1992

An analytical analysis for modeling accurate interprocess communication costs

Andrew William List
Lehigh University

AUTHOR:

List, Andrew W.

TITLE:

An Analytical Analysis for Modeling Accurate Inter-process communication costs.

DATE: January 17, 1993

An Analytical Analysis for Modeling Accurate Interprocess Communication Costs

by

Andrew William List

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Department of Electrical Engineering and Computer Science

Lehigh University

Bethlehem, Pennsylvania 18015

December 11, 1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

12/8/92
Date

Thesis Advisor

Chairperson of Department

Table of Contents

List of Tables v

List of Figures v

Abstract 1

1. Introduction 2
    1.1. Organization 2
    1.2. Parallelization 3
    1.3. Interprocess Communication Costs 4
    1.4. Future Research 7

2. Extracting Parallelism Information 8
    2.1. Approaches to Software Parallelizers 10
    2.2. Parallel Information Sources 12
    2.3. ANSI C Data Collection 14
        2.3.1. ANSI C Data Types 15
        2.3.2. ANSI C Global Variables 18
        2.3.3. ANSI C Functions 19
        2.3.4. Uncorrelated Data References Table . . 21

3. Control Flow Analysis and Data Dependency 23
    3.1. Loop Parallelism 26
    3.2. Functional Parallelism 27
    3.3. ANSI C Parallelizer 28

4. Literature Survey Of IPC Cost Functions 31
    4.1. Load Imbalance Plus Communication Time . . . 31
    4.2. Execution Time Plus Communication Cost . . . 34
    4.3. Link Delays With Traffic Density 36
    4.4. Forgotten Communication Costs 37
    4.5. Real Effect of Load Imbalance 38
    4.6. Architecture and Configuration Specifics . . 39
    4.7. Broadcast Communication Costs 41

5. General IPC Cost Function 42
    5.1. Guidelines and Assumptions 43
    5.2. Terms and Definitions 43
    5.3. A General IPC Cost Formula 51

6. Analysis of Topologies 54
    6.1. Parallel Links and Message Paths 55
    6.2. Bus Topologies 58
        6.2.1. Ethernet Networks 59
        6.2.2. Single and Multiple Parallel Bus . . . 62
    6.3. Static Network Topologies 66
        6.3.1. Store and Forward 66

6.3.2. Cut Through 68
6.4. Dynamic Network Topologies 70
6.4.1. Crossbar Networks 71
6.4.2. Circuit Switched Networks 72

7. Conclusions and Future Work 74

List Of References 76

Vita 79

List of Tables

Table I - Definition of Terms 51

List of Figures

Figure 1 - Data Dependencies 26

Figure 2 - Program Dependency Graphs 28

Figure 3 - An Undirected Program Graph 32

Abstract

Advances in parallel computing hardware have grown much faster than their software counterparts. With the cost of parallel computing solutions dropping, there exists a need for parallel software development tools and automatic parallelizers for converting the large installed base of sequential programs currently in use. A large portion of the parallelization problem resides in the efficient allocation of program tasks to the available hardware resources, minimizing the communication costs. This thesis will review the basic steps necessary for extracting parallelism from sequential programs and will detail other information sources that may be useful in making paralellization decisions. After presenting an overview of some interprocess communication cost (IPC) functions as shown in the literature, a general IPC cost function, that accurately models the communication costs, will be derived. This cost function will then be analyzed for several popular interconnection models including Ethernet, bus, packet and circuit switched networks.

"When we mean to build,
We first survey the plot, then draw the model;
And when we see the figure of the house,
Then we must rate the cost of the erection;
Which if we find outweighs ability,
What do we then but draw anew the model"
-Shakespeare

Chapter 1. Introduction

As parallel computers and parallel processing develop over the next few years, a large emphasis will be placed on understanding the characteristics of parallel systems and algorithms in an effort to speed up computations. Several research groups and commercial vendors currently produce hardware and software systems for special purpose parallel computing, yet the need exists to expand to more general capabilities.

1.1. Organization

A substantial area of research focuses on the problems associated with the relationships between sequential programs and their parallel counterparts. One goal of this research is to understand how to partition an existing general purpose sequential program into pieces that can subsequently be run in parallel on one or more computers[1,2,3,4,5,6,7,8]. This thesis will focus on the issues of extracting parallel information from existing sequential code written in ANSI C, partitioning of the code into parallel tasks, and analysis of the communication costs associated with distribution of the tasks to the remote machines. It will not directly address the problems of program task allocation to system resources or load balancing.

1.2. Parallelization

As hardware becomes more versatile and standardized, software development for parallel systems will become a very vital segment of the software industry. Development of large software systems is difficult on sequential computers and becomes exponentially complex in parallel environments. Language and debugging tools need to be developed to support new parallel features of advancing hardware technology and to efficiently parallelize existing applications software to run on the new platforms. While some programs and algorithms, such as image processing and VLSI chip routing, are natural candidates for parallelization due to computation or data organization[9,10,11,12,13,14,15], most programs outwardly do not exhibit such desirable characteristics. It is therefore necessary to develop automated software tools that can analyze existing source code and extract meaningful parallelism.

One popular method used in parallelizing or distributing applications is the remote procedure call (RPC)[4,5,6,27]. To the programmer, an RPC looks like a standard subroutine call, while the actual code for the procedure is executed on a remote machine. RPC's may either block until the results are received from the remote processor, or continue processing until the results of the procedure call are required. A program that is capable of extracting parallelism from ANSI C source code and automatically distributing various procedures of the program via non-blocking RPC will be referred to in this thesis.

