2005

An automatic code generation tool for partitioned software in distributed computing

Neeta S. Singh
University of South Florida

An Automatic Code Generation Tool For Partitioned Software in Distributed Computing

by

Neeta S. Singh

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Nagarajan Ranganathan, Ph.D.
Dewey Rundus, Ph.D.
Hao Zheng, Ph.D.

Date of Approval:
March 30, 2005

Keywords: Code Partition, Reverse Engineering, Procedural Language Application
Distribution, Parallel Virtual Machine, Heterogenous Systems

DEDICATION

To my dad, mom, sister Susheel and brother Jai.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ranganathan for giving me an opportunity to work under his guidance. Dr. Ranganathan has played a very important role in helping me with my thesis work, without whose time, endless encouragement and patience this work would not have been possible. I thank Dr. Rundus and Dr. Zheng for the time they took to review this thesis and their valuable suggestions to improve it. I would like to thank Viswanath Sairaman for his valuable ideas and help throughout the thesis work. I am indebted to the support and encouragement I have got from my family, friends and colleagues.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

AN AUTOMATIC CODE GENERATION TOOL FOR PARTITIONED SOFTWARE IN DISTRIBUTED COMPUTING

Neeta S. Singh

ABSTRACT

In a large class of distributed embedded systems, most of the code generation models in use today target at object-oriented applications. Distributed computing using procedural language applications is challenging because there are no compatible code generators to test the partitioned programs mapped on to the multi-processor system. In this thesis, we design a code generator to produce procedural language code for a distributed embedded system. Un-partitioned programs along with the partition primitives are converted into independently executable concrete implementations. The process consists of two steps, first translating the primitives of the un-partitioned program into equivalent code clusters, and then scheduling the implementations of these code clusters according to the data dependency inherent in the un-partitioned program. Communication and scheduling of the partitioned programs require the original source code to be reverse engineered. A reverse engineering tool is used for the creation of a metadata table describing the program elements and dependency trees. This data gathered, is used by Parallel Virtual Machine message passing system for communication between the partitioned programs in the distributed environment. The proposed Code Generation Model has been implemented using C and the experimental results are presented for various test cases. Further, metrics are developed for testing the quality of the results taking into consideration the correctness and the execution time.

CHAPTER 1

INTRODUCTION

In the realm of embedded systems, distributed computing is widely regarded as a potent means for maximum efficiency and optimum resource utilization. By mapping applications into distributed environments, costs due to task specific hardware commitments are mitigated. The procedure entails the use of a central controller which appropriates individual tasks to the various processing elements on the network. Furthermore, dynamic mapping ensures a coherent choice of the processing element, consistent with any demand of the code. Traditional methods involve manual partitioning of the code, followed by recoding of the application to utilize the middleware mechanism for communication. Recently, automatic partitioning of the application software and code generation are becoming critical in converting a regular centralized application into a distributed one.

Figure 1.1 elaborates the entire process sequence. The process begins with profiling the relevant application and obtaining important information about the corresponding tasks. Some common forms of storing extracted information include Task Graphs, CDFG - Conditional Data Flow Graph, DFG - a plain Data Flow Graph and FSM's (Finite State Machines). A partitioning algorithm is then applied which uses the task information to create clusters of code, and at the same time generate mapping information to assign these code clusters to various processing elements. The next step in the design process is Code Generation, wherein chunks of code are analyzed individually and the dependencies between code clusters are resolved by using a communication mechanism. Finally, the individual parts of the system are scheduled heterogeneously. The issues related to code partitioning and code generation are briefly introduced below.

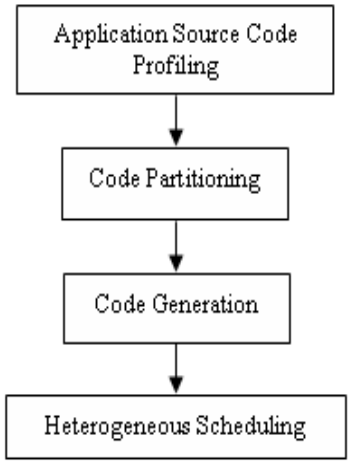

Figure 1.1. Process Flow For Mapping An Application On A Distributed Heterogeneous Environment

1.1 Automatic Code Partitioning

Application partitioning is the task of breaking up the functionality of an application into distinct entities that can operate independently over a distributed setting.

1.1.1 Code Partitioning Procedure

Any code partitioning technique involves the determination of the features related to: 1) system architecture; 2) application source code. The information about which processing element is most suitable for which task is gathered. Then, intelligent partitioning of the code is performed such that all tasks needing to be mapped to the same processor are grouped into a cluster. The partitioning algorithm also takes into consideration that there should be minimal expense in communication overhead and minimal imbalance of work load.

The next step to any code partitioning problem is code generation. The process involves modification of the code clusters, thus transforming them into individual programs.

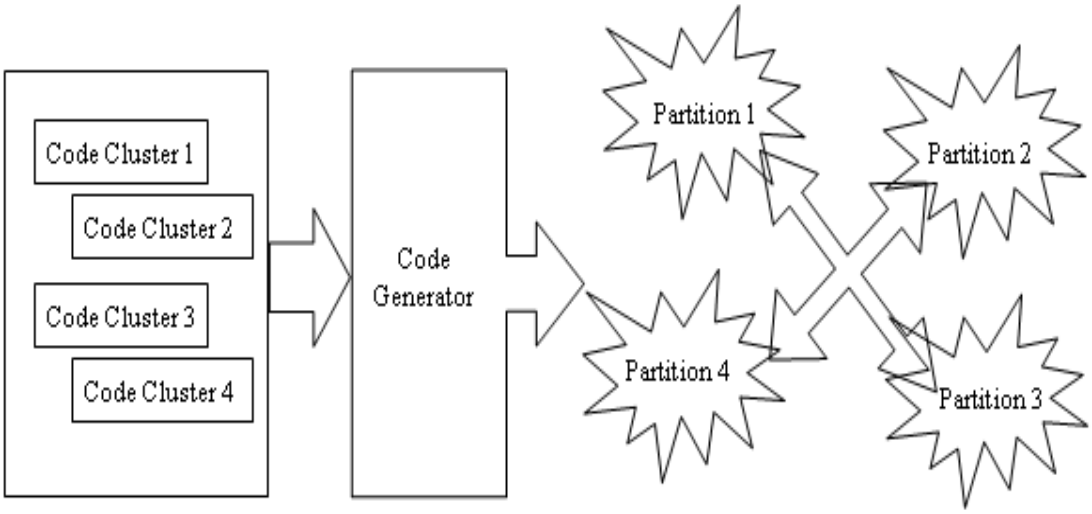

Figure 1.2. Code Generation Process For Code Partitions

1.2 Automatic Code Generation

Code generation is the process of converting the code blocks of an application into concrete implementations in the form of programs written in a high-level programming language. The process consists of the following steps: 1) translating the primitives of the application into equivalent implementations; 2) scheduling the implementations of primitives according to the data dependency inherent in the original application and, 3) handling the communication between these implementations for correct execution.

1.2.1 Code Generation Process

The code generation module takes input from the code partitioning unit. The annotated code, the partition primitives and the mapping data are given as input to the code generator. The code generator processes the chunks of code thus developing independently executable applications. Data dependency information is gathered by profiling the source code application. Based on this information, constructs are added to resolve the dependencies between the code partitions. A communication mechanism is introduced to

manage the synchronization between the subprograms, scheduling them as per the inherent dependencies in the original program.

The primary objective of the code generation module is correct and efficient execution of the distributed application. Most common optimization procedures are reducing the cost of communication and minimizing delay in execution. Factors that usually affect the execution time and synchronization efficiency are code size and the communication mechanism. Correct execution depends on the accuracy of the scheduling information.

1.3 Motivation

Traditional code generation approaches to evolving a centralized application into a distributed one involves employing conventional middleware (e.g. CORBA, DCOM, JAVA RMI) for communication purposes. The middleware programming has numerous complications and a programmer needs to perform several tedious steps to distribute an existing application. Thus there is a cognitive cost involved in addition to the inherent time and efficiency costs. More recent approaches to code generation [12] involve automatic distribution of the application, but unfortunately are suitable only for object-oriented applications. Another technique [17] uses a shared memory concept, and hence needs semaphores for scheduling purpose. Moreover, shared memory applications have unfavorable output bottle necks and are therefore not preferred.

As per the author's knowledge, no code generation technique exists that can produce code for procedural language applications in a distributed message passing environment.

1.4 Assumptions and Scope

This research makes certain basic assumptions. The objective is to focus on the code generation issues for procedural language sequential applications that need to be distributed. The research does not address the partition problem and assumes that the application has been annotated by a task clustering program, and that the partition primitives and mapping information are available as inputs.

The scope of this work includes a set of several of interrelated tasks:

- Partitioning original C-language multitasking application as per the primitives provided.
- Generation of code for the program segments.
- Mapping the sub-programs corresponding to the data provided.
- Implementation of communication layer in order to allow different program entities to communicate with each other
- Administering optimization for effective communication.
- Lastly, gauging correctness of produced code.

1.5 Thesis Contributions

In this thesis, we have developed a code generation scheme, with the integration of several code optimization strategies for executing a sequential procedural language software applications in C, on a message passing machine. An annotated multitasking C-Program, clustering information and mapping information are used as input. Partitioned blocks are created on the basis of the input data. Code generation process converts the program blocks into independent standalone applications. We then collect data (using a reverse engineering approach) from the source application to handle communication (using PVM) between the independent program elements.

The code optimization includes 1) reducing code duplication to curb code size. 2) removal of redundant communication primitives. 3) selection of one-to-all or multicast communication. The issues of data coherence and communication problems are considered and an analysis is presented for ensuring the correctness of the generated code.

