3

SYSTEM METHODOLOGY

The key function of the system is to transform the application by grouping objects based on an analysis of their communication affinities. The standard object-oriented application is then transformed to a form that can directly run on a distributed environment. The methodology consists of two stages, namely the object level analysis stage followed by the automatic distributed code generation stage. The whole process makes use of the Abstract Syntax Tree (AST) [22] to perform the function in a structured mode.

3.1 The Object Model of Applications

Before discussing the methodology of distributed code generation using object level analysis, the features of the objects of distributed applications are first summarized. This model of objects forms the basis for the methodology.

The applications under consideration and their objects have the following features:

– The applications are written in an object-oriented paradigm, and the only entity that can be manipulated at runtime is the object. The current implementation of JDGC is for Java applications, which are based on objects.

– The objects in the application are active objects. An active object is an object that potentially has its own process or thread of control. In a distributed application, the distributed objects may run on a different networked machine. Furthermore, an active object is the composition of two entities: a body and another standard Java object. In a distributed computing environment like JDGC, the body entity is the runtime environment on each machine, responsible for receiving incoming calls from the network, and invoking local Java objects.

– The runtime objects can communicate with each other by message passing via the network. In object-oriented paradigm, objects provide a structured interaction interface defined in the corresponding classes. In Java, the communication between objects could be done through method invocation or field referencing.

3.2 The Methodology of Distributed Code Generation using Object Level Analysis

The purpose of the method is to efficiently map the concurrent application to the system's computational resources, and to make this process automatic. The method strives to reduce the communication overhead between objects. Inter-machine communications are much more expensive than intra-machine communications. The method groups heavily communicating objects in the same machine so that they could use lightweight communication mechanisms in the generated code.

To achieve this, the method first analyzes the original application, and determines the volume of communications between its objects. Our method provides a strong and detailed analysis of the application, which could give object level communication relations. This object level analysis is an independent method, which is not only useful in our system but also a general method for extracting communication affinity metrics of object-oriented applications in the runtime object level. In our system, this information serves as the guidance in the object placement, which is incorporated into the generated distributed code.

The second part of the method is distributed code generation. The generated code runs directly on the distributed environment. The code generation includes a transformation process, which ports the program for single machine to the network machines. At runtime the distributed code selects a proper machine to distribute a newly created object according to its affinities to other objects in the system. The communications between objects are properly implemented in the generated code using either local or network communication mechanisms. After this process, the code generated integrates seamlessly into the network layer API and is able to run directly in the distributed environment.

As is indicated in Fig.2.2, in the process of executing an application in a distributed environment, the position of the above method is after the submission of standard (single-machine oriented) application by the user and before the system submission of the transformed code to the distributed environment. Users also have control over the method through parameters, such as the number of hosts to be recruited and whether to generated transformed source code aside from the bytecode.

3.3 Flow of Processing

The whole process of analysis and code generation is based on the Abstract Syntax Tree (AST) of the Java application. We firstly examine the structure and features of an AST and attributed AST, which is used throughout the discussions in the following chapters.

An AST is a tree representation of the semantics of a program. The representation is independent of the language. Whereas the concrete syntax of a certain programming languages incorporates many keywords and tokens, the abstract syntax is rather simpler, retaining only those components of the language needed to capture the real content and ultimately meaning of the program.

The AST is usually produced by a parser according to the grammar rules (or usually give as the BNF) of a language. Every node in the AST represents a terminal or non-terminal of the grammar. All the leaf nodes are terminals, while all the inner nodes are non-terminals. In every sub-tree of an AST, the parent node and its child nodes must conform to a rule in the grammar.

The AST itself does not contain all the semantic information of the program, because it only represents a context-free language. To add the contextual information in a computer program, the AST must be attributed or decorated (usually done by an attributor) to produce an attributed AST. The attributed tree then has the typing and other contextual information attached to various nodes. After this, the attributed AST

contains all the semantic information of the original program. Thus it is usually used as an intermediate but structured representation to facilitate further processing for a computer program.

Our proposed method uses the attributed AST representation throughout the object level analysis and the automatic distributed code generation. The flow of processing, from the input to the output, is described as follows.

The input to our method is standard multiple threaded Java application. This is done through a graphical user interface. One or more applications can be opened and processed simultaneously, and an application may contain multiple source files.

The entire procedure consists of six parts: namely parser, attributor, class-level analyzer, object-level analyzer, tree transformer and code generator. This is depicted in Fig.3.1.

The parser and attributor extracts the AST of a given Java program. The parser and attributor used here are the standard components in compilers. The obtained attributed AST is constructed to facilitate analysis of the following steps. The AST eliminates information that relates only to lexical or syntactic issues, but retains all the semantic of the original program.

The Class-Level Analyzer calculates the communication relationship between classes by analyzing the methods and their invocations. Note that existing forward/reverse software engineering tools only provide class-to-class analysis of interaction. Our

method uses a three dimensional class level analysis to give more detailed information. This is necessary for subsequent object-level analysis.

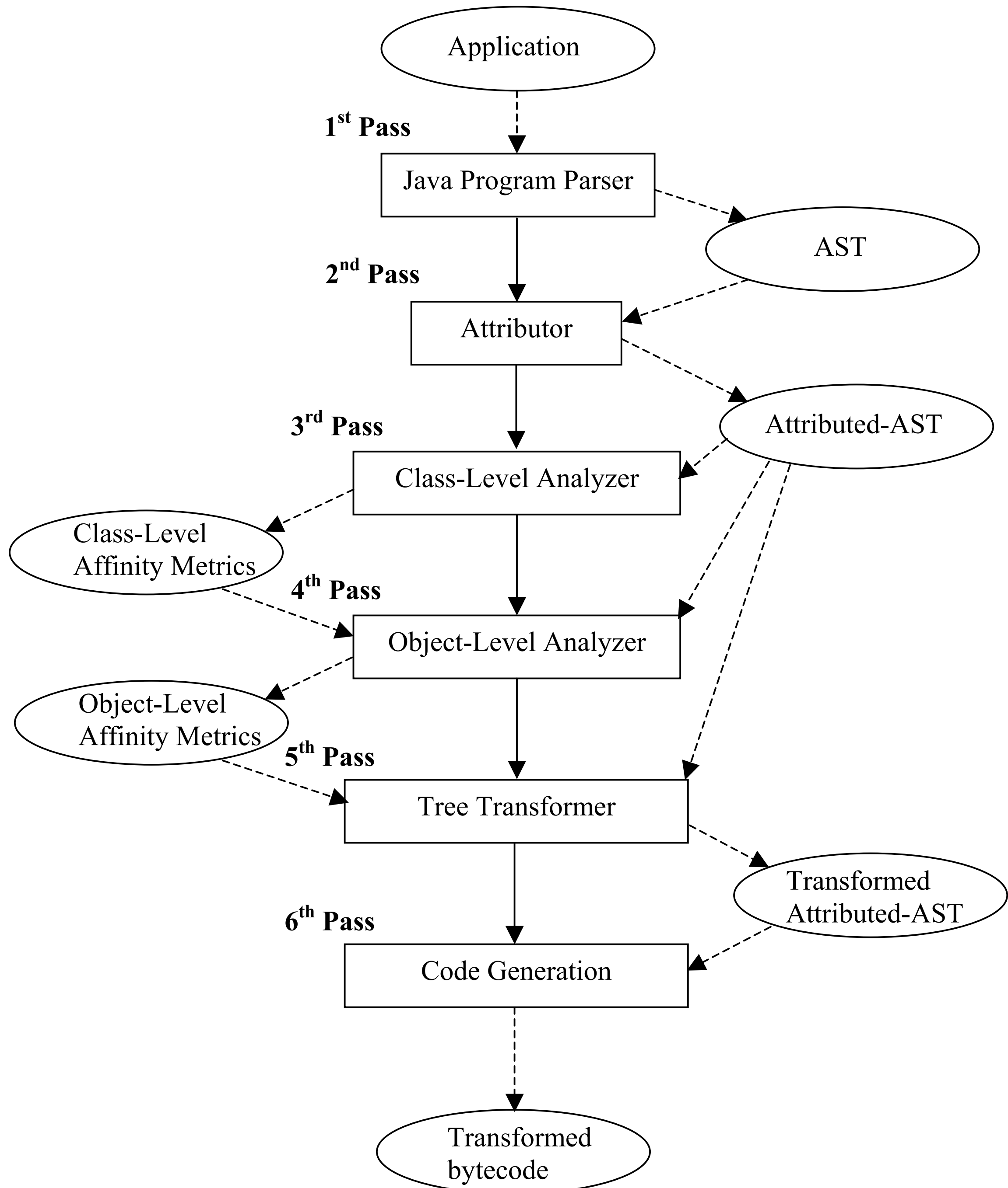

Fig.3.1. The Flow of Processing for the Object-Level Analysis and Distributed Code Generation

The Object-Level Analyzer analyzes every statement involving creating a new object (instance) and calculates the communication relationship between the new created instances and other object instances. The calculation of Object-level affinity is based on the 3 dimensional class-level affinity metrics. From this step, any instance created at runtime could be found in the resulting object-level affinity metrics, which gives the communication metrics between this object and others.

The tree transformer incorporates the object placement information derived from the object-level metrics into the AST. The object manipulations and interactions are re-implemented by the transformer using the network layer API functions. The result is a transformed attributed AST.

The code generation generates the distributed code and optionally the transformed source code using the transformed attributed AST.

Output of the process is the bytecode incorporated with the network layer API. It can be submitted to the distributed environment and be managed by the runtime application management system.

3.4 Summary

Based on an introduction to the object model of applications, this chapter described the methodology of distributed code generation using object-level analysis. The details of the two stages of this methodology are to be discussed in the next two chapters.

CHAPTER 4

THE METHOD OF OBJECT LEVEL ANALYSIS

The object level analysis is a general analysis method for object-oriented applications, providing detailed communication affinity information in the runtime object level. It uses a two level analyzer with an intermediate 3 dimensional communication metrics. The resultant metrics are used in the subsequent distributed code generation. However, the method itself is a general approach for extracting object-to-object communication affinity metrics and can also be used in other contexts.

4.1 Class-Level Analyzer and Three Dimensional Affinity Metrics

As described in Section 3.3, the parser and attributor are standard component in compilers. The output is an attributed AST of the application. After the AST is obtained, the next step is the analysis of communication affinity. This is done in two levels.

Currently there are many class-based analyzers. However, these class level methods are inadequate because they do not account for runtime instances. One class may have multiple instances, which could be accessed in very different ways and thus have different communication affinity metrics with other parts of the application. In our method, an enhanced class level analysis is employed as the first step of analysis. This section presents the key concepts and the details of class level analysis. Based on the result, an object level analysis is performed to produce the detailed object level metrics, which is the topic of Section 4.2.

4.1.1 The Three Dimensions

A typical 2-dimensional class-to-class affinity metrics is not sufficient for the object level analysis to be performed in the subsequent level. We therefore enhanced the Class-Level Analyzer to generate a 3-dimensional affinity metrics. The third dimension is the entry point from which (an instance of) a class is accessed.

Classes are executed by an invocation of their public methods. We analyze the communication from Class A to any other object by examining every entry point to Class A. An entry point refers to the point where the application's control flow can go into the Class A. In an object-oriented program, the entry points of a class are the beginning of every public method. Then we measure the communication from Class A to any other object if A is accessed through a certain entry point (e.g. through the invocation of a certain public method).

The Class-Level Analyzer will record every class and the entry points to each of them. This gives the first two dimensions of the affinity metrics, that is, (ClassFrom, EntryMethod). The third dimension is the visible objects within a certain entry point of a certain class. Note that in any method of class A, A only can communicate with the objects that are visible in this method. In object-oriented program these visible objects are accessed through their references. Therefore for each class, say A, and a certain entry method M, we analyze the communication from A to all the visible objects, or more specifically, the reference of a visible instance. A visible object can be a member variable of class A, a local variable defined in method M or a parameter passed to M. The object must have a unique reference so that it can be accessed in A. The Class-Level Analyzer analyzes every visible object for the communication affinity from A with respect to method M. The value is the number of bytes to be transferred to the object within this method.

Formally a single affinity index is associated with every 3-tuple (ClassFrom, EntryMethod, VisibleObjectReference), where VisibleObjectReference is one of the visible objects in the method EntryMethod of the class ClassFrom. The affinity index of (class A, method M of A, object reference O) is x, is interpreted as follows. If A is accessed through method M, then the communication affinity, or, the number of bytes to be transferred, from A to the visible object O is x. The concept of the three dimensions of object-oriented programs is illustrated in Fig.4.1.

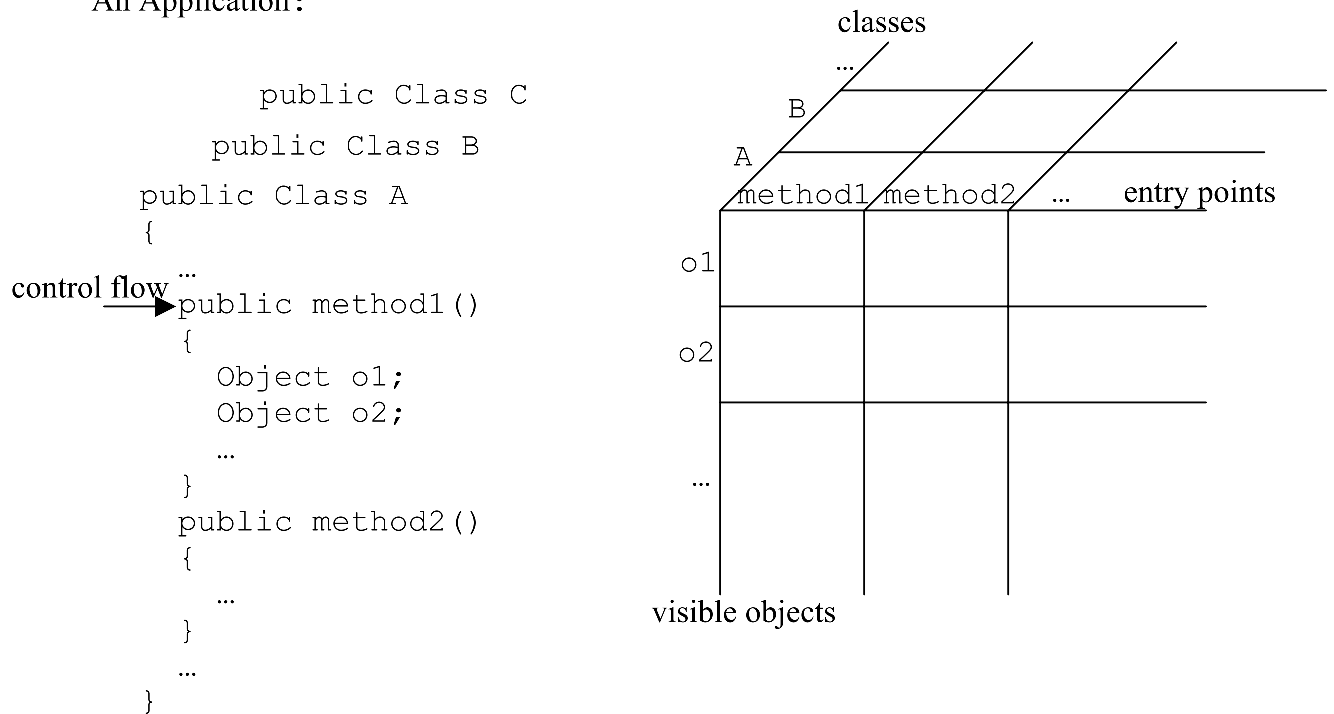

Fig.4.1. The Three Dimensions of Object-Oriented Program

4.1.2 Analyzing Procedure

The analyzing procedure uses the attributed AST obtained previously. The Class-Level Analyzer analyzes the public methods in all the classes one at a time, calculating the affinity index for each 3-tuple (FromClass, EntryMethod, VisibleObjectReference). The procedure is described as follows:

1. Search for all the classes defined in the application, and for each class, whose AST node is denoted by A, do the following steps.
    2. Search for all the methods (entry point) in A, and for each method in A, whose AST node is denoted by M do the following steps.