Since C is not inherently a parallel language, there is a significant amount of work involved in searching for parallelism in a user's program. Chapters 2 and 3 of this thesis will give an overview of a software tool currently being implemented to perform the task of automatically partitioning sequential user programs into parallel segments, devising a distribution strategy based on the computing environment and user restrictions, and generating the necessary source code level output files for compilation and execution of the original program in parallel. The tool is being designed to work with many types of parallel multicomputer and distributed system platforms.

1.3. Interprocess Communication Costs

When looking at the possibilities of segmenting a program or group of processes among a fixed number of host processors or nodes, the issue of communication time is usually the most important factor, assuming that all of the processors are likely candidates able to process any of the jobs. To minimize the communication overhead and maximize the system throughput, processes that communicate frequently should be grouped together at the same processor, or as close as possible to one another. There are two approaches to allocating tasks in an evenly distributed fashion, static and dynamic. In the static allocation algorithms, once a task has been placed on a specific processor, it remains there until the process completes. On the other hand, the dynamic algorithms allow processes to migrate from one processor to another as necessary to try to keep the work load evenly balanced.

Several papers have been written on various approaches to the problem of static task allocation, with the predominant study focusing on how to most efficiently map a program graph to the graph of a distributed system[16,17,18,19,20, 21,22,23]. Similarly, research has been performed in the area of dynamic load balancing of processes once static allocation has been performed and run time information about the processes becomes available, but this will not be presented herein.

The central theme to both the task allocation and load balancing issues is to establish an algorithm that provides a good solution to a problem that has been proven NP complete [20,21,22,23]. This means that approximate or heuristic algorithms must be used in finding a near optimal solution to the problem. At the heart of the algorithm, there must be a cost function for estimating communication between processes and processors, such that the distribution achieved is optimized and system throughput maximized. While a few papers provide some insight into the cost function used for a specific algorithm [16,17,18], others use an approximation model which leaves out many parameters that could add significantly to the overall communication time. Most of the literature also makes the assumption that the distributed system is composed of a network of homogeneous processors in one standard configuration. However, in order to keep costs under control, interconnection of various non-homogeneous computers with specialized resources will increase in the future as workstations, multicomputers, and parallel computers will be networked to form very powerful computing environments.

Chapter 4 of this paper will present a survey of network cost functions seen in some of the literature. I will present each of the cost functions with a discussion of the advantages and disadvantages of each, as applied to general interprocess communication costs. Chapter 5 of this paper will derive a general form of a communication cost function for interprocess communication, taking into account many of the real world cost factors. Chapter 6 then uses the derived cost function to analyze several popular network configurations including Ethernet, packet and circuit switched networks, multiple bus, and packet switched networks with cut-through capability.

1.4. Future Research

As processors become increasingly smaller and faster, the communication cost will become a greater and greater portion of the computational overhead of parallel programs. Faster physical networks may reduce the transit time of a message, but still not affect significantly the communication overhead incurred. Accordingly, one must study the communication cost carefully if a truly near optimal, long term solution to the allocation and load balancing problems are to be found. Chapter 6 presents some conclusions of the study and proposes areas of further research.

Chapter 2. Extracting Parallelism Information

With the development of parallel computers and distributed systems, new frontiers of parallel programming and parallel operating systems have been unveiled. The complexity of these parallel environments create new problems not seen in sequential computers and demand innovative solutions to a wide range of programming and operating system problems. One of the largest problems of writing software for parallel architectures is the lack of language tools and programming guidelines which are well established for sequential development environments. Although specific applications have been developed for certain architectures, general purpose software development tools need to be implemented.

When a new type of computer enters the market, it generally takes quite some time for language and software development tools to be created for the platform. Programmers need these tools for rewriting and porting their applications to the new environment. Although most of the computers making up distributed computing systems are not new, to utilize the full capabilities of parallel program execution that networking offers, requires these new tools. Since there is a large installed base of software applications currently available to run on the individual computers of the distributed system, software developers could benefit greatly from an automated tool that converts the source code from a sequential format into a parallel equivalent.

When developing parallel applications or converting sequential programs into parallel versions, the question of compatibility arises. A program is usually written for one specific type of machine with an equally specific operating system. Dedicated parallel computers with an array of homogeneous processors represent the parallel extension of the serial computer. However, this is not necessarily true for distributed systems which may be made up of different computers each running their own special operating systems. And so, one has to first decide if the parallel application will be run on homogeneous or heterogeneous platforms.

Another problem arises if heterogeneous systems are chosen for parallel program execution rather than homogeneous ones. As different computers use different internal representations for data structures, any program that is to be run in parallel on heterogeneous systems must be able to handle the problems of data format translation when communicating between various modules on different hosts. This means that translation routines for all data types used in communicating between program modules must be imbedded into the original source code. Although this adds some additional processing overhead and is not necessary when using a homogeneous system, it allows the user to take full advantage of the computing resources available.

2.1. Approaches to Software Parallelizers

There are a couple of software vendors that currently have software parallelizers under development with the philosophy that executable parallel programs should be generated from sequential source code with no intermediate programmer intervention. To perform this translation, the source code, FORTRAN, C, or another language is converted into a generic representation, which is subsequently used to parallelize the program into object modules. These modules are in turn sent to a specialized compiler to create executable code for the designated system. While this approach is not language specific and requires the least knowledge of parallel programming, the programmer has no control over the parallel code generated or optimization provided.

The approach presented in this thesis is that the programmer should be able to optimize and interact with the parallel program generation at all levels of the process. This requires access to the original source code and the parallel equivalent source code that is generated by the parallelization process. And while the programmer is not required to work directly with the parallel code, it allows him to develop familiarity with extracted parallel structures from within the sequential program and to optimize the execution paths. One advantage of this philosophy is the output of standard source code files for the parallel program. This allows a generic compiler to generate machine specific implementations of parallel structures independently of the parallelization software tool. Although this approach is more versatile and flexible, one drawback is that each language requires its own converter to maintain consistent source code views.