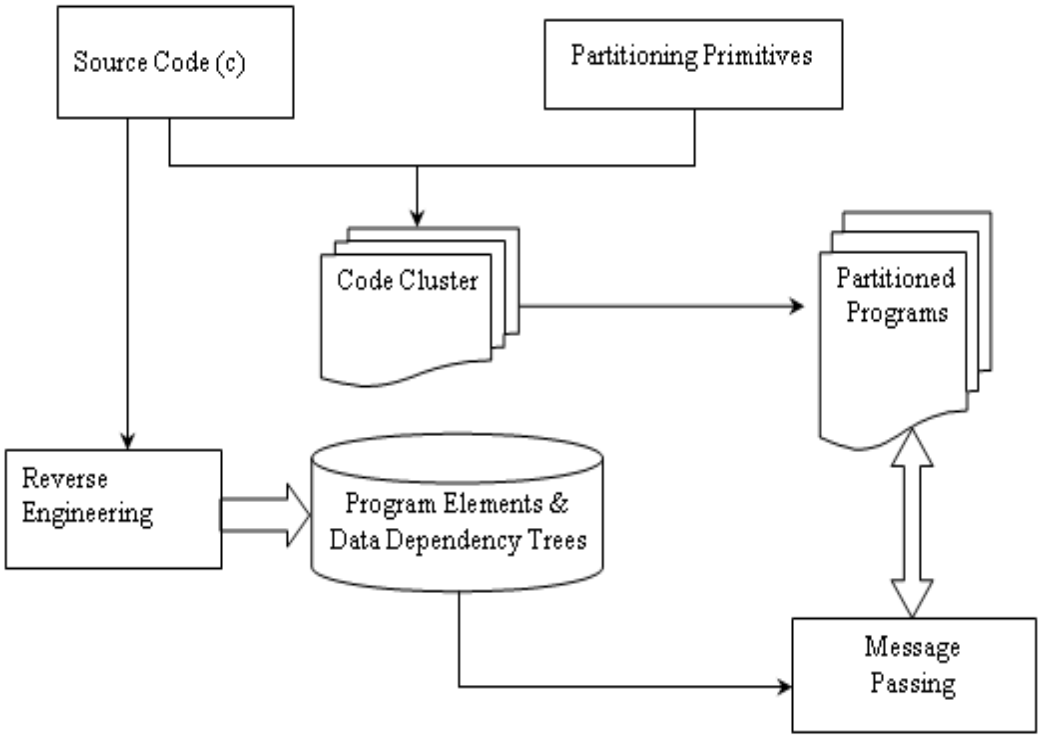

Figure 1.3. Complete Process Of Code Generation Proposed

1.6 Thesis Organization

The thesis is organized as follows: In chapter 2, a brief background of the topic and the related work in the area of code partition and code generation are presented, followed by, detailed description of code generation techniques, most relevant to this work.

Chapter 3 presents the intricacy of the code generation and gathering information about the data dependency and control flow which is needed for scheduling and correct execution. Chapter 4 discusses the communication technique used in this approach. Chapter 5 presents the experimental results. The conclusion and an overview of future work is discussed in Chapter 6.

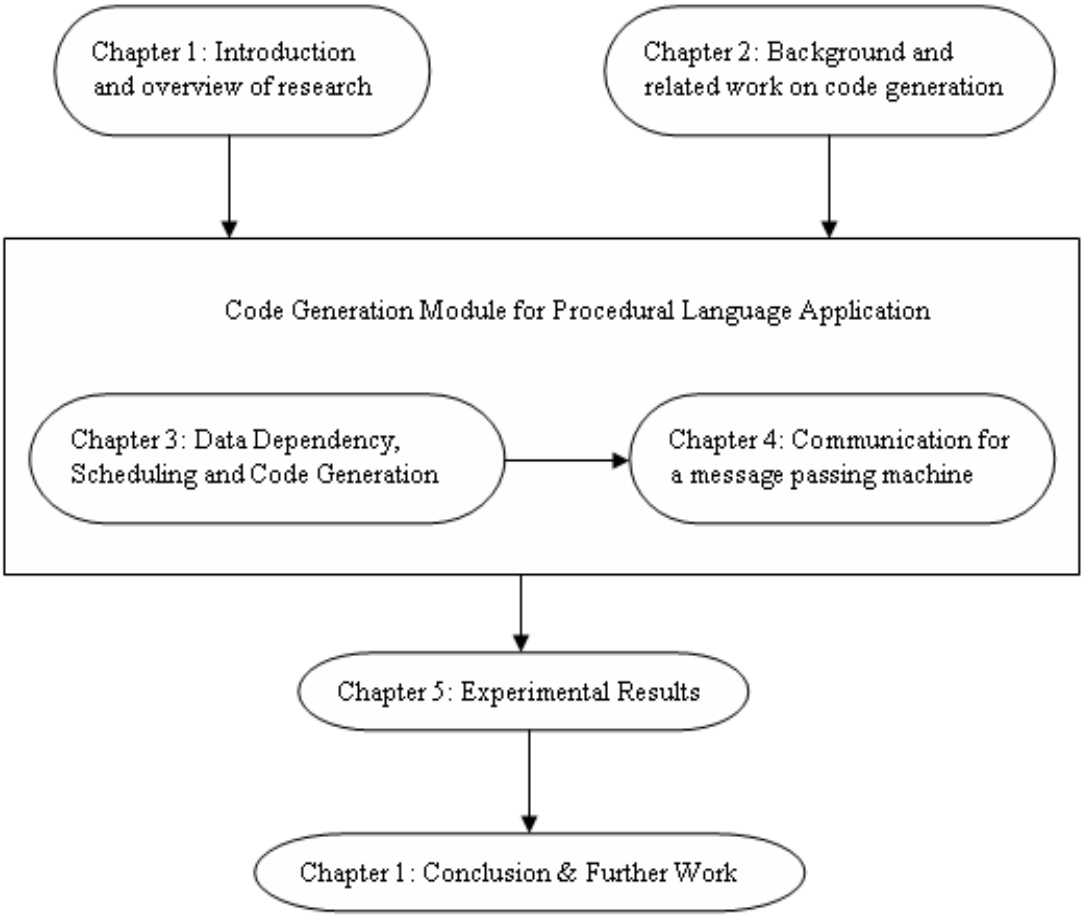

Figure 1.4. Thesis Outline

CHAPTER 2

BACKGROUND AND RELATED WORK

Popular acceptance of the internet has fueled a renewed interest in distributed applications due to the compelling advantages such as efficient use of data and resources. Although, the creation of distributed applications proves to be difficult there has been a lot of work in distributing object-oriented applications. However, creating a distributed procedural language application has still remained a challenging problem. In this work, we present a new code generation module to assist the execution of a procedural language application in a distributed manner.

As we have seen in the previous chapter, in order to aid distributed computing of a regular application the first step would be to partition the application followed by code generation to actuate the mapping of the code partitions. Hence it is important to first understand the partition problem and view the previous work done in this field.

The prior works that tackle the problem of code partitioning and code generation will be reviewed and the key projects and the techniques used in these areas will be discussed in this chapter, followed by the general communication techniques adopted in distributed architectures.

2.1 Code Partitioning

Code partitioning has been studied under different architectures with varied constraints. One of the earliest approaches was based on the ADA [22] programming language. This technique involves parsing ADA applications through a parser and subsequently through a partitioner. The partitioner isolates tasks based on Abstract Data Types. The approach is not suitable for large and complex applications as the partitioner is limited to a fewer

number of tasks and has a high run time and is targeted at object based applications. Partitioning of instructions for wide issue superscalar processors is dealt with in [2]. The focus of their work is on mapping code to the functional units of a clustered architecture with the objective of optimizing the tradeoff between work load balancing and reducing communication overhead. Nacul et al [17], have developed an automated code partitioner with a code generator for dynamic multitasking applications. The code partitioner is based on a generic clustering algorithm. But the clustering lacks efficiency as iterative improvements are randomly chosen for which partition metrics are computed and tested for improved quality. A genetic algorithm based approach for scheduling and mapping conditional task graphs for synthesis of low power embedded systems was investigated in [23]. The authors of the J-orchestra approach [12] propose an automatic partitioning for real time java based systems. The model proposed works only for java based object oriented applications.

The wide variety of techniques investigated in hardware/software partitioning can also be applied to code partitioning. In [1], the authors use finite automaton and constraint based approach for a target architecture consisting a general purpose core and a reconfigurable functional unit. Another notable contribution in considering power in the process of partitioning was investigated in [9] and [8]. A comparative study of simulated annealing and Tabu search based algorithms applied to system level partitioning is provided in [18]. In [21], the authors use binary code of the application to extract information necessary for partitioning. This technique is efficient as binaries provide accurate runtime information, the only drawback being the overhead in applying the technique to different target architecture. The approach of clustering for hardware/ software partitioning was attempted in [13] but it does not take into account the effects of power dissipation.

2.2 Communication Mechanisms

Communication methods used in distributed applications depend on the target architecture. the application language and the general methodology of partitioning. General

techniques in most distributed applications include CORBA, MPI, PVM and Java RMI. Phantom [17] uses a shared memory technique employing semaphores for synchronization. Shared memory systems do not require remote call communication procedures but synchronization of the shared data in order to avoid deadlock and for data coherency. Yang [24] uses message passing for communication in a shared memory environment. In [12], Java RMI is used for communication and hence useful only for java based applications. We briefly discuss the features of some the communication methods used in distributed systems.

2.2.1 JAVA RMI

Remote method invocation (RMI) allows Java developers to invoke object methods, and have them execute on remote Java Virtual Machines (JVMs) [6]. Under RMI, entire objects can be passed and returned as parameters, unlike many other remote procedure call based mechanisms which require parameters to be either primitive data types, or structures composed of primitive data types. That means that any Java object can be passed as a parameter - even new objects whose class has never been encountered before by the remote virtual machine. RMI is Java specific, and writing a bridge between older systems becomes the responsibility of the programmer. Additionally, it is designed for object based applications only.