(i) Find all the visible object references at the beginning of method M, including the class A defined variables and the parameters passed to M. The visible objects forms a list OL.

(ii) Initialize the affinity index (aff) for each entry (A, M, OL[i]) as aff=0.

(iii)Given the AST node of method M, recursively scan its children AST nodes in order.

(iv)If a statement that contains invocation to OL[i]'s method is encountered, calculate the size of the parameters and return values in that invocation. If this statement is within a branch statement, the calculated size is multiplied by 0.5, and this size is added to the affinity of entry (A, M, OL[i]).

(v) If a statement that contains referencing to OL[i]'s member variable is encountered, calculate the size of the variable. If this statement is within a branch statement, the calculated size is multiplied by 0.5, and this size is added size to the affinity of entry (A, M, OL[i]).

(vi)If a method invocation of A itself (method invocation without naming the object or with the object name `this`) is encountered, scan the child nodes of that method and apply this procedure recursively.

The Class-Level Analyzer output affinity metrics of three dimensions and a single value is associated with each (ClassFrom, EntryMethod, VisibleObjectReference). These affinity metrics will be used in the Object-Level Analyzer.

4.2 Object-Level Analyzer

Once the detailed 3-dimensional class-level affinity metrics of Section 4.1.2 is obtained, the next step is to obtain the instance-to-instance affinity metrics. Object-Level Analyzer analyzes the affinity between any two instances that are potentially created at runtime. It outputs a two dimensional affinity metrics. This information is then incorporated into the transformed distributed code, which at runtime distribute the objects according to the affinities.

4.2.1 Visible Scope Analysis and Mapping of Affinity Metrics

The purpose of the Object-Level Analyzer is to extract affinity between runtime instances. In an object-oriented program, any runtime instance must be created using certain statements in the class definitions. Therefore, any potentially existing instance could be associated with a new instance statement. In Java this is typically a `new` statement. The Object-Level Analyzer steps through the application's AST for every `new` statement, and extracts the affinity metrics between the created object in the statement and other objects in the application. This passing of the AST follows the order of the execution sequence and uses a depth-first method.

The affinity values are calculated by mapping the class-level affinity values to runtime objects. A class-level affinity value is associated with a class, a method and a variable reference, while an object-level affinity value is associated with any two `new` statements in the application. The Object-Level Analyzer performs this translation.

This is done mainly through analyzing the visible scope of objects, and the propagation of object references.

In any object-oriented application, one object, say object O1, can invoke the method of, or reference a member variable of, another object O2, only if O1 has the reference of O2, namely O2 is visible to O1. Therefore, in order to analyze the communications to an object, we need to first analyze the visible scope of the object, because all possible communications will only come from the visible scope. The visible scope of an object is described as follows.

When an object is initially created, it is only visible within the object that creates the new one, namely parent object, as well as the new object itself (referred as `this`). During the subsequent execution, the reference of this object may be propagated to other objects. The reference can be passed to a destination object as a parameter in an invocation of the object's method, or the reference may be assigned to a member variable of the destination object. In both cases the visible scope of the propagated (reference of) object extends to the destination object.

The propagation of an object's reference can only initiate from its parent object or itself. Once the visible scope extends to a third object, this object may in turn propagate the reference further to other objects. In summary, any object in an object-oriented application is only visible in the object itself (referred as `this`), in the parent object of it, and in all the objects reached in the propagation.

Based on the above discussion, the analyzer performs the visible scope analysis and mapping of affinity metrics in one pass of the AST. The analyzer first finds the `main` method of the application. It then steps through all possible control flows of the application. On encountering a `new` statement, the analyzer calculates the affinity of this newly created object in two parts. First it obtains the communication affinity from the object that creates the new one, namely parent object, to the new object. Secondly it obtains the affinity from any other objects to the new object.

In the first case we analyze the affinity from the parent object P to the new object N. This analysis is based on the visible scope of N within P. Suppose the new object N is created within the method M of P, and assigned to the object variable name O. Then the affinity from P to N is directly obtainable through mapping the instance N to the name O. The affinity from P to N is given by the affinity value of the 3-tuple (P, M, O) in the resultant metrics of the Class-Level Analyzer.

The affinity from any other object A to the new object N also depends on the visible scope of N, as well as the propagation of the new object. As discussed above, any other object A can access N only if A has the reference of object N, namely N is visible to A. Originally the new object N is only visible in P (parent object of N), as well as N itself. Then the reference of N can be propagated from P or N to other objects. In our method we first analyze the propagation of instance N and find the visible scope of N. For example, if the reference of N is propagated to the method M of object A through parameter passing, then N is visible in A and object A will possibly have communication to object N. If in A’s class, object N is accessed through the reference O, then the affinity from object A to object N is directly obtainable through mapping

the visible object reference O to the object N, that is, the value is given by the affinity value of the 3-tuple (A's class, M, O). The Object-Level Analyzer analyzes all propagation of object N, and sums up the obtained affinity values for each pair of objects to get the object-level affinity related to object N.

The resultant metrics are two dimensional. Each entry is associated with two `new` statements. The value represents the affinity between the two objects created by these two `new` statements.

4.2.2 Analyzing Procedure

The implementation procedure of the Object-Level Analyzer is described as follows:

1. Find the main() method of the application, where the control flow starts, and get the AST node for main().
2. Given the AST node of the current analyzing method, recursively scan its children nodes in order.

   A. If a `new` statement is encountered, then do the following:

      – Add an entry for this statement, and init the affinity (aff) between this entry to any others as aff = 0.

      – Find the class and method that is currently being analyzed, denoted by C and M. Find the reference of the new created object, denoted by O.

      – Get the value in class-level affinity metrics (C, M, O), which is the affinity from the parent object to the new created object.

      – Add this value into the proper entry in the object-level affinity metrics.

B. If a statement where any previously created object, denoted by N, is passed to another object A through parameter passing of one of A's method is encountered, then object N is said to be propagated to object A. Therefore N is visible in A, and A possibly has communication to N. Do the following:
   - Find the class of A and invoked method, denoted by C and M. Find the reference of the object N within method M, denoted by O.
   - Get the value in class-level affinity metrics (C, M, O), which is the affinity from the non-parent object A to the propagated object N.
   - Add this value into the proper entry in the object-level affinity metrics.

C. If any method invocation is encountered, then get the AST node for this method, and apply this procedure recursively.

The above recursive analysis procedure is summarized in the flow chart Fig.4.2. The initial value of `NODE` is the AST node for the main() method of the application.

After the two levels of analyzers, a 2 dimensional affinity metrics is obtained. Each entry represents the communication affinity between two runtime objects in the application.

4.3 Summary

The resultant metrics directly give the communication affinity between two objects that are potentially created at runtime. They are used as a guidance of object placement in the distributed code generation.

The method proposed in this chapter is independent of the JDGC platform or the subsequent distributed code generation method. It is able to analyze any object-oriented application, and to extract the object-to-object communications.

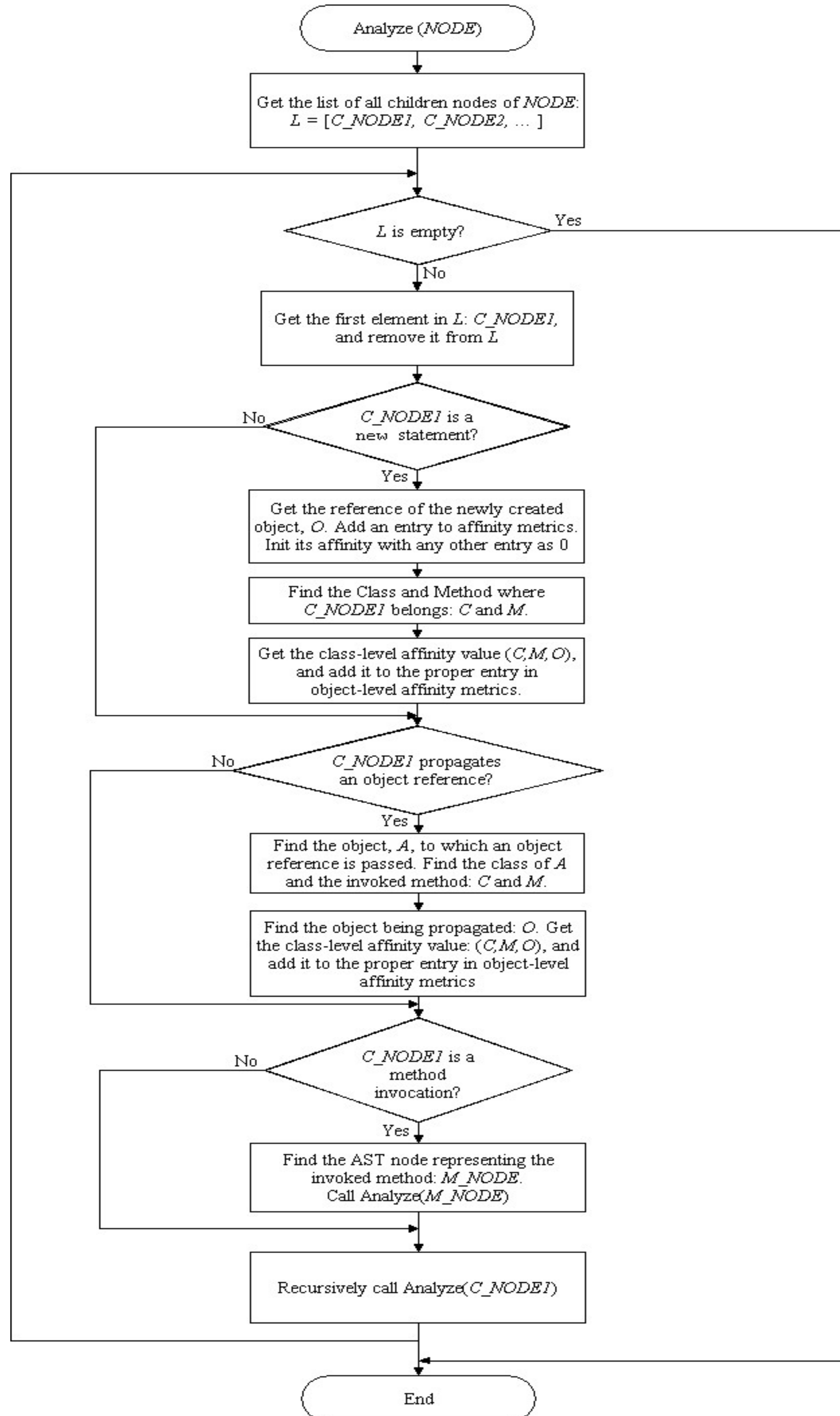

Fig.4.2. Flow Chart of the Object-Level Analysis Procedure

CHAPTER 5

DISTRIBUTED CODE GENERATION

Using the object level affinity metrics, the original application program is automatically ported to the distributed code that runs on the network machines. The generated code incorporates the runtime object placement according to the affinities computed. This process is completely automatic and does not require a special programming paradigm for the user's program. A normal Java program for a single machine is transformed to a program that runs on the network using the underlying API, handling object manipulations and communications on network machines. The transformer performs the code transformation, traditionally performed by hand, in a structured mode.

5.1 Code Generation in Two Steps

The code generation is the last step of compilers. The input of a code generator is the attributed AST obtained from the parser and attributor. Since the AST records all the semantic information of the original program, the code generator is able to synthesize the object code, which is another form of the program able to run on the underlying executing environment. For Java applications, the object code conforms to the

specification of bytecode, the code format that is interpreted by the Java virtual machines.

In the case of single machine, the executing environment is one Java virtual machine. The bytecode produced by standard Java compilers is sufficient to run on a single virtual machine. However, in the case of distributed computing, the executing environment is a group of Java virtual machines connected to a network. The object code in this case needs to handle the acquirement of networked machines, the object distribution according to the communication requirements and the transmission of data between the objects on different machines.

One solution for the production of distributed code is to put the above special considerations in the application developer's side. In many existing systems, the programming methods for individual systems were proposed to facilitate the development of distributed code. Developers use these system-specific programming rules to handle the object distribution and communications explicitly in the source code level. However, these solutions have two main drawbacks:

- The distributed code is not generated automatically. Developers have to program for the coordination functions, aside from the computation tasks.
- The distributed code is not portable. Because the programming rules are system-specific.

In consideration of the above problems, we propose a method of distributed code generation, which possesses the following features:

- The input code is standard Java applications. There is no requirement for any special programming techniques or language features.
- Perform completely automatic transformation of a concurrent program for single machine to one that runs on networked machines.

To achieve this objective, our distributed code generation method consists of two consecutive steps.

Firstly, the AST for the original program is transformed. The sub ASTs representing the object creations and object interactions are re-implemented. The transformed program incorporates the object placement according to their communication affinity metric, and contains the functions of handling object distribution and inter-machine communications using the network layer API. The transformed AST will represent the complete semantic of a distributed program that correctly runs on the networked machines.

Secondly, the transformed AST is ported to bytecode using a Java code generator. This second step is standard. Thus, we only examine the tree transformation method in detail.

5.2 Tree Transformation

In order to transform general program, the tree transformer uses the AST of the original program. The transformation is performed as the 5th pass of the AST.

5.2.1 Changes for Object Placement and Communications

Having the communication metrics between runtime objects, the placement of any object when it is created can be calculated. If two objects are placed on different machines at runtime, the communications between them also need special treatment. The system makes use of the underlying network layer API to re-implement the code for object placement and communications.

Firstly, the `new` statement will be re-implemented by a procedure that places the new object according to the affinity metrics. Initially every potential object in the object level affinity metrics is regarded as a separate group. When an object is created during the execution, it is assigned to one of the recruited machines, each of which hosts a group of objects at runtime. The procedure calculates the placement of object in the following way. The system has the information for all the available machines recruited by the application, as well as the existing objects in every machine. Then, according to the object-level affinity metrics, it calculates the affinity between the new object and every machine, that is, the sum of affinity values between the new object and every object in the machine. The new object will be placed in the machine that has the largest affinity value with the new object. That is, suppose the application recruits m machines $M_1, M_2, \ldots, M_m$, the ith machine has the existing objects $O_{i,1}, O_{i,2}, \ldots, O_{i,ni}$, and a new object O_{new} is created. The machine to place O_{new}, denoted by M_{place}, satisfies:

$$Affinity(O_{new}, M_{place}) = \max\{ Affinity(O_{new}, M_i) \},\ i = 1,2, \ldots, m,$$

$$\text{where } Affinity(O_{new}, M_i) = \Sigma\{ Affinity(O_{new}, O_{i,k}) \},\ k = 1,2, \ldots, n_i$$

The affinity between two objects, $Affinity(O_{new}, O_{i,k})$, is directly obtainable from the object-level affinity metrics. The system uses the network layer API functions to implement the creation operation on the selected machine. If the new instance should

be placed in a certain network machine, then the network layer API will be used to do the inter-machine creation. If the new instance should be placed in the local machine, then it is created as normal.

Secondly, the inter-object communication statements should also be re-implemented to correctly handle intra or inter machine data delivery. This also makes use of the network layer API functions. Every communication statement will be replaced by a procedure, which first decides whether the destination instance is currently located at local machine or a certain network machine. If it is local, then the communication is done as normal. If it is in a certain network machine, then the network layer API functions will be used to handle inter-machine communication.

5.2.2 The Wrapper Class

To facilitate the automatic transformation, a wrapper class is used to replace the reference to any user-defined classes. The Wrapper class hides the difference between a local and a remote object at compile time and runtime. Any created user-defined object in the program is represented in the transformed code by a Wrapper class. If the object is local, then the corresponding Wrapper just encapsulates the normal object in the same virtual machine. If the object is remote, then the corresponding Wrapper encapsulates a RObject object, which is the reference to this remote object. The Wrapper class has a flag to indicate whether the inner object is local or remote.