Included with a general discussion of the above approach, which is applicable to any language, are references to a software tool, currently under development at Paralogic, Inc., capable of automatic parallelization of ANSI C at the source code level. Tasks performed by the tool include extracting parallelization information from the user's application code, providing a partitioning algorithm that breaks the original code into tasks and statically assigns them to available resources in a balanced fashion, and generating output files containing source code for the parallel version of the program. This chapter will describe what information must be gathered, from the raw source code as well as other sources, to make the best decisions about parallelizing the original program.

Since the output of the parallelization tool is ANSI C source code, no assumptions about the system on which the program will be run are made. It is left to the compiler and the libraries used to finalize the implementation specifics. The parallelizer attempts to find parallelism in the user's code at the subroutine level and inserts asynchronous remote procedure call and semaphore structures to partition the work load among all available processors in the distributed system.

2.2. Parallel Information Sources

To best parallelize a sequential program, the more information available concerning program behavior and the environment in which it will be executed, the better it may be parallelized. Parallelization of the internals of a program requires collection of information in three major categories, data types, global variables, and function definitions. Within function definitions, local variables, variable references, and function references must be collected for data flow analysis and data dependency analysis. Depending on the type of parallelism sought, expressions regarding looping and pointer manipulation may also have to be collected. Most of this information can be extracted from the code directly by the use of a parser or front end to a compiler. Some of the most difficult items about which to collect information are indirect variable references in the code. To properly account for these items, run time dynamic variable tracking, in addition to the compiler support, may be necessary to properly handle manipulations of indirect variables[1].

To aid in expected execution time and frequency of function calls, information from code profiling of the user's application is necessary. A profiler is a program that monitors a process while it is executing, recording each function call and execution time in every subroutine. Before the parallelization is performed, profiling occurs on the original sequential source code while executing on one machine. Two major determining factors that of accurate profile information are processor speed and input data.

Since the expected execution time recorded by the profiler is machine dependent, it is important that the information is collected on a machine representative of the type on which the parallel version will be executed. On programs whose execution time depends on the size of the input data, the size of the input used when profiling should be representative of the expected size to be used when the program is parallelized. More accurate statistics may be generated by averaging multiple profile runs on various input data sizes and machines. I will refer to these quantities as the average expected execution time and average data input size.

Another piece of information that is useful in making decisions regarding parallelization is a set of statistics on average network traffic and bandwidth utilization in the distributed system on which the parallel program will be executed. This information can be collected by monitoring the distributed system interconnection network traffic levels and averaging the data over the period of time when the parallel application will most likely be running. This information can be used by the task allocation algorithm when determining the communication cost of which subroutines are the best to distribute.

2.3. ANSI C Data Collection

For the ANSI C parallelizer implementation under development, all code processed is assumed to be compilable by an ANSI C compiler. For simplicity, the parallelizer makes assumptions about structures in the input source code that a compiler would not allow, so errors may be masked if the source code is not compilable. All code presented to the parallelizer also must be passed through an ANSI C preprocessor so as to include all relevant header files and conditional compilation information. If source code is available for system libraries or parallelized libraries exist, some system calls may also be able to be parallelized.

An ANSI C program is made up of data type definitions, internal and external global variable declarations, function prototypes, and functions definitions themselves. Functions may not be nested inside one another, but the scope, or current level in the program, must be accurately maintained for proper variable, data type, and function name differentiation. The scope of a program must include module file names, levels of recursion within data type definitions, and level within functions to uniquely track names used in the source code. Unique names in all contexts must be distinguishable because ANSI C allows the programmer to use the same name for different uses within the same program.

Functions may contain their own data type definitions, local variables, and a list of arguments passed to the function when it is called. Within a function, references to arguments, global variables, local variables, enumeration constants, and other functions may be found. As previously stated, information regarding data types, global variables, function definitions, local variables, and variable and function references must be collected for data flow and data dependency analysis. All data is stored in tables, created in sorted order by the name field of the respective structure. By storing the tables in sorted order, subsequent searches can utilize binary search techniques for quick access.

Most of the necessary information can be collected through the use of a parser able to scan ANSI C source code and extract the relevant items. Because ANSI C is almost entirely unambiguous, it is well suited for use with automatic parser generation tools such as YACC and LEX, two UNIX utilities. (A complete grammar for ANSI C may be found in the appendix of Kernighan and Ritchie's book on ANSI C, along with information regarding the scope of names and their duration[24]). In the following sections, each of the various types of information and its significance is briefly discussed.

2.3.1. ANSI C Data Types

All variables, arguments, and function return values in a C program must be of some defined data type, each stored so that format and size information associated with variables and constants can be determined when distributing specific procedures. As ANSI C contains only five basic data types, **int**eger, **char**acter, **float**ing point, **double** length floating point, and **void**; complex data types may be created by combining arbitrary groups of the basic data types into structures or unions. Enumeration data types allow the definition of a set of constants to be used within the program for variables that only take on fixed values. Using the **typedef** keyword, ANSI C allows the programmer to create synonym names for data types, which helps to simplify naming conventions.

Every type definition in the user program will be stored in a dynamically allocated data types table. The data types table can have a subtable with data type subfields to represent **struct**ures, **unions**, and **enum**erations, or may point to another type for which the entry is a synonym. The first entries in the array will be the fundamental types of ANSI C from which all **struct**ures, **unions**, **enum**erations and **typedef**s will be created. Four other keywords, **short**, **long**, **signed**, and **unsigned** can be used in defining types. These may either be a modifier of a base type, such as **char**acter, or if a variable's type is defined only with a modifier, the base type is assumed to be **int**eger. Each variable can have up to two modifiers defined along with the base type, **short unsigned char**, for example.