2.2.2 CORBA

Common Object Request Broker Architecture (CORBA) [6] is a competing distributed systems technology that offers greater portability than remote method invocation. Unlike RMI, CORBA is not tied to a particular language, and as such, can integrate with legacy systems of the past written in older languages, as well as future languages that include support for CORBA. CORBA is not tied to a single platform (a property shared by RMI), and shows great potential for use in the future. In contrast, for Java developers, CORBA offers less flexibility, because it does not allow executable code to be sent to remote systems. CORBA services are described by an interface, written in the Interface Definition Language

(IDL). IDL mappings to most popular languages are available, and mappings can be written for languages written in the future that require CORBA support. CORBA allows objects to make requests of remote objects (invoking methods), and allows data to be passed between two remote systems. Remote method invocation, on the other hand, allows Java objects to be passed and returned as parameters. This allows new classes to be passed across virtual machines for execution (mobile code). CORBA only allows primitive data types, and structures to be passed - not actual code.

2.2.3 MPI

Message Passing Interface is a standard for writing message passing programs allowing efficient communication by avoiding memory-to-memory copying and allowing overlap of computation and communication. MPI [6] is more suitable for large multiprocessor homogeneous systems.

2.2.4 PVM

PVM is built around the concept of a virtual machine and is useful when the application is executed on a networked collection of hosts, particularly if the hosts are heterogeneous. PVM [6] contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPP. The larger the cluster of hosts, the more important PVM's fault tolerant features become. The ability to write long running PVM applications that can continue even when hosts or tasks fail, or loads change dynamically due to outside influence, is quite important to heterogeneous distributed computing.

We choose PVM as the message passing technique in our method because it has these salient features. It is best suited for largely heterogeneous distributed systems. Other features like easy portability and robustness make it a better choice. MPI is good but not easily portable and works better in a homogeneous environment.

2.3 Code Generation

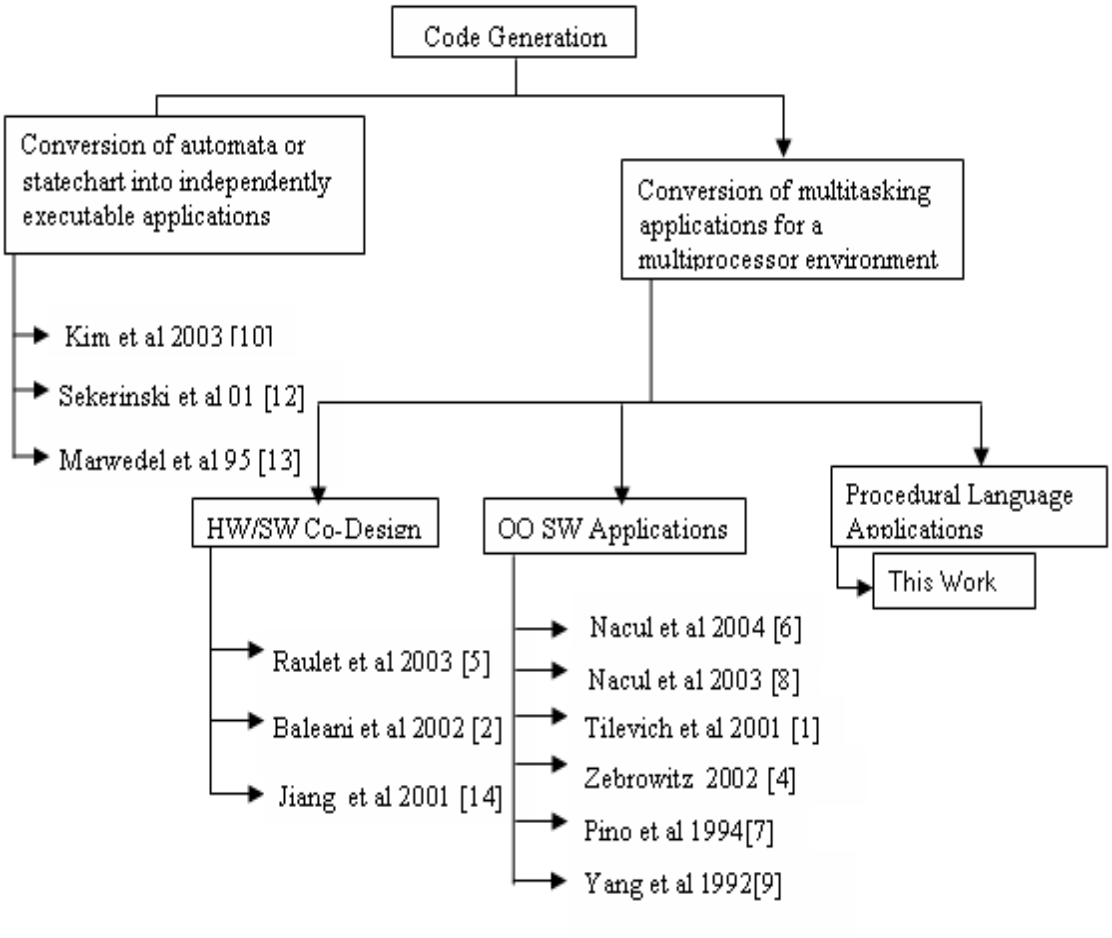

Figure 2.1. A Taxonomy Of Prior Works In Code Generation

Automatic Code Generation has been used in a lot of scenarios in some cases as a mere translator for conversion from one language to different other target languages. Pertaining to partitioning problems code generation can be studied for the two distinct methods : a) code generation from Statechart or automata [5], [25], [10], [14], which (mostly seen in HW/SW partition problems) and b) code generation for software application partition problems. The closest to our approach are Pangea, Coign and J-orchestra.

2.3.1 Automata and Statechart

In [24], the problem of parallelization of applications for distributed memory architectures is addressed. A directed acyclic graph (DAG) was used to model parallel computation. The tasks in the DAG were clustered with the help of a scheduling algorithm called DSC (Dominant Sequence Clustering). The code generation method proposed by them involves executing a schedule of an arbitrary task graph based on an asynchronous communication model.

The tasks in the DAG access data in a shared memory programming style. Message Passing was used for communication. The authors in [10], propose a code generation framework for hybrid automata which deals with continuous and discrete data dependency. The automatic code generation process proposed is decomposed into two phases: one translating each primitive into a piece of code and the other scheduling the pieces of code consistent to data dependency. In [7], a CAD tool, SynDEx is used for partitioning and code generation. The authors develop fast automatic prototyping process dedicated to parallel architectures. In [1], the authors introduce an automated hw/sw partition and code generation flow for control applications. It uses the EFSM model and derives hw/sw implementation automatically from high-level synchronous specifications. Using the automated synthesis flow, embedded applications can be mapped from high-level language specifications like Esterel, down to hw/sw implementation on the reconfigurable architecture platform. However, this approach is limited to the applications that can be modeled and programmed by extended finite state machines. Recently, there has been a lot of work in automatic code generation for embedded systems pertaining to software synthesis of hardware/software Codesign problems. Most of these menthods use conversion of automata into code. This method is more suitable for hw/sw Codesign partitioning problems. Though this method can also be used for software applications, there is an overhead of translating of the automata into software code.

2.3.2 Application Partitioning

There has not been much focus to the problem of code generation for distributed software. Mostly, when applications need to be distributed, programmers manually partition the application and assign those sub programs to machines. With the help of middleware such as RRPC, CORBA and DCOM, these sub programs are successfully executed on the respective machines. Often the techniques used to choose a distribution are ad hoc and create one-time solutions biased to a specific combination of users, machines, and networks. To address this issue, the authors in [13] proposed a automatic distributed partitioning system for binary software applications based on the COM model. Given an application (in binary form) built from distributable COM components, Coign constructs a graph model of the application's inter-component communication through scenario-based profiling. Later, Coign applies a graph-cutting algorithm to partition the application across a network and minimize execution delay due to network communication. Using Coign, even an end user (without access to source code) can transform a non-distributed application into an optimized, distributed application. This method cannot be used as a generalized approach as very few of the real world applications are written as collections of the COM component.

J-Orchestra operates at the Java bytecode level and rewrites the application code to replace local data exchange (function calls, data sharing through pointers) with remote communication (remote function calls through Java RMI [18], indirect pointers to mobile objects). The resulting application is guaranteed to have the same behavior as the original one (with a few, well-identified exceptions). J-Orchestra receives input from the user specifying the network locations of various hardware and software resources and the code using them directly. A separate profiling phase and static analysis are used to automatically compute a partitioning that minimizes network traffic.

The main objective of our program and J-orchestra is the same - Automatic application partition in a distributed embedded environment. Only thing is that their model supports only object oriented applications written in JAVA. Our approach caters to procedural language applications written in C. In the J-orchestra approach the whole application is

rewritten and later the partitions are computed. We on the other hand first compute the partitions then only where necessary we insert additional code for PVM calls. In J-orchestra all the local data exchange is replaced with remote communication which causes more overhead.

Pangaea [16][17] is an automatic partitioning system that has very similar goals to our approach. Pangaea is based on the JavaParty [13] infrastructure for application partitioning. Since JavaParty is designed for manual partitioning and operates at the source code level, Pangaea is also limited in this respect. Therefore, Pangaea has little applicability to real world situations, especially with limited manual intervention.

2.4 Summary

This thesis describes a new code generation methodology for executing partitioned code of procedural language applications in distributed environments. In this chapter, prior work in the area of code partitioning and code generation was studied with emphasis to those directly related to this work. Unlike other methods, we use Parallel Virtual Machine (PVM) as the message passing technique for communication between the application partitions which works well independent of the number of partitions.