As an auxiliary class in any transformed application, Wrapper provides the unified creation facility that handles both local and remote objects. The object-level affinity

metrics are available to the Wrapper class. The transformed program is also aware of the current correspondence between created objects and available hosts in the data structure for runtime application management of Section 6.2. When a new object is created, the corresponding Wrapper object selects the host that has the largest affinity value to the new object. Then it can check whether the selected host is the local machine or a certain remote host recruited. In the case of a remote host, the Wrapper object will use the network layer API functions to make a remote creation.

```
Template: Class Wrapper
{
  Field: boolean flag;

  Field: Object localObj;

  Field: Robject remoteObj;
  …

  Method: Constructor(args)

  Method: Invoke(method, args)
  …
}
```

If `flag` indicates local:
`localObj` points to the normal instance
`remoteObj` is null

If `flag` indicates remote:
`localObj` is null
`remoteObj` points to the reference to remote object

Unified interfaces for object creations and invocations

Fig.5.1. Template of Wrapper Class

Wrapper also provides the unified method invocation facility that handles both normal invocation on the same machine or those that involves inter machine communication. When an existing Wrapper object gets an invocation, the object could decide from the flag whether the wrapped actual object is remote or local. Fig.5.1. describes the template of the construction and the interfaces of the Wrapper class.

With these facilities in Wrapper class, it is sufficient to replace any operations on user-defined objects by the operations on corresponding Wrapper objects. The transformed code will not contain manipulated code for the user-defined classes, but contain code for Wrapper objects.

Wrapper objects are local. They act as local represents of created objects. They hide the difference of local and remote cases. The transformed code only operates on Wrapper objects, and not on the represented objects.

5.2.3 Addition and Substitution of Sub AST

With the aid of Wrapper class, the transformation is well structured. The transformation rules are simple, without any contextual logic. For example, there is no need to analyze the location of involved objects before the transformer can decide which mechanism should be used in an inter-object method invocation. All the transformations are done through adding or substituting a sub AST in the original AST. To ensure the code has the necessary functions and conform to the underlying protocol, the transformation rules are summarized as follows:

- Import the necessary classes, including object level affinity metrics data structure, the Wrapper class, and the network layer API. Several import clauses should be added to the beginning of the classes. This is done by adding a sub AST with the root of an Import node.
- Make every defined class serializable, that is, it should implement the Serializable interface. This is required because the underlying layer is based on

Java RMI protocol. An Implement sub AST is appended as a child node of every Class node.

- At the beginning of program, generate and add the procedure of recruiting hosts. The generated sub AST represents a block statement, which contains the operations of recruiting hosts. This sub AST is then added to the MethodBody node of the `main()` method.
- Pass through the whole AST of the program, and for any sub AST of creating an object of a defined class, generate the corresponding sub AST of creating a Wrapper object, and make the tree substitution.
- Pass through the whole AST of the program, and for any sub AST of method invocation to object of defined class, generate the corresponding sub AST using previously created Wrapper object, and make the tree substitution.
- The parameters of invocation must also be transformed to objects for primitive type, as the underlying API only transfers objects. For method with primitive return type, certain functions of Wrapper are called to strip the returned object into corresponding primitive data, so that the function call incorporates with the program seamlessly.

The tree transformer performs the 5th pass of the AST. Each class and their methods are examined. All the generated sub AST are parsed at generation time. The transformed AST is again attributed and directly ported to bytecode. Optionally, the source code that represents the transformed program can also be generated.

5.3 Summary

Chapter 4 and 5 have examined in detail the key function of the JDGC platform, that is, the distributed code generation with object level analysis. The resultant code contains the auxiliary classes and the network layer API. It will be submitted to the networked machines, and be managed by the runtime application manager.

CHAPTER 6

JDGC SYSTEM AND APPLICATIONS

A Java Distributed code Generating and Computing (JDGC) platform is proposed in this thesis. A graphical user interface integrates the functions for a host to run as a server and as a client. The runtime application manager is an essential component of JDGC, which provides the runtime environment for the analysis, transformation and execution of applications. Two sample applications running on JDGC platform are presented. We test the execution time for the two applications in different situations and discuss the result of the tests.

6.1 User Interface

The graphical user interface integrates the functions of a server host and a client host. Users can open and transform their own Java applications, and to submit the generated distributed code to the system, as well as to start client process all in this unified environment. The UI is built on the swing API of Java.

Fig.6.1 is a snapshot of the graphical user interface of JDGC running on Win32 system. The figure shows the situation where a client application is opened and transformed.

The currently opened files are shown in the source code pane. Pressing the transform button will transforms the source standard Java program to the one that runs on the network using the underlying API. The transformation uses the parameters specified in the left part. If the user checked the emit source box, then the result transformed Java code will be shown in the result pane. The system outputs of the transformation are redirected to the application log pane. Any compiling error and error during analyzing and transformation will be displayed in the pane.

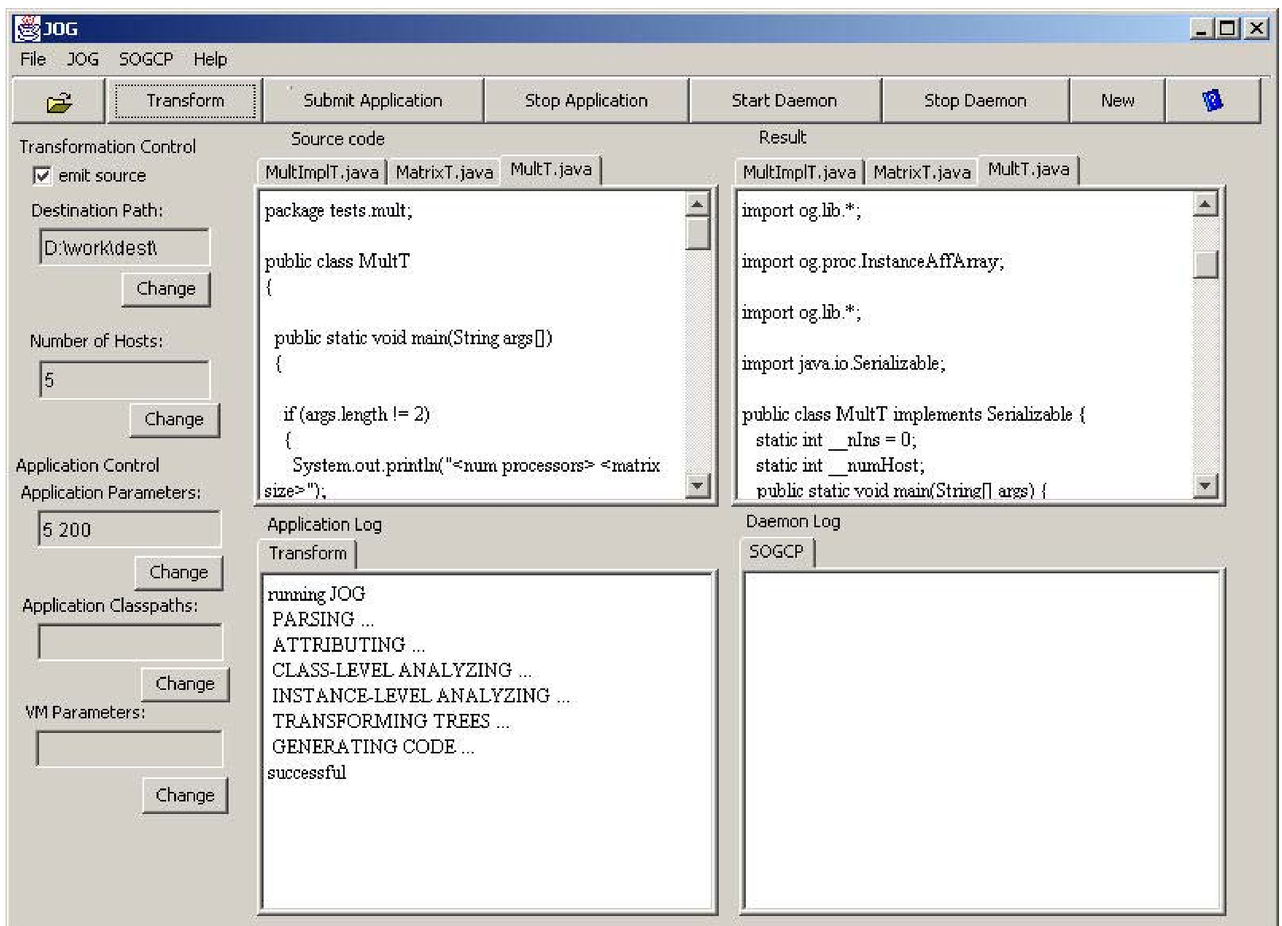

Fig.6.1. User Interface: Transformation of Application

After submission of the transformed application, the system will create a new pane within the application log pane for the running application. The system extracts the I/O

stream from the application's process so that any output from the process can be captured and displayed in the application's pane. Users can also kill the application manually using the stop application button.

The start daemon and stop daemon buttons are to start and stop a server daemon which responses to recruiting request and perform computations from others. The daemon pane will display the actions to the server daemon process. Fig.6.2. shows a typical output from the submitted application and the server daemon.

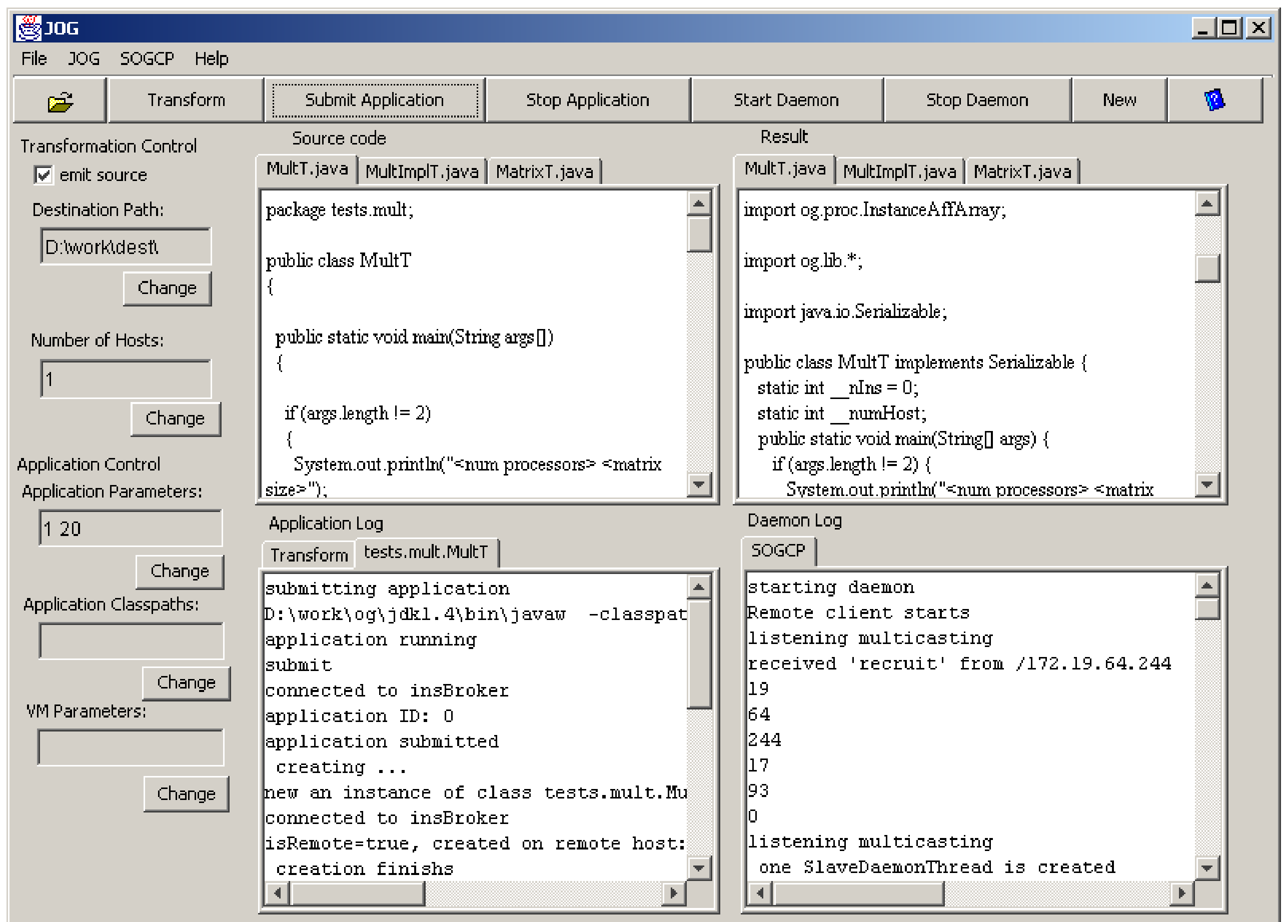

Fig.6.2. User Interface: Submission and Execution of Application

The left part includes the parameters that users have control over. In the transformation control, users can specify whether to emit source. If not, then the system only generates the bytecode, but not the transformed source code. Users can also change the destination path to save the generated files, and the number of hosts to be recruited.

In the application control, users can specify the parameters for their individual applications. The application classpaths are the paths in which the application search for its necessary classes. The VM parameters are for the options of the virtual machine, such as JIT options, garbage collection options and heap size options.

Users are allowed to open and submit several applications simultaneously. Each application will have a separate window, which shows its original code, transformation result and execution output. Pressing the new button will bring up a new window. Closing one window will only exit the application contained in that window, but may not result in the whole system to exit. The system exits when all windows are closed. The system maintains a global and dynamic data structure holding the information for all applications at a certain time. All the above functions are also provided in the menu.

6.2 Runtime Application Manager

For every participant host, all the applications and recruited hosts are recorded in a runtime application manger in the local JDGC platform. It always contains the current scenario/status of the dynamic system. More concretely, a local data structure in the manager contains:

– The global information about every application submitted in this host.

- The objects currently exist in every running application. Once an object is created during the execution of an application, the manager adds the information of the object in the data structure.
- The hosts used by running applications. Once an application successfully recruits a remote server host, the manager adds the information of this host in the data structure.

The data structure for runtime application management has three kinds of nodes, namely *application nodes*, *object nodes* and *host nodes*. The data structure is organized as a vector of application nodes. The structure of an application node is as follows.

Application node:

- Application ID.
- Affinity metrics of this application.
- Reference to a vector of object nodes that belongs to this application.
- Reference to a vector of host nodes that are used in this application.

In the structure, the application ID is used for uniquely identifying the application in the local system. The affinity metrics is used for object placement. The reference to a vector of object nodes contains the information for the objects in the application at the current time. The structure of an object node is as follows.

Object node:

- Object ID.
- A reference to a remote object if the object resides in a remote machine, or an object reference if the instance resides in local machine.
- A reference to a host node, which represents the host where this object resides.
- Miscellaneous information about the object.

- Backup image information for the object.

The object ID is not global to the local system. It is local to each application and identifies a certain object within the application. The node also contains a reference to access the real object. In a remote case it is a reference to a remote object. In a local case it is a reference to a local object. This reference enables the manager to access the local/remote object when it performs the object backup and recovery. Miscellaneous information includes the class name, instance name, index in the affinity metrics, etc. The backup image information is also used for the crash recovery discussed later. A reference to a host node represents the host where this object resides. All the host nodes in all applications are linked in a global vector of host nodes. One host node is maintained for each physical host. If different applications/objects use the same hosts, their reference of host nodes will point to the same host nodes. The structure of a host node is as follows.

Host node:

- Host ID.
- A reference to a remote server object, which represents this host.
- The number of objects in this host.

The host ID is global to the local system. The structure contains the reference to the remote server object to access the remote host. It also has the information for the number of objects that reside in this host at the current time.

All these nodes are organized in two global cross-linked lists. One global list is the list of all application nodes. As described above, the application node will contain its object nodes, which in turn points to a host node where the object resides. The data structure is illustrated in Fig.6.3.