Each variable or pointer may have a qualifier of **const** or **volatile** associated with it. Although acceptable, but not meaningful, both **const** and **volatile** can be present and will be recorded. Since one qualifier is associated with the base type and all others refer to pointers to the base type, storage must be allotted for **const** and another for **volatile** allowing the user to specify levels of indirection on a per variable basis. Each variable may also have a storage class specifier defined in its declaration. Although the compiler enforces when specifiers may be used, exactly one specifier is allowed for each variable. The legal storage class specifiers are **auto**, **extern**, **register**, **static**, and **typedef**.

All variables defined in the user program, whether local, global, or procedure arguments, must reference a defined data type in the data types table. All synonyms for data types must reference another data type in the data types table and the chain must be anchored by a **struct**ure, **union**, **enum**eration, or base type. Each entry in the data types table contains a pointer which can point to either another data type, to a dynamic array of pointers to field structures, to a dynamic array of pointers to enumeration structures, or to NULL, which indicates that this entry is a base type definition.

Data types can be explicitly defined by use of a typedef statement, or implicitly defined by declaring the data type as part of a global or local variable declaration, function return type, or even as the data type for an argument passed to a function. While this is not good programming practice, it is legal in ANSI C. When collecting the data type definitions from the user's source code, it is important to maintain consistency, therefore, a type or variable name should only have one unique corresponding entry in the appropriate table. While data types may not be redefined within one file unless the definitions are identical, which is enforced by the compiler, multiple definitions may be present pre-processed files.

structures and **union**s may be processed in the same fashion because they have the same definition rules, although their storage size rules do differ. The declarations of **struct**ures and **union**s may contain recursive declarations, where the **struct**ure or **union** contains subfields that are pointers to structures of the type being defined.

2.3.2. ANSI C Global Variables

Each global variable definition in the user program will be stored in a dynamically allocated table, which does not require a subtable. The recording of global variables not only involves storage of the global variable information, but may also involve the definition of data types, and the processing of function prototypes. When a global variable definition is encountered which also defines a **struct**ure, **union**, or **enum**eration data type, the type is processed before the global variable information is collected. For definitions that do not define new data types, the type given must already exist and have been processed.

The collection of global variable information is relatively straightforward compared with data types and functions. Individual definitions of global variables contain all information necessary including base type, specifier, and qualifier, but multiple global variables may be included in one definition. Since variables may be defined in lists referring to the same base type, all members in the list of globals must be entered into the table with references to the base type defined. Likewise, base type qualifier and specifier information must also be stored with each variable defined in the list.

2.3.3. ANSI C Functions

All functions prototyped or defined in the user's code will be stored in another dynamically allocated table. Each entry in the subroutine table contains pointers to three other dynamically allocated subtables per entry, one for the argument list, one for the local variables, and one for the variable and function references. The latter's is the most complicated of all of the tables and will be the largest and most used table used in the analysis of routines which may be distributed. As with global variables, data types may be defined in the function definitions in either the return data type or parameter list positions.

The difference between processing a prototype and processing a function definition is that a function definition requires changing the current scope. ANSI C allows both the older standard C function prototypes and function argument list definitions, as well as the new versions which allow more rigorous type checking. This means that the parser and collection routines need to be able to distinguish between and handle each appropriately. Old style arguments are listed simply by the name of the parameter; the names are assigned types in a declaration section prior to the start of the function. New style arguments specify the base type and qualifiers for each parameter within the definition of the argument list.

The argument subtable consists of each argument passed to the subroutine as defined in the procedure declaration. The local variable subtable contains all variables locally defined within the procedure, and the variable reference subtable contains an entry for each variable, local or global, and each other function that is referenced or altered in some fashion by the procedure. A reference to a variable or function is defined as a set of operations including reading, writing, allocating, deallocating, and calling. These references may be direct via assignment operators, or indirect through the use of pointers.

To completely analyze all indirect references, post parsing analysis is necessary to correctly identify variable function arguments and indirectly altered memory items. Since the parser cannot fill in the value of variables to determine all memory usage, especially in programs that use dynamically allocated memory structures, some form of dynamic variable tracking is necessary while the parallel program is executing. Dynamic variable tracking entails keeping track of common memory segments and references to them by any of the processors, local or remote, that are part of the program. While this is an important part of maintaining a consistent view of the memory used by a distributed application, a thorough discussion of the details is beyond the scope of this thesis.

2.3.4. Uncorrelated Data References Table

Yet another table, to store uncorrelated references, is needed for the information gathering phase of parallelization. This table will hold the names of all variable or function references for which there has been no definition. There are two cases under which this could occur. A function could be defined later in a source file and not have a function prototype, in which case it will end up in the uncorrelated data reference table temporarily until it is defined. Or, if a reference exists for which no definition is seen, in which case the object referenced must be in another compiled object module and the source code is not available. Consequently, any routine that has not been defined completely cannot be distributed because there is no way of knowing what exactly the routine does to memory locations.

Once the data described above has been collected, a complete analysis must be performed to determine distribution eligibility for portions of the code. By using the collected information to examining data and control interdependencies, the program may be successfully divided into tasks which may be scheduled and executed in parallel which is described in the following chapter.

Chapter 3. Control Flow Analysis and Data Dependency

The second step in parallelizing a sequential program, after extracting relevant parallelization information, is to analyze the control flow and data dependencies inherent in the code so as to decide which structures may be parallelized. There are two levels of parallelism that can be defined, fine grain and coarse grain. Fine grain parallelism occurs at the statement level of a program and is suitable for certain types of parallel computers. Coarse grain parallelism is usually found at a procedural or functional level in a program. For the analysis presented in this thesis, it is assumed that the program will be analyzed to extract large grain parallelism.

There are two basic approaches to identifying parallelism that are commonly used, data or loop parallelism, and functional or task level parallelism. Depending on the function of the program to be parallelized and the type of parallel system available, one approach may be better suited than the other. The loop or data approach looks for fine grain parallelism, while the functional or task approach searches for large grain parallelism.