CHAPTER 3

PROPOSED FRAMEWORK FOR CODE GENERATION

This work presents an automatic code generation tool for partitioned centralized procedural language applications without manually changing the application source code. Instead, the user expresses how the application is to be partitioned and the tool can then rewrite the existing application code to replace local data exchange (e.g., function calls, data sharing) with specialized communication primitives. This automatic approach to code generation has significant potential. It can simplify drastically the process of partitioning applications.

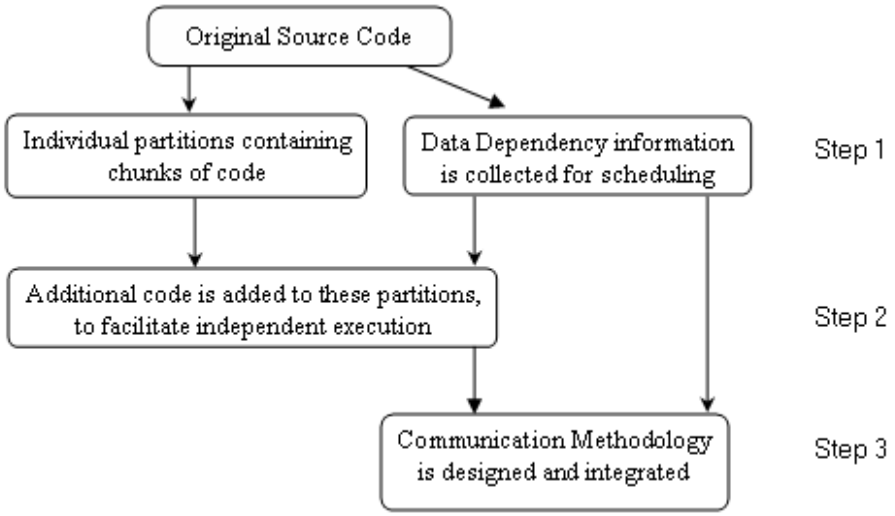

Figure 3.1. Steps Involved In Code Generation

The design and implementation of the proposed code generation tool is shown in Figure 3.1 above. The first step involves gathering data and the creating a metadata information repository. This is the most crucial stage as the accuracy of the code generator depends

on the data accumulated. The next step after is partitioning of the program, followed by adding constructs to create standalone programs.

The third step involves the inclusion of a communication mechanism. Data communication and synchronization are implemented via message passing. Optimization methods used to improve the quality and performance of code such as using multicast or broadcast communication, using aggregate communication and elimination of unnecessary communication are also explained in detail.

3.1 Need For A Data Repository

For synchronization and correct execution of the sub-programs of a distributed application, managing the communication between them becomes crucial. A communication methodology needs to be introduced that will be responsible for scheduling these sub-programs. In order to implement this communication layer, detailed information about the source code is required. The scheduling is affected by important data like

1. control flow,
2. data dependency and
3. functionality of the software program.

These details are not available from the code itself. Subsequently, reverse engineering is applied to acquire this data about the program.

The discussion below sheds light on the process of reverse engineering and its utility.

3.1.1 Reverse Engineering

Reverse Engineering [15] is a process of extracting information from the existing software system for its better understanding. During this process, the source code is not altered; although additional information about it is generated. The subject software system is represented in a form where many of its structural, functional and behavioral characteristics can be analyzed.

A source code itself cannot answer all the questions about the entities used in it. The reverse engineering [11] of the code helps answer some important questions about the program such as:

- What are the entities used in the program?
- Where are they changed?
- Where are they referenced?
- What entities do they depend on?
- What other entities depend on them?

3.1.2 The "Understand C" Tool

In order to gather functional and behavioral information of the application that needs to be partitioned, we reverse engineer it using the UNDERSTAND C Tool developed by Scientific Toolworks, Inc. [20]. Several other tools like inSight, Essential Metrics and Imagix were tested and found unsuitable. The "Understand C" tool was preferred because of its ease of use, speed of generating reports and accuracy.

3.1.2.1 Functionality and Utility of Tool

The UNDERSTAND C tool analyses the application code to identify the program components and their interrelationships and creates a higher level representation of the software program that is easily understandable. It uses a graph model to visualize the flow of the code. The application to be reverse engineered is given as input, the RE tool profiles the code creates a complete documented report giving intricate details of the application at hand.

This tool gives two basic reports. The first contains a list of the program components i.e. a list of all the variables and functions with complete details of their data types. The second report contains a list of the locations where the program entities where referenced

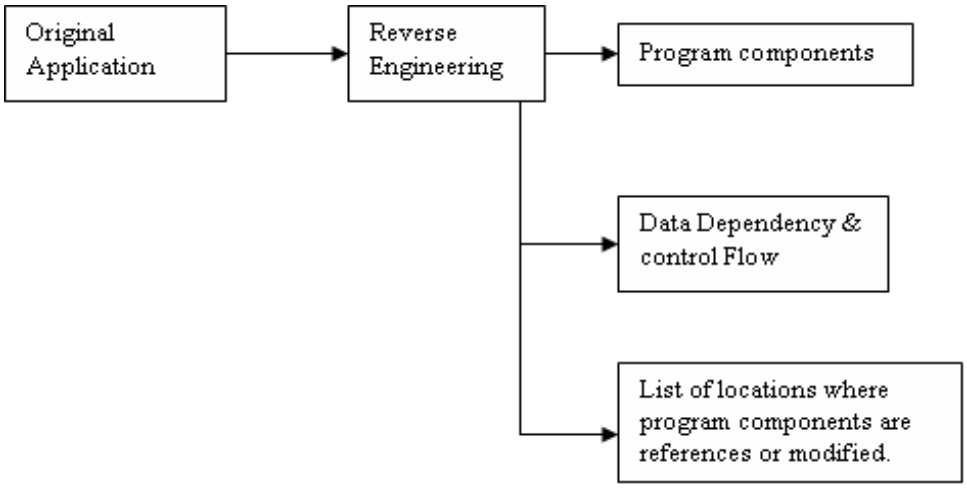

Figure 3.2. Reverse Engineering Process

or modified. The control flow and the data dependency of the software program are also illustrated by the tool.

3.2 Creation of Metadata Table

The Code Generation tool proposed requires knowledge about the working of the application, before it can be distributed over several processors. This information includes the data dependency, control flow and list of program components. Although the reverse engineering tool provides a multitude of information about the application, there is a need to glean out and store appropriate information that will aid the distribution. This is done with the help of maintaining a Metadata table.

The steps needed to create the Metadata table are:

1. A gawk script is used to format the information given by the reverse engineering tool.
2. The locations of where the program objects were modified, used or declared are noted.
3. The annotated source code is profiled to obtain the task information.
4. Program components are mapped according to the tasks in which they are referenced.
5. All the information gathered by the previous steps is stored.

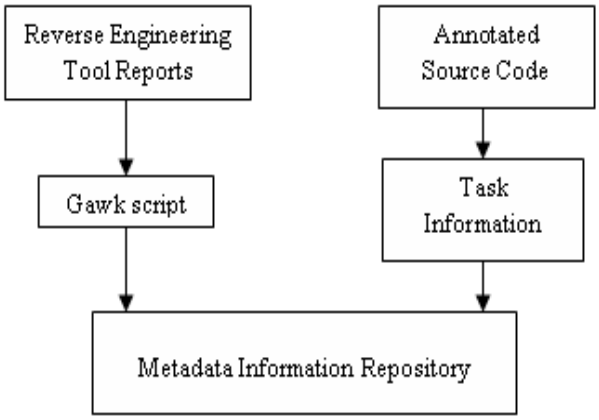

Figure 3.3. Creation of Metadata table

At the end of the preprocessing step the metadata table contains the following:

- List of program entities
- Location in which the program entities are referenced
- List of tasks
- Entities used in each task
- Task dependencies

Although the creation of a Metadata table accounts for an overhead, this process is required to be performed just once. Thereafter, distributing the application by varying the number of clusters does not require this process to be performed again.

3.3 Partitioning Program To Form Independent Executions

The annotated C-Language application is parsed though a separator program and partitions are created using clustering. These partitions or blocks of code are called code clusters. In order to execute the code clusters on specified processing elements, it is necessary to convert the code clusters into independently executable programs. The process of clustering and program formation is explained in following sections.

3.3.1 Clustering

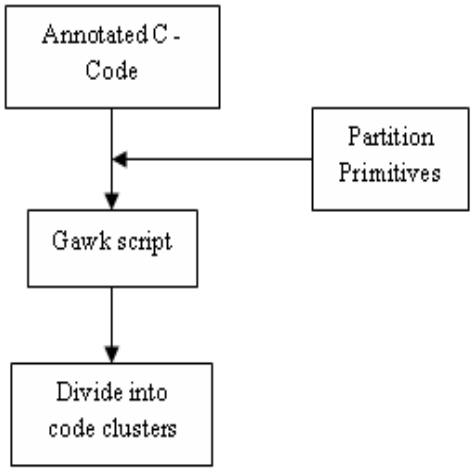

Figure 3.4. Clustering

A scripting program is executed on the annotated source application. Each cluster is a group of several tasks.

We will refer to an example shown in Figure 3.5. In the example the application is partitioned into three clusters:

- Code Cluster 1 containing Task 1 and Task 7
- Code Cluster 2 containing Task 3
- Code Cluster 3 containing Task 5

When partitioning is dynamic, the user can select the tasks that need to be grouped in the same cluster as well as the number of clusters that the application is to be partitioned into.

3.3.2 Adding constructs

The basic goal is to convert the code clusters into independently executable programs. This requires adding constructs like declaration information and functions to the code clusters.