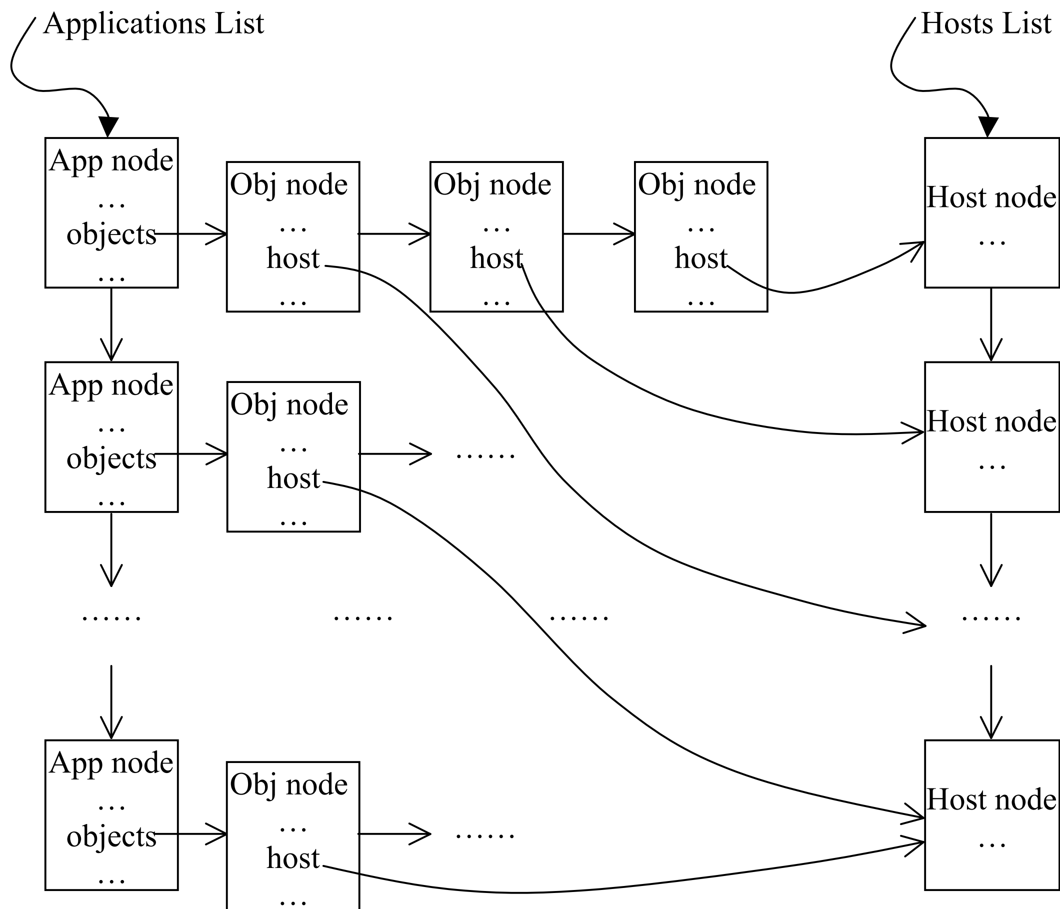

Fig.6.3. The Data Structure in Runtime Application Manager

An important function provided in the runtime application manager is the backing up and crash recovery. The system will backup the objects in local host during the lifetime of the objects. The backup image information will be held in each object node. During an invocation, if the remote host crashes, does not response or has other exceptions, the system will automatically retry the invocation for n number of times. After that, the remote host or server daemon is considered crashed. The system will follow a similar procedure as the object creation procedure to select a host and migrate the object using the backup image information in the corresponding local object node. After the object

is created, the system re-performs the invocation. All these will be done by the system automatically without the user's intervention.

6.3 A Discussion on Security Issues

The distributed computing environment discussed in this thesis is an open system, where any host can participate in or quit from the system at will. In such an environment, the security is another important aspect. In this section, we classify the major security issues in a distributed computing environment and provide a discussion on the possible solutions for these issues.

There are two separate security issues in the discussed computing environment. The first is the protection of the server hosts from malicious programs. This aspect of the security problem is similar in nature to that of a multi-user computer system, where individual programs must be protected from other programs running in the same physical system. This problem has been investigated and addressed in the Java technology. The virtual machine concept proposed for Java provides a practical solution for this problem. The Java virtual machine has a byte code verification mechanism [23]. Before loading the bytecode of a defined class, the code verifier checks whether it possibly poses security threats to the local system. The next line of defense is the Security Manager mechanism [24]. The Security Manager is a single module that can perform runtime checks on dangerous operations of the distributed code. Code in the Java library consults the Security Manager whenever a dangerous operation is about to be attempted. According to a predefined security policy, the Security Manager is given a chance to restrict the operation by generating a Security

Exception at runtime. These security solutions in Java are one important reason for choosing Java as the implementation technique of JDGC platform.

The second issue, which is the protection of the distributed program executing in a malicious environment (remote server), can be divided into two parts. The first part is secrecy, which requires the content of data and semantic of codes in the program be shielded from the remote server. The second part is on integrity, which requires the un-tampered execution of the distributed program by the un-trusted server, and therefore ensuring the correctness of the results.

For secrecy problems, the distributed computing architecture could provide an intermediate level of protection. The risk of secrecy is reduced somewhat by ensuring that no one server host receives the complete set of private information. For stronger protection of secrecy, the main solution is to use encryption techniques. The most widely used technique is the encoding or obfuscation of code. Popular Java obfuscators are RetroGuard, CodeShield, JProof, etc. These tools apply three main strategies to the bytecode; namely layout obfuscation, data aggregation/decomposition and control aggregation/computation [25]. Because the three strategies do not change the semantic of the code, these tools can also be used for the distributed code generated in our proposal. If strong secrecy protection is necessary, it is straightforward to insert an obfuscator into the JDGC architecture between the distributed code generation and the submission of application.

Although the above secrecy solutions are widely used, these techniques are not able to provide mathematically provable encryption for the content of distributed code. A

similar problem in data encryption has been achieved in the RSA cryptography [26], which can encrypt the data with a provable strength. However, in the case of program encryption, a viable protection strategy has to ensure that the encrypted program is still executable, and at the same time hides the content of the program. The only available program encryption method is the mobile cryptography proposed in [27]. The strength of this method is based on function decomposition. However, this technique only applies to a certain class of program.

The second part of protecting distributed code is on integrity. This requires the un-tampered execution of the distributed program by the un-trusted server host. Violation of integrity must be detected as early as possible in order to recognize the malicious clients and to avoid using the wrong results. To deal with the integrity problem, there are three general approaches; organizational, hardware-based, and software-based solutions.

Firstly, the organizational approach relies on "trusted" server hosts. In the proposal of [28], integrity is guaranteed by not utilizing un-trusted hosts in the system. A drawback of this approach is that it requires an authentication for every usable host. This may limits the number of participants in the system. Secondly, hardware-based schemes employ tamperproof hardware in all potentially malicious servers to guarantee the correctness of execution [29]. Similar to the organizational based scheme, this approach also has the drawback of limiting the number of participant hosts. The third class of solution is the software-based schemes, which rely on the software structure in the recruiting host and the server host to ensure integrity. Such a design scales better, and is more suitable for an Internet computer environment. Generally there are two

categories of approaches in software-based methods: tampering prevention and tampering detection.

In tampering prevention, safeguards are inserted before execution. Proposals of [25][27] are examples of tampering prevention approaches. These methods employ the similar strategies as the secrecy protection to hide the information of the distributed programs from server hosts. The prevention approaches mainly address the purposeful tampering behaviors based on the semantics of a program.

Tampering detection allows detecting both tampering based on the knowledge of the program, and non-semantic-based tampering done without such knowledge. It observes the execution and discovers any divergence from the correct execution. Software-based tampering detection scheme often uses techniques in Fault-Tolerant Computing, which provides various mechanisms to detect execution fault. An efficient approach is software signature, widely investigated in Fault-Tolerant Computing. It is used to detect both control flow fault and single instruction fault. Task replication [30] and integrity constraint [31] are another two methods. In task replication, the same task is executed in two or more different sites. Consistency of results from different sites are checked. In integrity constraint method, assertions (invariants) are inserted into the program to identify wrong execution.

6.4 Applications

The proposed JDGC platform is an integrated distributed computing system supporting Java applications. In this section we present two sample applications that run on the JDGC platform.

- Program MultT. This application performs multiplication of two large matrices of floating point numbers.
- Program Curve. This application performs curve fitting for a series of data using the least square error criteria.

The two applications are tested in different situations where one or more single threaded and multiple threaded applications are submitted to the system. In the single threaded case, all the computation is done within one object. Multiple such applications can be submitted, and their computations are distributed by the system to available hosts. In the tests for multiple threaded applications, the computation of an application is done in multiple concurrent objects that can be distributed to available hosts. More than one multiple threaded applications may also be submitted simultaneously. The applications' average execution times are recorded for these situations. The units of execution time in the following tables are in seconds (sec).

Firstly, the applications are tested as single threaded applications. For the program MultT, the order of the matrix is 1000. For the program Curve, the size of data series is 1600, and the precision of parameters is 10^{-4}. The single threaded applications are tested in the following situations:

– The system consists of only one available host. We increase the number of applications submitted simultaneously.

Table.6.1 The Average Execution Time with One Host

MultT (1000)	Num of Apps	1	2	4
	Execution Time	104.408 sec.	209.279 sec.	421.974 sec.
Curve (1600, 10^{-4})	Num of Apps	1	2	4
	Execution Time	246.943 sec.	489.810 sec.	975.951 sec.

As is show in Table.6.1, when the number of applications increases, the average execution time for the applications also increases. This is because all the applications share the CPU time of the only host. The execution time is roughly proportional to the number of applications.

– There are 4 applications submitted simultaneously. We increase the number of hosts in the system, and record the average execution time of the applications.

Table.6.2 The Average Execution Time for Four Simultaneous Applications

MultT (1000)	Num of Hosts	1	2	4
	Execution Time	421.974 sec.	217.812 sec.	115.123 sec.
Curve (1600, 10^{-4})	Num of Hosts	1	2	4
	Execution Time	975.951 sec.	488.902 sec.	260.160 sec.

In this test for 4 applications, as we increase the number of server host, the average execution time decreases. When there are two hosts available, each of them receives two applications. In the case of four hosts, each application is sent to one host to be computed simultaneously. The execution time with n hosts is roughly equal to, but slightly longer than, 1/n of the execution time with one host. This overhead is due to the time of launching the applications on a networked machine.

– We simultaneously increase the number of hosts and the number of submitted applications.

Table.6.3 The Average Execution Time When Num of Apps Equals Num of Hosts

MultT (1000)	Num of Apps	1	2	4
	Num of Hosts	1	2	4
	Execution Time	104.408 sec.	108.086 sec.	115.123 sec.
Curve (1600, 10^{-4})	Num of Apps	1	2	4
	Num of Hosts	1	2	4
	Execution Time	246.943 sec.	251.577 sec.	260.160 sec.

If we keep the number of applications equal to the number of hosts, then as is shown above, the average execution time of the applications does not increase significantly, but it remains within 104 to 116 seconds. The slight increase of execution time reflects the overhead as the number of applications increase. This overhead is solely due to the time for initializing application over network, but there are no communication overheads between the four independent applications.

Secondly, the applications are tested as multiple threaded applications. For the program MultT, the order of the matrix is 2400. For the program Curve, the size of data series is 3200, and the precision of parameters is 10^{-6}. The applications are tested in the following situations:

– Firstly we test the performance of one application with four concurrent threads (objects). The applications are submitted to the system with one, two and four hosts recruited. We also compare these execution times of multiple threaded application with a single threaded application for the same problem.

Table.6.4 The Average Execution Time of Multiple Threaded Applications

MultT (2400)	Threads	Single	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	1	2	4
	Execution time	1401.336 sec.	1582.071 sec.	809.992 sec.	426.507 sec.
	Speed up	1	0.886	1.730	3.286
Curve ($3200, 10^{-6}$)	Threads	Single	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	1	2	4
	Execution Time	1164.755 sec.	1255.500 sec.	652.834 sec.	335.265 sec.
	Speed up	1	0.928	1.784	3.474

The Table.6.4 shows the execution times and speedups of a multiple threaded application. If the application is implemented as multiple threaded and runs on a single machine, it is slower than a single threaded implementation. This is due to the context switching on the single machine, as well as the local communication between the multiple threads.

When there are more server hosts available to the client host, the multiple threaded application outperforms the single threaded one running on a single machine. The speedups, however, are below the increase of the number of hosts. The overhead is partly incurred by the time of transferring and initializing the objects on a networked machine. This cause is similar to the case of multiple single threaded applications. On the other hand, for a multiple threaded application, the overhead is also due to the communications between the multiple threads of an application. These overheads are measured and summarized in Table.6.5.

Table.6.5 The Overheads in Creation and Communication

MultT (2400)	Threads	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	2	4
	Creation Overhead	11.827 sec.	13.197 sec.	13.252 sec.
	Communication Overhead	2.205 sec.	15.678 sec.	20.067 sec.
Curve ($3200, 10^{-6}$)	Threads	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	2	4
	Creation Overhead	10.619 sec.	11.702 sec.	11.969 sec.
	Communication Overhead	0.366 sec.	11.347 sec.	25.874 sec.

The creation overhead in the above table is the time of initializing the four concurrent objects on the recruited hosts. The communication overhead is the time spent by one thread in its communications with other threads. It is obtained by summing up the execution time of all communication operations with other threads.

- Having measured the speedups of execution time, next we validate the effectiveness of our method by comparing it with two other object allocation methods; namely, the random allocation and the manual (worst case) allocation. We use the same problem parameters as above. In this test the two applications are implemented as 16 concurrent threads (objects). The number of recruited hosts is fixed to 4. In the random case, every recruited host is randomly assigned four objects without considering their communication relations. In the manual case, the objects are manually allocated to the hosts according to the worst case of all combinations. The comparison of their execution time is summarized in Table.6.6.

Table.6.6 Comparison of Different Object Allocation Methods

MultT (2400)		Random	Manual	Proposed Method
	Threads/Hosts	16/4	16/4	16/4
	Execution Time	473.956 sec.	505.918 sec.	431.518 sec.
Curve ($3200, 10^{-6}$)		Random	Manual	Proposed Method
	Threads/Hosts	16/4	16/4	16/4
	Execution Time	380.758 sec.	398.648 sec.	344.585 sec.

Comparing execution times, the proposed method outperforms the random allocation and the manual (worst case) allocation method. The difference of performance is due to the different communication overheads when using different methods. In the worst case, objects that have the largest communication affinity are manually separated into different hosts so that the overheads introduced are larger than the other cases. In the random case, the execution time fluctuates in different trials. The average execution

time, however, is between the manual method and the proposed method. These comparison tests are done in a local area network, where the inter-machine communications are fairly fast. If the test is extended to a wide area network or the Internet environment, larger difference between the three cases can be expected.

– Finally, to show the system's support for more than one multiple threaded applications, we submit two applications with 4 threads simultaneously, and increase the number of recruited hosts.

Table.6.7 The Support for Multiple Applications

2 * MultT (2400)	Threads	Single	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	1	2	4
	Execution Time	2950.384 sec.	3168.250 sec.	1628.010 sec.	852.082 sec.
	Speed up	1	0.931	1.812	3.463
2 * Curve (3200, 10^{-6})	Threads	Single	Multiple (4)	Multiple (4)	Multiple (4)
	Num of Hosts	1	1	2	4
	Execution Time	2465.154 sec.	2509.799 sec.	1310.220 sec.	675.155 sec.
	Speed up	1	0.982	1.881	3.651

More than one multiple threaded applications can also be submitted to the system simultaneously. In the case of two, a similar speedup as that of one application is observed as is shown in Table.6.7.