While tightly coupled parallel computers, especially those with shared memory, can take advantage of coarse and fine grained parallelism, loosely coupled distributed systems are better suited for coarse or large grain parallelism. Generally, in distributed systems, fine grain parallelism is not worth the overhead incurred to distribute the code segment, due to data conversion time, communication time, and operating system overhead. An ideal code parallelizer would attempt to combine the approaches in order to maximize the amount of parallelism obtained and to tailor the code generated to the available system resources.

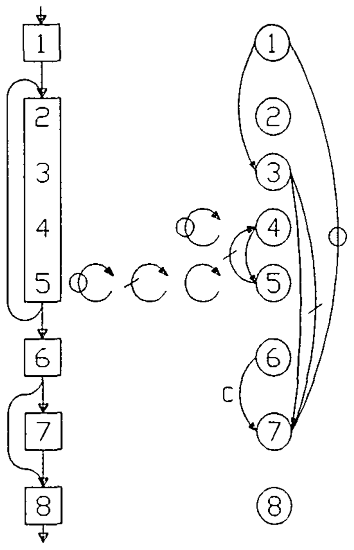
To perform the analysis of information collected, the program control structures and data dependencies are usually depicted in a graphical arrangement of either directed or undirected graphs. Using graph theory to analyze these graphs, interdependence between program sections and cyclical structures may be found and decisions can be made on parallelization of routines. While information in a directed graph can be reduced into an undirected graph, information in an undirected graph cannot be used to construct a directed graph. Even though either directed or undirected graphs may be used, I will only discuss the case of directed graphs in this thesis, for they will offer the most useful presentation of the interprocess communication cost function.

Both the loop and functional approaches to the parallelization problem require some sort of analysis of the program control, which is usually depicted in a control flow graph (CFG). This graph can be drawn as a directed graph with nodes representing basic blocks of a program and edges representing the control paths. A basic block is defined as a piece of code from which control may enter only at the top and exit only at the bottom. Once execution of the block is started, all instructions contained within the block will be executed.

Data dependency analysis is usually represented by a data dependency graph (DDG), which may be drawn as a directed graph with nodes representing the program's basic blocks and the edges symbolizing dependencies between variables used by the basic blocks. A DDG can be drawn for the internal behavior of each basic block for fine grain parallelism and a DDG may also be drawn between basic blocks or modules of a program for larger grain parallelism.

There are four general types of program dependencies, flow dependency, anti-dependency, output dependency, and control dependency, that may be analyzed for each statement in a program[4]. A statement S_j is flow dependent on S_i if S_i assigns a value to a variable which is subsequently used in S_j. S_j is anti-dependent on S_i if the value of the variable used in S_i is recomputed later by S_j. S_j is output dependent on S_i if both statements define the same variable but S_j writes to the location of the variable after S_i. Finally, S_j is control dependent on S_i if the execution of S_j depends on the path taken due to the evaluation of S_i.

Figure 1 shows a segment of code written in C which could be contained within a function. The numbers corresponding to the lines of code are shown in two graphs to the left, one showing the basic blocks, and the other depicting the dependencies as described above. Although this example is shown at the statement level, it could easily be expanded to the larger scope of modules. The dependency edges are shown as: flow - normal edge, anti - edge with a slash through it, output - edge with a circle on it, and control - edge with the letter C next to it. This example shows only the types of dependencies; it does not attempt to explain how such a code fragment could be parallelized.



```
1 scale = alpha * beta;
2 for (i=start; i<finish; i++)
3 {  dimension[i] = scale * items[i];
4    epsilon = items[i] * delta;
5    error += epsilon; }
6 if (done)
7    scale = dimension[0];
8 return;
```

Figure 1 - Data Dependencies

3.1. Loop Parallelism

Data parallelism tries to distribute loops and data structures across several machines, that each perform the same type of operations. Data parallelism is the most researched of the two methodologies and examples can be found in languages, compilers, and architecture designs. The concept of data parallelism examines loop structures which perform computations on data to determine if any dependency between data items exists. If no dependency is present or can be distributed in such a way as to maintain the data dependencies, the computations may be done in parallel. A direct application of data dependency analysis to parallelizing compilers is presented in a paper by Li, Yew and Zhu[2].

3.2. Functional Parallelism

Functional or task level parallelism uses interprocedural analysis to identify tasks or functional modules that may be distributed in parallel. Girkar and Polychronopoulos[3] present an intermediate parallel program representation that attempts to encapsulate data and control dependencies into modules representing functional tasks. With proper minimization of dependencies, these tasks can be distributed in parallel with minimal synchronization overhead. Since the functional parallelism approach offers potentially large grain parallelism with minimal module interdependency, it is well suited for parallelization of programs for use in distributed systems.

Figure 2A shows an example of a hierarchical program graph, Figure 2B a CFG, and Figure 2C a DDG that it could map to in terms of directed graphs. The edge weights in Figure 2B correspond to the calling frequencies of the subroutines for a specific input data stream. As will be shown in the following two chapters, this information is particularly important in accurately defining the interprocess communication costs between program modules. Minimization of a parallel program's execution time requires that program modules be placed so as to minimize the IPC costs. Thus, the static task allocation algorithm needs accurate calling frequency information for average input data sizes in order to properly calculate IPC costs and place modules at appropriate processors in the system.