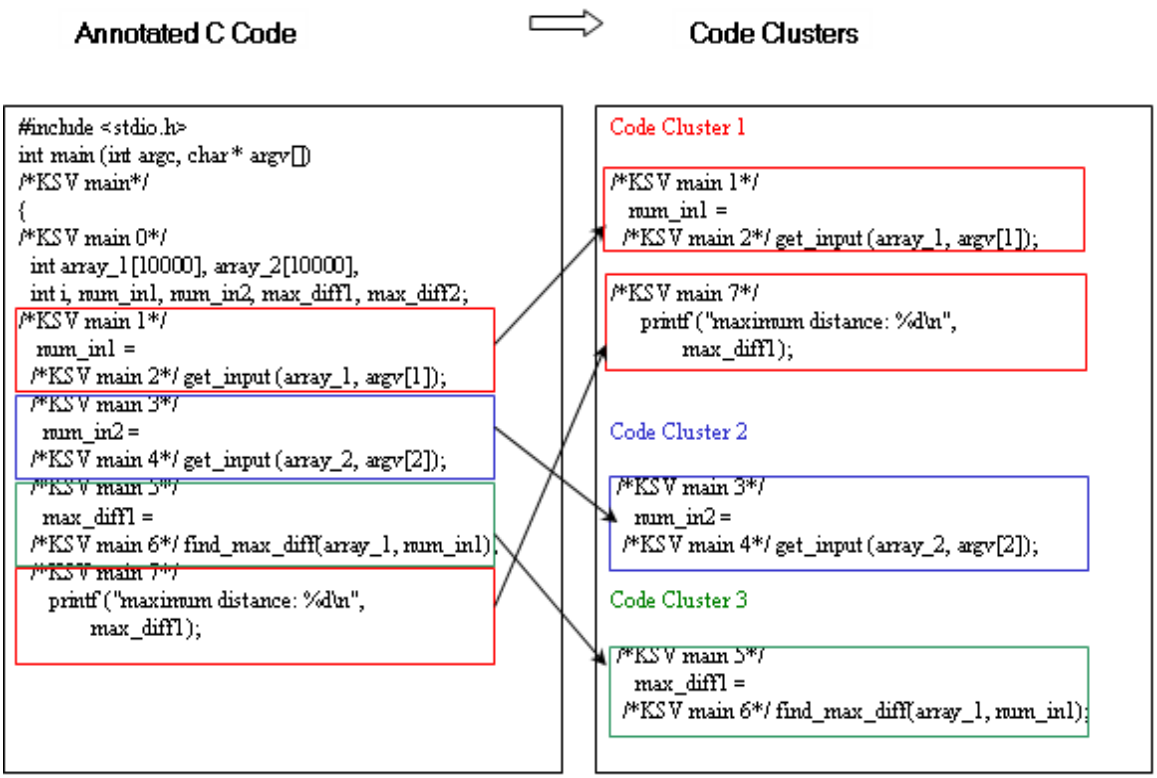

Figure 3.5. Code Cluster Formation

A naive approach to adding constructs would be to simply duplicate the functions in all the partitions. However, as the number of sub-programs increase (the number of partitions in which the program is distributed), the code size would explode.

The approach used in this work is to obtain certain information about the tasks included in each partition. Then, using the metadata table, find the corresponding entities and functions used in each of these tasks. Based on this information, declaration and functions for the entities present in the partition are added. This optimization helps in curbing the code size and reducing redundant code duplication.

As seen in Figure 3.7, code cluster 1 is converted into a program Partition 1 by adding constructs. This is accomplished by adding the appropriate header files and functions that are being referenced by the tasks in the code cluster. Later, introducing the declaration information of the entities in order to errors on compilation.

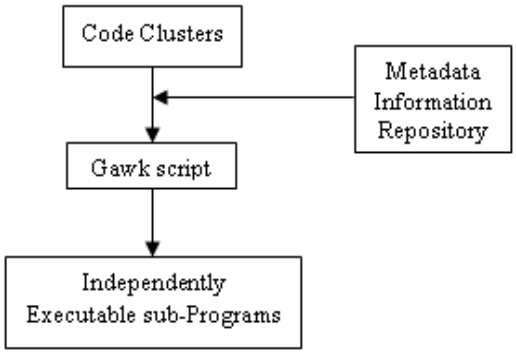

Figure 3.6. Adding Constructs

3.4 Communication Methodology

The main function of code generation is to produce code for each distributed processor such that every task is correctly executed. Before a task is executed, the task with precedence to it, needs to complete execution and the data items referenced by the task are made available in the local memory. Additionally, after each new data item is computed, it must be sent to the appropriate processors whose tasks need this data for further computation. This requires the implementation of a communication layer.

3.4.1 Selection Of PVM As The Communication Mechanism

Most common communication technologies include MPI, Java RMI and PVM. Java RMI is more suitable for Java applications, hence is not considered for this approach. We choose PVM [6] over MPI for its robustness and suitability for heterogenous architectures.

3.4.2 PVM Communication Process

The PVM message passing process is explained in the steps given below:

- *Initializing a new send buffer:* Before any message can be sent out from one processor to another, we need to allocate and initialize a send buffer.
- *Placing data into the send buffer:* This second stage of PVM message passing involves placing the data that is to be sent into the newly initialized send buffer.

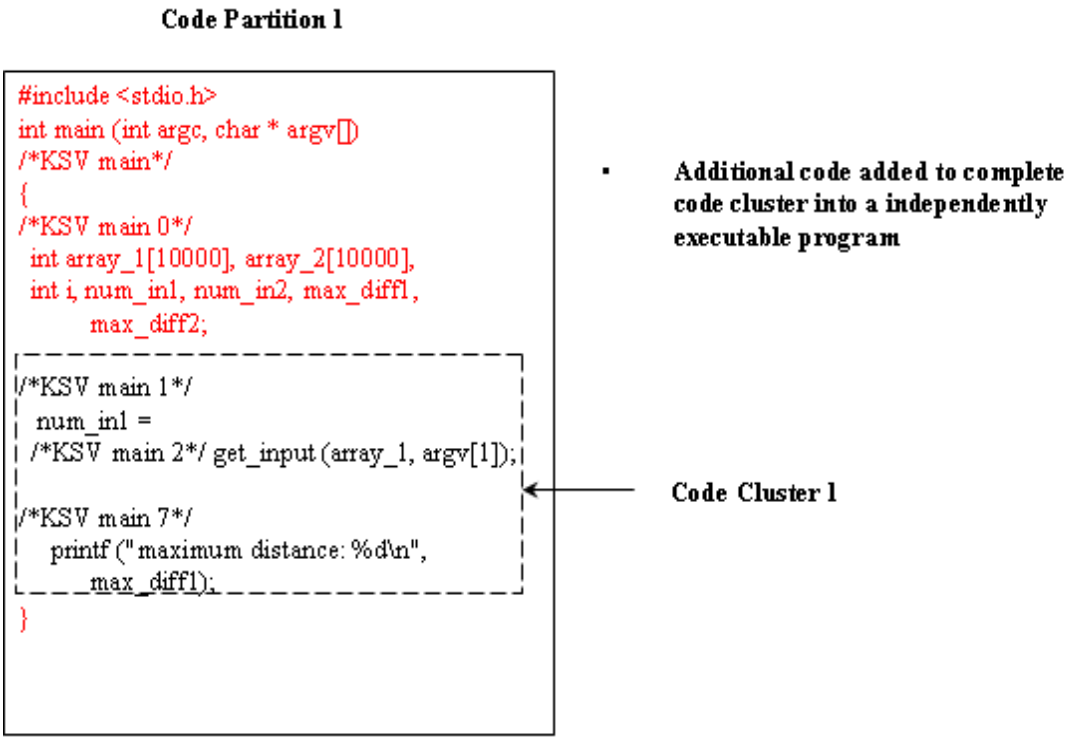

Figure 3.7. Creation Of Sub-Programs

- *Sending Out The Data In The Send Buffer:* This step involves sending the data in the send buffer to the destination processor.
- *"Catching" A Message and Placing It Into A Receive Buffer:* The destination processor receives the data from the sending processor and caches it in the receive buffer. There are two distinct type of receive commands, one is a blocking receive, wherein the destination processor will stall till the data arrives. The other type of receive command is non-blocking and resumes execution if the data does not arrive.
- *Unpacking the Data From the Receive Buffer:* The final step includes unpacking the data from the receive buffer.

3.4.2.1 Asynchronous Messaging Primitives

Several asynchronous PVM message passing primitives used in this work are mentioned below:

- pvm_recv(int tid, int msgtag)

  Executing a receive command will receive a message with ID *msgtag* if it is in the communication buffer of the processor. Otherwise it blocks the processor idle till the message arrives.

- pvm_mcast(int *tids, int ntask, int msgtag)

  Executing a multicast command will send the data in the active message buffer *ntask* number of tasks within *tids*. This is useful for broadcasting the data.

- pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

  A process named *task* will be initiated in the specified processor. The taskID of the spawned process will be sent to the parent process.

- pvm_initsend(PvmDataDefault)

  The routine *pvm_initsend* clears the send buffer and prepares it for packing a new message.

3.4.3 Implementing PVM Communication

PVM communication primitives are automatically interleaved in the sub-programs in a manner that the distribution appears invisible to the user. The technique used is fast, error free and does not involve intervention by the programmer unlike traditional methods that involved manual coding of the communication layer. However, it is necessary for the architecture to be PVM compliant.

In the transparent mode the mode, tasks are automatically executed on the most appropriate computer. Nevertheless, PVM gives the flexibility to allow the user to specify the processors to which the tasks need to be mapped in the architecture-dependent mode. In low-level mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of necessary data conversions from computer to computer as well as low-level communication issues.