In summary, for the above two applications, a sub-linear speedup of application runtime with respect to the number of recruited hosts is observed. For other multiple threaded applications, similar speedups could be achieved when the computations are relatively evenly distributed to the hosts. The speedup is in general inferior to linear, because there are creation overhead and communication overhead, as is shown in Table.6.5. On one hand, the creation overhead is incurred during initialization and it depends mostly on the number of concurrent threads to be launched on the network.

On the other hand, the communication overhead during runtime depends largely on the amount of inter-thread communication of the application. The more data are to be transferred between threads, the larger is the communication overhead. Also, the communication overhead increases when the number of recruited hosts increases. This is because the inter-host communication is much more expensive than the intra-host communication. For this reason, the proposed distributed computing platform is more suitable for coarse-grained computational applications, where the communications between concurrent threads are relatively small, compared with the computational tasks. When the application is very communicational intensive, the runtime speedup with respect to the number of hosts can be much inferior to the linear curve.

CHAPTER 7

CONCLUSIONS

7.1 Contributions

This thesis proposed a distributed code generation method with object level analysis of object-oriented applications. This method is a key component of the proposed Java Distributed code Generating and Computing platform. The major works in this research are summarized in the following three aspects.

Firstly, we developed a new method of extracting the object level communication affinity metrics for distributed object-oriented computing applications. These information are the basis for mapping the application to the networked machines. The communication affinity metrics obtained from this method are used in the object placement of the subsequent distributed code generation.

Unlike the existing software modeling/reverse engineering methods, which are only able to analyze static class-to-class information, the new method extract the communication affinities between runtime objects. Unlike existing dynamic modeling methods, which decompose the objects into separate scenario diagrams, the new

method explicitly outputs a two dimensional affinity metrics between objects, which can be directly used in the object placement.

This object level analysis method while motivated from our integrated system for object distribution, is nevertheless a general approach for extracting object-to-object communication affinity metrics and can also be used in other contexts.

Secondly, we developed an automatic distributed code generation method. Using the object level affinity metrics, the original application program is automatically transformed to run on the network machines. The generated distributed code incorporates the runtime object placement according to the affinities computed, as well as the network layer functions handling object manipulations and communications on network machines.

Unlike the existing schemes for the production of distributed code, the proposed method is completely automatic, and does not require special programming paradigm for the user's program. The input is a standard concurrent program for single machine. Without any intervention of the user, the method outputs the distributed code ready for submission to the distributed computing environment. This method on one hand, releases the developers from any special programming considerations on the underlying distributed environment, and on the other hand, solves the portability problem of distributed applications.

Thirdly, we developed an integrated JDGC platform that transforms and executes Java applications in a distributed environment. The entire system is developed using Java so

that it is portable to other operating systems. The key function in the integrated platform is the distributed code generation with object level analysis. As a fully functional system, JDGC receives user applications from a comprehensive interface, keeps tracks of the dynamic status of the system in a runtime application manager, and performs backing up and recovery.

The typical procedure of running an application on JDGC is as follows. Through the interface, a user provides the system with a standard concurrent application, as well as some controlling parameters. After this stage, the system automatically proceeds along without any further user's intervention. The system performs a preprocessing on the application, which includes parsing the application, analyzing it in two levels, and generating the object code. The generated code, while recorded in the runtime application manager, executes on the underlying networked virtual machines using the network layer functions.

7.2 Recommendations

Since the object level analysis is an independent method of program analysis, the method can also be incorporated into software modeling and reverse engineering tools. As a supplement to the models already in use, our method models the behaviors and interactions between runtime objects. This model may provide valuable information in several software engineering areas, such as software performance analysis, software reengineering, locating of bottleneck components, etc.

In the thesis, the proposed analysis method was applied to two applications: the matrix multiplication and curve fitting. Generally speaking, applications that are mostly

suitable for this analysis method are the non-interactive applications with relatively deterministic control flow logic between objects. For these kinds of applications, the inter-object communication costs are mostly determinable in the preprocessing stage, and thus the analysis would give more precise affinity information. For those application having frequent interactions with users, the runtime behavior of the application is largely dependent on the user inputs at runtime, which creates difficulty in the calculation of the communication affinities between objects. Similarly, the same difficulty arises if the application has many non-deterministic branches and may follows different control flows in different runs. In practice, many scientific computational applications are suitable for the proposed method. In these applications the pieces of computation tasks are usually well-structured and the interactions between these tasks are defined in the code before execution, instead of depending on runtime user input. Furthermore, the control flows for these computational applications are usually deterministic. As is the case for matrix multiplication and curve fitting, any normal execution of the computation would follow the same control flow except that certain exceptions or errors are encountered during the computation.

The implementation of the distributed code generation is currently only for Java language. However, this method can be extended to other object-oriented languages. For example, in the situation where portability is not an important consideration, the similar distributed code generation can be implemented for C++ applications, since the execution speed of C code is still faster than that of Java code, although JIT techniques have decreased the difference.

Furthermore, it is possible to build multi-language support for the distributed code generation method. As described in previous chapters, the method only uses the AST of the original program, which is an abstract representation of computer program and is independent of the language in use. Therefore, the object level analysis and AST transformation functions need not be changed for different languages. Only the standard parser and code generator are language dependent. However, these components have already existed in the compilers of individual language. Thus a system that supports multiple languages needs to have multiple parsers and code generators built in. But only one set of analyzers and transformers for general AST is necessary.

The multiple language support can be extended to build a distributed computing platform that automatically chooses the format of generated code depending on the executing environment of the recruited server host. For example, native object code runs faster than Java bytecode. But native code is not portable, that is, the native code generated for a certain operating system (or machine architecture) cannot run on a different operation system (or machine architecture). In order to enhance the execution speed and at the same time guarantee the code executable in the destination host, the system choose which code generator is used to generate object code, depending on the operation system (or machine architecture of the recruited host). The information can easily be obtained in the recruiting stage. If the host is able to execute a certain format of native code faster than the bytecode (e.g. machine code for PC architecture), then the system will port the source code to this native format. In the worst case, if the host cannot provide information about its executing environment, the system will generate Java bytecode, which is slower but is guaranteed to be executable.

Currently, the proposed analysis and transformation method only works for source code. A possible research direction is to enhance this method to be able to work for bytecode. Thus the user needs not to provide the source code of an application. This is possible because the bytecode format is also object-oriented. The analyzer needs to analyze the bytecode to extract object level affinity metrics, and the transformer needs to convert the bytecode for single machine to a form that runs on networked machines.

The proposed platform aims to utilize multiple machines to collectively solve computational applications. The user provided applications are distributed to server host for execution. A related design is to directly run the applications installed in the server host, but the input and output of the remote application is redirected to the client host in a same way as the distributed computing platform. For example, a client may run a word processor on an application server in order to save the consumption of memory and CPU time in the local machine. In such a system, there is no need to distribute code to the server. However, the client has to create a local represent for the remote application similar to the Wrapper objects used in our system.

APPENDIX A

KEY SOURCE CODE

Appendix A includes the key source code in the JDGC system that performs the class-level and object-level analysis.

Code for Class-Level Analysis

The following methods are implemented in the `class ClassAnalyzer`. The public method `execute()` calculates the class-level affinity values for a program represented by a given AST. The resultant affinity metrics are saved in an instance of `class ClassAff`. Other key methods that are invoked in the implementation of `execute()` are also given below.

```
public void execute()
{
  // init aff
  cAff = new ClassAff(trees.length);
  for (int i = 0; i < trees.length; i++)
  {
    cAff.addCls(findClassDef(trees[i]));
  } // for i

  // calculate
```

```
    for (int i = 0; i < trees.length; i++)
    {
      AST[] classBody = findClassDef(trees[i]).defs;
      // analyze the methods of this class
      // first get the class defined objects, and add them
to the visibleVarDef vector
      Vector visibleVarDef = new Vector();
      for (int j = 0; j < classBody.length; j++)
      {
        if (isInstanceOf(classBody[j],
"net.sf.pizzacompiler.compiler.AST$VarDef"))
        {
          visibleVarDef.add((Object)classBody[j]);
        }
      }
      // second analyze methods one by one
      for (int j = 0; j < classBody.length; j++)
      {
        if (isInstanceOf(classBody[j],
"net.sf.pizzacompiler.compiler.AST$FunDef")) // FunDef
          if (true)
          {
            FunDef method = (FunDef)classBody[j];
            // append the visibleVarDef to add the objects
passed to this method as parameters
            int whereToTruncate = visibleVarDef.size();
            VarDef[] parameters = method.params;
            for (int k = 0; k < parameters.length; k++)
            {
              visibleVarDef.add((Object)parameters[k]);
            }
            // get the method body
            AST[] stats = method.stats;
            // analyze the statement
            analyzeUnit(stats, findClassDef(trees[i]),
method, visibleVarDef, 1);
            // truncate the visibleVarDef to eliminate the
objects passed to this method as parameters
            // so that it only contains the class defined
objects
            for (int k = whereToTruncate; k <
visibleVarDef.size(); k++)
            {
              visibleVarDef.removeElementAt(k);
            }
          } // if true
      } // for j
    } // for i
  } // execute()
```

```
  private void analyzeUnit(AST[] stats, ClassDef cls,
FunDef method, Vector visibleVarDef, float w)
  { // analyze set of statements
    int whereToTruncate = visibleVarDef.size();
    for (int i = 0; i < stats.length; i++)
    {
      if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$VarDef")) // the child
is a VarDef
      {
        //append the vectorObjects to add the objects
local to this current analyze unit
        visibleVarDef.add(((Object)stats[i]));

        // THIS IS WHEN THE AFFINITY ARISES (THRU 'NEWING'
A USER-DEFINED OBJECT)
        if ( (((VarDef)stats[i]).init != null) &&
isInstanceOf(((VarDef)stats[i]).init,
"net.sf.pizzacompiler.compiler.AST$NewClass") )
          if ( isUserClass((VarDef)(stats[i]), trees) )
          {
            cAff.addAff(cls, method, (VarDef)(stats[i]),
NEW_INS_SIZE);
          }
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Block"))
      {
        analyzeUnit(((Block)stats[i]).stats, cls, method,
visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Synchronized"))
      {
        analyzeUnit(((Synchronized)stats[i]).body, cls,
method, visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$DoLoop"))
      {
        analyzeUnit(((DoLoop)stats[i]).body, cls, method,
visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$WhileLoop"))
      {
        analyzeUnit(((WhileLoop)stats[i]).body, cls,
method, visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$ForLoop"))
      {
        analyzeUnit(((ForLoop)stats[i]).body, cls, method,
visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Conditional"))
```

```
      {
        analyzeUnit(((Conditional)stats[i]).thenpart, cls,
method, visibleVarDef, (float)(w * 0.5));
        analyzeUnit(((Conditional)stats[i]).elsepart, cls,
method, visibleVarDef, (float)(w * 0.5));
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Switch"))
      {
        analyzeUnit(((Switch)stats[i]).cases, cls, method,
visibleVarDef, w / ((Switch)stats[i]).cases.length);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Case"))
      {
        analyzeUnit(((Case)stats[i]).stats, cls, method,
visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Try"))
      {
        analyzeUnit(((Try)stats[i]).body, cls, method,
visibleVarDef, w);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Exec"))
      {
        analyzeExpr(((Exec)stats[i]).expr, cls, method,
visibleVarDef, w);
      }

    } // for
    // truncate the visibleVarDef vector to eliminate the
objects local to this current statement
    for (int i = whereToTruncate; i < visibleVarDef.size();
i++)
    {
      visibleVarDef.removeElementAt(i);
    }
  } // analyzeUnit()

  private void analyzeUnit(AST stat, ClassDef cls, FunDef
method, Vector visibleVarDef, float w)
  {
    AST[] statA = new AST[1];
    statA[0] = stat;
    analyzeUnit(statA, cls, method, visibleVarDef, w);
  }

  private void analyzeExpr(AST expr, ClassDef cls, FunDef
method, Vector visibleVarDef, float w)
  {
    if (isInstanceOf(expr,
"net.sf.pizzacompiler.compiler.AST$Assign"))
    {
```

```
      AST callee = ((Assign)expr).lhs;
      AST value = ((Assign)expr).rhs;

      // THIS IS WHEN THE AFFINITY ARISES (THRU 'NEWING' A
VISIBLE USER-DEFINED OBJECT)
      if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Ident") )
      { // when the left-hand-side is an Identifier
        VarDef calleeDef = findVarDef((Ident)callee,
visibleVarDef);
        if ( (calleeDef != null) && (isInstanceOf(value,
"net.sf.pizzacompiler.compiler.AST$NewClass")) )
          if ( isUserClass(calleeDef, trees) )
          {
            cAff.addAff(cls, method, calleeDef,
NEW_INS_SIZE);
          }
      } // if Ident
      if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Index") )
      { // when the left-hand-side is an array indexing
        callee = ((Index)callee).indexed;
        if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Ident") )
        {
          VarDef calleeDef = findVarDef((Ident)callee,
visibleVarDef);
          if ( (calleeDef != null) && (isInstanceOf(value,
"net.sf.pizzacompiler.compiler.AST$NewClass")) )
            if ( isUserClassArray(calleeDef, trees) )
            {
              cAff.addAff(cls, method, calleeDef,
NEW_INS_SIZE);
            }
        }
      } // if Index

      analyzeExpr(((Assign)expr).rhs, cls, method,
visibleVarDef, w);
    } else if (isInstanceOf(expr,
"net.sf.pizzacompiler.compiler.AST$Apply"))
    {
      analyzeExpr(((Apply)expr).fn, cls, method,
visibleVarDef, w);
    } else if (isInstanceOf(expr,
"net.sf.pizzacompiler.compiler.AST$Select"))
    {
      AST callee = ((Select)expr).selected;
      Name selector = ((Select)expr).selector;
```

```
      // THIS IS WHEN THE AFFINITY ARISES (THRU A 'SELECT'
OPERATION, NO MATTER WHETHER IT IS S FIELD REF OR METHOD
INVOKE)
      if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Ident") )
      {
        VarDef calleeDef = findVarDef((Ident)callee,
visibleVarDef);
        if (calleeDef != null)
        {
          int affSize = calcSize(callee, selector,
calleeDef);
          if (affSize != java.lang.Integer.MIN_VALUE)
            if ( isUserClass(calleeDef, trees) )
              cAff.addAff(cls, method, calleeDef, w *
affSize);
        }
      } // if Ident
      if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Index") )
      {
        callee = ((Index)callee).indexed;
        if ( isInstanceOf(callee,
"net.sf.pizzacompiler.compiler.AST$Ident") )
        {
          VarDef calleeDef = findVarDef((Ident)callee,
visibleVarDef);
          if (calleeDef != null)
          {
            int affSize = calcSize(callee, selector,
calleeDef);
            if (affSize != java.lang.Integer.MIN_VALUE)
              if ( isUserClassArray(calleeDef, trees) )
                cAff.addAff(cls, method, calleeDef, w *
affSize);
          }
        }
      } // if Index
    }
  } // analyzeExpr()
```