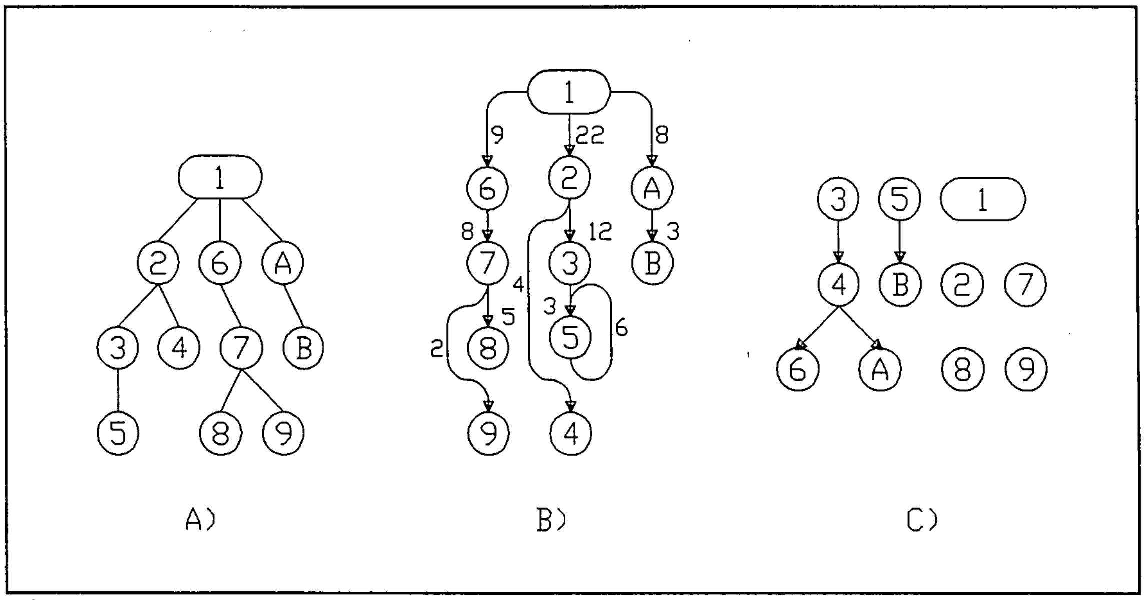


Figure 2 - Program Dependency Graphs

3.3. ANSI C Parallelizer

For the implementation of the ANSI C parallelizer, the approach taken is similar to the functional partitioning described in the previous section. By analyzing the code at the subroutine level, it can be segmented naturally into blocks. Similar to basic blocks, though they do not follow the strict definition, subroutines usually only have one entry point and one exit point. Since most programs can be drawn as a hierarchically arranged graph, subroutines generally represent the functional tasks performed at each stage in the program's execution.

Using the terminology of Bustard, Elder, and Welsh[5], a dominant module provides motive force for a program while a subordinate module provides a service for dominant components. Subordinate modules may be shared between more than one dominant module. It is by applying these properties to the subroutines of a program, groupings of subordinate modules and the dominant modules they service can be made. Subordinate routines that are shared may be duplicated in different blocks to help form larger modules, which I will call tasks. By analyzing the data and control dependencies between these module groups, one can determine which of the tasks may be distributed in parallel.

Constraints on partitioning or placement of tasks may not only be a function of dependencies of control or data flow but of resources. If a particular program uses resources that are not available on all processors in the system, it will have to be placed on a machine that has the capabilities required. If a significant number of tasks in a program exhibit this type of behavior, parallelization of a program may yield little or no benefit.

No specific algorithms are discussed in this thesis for the actual partitioning of a program into these task groupings. The remainder of the thesis will assume that the program has already been segmented into tasks and will explore the importance of task allocation with static load balancing among the processors available in the system. The most important aspect of both problems is the cost of interprocess communication.

Chapter 4. Literature Survey Of IPC Cost Functions

An excellent place to begin the evaluation of interprocess communication cost functions is in the recent technical literature. This survey does not investigate papers dated earlier than 1985, and while not inclusive, covers several interprocess communication cost functions. Each approach will be presented separately, detailing all of the terms in the equations, and the conditions under which it was designed to be used. Since the equations in all example cases were designed to be used in solving the problem of initial task assignment, some terms in the equations account for functions that are not of interest to this study. Following the presentation of the various IPC cost formulas, I will point out the drawbacks of each approach as it applies the general communication costs in a generic distributed environment. All of the approaches presented herein assume that the program to be distributed has been segmented into tasks or modules and that the communication patterns between the various modules are known.

4.1. Load Imbalance Plus Communication Time

Hwang and Xu[16] derive a cost function based on the sum of a load imbalance function and a communication cost function. To solve the partition problem, the authors use an undirected program graph to represent the tasks to be allocated to system processors with edge weights corresponding to communication patterns between program modules. An example of an undirected program graph can be seen in Figure 3. The circles represent program modules, with the numbers in the circles depicting the amount of code and data memory used (usually the numbers indicate the expected execution time). Since the graph is undirected, the edge weights coincide with the total number of messages that are passed between the two modules, regardless of the point of origination. The authors assume that the distributed system can be represented by a connected graph, where one bidirectional link exists between any pair of processors that are neighbors. They also assume that there is only one way that a message may be routed between any two processors.