Once the centralized application is split into equivalent independent executions, the parent partition is selected. PVM constructs are then inserted into the parent partition

to mobilize spawning of the other partitions on the specified appropriate processors. Then each partition is carefully profiled and message passing passing constructs are conservatively appended.

The message passing constructs are inserted into the subprograms based on the following:

- For every task that modifies an entity the metadata table is searched to check if there are other tasks that will be affected. If the tasks being influenced by the modification belong to different partitions the new value is broadcasted them.
- Similarly, before computing a task, the program checks whether the entities in that task have been modified previously. If the entities have changed, the new values are received from the respective partitions.

As the send command is non-blocking, the processor that sends the message can proceed with the execution. But the receive command is a blocking command and will stall till the message is not received. This aids any inherent parallelism in the program on the other hand ensures the correct scheduling and execution of the distributed program.

Figure 3.8 shows the need for communication. After the preprocessing and partition process is complete we obtain the above partitions 3.8 (Code Partition 1 and Code Partition 3). The tasks have been segregated in a manner that an entity that is computed in one partition needs to be printed in another partition. The entity needs to be sent to Code partition 1 as soon as it is computed for correct execution.

The communication between the two partitions is facilitated by addition of PVM primitives as shown in Figure 3.9. In this implementation, Code Partition 1 is chosen as the parent process which spawns another process (Code Partition 3).

The PVM header files are included in all the partitions. The program finds that the entity modified in task 6 of partition 3 is required by task 7 of partition1. PVM instructions to initialize the send buffer, pack the new value and send it to the parent processor are appended. Similary, in code partition1, PVM instructions to receive and unpack the new value are automatically appended.

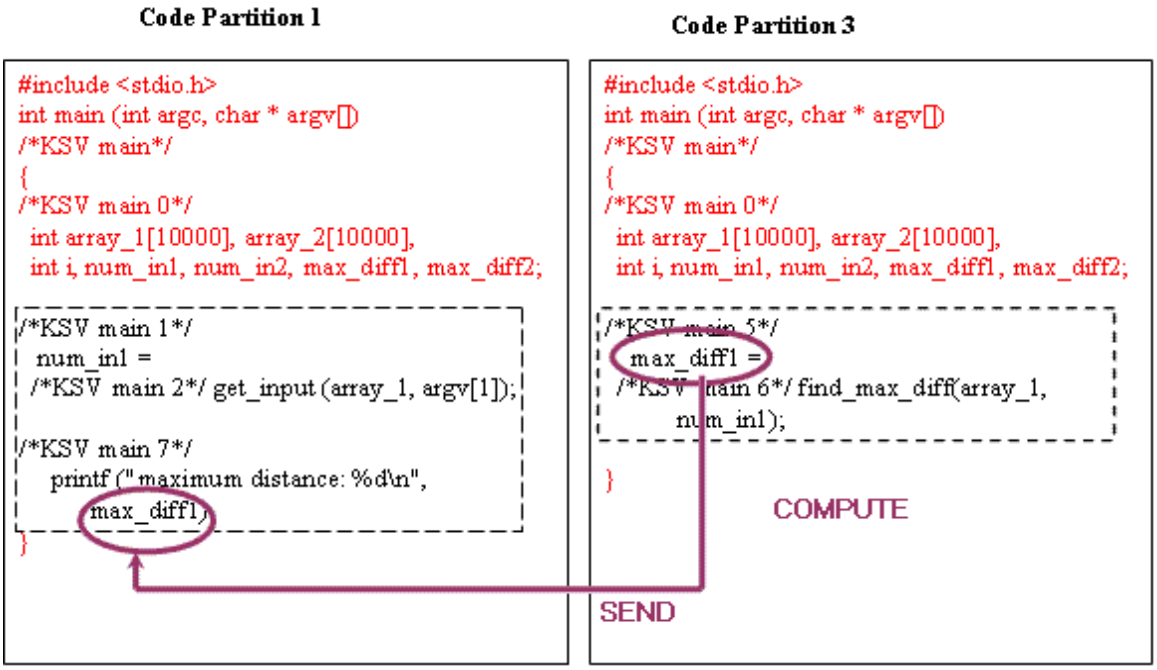

Figure 3.8. Communication Between Sub-Programs

Figure 3.10 depicts the algorithm for inserting PVM instructions in the partitioned code.

3.4.4 Optimizations

This section deals with the optimizations handled for improving communication time. The techniques of using broadcast communication and elimination of redundant communication are explained in the following sections.

3.4.4.1 Broadcast Communication

A task could send the same message to many other tasks in different processors. The incorrect way to do this would be using repeatedly one to one sending, which could create communication contention in the processor network. Thus a broadcasting scheme should be used to take advantage of the network topology in order to alleviate network message traffic.

Code Partition 1

```
#include <stdio.h>
#include "pvm3.h"
int main (int argc, char * argv[])
/*KSV main*/
{
/*KSV main 0*/
 int array_1[10000], array_2[10000],
 int i, num_in1, num_in2, max_diff1,
        max_diff2,cc,tid;
cc = pvm_spawn("partition3", (char**)0, 1,
        "grad", 1, &tid);
/*KSV main 1*/
  num_in1 =
 /*KSV main 2*/ get_input (array_1, argv[1]);

   pvm_recv(tid,2);
   pvm_upkint(&max_diff1,1,1);

/*KSV main 7*/
    printf ("maximum distance: %d\n",
        max_diff1);
}
```

Code Partition 3

```
#include <stdio.h>
#include "pvm3.h"
int main (int argc, char * argv[])
/*KSV main*/
{
/*KSV main 0*/
 int array_1[10000], array_2[10000],
 int i, num_in1, num_in2, max_diff1,
        max_diff2,ptid;

ptid = pvm_parent();
/*KSV main 5*/
   max_diff1 =
 /*KSV main 6*/ find_max_diff(array_1,
        num_in1);

        pvm_initsend(PvmDataDefault);
        pvm_pkint(& max_diff1, 1, 1);
        pvm_send(ptid, 1);
}
```

Figure 3.9. Interleaving PVM Primitives

In this implementation, after an entity is modified, the control flow information in the metadata table is searched to check if all other partitions require the modified entry. If they all require the modified value instead of having multiple send receive commands we replace it with a single broadcast command.

3.4.4.2 Eliminate Unnecessary Communication

Local Memory: If two tasks are assigned in the same processor, there is no need for communication between them. This is because even if one task modifies an entity, the new value still exists in the local memory. For example in the Figure 3.11 two tasks (task 2 and task 3), assigned to the same processor (processor 2), are each receiving data from another task (task 1) in a different processor (processor 1). Processor 1 issues two sends with the same message to processor 2, and processor 2 issues two receives for the task 2 and task 3.

For each task:

- Check the metadata table for all the entities used in the task.
- For each entity in the task:
  - Check if the entity was modified in an earlier task.
  - If the entity is modified check if the task in which the entity is modified is within the same partition.
    - If within the same partition then no need to have a receive statement.
    - If not then receive from the other task.
  - Similarly, if an entity is being modified in the task, broadcast the new value if the entity is being used in a different partition.

Repeat for all tasks

Figure 3.10. Communication Algorithm

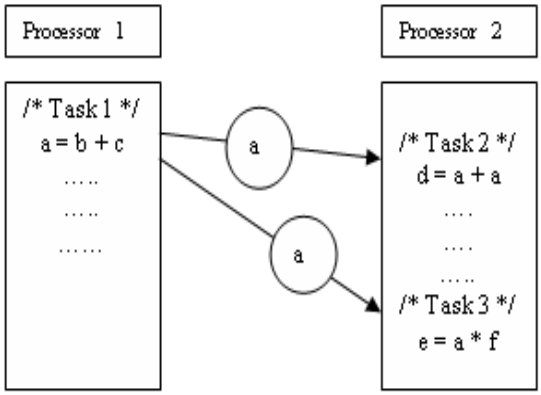

Figure 3.11. Local Memory Optimization

An optimized approach should eliminate such redundant communication. Processor 2 does not require receiving the same data for task 3 that it already has received for task2, as the data can be stored in the local memory. For task 3, data can be fetched from the local memory itself, reducing he communication expense. Thus a redundant receiving can be detected if the data item has already been in the local memory.

Dead Messages: Another optimization is to eliminate communication if an object is not going to be used any further. This is because even if the data item is modified it is not required by any other task.

3.4.4.3 Aggregate Communication For Consecutive Tasks

Using aggregate communication for consecutive tasks, when required, improves performance. For example, in the Figure 3.12, Task 3 in processor 2, requires two entities that are being changed in processor 1. One approach could be two send the entities separately, each time the entity was modified using two send and receive messages. This approach uses an optimized method of packing both the entities in a single send command. The basic aim is to send less number of long messages.

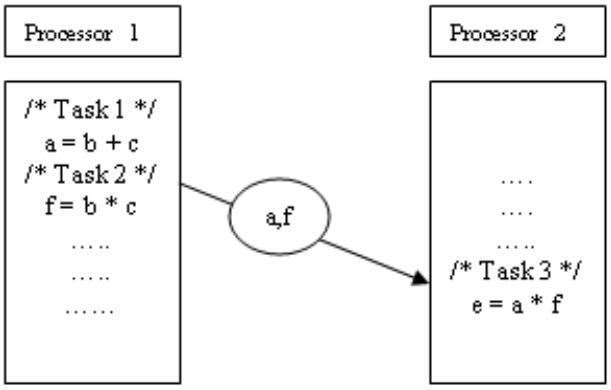

Figure 3.12. Aggregate Communication Optimization

3.5 Summary

This chapter covered the implementation of the automatic code generator tool. Important program information was collected and stored in a metadata table to aid interleaving the communication layer. Additionally, the application is segregated into independent sub-programs by conservatively adding constructs to the code clusters. In this chapter the rationale behind using PVM as the communication methodology was discussed. The chapter also covered the implementation and design of PVM as the communication methodology in our work. A detailed description of the optimization techniques employed to increase performance of the distributed code was presented in the last section of the chapter. The techniques include usage of aggregate communication, broadcasting information instead of using one-to-one communication and elimination of redundant communication.