Code for Object-Level Analysis

The following methods are implemented in the `class InstanceAnalyzer`. The public method `execute()` calculates the object-level affinity values from the class-level affinity metrics. It takes the class-level affinity metrics as input. The resultant object-level affinity metrics are saved in an instance of `class InstanceAff`. Other key methods that are invoked in the implementation of `execute()` are also given below.

```
  public void execute(ClassAff _cAff)
  {
    cAff = _cAff;
    // find the main() method
    ClassDef mainClassDef = findMainClassDef(trees);
    FunDef mainFunDef = findFunDef("main", mainClassDef);
    if (mainFunDef == null)
    {
      System.out.println("No main() found.");
      return;
    }
    // count new instance
    // init iAff, including the instanceVector and the
instanceAff
    numInstance = 0;
    iAff = new InstanceAff();
    iAff.v = new Vector();
    numInstance++;
    (iAff.v).add(mainClassDef);
    for (int i = 0; i < trees.length; i++)
    { // for every class
      AST[] classBody = findClassDef(trees[i]).defs;
      for (int j = 0; j < classBody.length; j++)
      {
        if (isInstanceOf(classBody[j],
"net.sf.pizzacompiler.compiler.AST$FunDef")) // method
definition
        { // for every method
          initUnit(((FunDef)classBody[j]).stats);
```

```
        }
      } // for j
    } // for i
    iAff.aff = new int[numInstance][numInstance];

    // calculate
    if (numInstance > 1)
    {
      // init visibleVar
      Vector visibleVar = initVisibleVar(mainClassDef,
mainFunDef, new Vector());
      // analyze
      analyzeUnit(0, mainFunDef.stats, mainClassDef,
mainFunDef, visibleVar);
    }
    else
    {
      System.out.println("No new statement found");
      return;
    }
  } // execute()

  // analyze units
  private void analyzeUnit(int PIndex, AST[] stats,
ClassDef cls, FunDef method, Vector visibleVar)
  {
    int whereToTruncate = visibleVar.size();
    for (int i = 0; i < stats.length; i++)
    {
      if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$VarDef"))
      {
        visibleVar.add(new VisibleVarNode((VarDef)stats[i],
null));
        if (((VarDef)stats[i]).init != null)
        {
          if (isInstanceOf(((VarDef)stats[i]).init,
"net.sf.pizzacompiler.compiler.AST$NewClass"))
          {
            VarDef NDef = (VarDef)stats[i];
            AST NNew = ((VarDef)stats[i]).init;
            addNewToVisibleVar(NDef, NNew, visibleVar);
            if ( isUserClass((NewClass)NNew, trees) )
            {
              int NIndex = iAff.getIndex(NNew);
              addP2N(PIndex, NIndex, cls, method, NDef);
            }
          } // if init == NewClass
          if (isInstanceOf(((VarDef)stats[i]).init,
"net.sf.pizzacompiler.compiler.AST$Ident"))
          { // Object a = b;
```

```
            VarDef NDef = (VarDef)stats[i];
            AST NNew = getVarNew(visibleVar,
((Ident)((VarDef)stats[i]).init).idname);
            addNewToVisibleVar(NDef, NNew, visibleVar);
            if ( isUserClass((NewClass)NNew, trees) )
            {
              int NIndex = iAff.getIndex(NNew);
              addP2N(PIndex, NIndex, cls, method, NDef);
            }
          } // if init == Ident
          analyzeUnit(PIndex, ((VarDef)stats[i]).init, cls,
method, visibleVar);
        } // if init != null
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Block"))
      {
        analyzeUnit(PIndex, ((Block)stats[i]).stats, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Synchronized"))
      {
        analyzeUnit(PIndex, ((Synchronized)stats[i]).body,
cls, method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$DoLoop"))
      {
        analyzeUnit(PIndex, ((DoLoop)stats[i]).body, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$WhileLoop"))
      {
        analyzeUnit(PIndex, ((WhileLoop)stats[i]).body,
cls, method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$ForLoop"))
      {
        analyzeUnit(PIndex, ((ForLoop)stats[i]).body, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Conditional"))
      {
        analyzeUnit(PIndex,
((Conditional)stats[i]).thenpart, cls, method, visibleVar);
        analyzeUnit(PIndex,
((Conditional)stats[i]).elsepart, cls, method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Switch"))
      {
        analyzeUnit(PIndex, ((Switch)stats[i]).cases, cls,
method, visibleVar);
```

```
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Case"))
      {
        analyzeUnit(PIndex, ((Case)stats[i]).stats, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Try"))
      {
        analyzeUnit(PIndex, ((Try)stats[i]).body, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Exec"))
      {
        analyzeUnit(PIndex, ((Exec)stats[i]).expr, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Assign"))
      {
        AST lhs = ((Assign)stats[i]).lhs;
        AST rhs = ((Assign)stats[i]).rhs;
        if (isInstanceOf(lhs,
"net.sf.pizzacompiler.compiler.AST$Ident"))
        {
          if (isInstanceOf(rhs,
"net.sf.pizzacompiler.compiler.AST$NewClass"))
          {
            VarDef NDef = getVarDef(visibleVar,
((Ident)lhs).idname);
            AST NNew = rhs;
            addNewToVisibleVar(NDef, NNew, visibleVar);
            if ( isUserClass((NewClass)NNew, trees) )
            {
              int NIndex = iAff.getIndex(NNew);
              addP2N(PIndex, NIndex, cls, method, NDef);
            }
          } // if rhs == NewClass
          if (isInstanceOf(rhs,
"net.sf.pizzacompiler.compiler.AST$Ident"))
          { // a = b;
            VarDef NDef = getVarDef(visibleVar,
((Ident)lhs).idname);
            AST NNew = getVarNew(visibleVar,
((Ident)rhs).idname);
            addNewToVisibleVar(NDef, NNew, visibleVar);
            if ( isUserClass((NewClass)NNew, trees) )
            {
              int NIndex = iAff.getIndex(NNew);
              addP2N(PIndex, NIndex, cls, method, NDef);
            }
          } // if rhs == Ident
        } // if lhs == Ident
```

```
        if (isInstanceOf(lhs,
"net.sf.pizzacompiler.compiler.AST$Index"))
        {
          lhs = ((Index)lhs).indexed;
          if ( isInstanceOf(lhs,
"net.sf.pizzacompiler.compiler.AST$Ident") )
          {
            if (isInstanceOf(rhs,
"net.sf.pizzacompiler.compiler.AST$NewClass"))
            {
              VarDef NDef = getVarDef(visibleVar,
((Ident)lhs).idname);
              AST NNew = rhs;
              addNewToVisibleVar(NDef, NNew, visibleVar);
              if ( isUserClass((NewClass)NNew, trees) )
              {
                int NIndex = iAff.getIndex(NNew);
                addP2N(PIndex, NIndex, cls, method, NDef);
              }
            } // if rhs == NewClass
            if (isInstanceOf(rhs,
"net.sf.pizzacompiler.compiler.AST$Ident"))
            { // a = b;
              VarDef NDef = getVarDef(visibleVar,
((Ident)lhs).idname);
              AST NNew = getVarNew(visibleVar,
((Ident)rhs).idname);
              addNewToVisibleVar(NDef, NNew, visibleVar);
              if ( isUserClass((NewClass)NNew, trees) )
              {
                int NIndex = iAff.getIndex(NNew);
                addP2N(PIndex, NIndex, cls, method, NDef);
              }
            } // if rhs == Ident
          } // if Ident
        } // if lhs == Index
        analyzeUnit(PIndex, rhs, cls, method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Apply"))
      {
        AST fn = ((Apply)stats[i]).fn;
        AST[] args = ((Apply)stats[i]).args;
        if ( isInstanceOf(fn,
"net.sf.pizzacompiler.compiler.AST$Ident") )
        { // intra-class invoke, fn is a function of
current class
            // add O2N, that is the current class (O) to
the parameter callees (Ns)
            for (int j = 0; j < args.length; j++)
            {
```

```
              if (!isInstanceOf(args[j],
"net.sf.pizzacompiler.compiler.AST$Ident"))
                continue;
              AST NNew = getVarNew(visibleVar,
((Ident)(args[i])).idname);
              if ( (NNew != null) )
              {
                int NIndex = iAff.getIndex(NNew);
                FunDef Omethod =
findFunDef(((Ident)fn).idname, cls);
                VarDef calleeDef = Omethod.params[j];
                addO2N(PIndex, NIndex, cls, Omethod,
calleeDef);
              } // if NNew
            } // for i
            // analyze the invoked method
            FunDef Omethod = findFunDef(((Ident)fn).idname,
cls);
            if (Omethod != null)
            {
              Vector OvisibleVar = initVisibleVar(cls,
Omethod, visibleVar);
              analyzeUnit(PIndex, Omethod.stats, cls,
Omethod, OvisibleVar);
            }
        } else if ( isInstanceOf(fn,
"net.sf.pizzacompiler.compiler.AST$Select") )
        {
          AST selected = ((Select)fn).selected;
          if ( isInstanceOf(selected,
"net.sf.pizzacompiler.compiler.AST$Ident") )
          {
            // check if the selected is a user-defined
class
            VarDef selectedVarDef = getVarDef(visibleVar,
((Ident)selected).idname);
            if ( !isUserClass(selectedVarDef, trees) )
              return;
            Name selector = ((Select)fn).selector;
            // add O2N, that is the selected class (O) to
the parameter callees (Ns)
            for (int j = 0; j < args.length; j++)
            {
              if (!isInstanceOf(args[j],
"net.sf.pizzacompiler.compiler.AST$Ident"))
                continue;
              VarDef ODef = getVarDef(visibleVar,
((Ident)selected).idname);
              AST ONew = getVarNew(visibleVar,
((Ident)selected).idname);
```

```
              AST NNew = getVarNew(visibleVar,
((Ident)(args[i])).idname);
              if ( (NNew != null) && (ODef != null) &&
(ONew != null) )
              {
                int OIndex = iAff.getIndex(ONew);
                int NIndex = iAff.getIndex(NNew);
                ClassDef Ocls = findClassDef(ODef.vartype,
trees);
                FunDef Omethod = findFunDef(selector,
Ocls);
                VarDef calleeDef = Omethod.params[j];
                addO2N(OIndex, NIndex, Ocls, Omethod,
calleeDef);
              } // if NNew
            } // for i
            // analyze the invoked method
            VarDef ODef = getVarDef(visibleVar,
((Ident)selected).idname);
            AST ONew = getVarNew(visibleVar,
((Ident)selected).idname);
            if ( (ODef !=null) && (ONew != null) )
            {
              int OIndex = iAff.getIndex(ONew);
              ClassDef Ocls = findClassDef(ODef.vartype,
trees);
              FunDef Omethod = findFunDef(selector, Ocls);
              if (Omethod != null)
              {
                Vector OvisibleVar = initVisibleVar(Ocls,
Omethod, visibleVar);
                analyzeUnit(OIndex, Omethod.stats, Ocls,
Omethod, OvisibleVar);
              }
            }
          } // if Ident
          if ( isInstanceOf(selected,
"net.sf.pizzacompiler.compiler.AST$Index") )
          {
            // check if the selected is a user-defined
class array
            selected = ((Index)selected).indexed;
            if ( isInstanceOf(selected,
"net.sf.pizzacompiler.compiler.AST$Ident") )
            {
              VarDef selectedVarDef = getVarDef(visibleVar,
((Ident)selected).idname);
              if ( !isUserClassArray(selectedVarDef,
trees) )
                return;
              Name selector = ((Select)fn).selector;
```

```
              // add O2N, that is the selected class (O)
to the parameter callees (Ns)
              for (int j = 0; j < args.length; j++)
              {
                if (!isInstanceOf(args[j],
"net.sf.pizzacompiler.compiler.AST$Ident"))
                  continue;
                VarDef ODef = getVarDef(visibleVar,
((Ident)selected).idname);
                AST ONew = getVarNew(visibleVar,
((Ident)selected).idname);
                AST NNew = getVarNew(visibleVar,
((Ident)(args[j])).idname);
                if ( (NNew != null) && (ODef != null) &&
(ONew != null) )
                {
                  int OIndex = iAff.getIndex(ONew);
                  int NIndex = iAff.getIndex(NNew);
                  ClassDef Ocls =
findClassDef(((ArrayTypeTerm)(ODef.vartype)).elemtype,
trees);
                  FunDef Omethod = findFunDef(selector,
Ocls);
                  VarDef calleeDef = Omethod.params[j];
                  addO2N(OIndex, NIndex, Ocls, Omethod,
calleeDef);
                } // if NNew
              } // for i
              // analyze the invoked method
              VarDef ODef = getVarDef(visibleVar,
((Ident)selected).idname);
              AST ONew = getVarNew(visibleVar,
((Ident)selected).idname);
              if ( (ODef !=null) && (ONew != null) )
              {
                int OIndex = iAff.getIndex(ONew);
                ClassDef Ocls =
findClassDef(((ArrayTypeTerm)(ODef.vartype)).elemtype,
trees);
                FunDef Omethod = findFunDef(selector,
Ocls);
                if (Omethod != null)
                {
                  Vector OvisibleVar = initVisibleVar(Ocls,
Omethod, visibleVar);
                  analyzeUnit(OIndex, Omethod.stats, Ocls,
Omethod, OvisibleVar);
                }
              }
            } // if Ident
          } // if Index
```

```
        } // if Select
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Binop"))
      {
        analyzeUnit(PIndex, ((Binop)stats[i]).lhs, cls,
method, visibleVar);
        analyzeUnit(PIndex, ((Binop)stats[i]).rhs, cls,
method, visibleVar);
      } else if (isInstanceOf(stats[i],
"net.sf.pizzacompiler.compiler.AST$Unop"))
      {
        analyzeUnit(PIndex, ((Unop)stats[i]).operand, cls,
method, visibleVar);
      } else
      {
      }

    } // for i

    // truncate the visibleVarDef vector to eliminate the
objects local to this current statement
    for (int i = whereToTruncate; i < visibleVar.size();
i++)
    {
      visibleVar.removeElementAt(i);
    }
  } // analyzeUnit()

  private void analyzeUnit(int PIndex, AST stat, ClassDef
cls, FunDef method, Vector visibleVar)
  {
    AST[] astA = new AST[1];
    astA[0] = stat;
    analyzeUnit(PIndex, astA, cls, method, visibleVar);
  }
```

APPENDIX B

EXAMPLES OF CODE TRANSFORMATION

Appendix B includes illustrative examples of code transformation. The original code and the transformed code for the matrix multiplication and curve fitting applications are provided.

Matrix Multiplication

The following are the original code that runs on single machine. The method `multiply()` in the `class MultImplT` is the main computational method.

MultT.java:

```
package tests.mult;
public class MultT
{
  public static void main(String args[])
  {
    if (args.length != 2)
    {
      System.out.println("<npieces> <matrix size>");
      System.exit(1);
    }
    int npieces = Integer.parseInt(args[0]);
    int size = Integer.parseInt(args[1]);
    MatrixT a, b;
    MatrixT[] sub_a = new MatrixT[npieces];
    MatrixT[] sub_result = new MatrixT[npieces];
```

```
MultImplT[] mm = new MultImplT[npieces];

try
{
  long beginTime = System.currentTimeMillis();
  System.out.println("begin time = " + beginTime);
  System.out.println(" creating ... ");
  for (int i = 0; i < npieces; i++)
 {
      mm[i] = new MultImplT();
  }
  System.out.println(" creation finishs " );

  b = MatrixT.createIdentityMatrix(size);
  a = MatrixT.createRandomMatrix(size, size);
  int start_row = 0;
  int end_row = -1;
  int nrows = size / npieces;
  for (int i = 0; i < npieces; i++)
  {
    if (i >= size % npieces)
    {
   start_row = end_row + 1;
   end_row = start_row + nrows - 1;
 } else
 {
   start_row = end_row + 1;
   end_row = start_row + nrows;
 }
 sub_a[i] = a.getSubMatrixByRows(start_row, end_row);
} // for i

  // execute
  System.out.println(" executing ... ");
  for (int i = 0; i < npieces; i++)
  {
      mm[i].init(sub_a[i], b);
      mm[i].start();
  } // for i

  for (int i = 0; i < npieces; i++)
  {
    mm[i].join();
  } // for i
  System.out.println(" execution finishes. ");
  long endTime = System.currentTimeMillis();
  System.out.println("end time = " + endTime);
  System.exit(0);
} catch (Exception ex)
{
  System.out.println("test fails " + ex);
```

```
    }
  } // main()
}
```

MultImplT.java:

```
package tests.mult;

public class MultImplT  extends Thread
{
  MatrixT a, b;
  public MultImplT()
  {
  }
  public void init(MatrixT _a, MatrixT _b)
  {
    a = _a;
    b = _b;
  } // init()

  public void run()
  {
    MatrixT result = multiply();
  } // run()

  public MatrixT multiply()
  {
    int brows, rows, columns;
    MatrixT result = null;
    int value;
    if (a.getColumns()== b.getRows())
    {
          rows = a.getRows();
          columns = b.getColumns();
          brows = b.getRows();
          result =  new MatrixT();
          result.init(rows, columns);
          for (int i = 0; i < rows; i++)
            for (int j = 0; j < columns; j++)
            {
              value = 0;
              for (int k = 0; k < brows; k++)
              value += a.getValue(i,k) * b.getValue(k,j);
              result.setValue(i, j, value);
            }
    } // if
    return result;
  } // multiply()
}
```