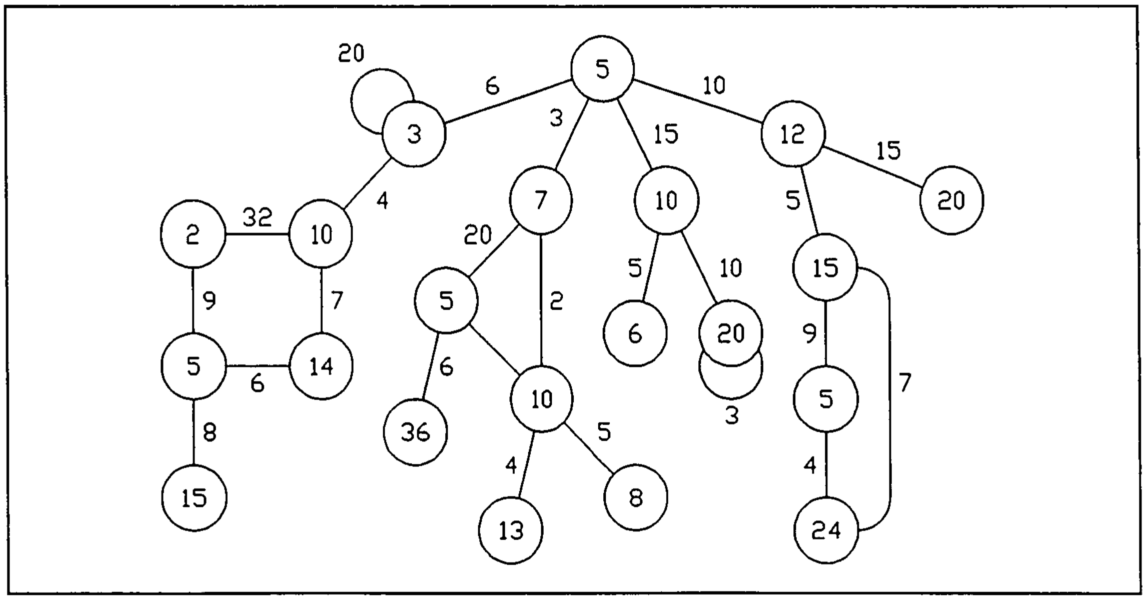


Figure 3 - An Undirected Program Graph

The imbalance function is calculated by summing the expected computational load on each of the local processors, based on a specific mapping of program tasks to the available processors. The total expected load is divided by the number of processors and an imbalance vector is generated by taking the absolute value of the local load minus the average load. This vector is then summed to form the imbalance function.

To compute the communication cost, the authors calculate node to node communication delays and store them in an N by N matrix, where N is the number of nodes in the system. Each cost entry, C_{jk}, in the matrix corresponds to the total cost of sending all messages from node j to node k and from node k to node j. Each C_{jk} cost entry is calculated by multiplying the distance between nodes j and k by the edge weights between all program modules residing on node j and all program modules residing on node k. The final communication cost of a particular distribution of program tasks is found by summing all entries in the cost matrix and dividing the sum by two. The division by two is required because the cost between node j and node k will be counted twice, once using the entry C_{jk} and again using the entry C_{kj}. As the authors present in their paper, the following equation represents their allocation cost function. In the equation, N equals the number of processors, L denotes the load, either local or average, and C is the entry in the communication cost matrix.

$$Cost=E_{Imb}+E_{Com}=\sum_{i=1}^{N}|L_{Loc}-L_{Avg}|+\frac{1}{2}\sum_{j=1}^{N}\sum_{k=1}^{N}C_{jk} \qquad (1)$$

The complexity of this algorithm may be reduced without changing the content of the equation or its behavior, by noting that cost entries C_{jk} where j = k, are always zero because the distance between nodes is zero. This case corresponds to both program modules under evaluation residing on the same processor. If the matrix is split along the j = k diagonal, it can also readily be seen that the lower half of the matrix is a reflection of the upper half of the matrix because the node indices are reversed, however, the communication cost based on undirected graph edge weights is the same. To make the evaluation of the equation simpler, only the upper triangular half of the cost function matrix needs to be computed and any entry in the matrix where index j = index k does not have to be calculated. Since the entries in the cost matrix would only be counted once, the equation could be changed as follows:

$$Cost = E_{Imb} + E_{Com} = \sum_{i=1}^{N} |L_{Loc} - L_{Avg}| + \sum_{j=1}^{N-1} \sum_{k=j+1}^{N} C_{jk} \qquad (2)$$

where j corresponds to the rows and k corresponds to the columns of the cost matrix.

4.2. Execution Time Plus Communication Cost

Ezzat, Bergeron, and Pokoski[17], take a similar approach to their cost function, which utilizes the expected execution time of the process plus a communication cost term. Although not directly stated in the paper, but implied in very confusing notation, the authors make some assumptions about the interconnection structure of the distributed system. They assume that the system consists of a group of processors connected by a communication structure, over which a message has only one way to be routed between any two nodes. It is assumed that there is a matrix, C_{ij}, which contains in each entry the number of messages passed from process i to process j. From this, I concluded that the graph representing the program to be distributed was a directed graph.

The cost function uses the assumption that communication cost is directly proportional to the distance, D_{kl}, that the message has to travel and that two processes executing on the same node have a distance of zero. The authors use the variable Q_{ikm} to represent the expected process execution time of process i on host processor k with resource m. They use W_m as a scaling factor, to account for the differences in measurement units, and introduce a function X_{ikm}, which is a 1 if process i is assigned to processor k at resource m, zero otherwise. The authors also introduce X_{ikc} to be the communication resource of processor k assigned to process i, which I will rename Y_{ikc}. The cost formula is then represented as follows:

$$Cost=\sum_{i}\sum_{k}\left(\sum_{m} W_m Q_{ikm} X_{ikm}+\sum_{l}\sum_{j}\left(C_{ij} D_{kl}\right) Y_{ikc} Y_{jlc}\right) \qquad (3)$$

where i and j represent processes residing on processor nodes k and l respectively. The sum over m represents the cost associated with the expected execution time of all modules utilizing resource m on processor k by process i. The interior double sum term represents the communication cost incurred between processor nodes k and l using communication resources at the respective nodes by processes i and j. The addition of these two terms summed over all processes on all nodes yields the total cost.