CHAPTER 4

EXPERIMENTAL RESULTS

This chapter presents a set of experimental results to verify the efficacy of the automatic code generator tool. The implementation involved three steps - partition into sub-programs, addition of constructs for stand alone execution and finally, application of the communication methodology. For the integration of the three phases, the output of each phase is made compatible for the next step.

4.1 Architecture

The architecture under consideration is a distributed system with eight 900Mhz processors. The processing elements have individual memory storages with a total of 32GB RAM. All the components are connected together by means of a shared bus, with known data transfer speeds. The experiments were executed on a SunOS system, specification of which is listed in Figure 4.1. The execution time of the algorithm was calculated using the functions of ¡sys/timeb.h¿ library. The sparc processor acts as the master processor in this case and then sends out data to the other processors and collects the computed results. The key assumption is that all tasks are capable of executing on all processing elements, although the time taken to execute might vary.

Our results and analysis are based on the experiments performed by testing on sample C-programs, in table 4.1, that were split into tasks by a task clustering and partitioning algorithm. The first sample program, parallel.c was split into 44 tasks. This program generates mean, variance, correlation and autoregressive values for the input arrays. The other program is fft.c that computes the Fast Fourier transform. This application was split into 22 tasks by the code partitioning module. The code generation tool was also

Table 4.1. Experimental Sample Set

No.	Sample Program	Number of tasks	Brief Description
1	Parallel.c	44	Compute mean, variance, co-relation and auto regression
2	FFT.c	22	Computes Fast Fourier transform
3	Edgesob.c	52	Soble Edge detector program
4	RGBconv.c	93	Program to convert color images to grayscale
5	Hough.c	38	Hough transform program

tried for few image processing programs that compute edge detection for an input image, convert color images to grayscale and compute Hough transforms to determine shapes in images. We chose programs that could be split into a large number of tasks, unlike many benchmark programs that are very short programs that are executed for a large number of iterations, to test distribution on the network. The code generator was tested on programs that varied from a few hundred lines to a few thousand lines. Each of these applications was split into partitions varying from 3 to 7. In order to illustrate the effectiveness of the model, the experiment is executed for an average of twenty times, varying the task clusters for each partitioning. The results of the distributed application are compared with the results of the original un-partitioned program for all iterations and the code generated tested for correctness. Optimization schemes have been applied to reduce overhead and therefore minimize execution time.

4.2 Correctness

The two sample programs that have been used in the experiment have been partitioned into clusters ranging from 3 to 7, each executed for twenty iterations with different set

System	SunOS
Node	sunblast
Release	5.9
KernelID	Generic_112233-11
Machine	sun4u
OEM#	0
Origin#	1
NumCPU	8

Figure 4.1. System Details

of tasks clustered into a partition. For each execution of the code generated by the tool correctness was verified by checking:

- if the code generated for the sub-programs created were in accordance to the tasks in the partition primitive list given as input
- if the mapping was according to the mapping data provided at input
- if the results generated by executing the distributed application was in accordance to the results generated by the original program.

The original source code is not re-coded, only additional PVM constructs are added for communication, it can be safely said that the code generation tool does not change the functionality of the program.

4.3 Overhead

Although the the first two steps in the implementation of the tool that involve partitioning and creation of the metadata table, account for some of the overhead, similar to profiling, this process is required to be performed just once. Thereafter, distributing the

application by varying the number of clusters does not require this process to be performed again. The expense caused by the data gathering process is neglected and the overhead is computed in terms of percentage increase in lines of code.

Table 4.2. Overhead In Terms Of Lines Of Code

Number of clusters	% increase in Number of Lines using Naïve approach	% increase in Number of Lines using Optimized approach
3	35.78	18.97
4	38.63	19.97
5	42.81	20.88
6	45.24	21.31
7	47.22	22.56

The insertion of PVM constructs, to send and receive data between the partitions, increases the lines of code. The table 4.2 shows that percentage increase in lines of code, using the proposed tool, on an average is 20 %.

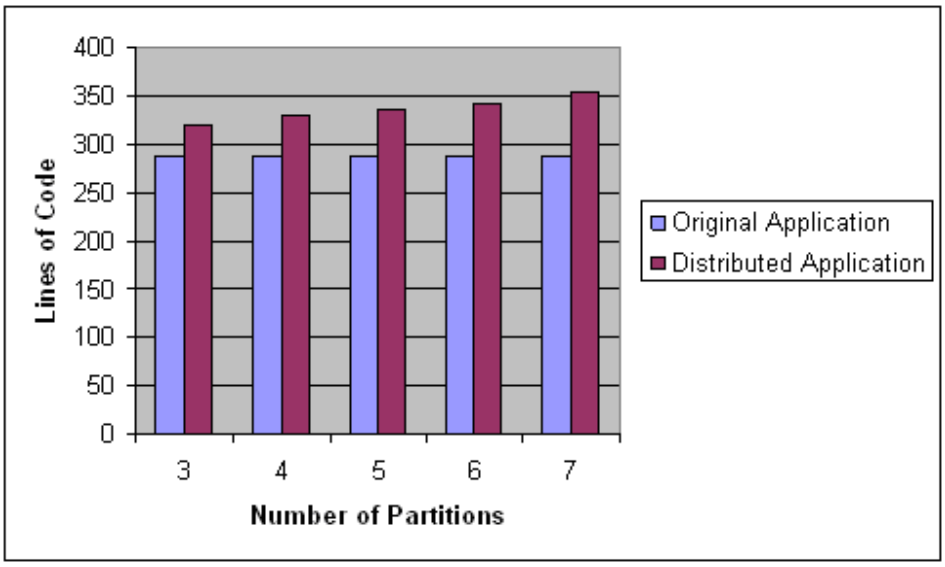

Figure 4.2. Increase in Lines of Code

Analysis of the generated code also shows that the overhead is proportional to number of partitions the application is split into. One of the sample programs is split into 44 tasks and partitioned into clusters ranging from three to seven. The bar graph 4.2 shows the increase in the lines of code compared to the original lines of code. We see that as the number of partitions increase in 4.3 , the percentage of increase in the lines of code, of the distributed application, is higher. This is due to the fact the communications between the partitions increases as a result of which the more number of PVM instructions for sending and receiving data have to be introduced.

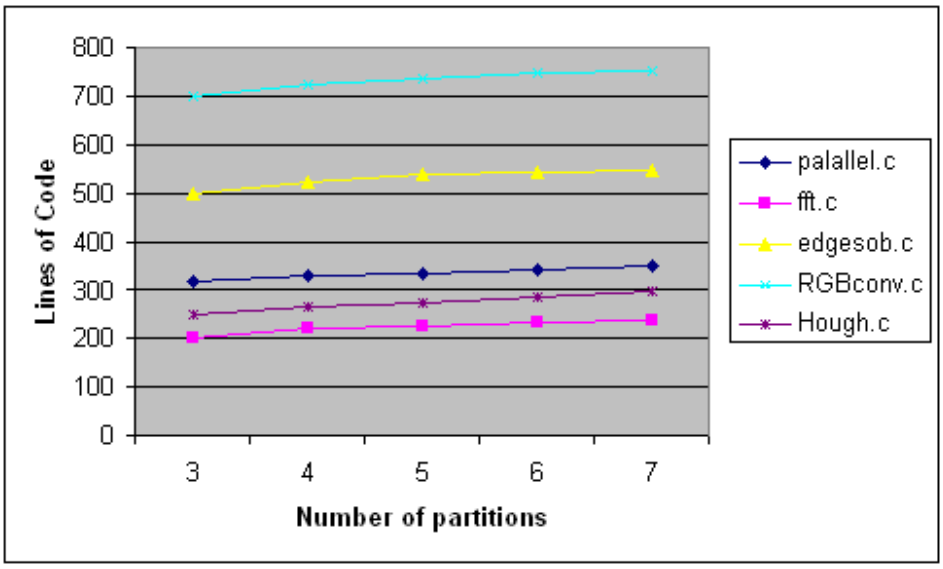

Figure 4.3. Effect Of Increasing Number Of Partitions On Number Of Lines Of Code

4.4 Execution Time

Any centralized application, when distributed will have a visible speedup in execution time. This is because the distribution speeds up the inherent parallelism in the application. However, this work does not concentrate on decreasing execution time by increasing parallelism. The main goal of the proposed method is to distribute procedural language applications to reduce network load and have efficient utilization of resources. The graph 4.4 shows that the execution time is reduced due to distribution, however, when the number

of partitions increase the execution time increases. The increase in data communication can be held responsible for the increase in execution time.