MatrixT.java:

```
package tests.mult;
import java.util.Random;

public class MatrixT
{
  int rows = 10;
  int columns = 10;
  int[][] m;
  private static int randomRange = 10;
  public MatrixT() {
  }
  public void init(int rows, int columns)  {
    this.rows = rows;
    this.columns = columns;
    m = new int[rows][columns];
  }
  public int getValue(int x, int y)  {
    return m[x][y];
  }
  public int getRows()  {
    return rows;
  }
  public int getColumns() {
    return columns;
  }

  public void setValue(int x, int y, int value) {
    m[x][y] = value;
  }
  public static MatrixT createIdentityMatrix(int size) {
    MatrixT result = new MatrixT();
    result.init(size, size);
    for (int i = 0; i < size; i++)
     for (int j = 0; j < size; j++)
       if (i == j)
         result.setValue(i, j, 1);
        else
         result.setValue(i, j, 0);
    return result;
  }
  public static void setRandomRange( int range )
  {
     randomRange = range;
  }
  public static MatrixT createRandomMatrix(int rows, int
columns) {
    MatrixT result = new MatrixT();
    result.init(rows, columns);
    Random random = new Random();
    for (int i = 0; i < rows; i++)
     for (int j = 0; j < columns; j++)
```

```
        result.setValue(i, j, random.nextInt(randomRange));
    return result;
  }
  public MatrixT getSubMatrixByRows(int startAtRow, int
endAtRow) {
    if (endAtRow < startAtRow)
     return null;
    int nrows = endAtRow - startAtRow + 1;
    MatrixT result = new MatrixT();
    result.init(nrows, columns);
    for (int i = 0; i < nrows; i++)
     for (int j = 0; j < columns; j++)
       result.setValue(i, j, m[i+startAtRow][j]);
    return result;
  }
  public MatrixT getSubMatrixByColumns(int startAtCol, int
endAtCol) {
    if (endAtCol < startAtCol)
      return null;
    int ncols = endAtCol - startAtCol + 1;
    MatrixT result = new MatrixT();
    result.init(rows, ncols);
    for (int i = 0; i < rows; i++)
     for (int j = 0; j < ncols; j++)
       result.setValue(i, j, m[i][j+startAtCol]);
    return result;
  }
}
```

The following are the transformed code that runs on 4 machines in the distributed environment.

MultT.java:

```
package tests.mult;
import sogcp.recruit.*;
import sogcp.control.*;
import java.rmi.*;
import java.util.*;
import java.net.InetAddress;
import java.io.*;
import jdgc.lib.*;
import jdgc.proc.InstanceAffArray;
import jdgc.lib.*;
import java.io.Serializable;

public class MultT implements Serializable {
  static int __nIns = 0;
```

```
    static int __numHost;

    public static void main(String[] args) {
        __numHost = 4;
        {
            try {
                String iAffFileName = ".\\tmp\\iAff";
                String jarFileName = ".\\tmp\\dist.jar";
                Wrapper.submit(__numHost, iAffFileName,
jarFileName);
                System.out.println("application submitted");
            }  catch (Exception e) {
                System.out.println("error submitting
application");
                e.printStackTrace();
                System.exit(- 1);
            }
        }
        if (args.length != 2) {
            System.out.println("<npieces> <matrix size>");
            System.exit(1);
        }
        int npieces = Integer.parseInt(args[0]);
        int size = Integer.parseInt(args[1]);
        Wrapper a;
        Wrapper b;
        Wrapper[] sub_a = new Wrapper[npieces];
        Wrapper[] sub_result = new Wrapper[npieces];
        Wrapper[] mm = new Wrapper[npieces];
        try {
            long beginTime = System.currentTimeMillis();
            System.out.println("begin time = " + beginTime);
            System.out.println(" creating ... ");
            for (int i = 0; i < npieces; i ++) {
                mm[i] = new Wrapper("tests.mult.MultImplT",
"ins" + __nIns ++, 2, ".\\tmp\\dist.jar", "localhost");
            }
            System.out.println(" creation finishs ");

            b =
(Wrapper)Wrapper.invokeStatic("tests.mult.MatrixT",
"createIdentityMatrix", new Class[]{int.class}, new
Object[]{new Integer(size)});
            a =
(Wrapper)Wrapper.invokeStatic("tests.mult.MatrixT",
"createRandomMatrix", new Class[]{int.class, int.class},
new Object[]{new Integer(size), new Integer(size)});

            int start_row = 0;
            int end_row = - 1;
            int nrows = size / npieces;
```

```
         for (int i = 0; i < npieces; i ++) {
            if (i >= size % npieces) {
               start_row = end_row + 1;
               end_row = start_row + nrows - 1;
            } else {
               start_row = end_row + 1;
               end_row = start_row + nrows;
            }
            sub_a[i] =
(Wrapper)a.invoke("getSubMatrixByRows", new Object[]{new
Integer(start_row), new Integer(end_row)});
         }
         System.out.println(" executing ... ");

         for (int i = 0; i < npieces; i ++) {
            mm[i].invoke("init", new Object[]{sub_a[i],b});
            mm[i].invoke("start", null);
         }
         for (int i = 0; i < npieces; i ++) {
            mm[i].invoke("join", null);
         }
         System.out.println(" execution finishes. ");
         long endTime = System.currentTimeMillis();
         System.out.println("end time = " + endTime);
         System.exit(0);
      }  catch (Exception ex) {
         System.out.println("test fails " + ex);
      }
   }

   public MultT() {
      super();
   }
}
```

MultImplT.java:

```
package tests.mult;
import jdgc.lib.*;
import java.io.Serializable;

public class MultImplT extends Thread implements
Serializable {
   static int __nIns = 0;
   Wrapper a;
   Wrapper b;
   public MultImplT() {
      super();
   }
   public void init(Wrapper _a, Wrapper _b) {
      a = _a;
```

```
      b = _b;
   }
   public void run() {
      Wrapper result = multiply();
   }

   public Wrapper multiply() {
      int brows;
      int rows;
      int columns;
      Wrapper result = null;
      int value;

      if (Wrapper.retInt(a.invoke("getColumns", null)) ==
Wrapper.retInt(b.invoke("getRows", null))) {
         rows = Wrapper.retInt(a.invoke("getRows", null));
         columns = Wrapper.retInt(b.invoke("getColumns",
null));
         brows = Wrapper.retInt(b.invoke("getRows", null));
         result = new Wrapper("tests.mult.MatrixT", "ins"
+ __nIns ++, 1, ".\\tmp\\dist.jar", "localhost");
         result.invoke("init", new Object[]{new
Integer(rows), new Integer(columns)});
         for (int i = 0; i < rows; i ++) for (int j = 0; j
< columns; j ++) {
            value = 0;
            for (int k = 0; k < brows; k ++) value +=
Wrapper.retInt(a.invoke("getValue", new Object[]{new
Integer(i), new Integer(k)})) *
Wrapper.retInt(b.invoke("getValue", new Object[]{new
Integer(k), new Integer(j)}));
            result.invoke("setValue", new Object[]{new
Integer(i), new Integer(j), new Integer(value)});
         }
      }
      return result;
   }
}
```

Matrix.java:

```
package tests.mult;
import java.util.Random;
import jdgc.lib.*;
import java.io.Serializable;

public class MatrixT implements Serializable {
   static int __nIns = 0;
   int rows = 10;
   int columns = 10;
   int[][] m;
```

```
   private static int randomRange = 10;

   public MatrixT() {
      super();
   }
   public void init(int rows, int columns) {
      this.rows = rows;
      this.columns = columns;
      m = new int[rows][columns];
   }
   public int getValue(int x, int y) {
      return m[x][y];
   }
   public int getRows() {
      return rows;
   }
   public int getColumns() {
      return columns;
   }
   public void setValue(int x, int y, int value) {
      m[x][y] = value;
   }
   public static Wrapper createIdentityMatrix(int size) {
      Wrapper result = new Wrapper("tests.mult.MatrixT",
"ins" + __nIns ++, 3, ".\\tmp\\dist.jar", "localhost");
      result.invoke("init", new Object[]{new Integer(size),
new Integer(size)});
      for (int i = 0; i < size; i ++) for (int j = 0; j <
size; j ++) if (i == j) result.invoke("setValue", new
Object[]{new Integer(i), new Integer(j), new Integer(1)});
else result.invoke("setValue", new Object[]{new Integer(i),
new Integer(j), new Integer(0)});
      return result;
   }
   public static void setRandomRange(int range) {
      randomRange = range;
   }
   public static Wrapper createRandomMatrix(int rows, int
columns) {
      Wrapper result = new Wrapper("tests.mult.MatrixT",
"ins" + __nIns ++, 4, ".\\tmp\\dist.jar", "localhost");
      result.invoke("init", new Object[]{new Integer(rows),
new Integer(columns)});
      Random random = new Random();
      for (int i = 0; i < rows; i ++) for (int j = 0; j <
columns; j ++) result.invoke("setValue", new Object[]{new
Integer(i), new Integer(j), new
Integer(random.nextInt(randomRange))});
      return result;
   }
```

```
   public Wrapper getSubMatrixByRows(int startAtRow, int
endAtRow) {
      if (endAtRow < startAtRow) return null;
      int nrows = endAtRow - startAtRow + 1;
      Wrapper result = new Wrapper("tests.mult.MatrixT",
"ins" + __nIns ++, 5, ".\\tmp\\dist.jar", "localhost");
      result.invoke("init", new Object[]{new
Integer(nrows), new Integer(columns)});
      for (int i = 0; i < nrows; i ++) for (int j = 0; j <
columns; j ++) result.invoke("setValue", new Object[]{new
Integer(i), new Integer(j), new Integer(m[i +
startAtRow][j])});
      return result;
   }
   public Wrapper getSubMatrixByColumns(int startAtCol,
int endAtCol) {
      if (endAtCol < startAtCol) return null;
      int ncols = endAtCol - startAtCol + 1;
      Wrapper result = new Wrapper("tests.mult.MatrixT",
"ins" + __nIns ++, 6, ".\\tmp\\dist.jar", "localhost");
      result.invoke("init", new Object[]{new Integer(rows),
new Integer(ncols)});
      for (int i = 0; i < rows; i ++) for (int j = 0; j <
ncols; j ++) result.invoke("setValue", new Object[]{new
Integer(i), new Integer(j), new Integer(m[i][j +
startAtCol])});
      return result;
   }
}
```

Curve Fitting

The following are the original code that runs on single machine. The main computation is performed in the Fit.java.

Curve.java:

```
package curve;

import java.lang.reflect.*;

public class Curve
{
  public Curve()
```

```
  {
  }
  public static void main(String[] args)
  {
    if (args.length != 3)
    {
      System.out.println("<npieces> <piece size> <order>");
      System.exit(1);
    }
    int nPieces = Integer.parseInt(args[0]);
    int pieceSize = Integer.parseInt(args[1]);
    int order = Integer.parseInt(args[2]);
    Fit[] fits = new Fit[nPieces];

    try
    {
      long beginTime = System.currentTimeMillis();
      System.out.println(" creating ... ");
      for (int i = 0; i < nPieces; i++)
      {
          fits[i] = new Fit();
      }
      System.out.println(" creation finishs " );
      double[] datax = new double[pieceSize];
      double[] datay = new double[pieceSize];

      // execute
      System.out.println(" executing ... ");
      double xCount = 0.0;
      for (int i = 0; i < nPieces; i++)
      {
        for (int j = 0; j < pieceSize; j++)
        {
          datax[j] = xCount;
          xCount = xCount + 1.0;
        }
        for (int j = 0; j < pieceSize; j++)
        {
          datay[j] = Math.random() * (nPieces * pieceSize);
        }
          fits[i].init( new Integer(pieceSize), new
Integer(order), datax, datay );
        long size = (datax.length + datay.length)*4;
      } // for i

      for (int i = 0; i < nPieces; i++)
      {
          fits[i].start();
      } // for i

      for (int i = 0; i < nPieces; i++)
```

```
      {
         fits[i].join();
      } // for i

      System.out.println("collecting ... ");
      double[][] coefficients = new
double[nPieces][order+1];
      double[] lsq = new double[nPieces];
      for (int i = 0; i < nPieces; i++)
      {
        long begin = System.currentTimeMillis();
        coefficients[i] = fits[i].getCoefficients();
        long end = System.currentTimeMillis();
        System.out.println("communication time for " +
coefficients.length*4 + " bytes:" + (end-begin));

        for (int j = 0; j < order + 1; j++)
        {
          System.out.print(coefficients[i][j] + " ");
        }
        System.out.println();
      } // for i
      System.out.println(" execution finishes. ");
      long endTime = System.currentTimeMillis();
      System.out.println("time = " + (endTime -
beginTime));
      System.exit(0);
    } catch (Exception e)
    {
    }
  } // main()
}
```

Fit.java:

```
package curve;

import java.lang.reflect.*;

public class Fit extends Thread
{
  double[] datax, datay;
  int pieceSize, order;
  double[] c;
  int lsq;
  double basicPrecision = 100000.0;
  double tunedPrecision = 1000000.0;

  public Fit()
  {
  }
```

```
 public void init(Integer _pieceSize, Integer _order,
double[] _datax, double[] _datay)
  {
    pieceSize = _pieceSize.intValue();
    order = _order.intValue();
    datax = new double[pieceSize];
    for (int i = 0; i < pieceSize; i++)
    {
      datax[i] = _datax[i];
    }
    datay = new double[pieceSize];
    for (int i = 0; i < pieceSize; i++)
    {
      datay[i] = _datay[i];
    }
  } // init()

  public void run()
  {
    // least square
    int rows = pieceSize;
    int cols = 1;
    int numCo = order + 1;
    c = new double[numCo];

    double basis[] = new double[numCo * rows];
    double alpha[] = new double[numCo * numCo];
    double alpha2[] = new double[numCo * numCo];
    double beta[] = new double[numCo];
// calc basis
for (int i = 0; i < numCo; i ++)
     {
     for (int j = 0; j < rows; j ++)
          {
          int k = i + j * numCo;
          if (i == 0) basis[k] = 1;
          else basis[k] = datax[j] * basis[k - 1];
          }
     }
// calc alpha2
for (int i = 0; i < numCo; i ++)
     {
     for (int j = 0; j <= i; j ++)
          {
          double sum = 0;
          for (int k = 0; k < rows; k ++)
               {
               int k2 = i + k * numCo;
               int k3 = j + k * numCo;
```

```
                sum += basis[k2] * basis[k3];
                }

            int k2 = i + j * numCo;
            alpha2[k2] = sum;

            if (i != j)
                {
                k2 = j + i * numCo;
                alpha2[k2] = sum;
                }
            }
        }

for (int i = 0; i < cols; i ++)
        {
        // calc beta
        for (int j = 0; j < numCo; j ++)
            {
            double sum = 0;
            for (int k = 0; k < rows; k ++)
                {
                int k3 = j + k * numCo;
                sum += datay[k] * basis[k3];
                }
            beta[j] = sum;
            }
        // get alpha
        for (int j = 0; j < numCo * numCo; j ++) alpha[j] =
alpha2[j];
        // solve for params
        if (!solve(alpha, beta, numCo))
            {
            }
        for (int j = 0; j < numCo; j ++)
            {
            c[j] = Math.floor(beta[j] * basicPrecision) /
basicPrecision;
            }
        }
          int sum = 0;
          for (int j = 0; j < pieceSize; j++)
          {
            sum += Math.pow(datay[j] - func(c, datax[j]),
(double)2);
          }
          lsq = sum;

          for (int i = 0; i < Math.pow(10, numCo) - 1; i++)
          {
            String str = Integer.toString(i);
```

```
          int l = str.length();
          for (int j = 0; j < numCo - l; j++)
            str = "0" + str;

          double[] cTry = new double[numCo];
          for (int j = 0; j < numCo; j++)
          {
            cTry[j] = c[j] + (1/basicPrecision) *
Integer.valueOf(str.substring(j, j+1)).intValue();
            cTry[j] = Math.floor(cTry[j] * tunedPrecision)
/ tunedPrecision;
          }
          sum = 0;
          for (int j = 0; j < pieceSize; j++)
          {
            sum += Math.pow(datay[j] - func(cTry,
datax[j]), 2);
          }
          if (sum < lsq)
          {
            lsq = sum;
            c = cTry;
          }
        } // for i
  } // run()

  double func(double[] c, double x)
  {
    double y = 0;
    for (int i = 0; i <= order; i++)
    {
      y += c[i] * Math.pow(x, i);
    }
    return y;
  } // func()

 public double[] getCoefficients()
  {
    return c;
  } // getCoefficients()

  public double getlsq()
  {
    return lsq;
  } // getlsq()

  boolean solve(double a[], double b[], int n)
  {
    for (int i = 0; i < n; i ++)
    {
      // find pivot
```

```
      double mag = 0;
      int pivot = -1;
      for (int j = i; j < n; j ++)
      {
        double mag2 = Math.abs(a[i + j * n]);
        if (mag2 > mag)
        {
          mag = mag2;
          pivot = j;
        }
      }
      // no pivot: error
      if (pivot == -1 || mag == 0)
      {
        return false;
      }
      // move pivot row into position
      if (pivot != i)
      {
        double temp;
        for (int j = i; j < n; j ++)
        {
          temp = a[j + i * n];
          a[j + i * n] = a[j + pivot * n];
          a[j + pivot * n] = temp;
        }
        temp = b[i];
        b[i] = b[pivot];
        b[pivot] = temp;
      }
      // normalize pivot row
      mag = a[i + i * n];
      for (int j = i; j < n; j ++) a[j + i * n] /= mag;
      b[i] /= mag;
      // eliminate pivot row component from other rows
      for (int i2 = 0; i2 < n; i2 ++)
      {
        if (i2 == i) continue;
        double mag2 = a[i + i2 * n];
        for (int j = i; j < n; j ++)
          a[j + i2 * n] -= mag2 * a[j + i * n];
        b[i2] -= mag2 * b[i];
      }
    }

    // result should be identity matrix in a and solution
vector in b
    return true;
  } // solve()
}
```

The following are the transformed code that runs on 4 machines.

Curve.java:

```
package curve;

import java.lang.reflect.*;
import sogcp.recruit.*;
import sogcp.control.*;
import java.rmi.*;
import java.util.*;
import java.net.InetAddress;
import java.io.*;
import jdgc.lib.*;
import jdgc.proc.InstanceAffArray;
import jdgc.lib.*;

import java.io.Serializable;

public class Curve implements Serializable {
   static int __nIns = 0;
   static int __numHost;
   public Curve() {
      super();
   }
   public static void main(String[] args) {
      __numHost = 4;
      {
         try {
            String iAffFileName = ".\\tmp\\iAff";
            String jarFileName = ".\\tmp\\dist.jar";
            Wrapper.submit(__numHost, iAffFileName,
jarFileName);
            System.out.println("application submitted");
         }  catch (Exception e) {
            System.out.println("error submitting
application");
            e.printStackTrace();
            System.exit(- 1);
         }
      }
      if (args.length != 3) {
         System.out.println("<npieces> <piece size>
<order>");
         System.exit(1);
      }
      int nPieces = Integer.parseInt(args[0]);
      int pieceSize = Integer.parseInt(args[1]);
      int order = Integer.parseInt(args[2]);
      Wrapper[] fits = new Wrapper[nPieces];
```

```
      try {
         long beginTime = System.currentTimeMillis();
         System.out.println(" creating ... ");
         for (int i = 0; i < nPieces; i ++) {
            fits[i] = new Wrapper("curve.Fit", "ins" +
__nIns ++, 1, ".\\tmp\\dist.jar", "localhost");
         }
         System.out.println(" creation finishs ");

         double[] datax = new double[pieceSize];
         double[] datay = new double[pieceSize];
         System.out.println(" executing ... ");
         double xCount = 0.0;

         for (int i = 0; i < nPieces; i ++) {
            for (int j = 0; j < pieceSize; j ++) {
               datax[j] = xCount;
               xCount = xCount + 1.0;
            }
            for (int j = 0; j < pieceSize; j ++) {
               datay[j] = Math.random() * (nPieces *
pieceSize);
            }
            fits[i].invoke("init", new Object[]{new
Integer(pieceSize), new Integer(order), datax, datay});
            long size = (datax.length + datay.length) * 4;
         }
         for (int i = 0; i < nPieces; i ++) {
            fits[i].invoke("start", null);
         }
         for (int i = 0; i < nPieces; i ++) {
            fits[i].invoke("join", null);
         }
         System.out.println("collecting ... ");
         double[][] coefficients = new
double[nPieces][order + 1];
         double[] lsq = new double[nPieces];
         for (int i = 0; i < nPieces; i ++) {
            long begin = System.currentTimeMillis();
            coefficients[i] =
(double[])fits[i].invoke("getCoefficients", null);
            long end = System.currentTimeMillis();
            System.out.println("communication time for " +
coefficients.length * 4 + " bytes:" + (end - begin));
            for (int j = 0; j < order + 1; j ++) {
               System.out.print(coefficients[i][j] + " ");
            }
            System.out.println();
         }
         System.out.println(" execution finishes. ");
         long endTime = System.currentTimeMillis();
```

```
         System.out.println("time = " + (endTime -
beginTime));
         System.exit(0);
      }  catch (Exception e) {
      }
   }
}
```

Fit.java:

```
package curve;

import java.lang.reflect.*;
import jdgc.lib.*;

import java.io.Serializable;

public class Fit extends Thread implements Serializable {
   static int __nIns = 0;
   double[] datax;
   double[] datay;
   int pieceSize;
   int order;
   double[] c;
   int lsq;
   double basicPrecision = 100000.0;
   double tunedPrecision = 1000000.0;
   public Fit() {
      super();
   }
   public void init(Integer _pieceSize, Integer _order,
double[] _datax, double[] _datay) {
      pieceSize = _pieceSize.intValue();
      order = _order.intValue();
      datax = new double[pieceSize];
      for (int i = 0; i < pieceSize; i ++) {
         datax[i] = _datax[i];
      }
      datay = new double[pieceSize];
      for (int i = 0; i < pieceSize; i ++) {
         datay[i] = _datay[i];
      }
   }

   public void run() {
      int rows = pieceSize;
      int cols = 1;
      int numCo = order + 1;
      c = new double[numCo];
      double[] basis = new double[numCo * rows];
      double[] alpha = new double[numCo * numCo];
```

```
      double[] alpha2 = new double[numCo * numCo];
      double[] beta = new double[numCo];
      for (int i = 0; i < numCo; i ++) {
         for (int j = 0; j < rows; j ++) {
            int k = i + j * numCo;
            if (i == 0) basis[k] = 1; else basis[k] =
datax[j] * basis[k - 1];
         }
      }
      for (int i = 0; i < numCo; i ++) {
         for (int j = 0; j <= i; j ++) {
            double sum = 0;
            for (int k = 0; k < rows; k ++) {
               int k2 = i + k * numCo;
               int k3 = j + k * numCo;
               sum += basis[k2] * basis[k3];
            }
            int k2 = i + j * numCo;
            alpha2[k2] = sum;
            if (i != j) {
               k2 = j + i * numCo;
               alpha2[k2] = sum;
            }
         }
      }
      for (int i = 0; i < cols; i ++) {
         for (int j = 0; j < numCo; j ++) {
            double sum = 0;
            for (int k = 0; k < rows; k ++) {
               int k3 = j + k * numCo;
               sum += datay[k] * basis[k3];
            }
            beta[j] = sum;
         }
         for (int j = 0; j < numCo * numCo; j ++) alpha[j]
= alpha2[j];
         if (! solve(alpha, beta, numCo)) {
         }
         for (int j = 0; j < numCo; j ++) {
            c[j] = Math.floor(beta[j] * basicPrecision) /
basicPrecision;
         }
      }
      int sum = 0;
      for (int j = 0; j < pieceSize; j ++) {
         sum += Math.pow(datay[j] - func(c, datax[j]),
(double)2);
      }
      lsq = sum;
      for (int i = 0; i < Math.pow(10, numCo) - 1; i ++) {
```

```
         String str = Integer.toString(i);
         int l = str.length();
         for (int j = 0; j < numCo - l; j ++) str = "0" +
str;
         double[] cTry = new double[numCo];
         for (int j = 0; j < numCo; j ++) {
            cTry[j] = c[j] + 1 / basicPrecision *
Integer.valueOf(str.substring(j, j + 1)).intValue();
            cTry[j] = Math.floor(cTry[j] * tunedPrecision)
/ tunedPrecision;
         }
         sum = 0;
         for (int j = 0; j < pieceSize; j ++) {
            sum += Math.pow(datay[j] - func(cTry,
datax[j]), 2);
         }
         if (sum < lsq) {
            lsq = sum;
            c = cTry;
         }
      }
   }

   double func(double[] c, double x) {
      double y = 0;
      for (int i = 0; i <= order; i ++) {
         y += c[i] * Math.pow(x, i);
      }
      return y;
   }
   public double[] getCoefficients() {
      return c;
   }
   public double getlsq() {
      return lsq;
   }
   boolean solve(double[] a, double[] b, int n) {
      for (int i = 0; i < n; i ++) {
         double mag = 0;
         int pivot = - 1;
         for (int j = i; j < n; j ++) {
            double mag2 = Math.abs(a[i + j * n]);
            if (mag2 > mag) {
               mag = mag2;

              pivot = j;
            }
         }
         if (pivot == - 1 || mag == 0) {
            return false;
         }
```

```
         if (pivot != i) {
            double temp;
            for (int j = i; j < n; j ++) {
               temp = a[j + i * n];
               a[j + i * n] = a[j + pivot * n];
               a[j + pivot * n] = temp;
            }
            temp = b[i];
            b[i] = b[pivot];
            b[pivot] = temp;
         }
         mag = a[i + i * n];
         for (int j = i; j < n; j ++) a[j + i * n] /= mag;
         b[i] /= mag;
         for (int i2 = 0; i2 < n; i2 ++) {
            if (i2 == i) continue;
            double mag2 = a[i + i2 * n];
            for (int j = i; j < n; j ++) a[j + i2 * n] -=
mag2 * a[j + i * n];
            b[i2] -= mag2 * b[i];
         }
      }
      return true;
   }
}
```

REFERENCES

[1] Cappello, P.; Christiansen, B.O.; Neary, M.O.; Schauser, K.E., "Market-based massively parallel Internet computing", Proceedings of the 3rd Working Conference on Massively Parallel Programming Models, 1998.

[2] Nisan, N.; London, S.; Regev, O.; Camiel, N., "Globally distributed computation over the Internet-the POPCORN project", Proceedings of the 18th International Conference on Distributed Computing Systems, 1998.

[3] Plasil, F. ; Stal, M., "An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM", Software-Concepts and Tools, Volume 19, Issue 1, 1998, Pages 14-28.

[4] Wei-Jin Park; Sang-Yoon Min; Doo-Hwan Bae; Pyeong-Soo Mah, "Object-oriented model refinement technique in software reengineering", Proceedings of the Twenty-Second Annual International Computer Software and Applications Conference (COMPSAC '98), 1998.

[5] Grundy, J.; Hosking, J., "Directions in modelling large-scale software architectures", Proceedings of International Conference on Software Methods and Tools (SMT 2000), 2000.

[6] Theodoros, L.; Edwards, H.M.; Bryant, A.; Willis, N., "ROMEO: reverse engineering from OO source code to OMT design", Proceedings of Fifth Working Conference on Reverse Engineering, 1998.

[7] Systa, T., "Understanding the behavior of Java programs", Proceedings of Seventh Working Conference on Reverse Engineering, 2000.

[8] Bujor D. Silaghi and Peter J. Keleher, "Object Distribution with Local Information", In Proceddings of the 21st ICDCS'01, Mesa, AZ, April 2001.

[9] Y. Gourhant, S. Louboutin, V. Cahill, A. Condon, G. Starovic, and B. Tangney, "Dynamic clustering in an object-oriented distributed system", Proceedings of OLDA-II (Objects in Large Distributed Applications), Ottawa, Canada, October 1992.

[10] Henri E. Bal and M. Frans Kaashoek., "Object Distribution in Orca using CompileTime and Run-Time Techniques", Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993.

[11] Shmulik London, "POPCORN - A Paradigm for Global-Computing", MSc Thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel June, 1998.

[12] Brecht, T., Sandhu, H., Shan, M., Talbot, J., "ParaWeb: Towards World-Wide Supercomputing", Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications. Sep. 1996.

[13] Fahringer, T., "JavaSymphony: a system for development of locality-oriented distributed and parallel Java applications", Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2000) 2000, Pages 145-152.

[14] A. Baratloo, P. Dasgupta, V. Karamcheti, and Z. Kedem, “Metacomputing with MILAN”, Proceedings of the Heterogeneous Computing Workshop (HCW'99), International Parallel Processing Symposium (IPPS/SPDP 1999), April 1999.

[15] P. Liu, “A Self Organizing Global Computing Architecture”, Master’s Thesis, National University of Singapore, 2001.

[16] Anderson, T.E.; Culler, D.E.; Patterson, D.A., “The Berkeley Networks of Workstations (NOW) Project”, Digest of Papers. COMPCON '95. Technologies for the Information Superhighway (Cat.No.95CH35737), 1995, Pages 322-326.

[17] Post, G. ; Kagan, A., “OO-CASE tools: an evaluation of Rose”, Information and Software Technology, Volume 42, Issue 6, 2000, Pages 383-388.

[18] U. Nickel, J. Niere, and A. Zündorf, “Tool demonstration: The FUJABA environment”, Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pp. 742--745, ACM Press, 2000.

[19] Kai Koskimies, Tatu Männistö, Tarja Systä, Jyrki Tuomi, “SCED - An environment for dynamic modeling in object-oriented software construction”, Proceedings of Nordic Workshop on Programming Environment Research '94 (NWPER'94), Lund, University, June 1994, pp. 217-230.

[20] “The Pizza Compiler”, http://pizzacompiler.sourceforge.net/doc/.

[21] Terence Parr, “Getting Started with ANTLR”, http://www.antlr.org/fieldguide/start/index.html.

[22] P.D. Terry, “Compilers and Compiler Generators”

[23] Leroy, X., "Java bytecode verification: an overview", Proceedings of 13th International Conference on Computer Aided Verification (CAV 2001) (Lecture Notes in Computer Science Vol.2102, 2001), Pages 265-285, 2001.

[24] B. Venners, "Java's Security Architecture", http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html.

[25] D. Low, "Protecting Java Code via Code Obfuscation", http://www.acm.org/crossroads/xrds4-3/codeob.html.

[26] Shand, M.; Vuillemin, J., "Fast Implementations of RSA cryptography", Proceedings of Symposium on Computer Arithmetic, 1993, Pages 252-259.

[27] Sander, Tomas; Tschudin, Christian F., "Towards mobile cryptography", Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, 1998, Pages 215-224.

[28] Farmer, William; Guttmann, Joshua; Swarup, Vipin, "Security for Mobile Agents: Authentication and State Appraisal", Proceedings of the European Symposium on Research in Computer Security (ESORICS), 1996.

[29] Palmer, E, "An Introduction to Citadel - a secure crypto coprocessor for workstations", Proceedings of the IFIP SEC'94 Conference, 1994.

[30] Aguilar, J.; Hernandez, M., "Fault tolerance protocols for parallel programs based on tasks replication", Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000.

[31] Gates, A., “On Defining a Class of Integrity Constraints”, Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering, Lake Tahoe, NV, 1996.