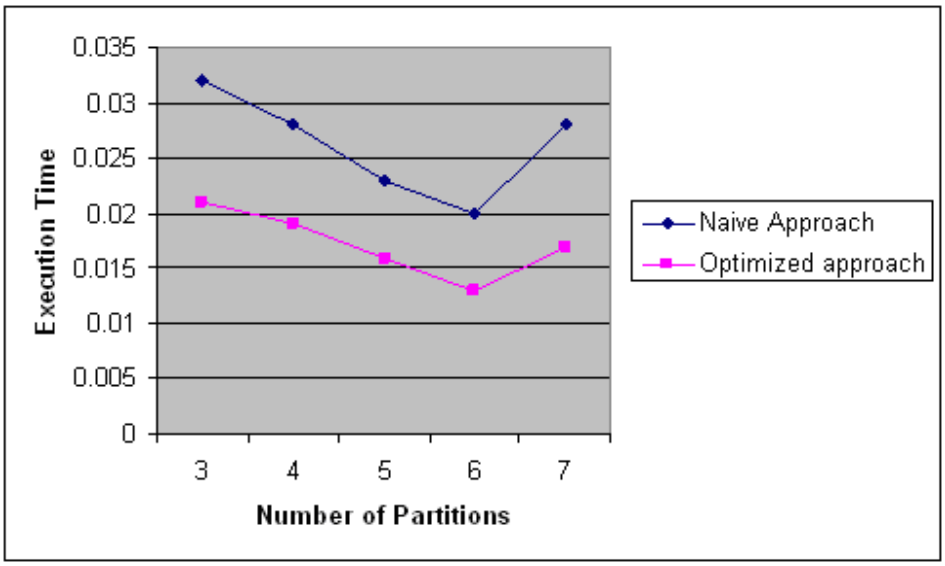

Figure 4.4. Execution Time Of Generated Code

4.5 Communication Cost

In the previous chapter we mention a nave method of code generation which increases the communication cost. This approach uses a conservative communication methodology that helps reduce the communication as seen in the graph 4.5.

For the two sample programs, the above graph shows that the communication is dependent on the number of clusters the application is partitioned into.

4.6 Summary

In this chapter we have discussed in detail the experimental procedure followed to validate the effectiveness of our Code Generation Tool. This is the first code generator tool catering to procedural language application distribution. We have successfully automated what would otherwise be a tedious manual process. The correctness and minimal overhead

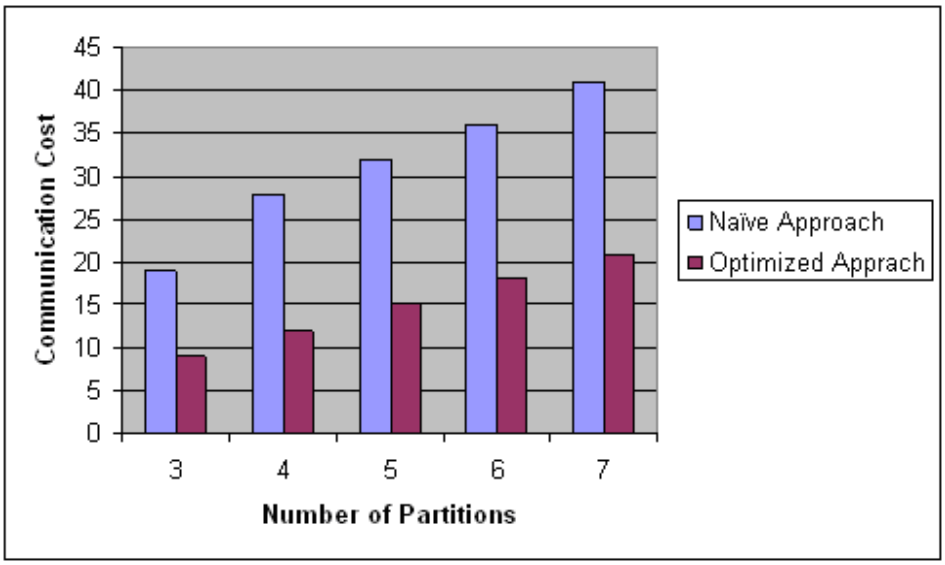

Figure 4.5. Cost Of Communication

make it a favorable solution. We have enumerated the causes of overhead. It was seen that the code generated has an average of 20 % increase in lines of code.

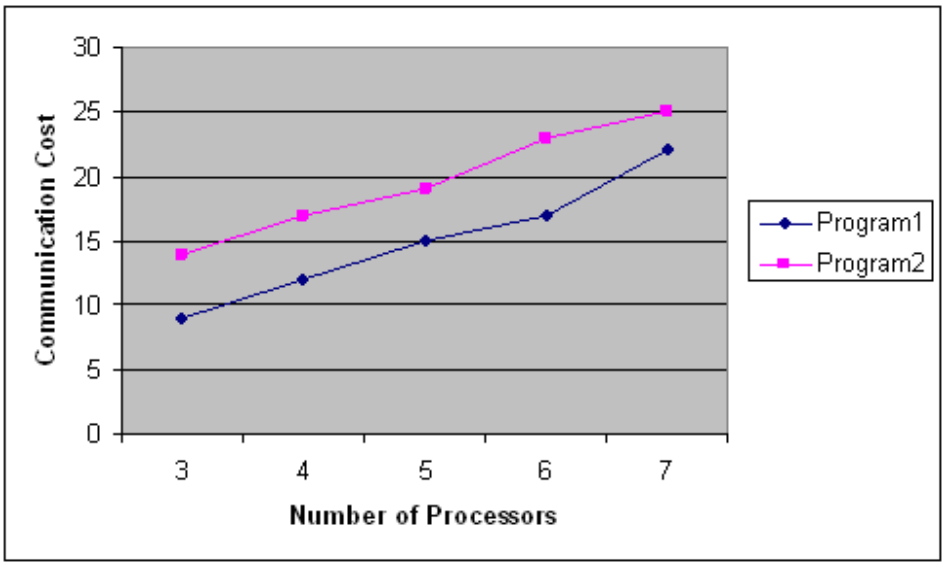

Figure 4.6. Effect Of Increasing Partitions On Communication

CHAPTER 5

CONCLUSION

We have presented a novel code generation module for procedural language application distribution problems. This is different from prior works in code generation, which have focused on partitioning of object oriented applications. Unlike other approaches where the original code is converted into a task graph and then the code is produced based on the scheduling algorithms, our approach for code generation deals with code generation by translation of the original source code.

In the experiments that we performed, we check for increase in lines of code introduced due to code duplication and addition of PVM constructs for communication and the overhead involved is found to be 22% at the maximum.

We also see that as the number of partitions increases, the communication time and hence the execution time increases.

Our Code Generator tool successfully generated code for the sample program partitions, mapped them to the appropriate processors and expertly manages the communication between the partitions.

In the future a partitioning methodology can be incorporated with a feedback loop from the code generator tool to obtain the best partitions.

REFERENCES

[1] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. Brayton, and A. S.-V. . Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES), Estes Park, Colorado, USA*, May 2002.

[2] Canal,R. and Parcerisa,J-M. and Gonzalez,A. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, 29:59–71, Feb 2001.

[3] Caracciolo and Pridmore. Reuse-oriented model year architectures for rapid prototyping.

[4] Carnegie Mellon University. *Message Passing using PVM Library.*

[5] D. Chandra, C. Fensch, W.-K. Hong, L. Wang, E. Yardimci, and M. Franz. Code generation at the proxy: An infrastructure-based approach to ubiquitous mobile code. In *Proceedings of the Fifth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS 2002), Malaga, Spain*, June 2002.

[6] G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

[7] L. Guthier, S. Yoo, and A. Jerraya. Automatic generation and targeting of application specific operating systems and embedded systems software. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 679–685. IEEE Press, 2001.

[8] J. Henkel. A Low Power Hardware/Software Partitioining for Core-based Embedded. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference*, pages 122–127, 1999.

[9] J. Henkel and Y. Li. Energy Conscious Hardware Software Partitioining of Embedded Systems: A case study on an MPEG-2 Encoder. In *Proceedings of the Sixth International workshop on Hardware/software codesign*, pages 23–27, Mar 1998.

[10] J. Kim and I. Lee. Modular code generation from hybrid automata based on data dependency. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, 2003.

[11] W. Li and C. McDonald. Full support for textual editing in a syntax-directed editor. Technical Report 94/8, Department of Computer Science, The University of Western Australia, 1994.

[12] N. Liogkas, B. MacIntyre, E. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning for prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 3(3), July 2004.

[13] M. Lopez-Vallejo and J. C. Lopez. On the hardware-software partitioning problem: System modeling and partitioning techniques. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):269–297, 2003.

[14] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors [Dagstuhl Workshop, August 31 - September 2, 1994]*. Kluwer, 1995.

[15] H. A. Muller, S. R. Tilley, and K. Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 217–226. IBM Press, 1993.

[16] A. Nacul, S. Choudhuri, and T. Givargis. Posix-compliant portable code synthesis for embedded systems. Technical report, Center for Embedded Computer systems, University of California, Irvine, November 2003.

[17] A. Nacul and T. Givargis. Code partitioning for synthesis of embedded applications with phantom. In *To be presented in Proceedings of the Internation Conference on computer aided design*, 2004.

[18] Z. P. Petru Eles and K. Kuchcinski. System level hardware/software partitioning based on simulated annealing and tabu search. *Kluwer Journal on Design Automation for Embedded Systems*, 2(1):5–32, Jan 1997.

[19] M. Raulet, M. Babel, O. Deforges, J. Nezan, and Y. Sorel. Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures. In *Proceedings of IEEE Workshop on Signal Processing Systems, SiPS'03*, Seoul, Korea, August 2003.

[20] Scientific Toolworks, Inc. *Understand C*.

[21] Stitt,G. and Vahid,F. Hardware/Software Partitioning of Software Binaries. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 164–170, Nov 2002.

[22] Welch,L.R. and Ravindran,B. and Henriques,J. and Hammer,D.K. Metrics and Techniques for Automatic Partitioning and Assignment of Object-based Concurrent Programs. In *Proceedings of The Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 440–447, Oct 1995.

[23] Wu,D. and Al-Hashimi,B.M and Eles,P. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In *IEE Proceedings-Computers and Digital Techniques*, volume 150, pages 262–273, Sept 2003.

[24] T. Yang and A. Gerasoulis. Pyrros: static task scheduling and code generation for message passing multiprocessors. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 428–437. ACM Press, 1992.

[25] H. Z. Zebrowitz. *GEDAETM - A Graphical Programming and Autocode Generation Tool for Signal Processor Applications.* Lockheed Martin Corporation.