4.3. Link Delays With Traffic Density

Bollinger and Midkiff[18] take a different approach to the derivation of a cost function for total communication cost. Instead of computing a set of delays based on the number of nodes in the system, they base their calculations on the number of communication links in the system. They also use a directed graph as their model of the program tasks to be partitioned, not undirected. In their program graph representation, each edge weight corresponds to the number of messages that must be transmitted between a specific source process and a specific destination process. They assume that the processors in the system may be represented by a connected graph, disallowing isolated processors. Further, they assume that only one bidirectional communication link exists between any two processors (multiple paths may exist between two processors) and that messages must always be routed over a specific path from process j to process k and may not sent over different network connections. The authors also restrict the number of processes in the system to be exactly equal to the number of processors and only allow one process to be placed on each processor in the system. Thus, they avoid the issue of load balancing by only examining an evenly distributed case, assuming all processes have roughly the same execution time.

The authors calculate the communication cost between any two processes j and k using a traffic density term, W_{jk}, times the distance between processes j and k. The distance term must include the sum of the delays of all links used in establishing communication between processes j and k and the W_{jk} term is defined as the sum of all messages that must traverse this path. They then derive the delay, D_i, at each link i in the system in terms of all of the processes in the system that use the link. U_{st} defines a usage function that is true if the link is used and false otherwise. The total communication cost between processes j and k can thus be written as,

$$Cost_{jk}=\sum_{i=1}^{d_{jk}} D_i \quad Where\ D_i=\sum_{s,t} W_{st}\cdot U_{st}(L_i) \tag{4}$$

which adds the cost of each link used to send a message from process j to process k.

4.4. Forgotten Communication Costs

Unfortunately, the cost functions presented in all of the above cases do not yield an accurate analysis of the communication costs in all cases. The communication cost of two processes on the same processor node is assumed to be zero because the distance between the processes is zero, however, this is not true in practice. Even though two processes may reside on the same host processor, there is a communication cost associated with the transmission of a message from one to the other. In terms of the delay time, the internally passed message will be considerably faster than any external transaction, although there are several cases where the internal communication time may be significant.

Since processes that communicate heavily with one another should be grouped together, or as close as possible to minimize the communication cost, a large number of internal messages may add up to a significant delay. If the computational load on a processor is large or the number of processes assigned to one processor is high, then the operating system overhead to concurrently process all of the tasks may slow the response time of the communication resources. Another factor inherent to the internal communication time is the number of communication resources available and the number of processes on the node that wish to use them. The addition of another term in the equation to handle this would yield a better approximation.

4.5. Real Effect of Load Imbalance

Hwang and Xu[16] and Ezzat, Bergeron, and Pokoski[17] both use a term to represent processor load in their allocation equations. While the processor load is important to the even allocation of tasks and load balancing, the term usually plays a very small role in the communication costs. For this reason, the above mentioned cost equations separate this term from the communication cost term in their respective cost functions, but the inclusion of the term may yield inaccurate results for the intended cost function because of the differences between the behavior of the two terms.

A partitioning in which computationally intensive, yet low communication, segments of a program are placed on the same node may make the evaluation of the cost function disproportionately large. The distance between two nodes only counts the number of links used and does not include the incurred delays of the message transmission along the path taken between two nodes or the bandwidth of the communication links themselves. This may result in a choice to move the allocation of these processes to more evenly distribute the load, even though the communication delays may have been minimized. In fact, it would be difficult to minimize the communication delays and find a mapping that yielded a balanced processor load using either of the two cost functions.

4.6. Architecture and Configuration Specifics

None of the above cost functions utilize terms to represent the real cost of message transmission with respect to architecture or system configuration. The delay and link terms in the equations of Bollinger and Midkiff[18], D_i and L_i, are generic in terms of system architecture or configuration; the delay terms only account for total traffic density. While this yields an overall estimate of the communication time, it does not include a term which corresponds to the delay encountered while waiting on links in use. If one assumes that link busy delays are linearly related to the traffic density, then this term would not affect the use of this equation. If the relationship is not linear, then such a term should be in the equation because it may change the behavior significantly as link message densities increase.

The delay terms should also contain architecture and configuration specific timing as a function of message length for the cost function to be accurate. Each of the authors assume that a message is atomic and that message length is either equal in all cases or that it is inconsequential. In real systems, messages do not have to be of the same length and the transmission time is usually partially dependent on the amount of data sent. While larger messages are often packetized for maintaining data integrity and the above formulas could treat each packet as individual messages, this still does not account for varying packet sizes. In fact, multistage interconnection networks, bus architectures, and packet switched networks all require different terms, not shown in these equations, to accurately represent the communication costs (this includes cut-through routing used in point-to-point networks).

4.7. Broadcast Communication Costs

None of the articles specifically mention broadcast messages, only that messages are passed directly from one process to another. To account for broadcast type messages using the above cost formulas, individual messages would have to be sent from the originating process to all destinations, but this may not accurately model the cost. Most distributed systems contain one or more mechanisms for broadcast messages, which none of these formulas account for. This may be an important distinction in heterogeneous distributed systems where the processor nodes and networks can have different characteristics depending on the point of origination and routing algorithms used.

A broadcast message initiated by one process must filter through the system to be received at all processes. The organization of the system and the mapping of the task units to the processors in the system may greatly affect the time required to transmit a broadcast message. In fact, the paths taken in the broadcast message distribution can change considerably with different mappings. While this may be accounted for in the delays based on known communication patterns and routing algorithms, it becomes difficult to handle as the system size and complexity increases. Again, although this is a problem to consider in designing an appropriate cost function for a specific application, it is beyond the scope of this paper.

Chapter 5. General IPC Cost Function

In much of the technical literature pertaining to task allocation and load balancing, the primary focus is on the problem of partitioning a program graph onto a distributed system graph. Most of the presentation is done at a theoretical level without reference to a specific architecture or interconnection structure. While analysis of the problem in this arena is excellent for studying and understanding the nature of the problem, it does not necessarily lead from theory into implementation easily. When specific machines are mentioned, most researchers make the assumption that machine architecture is uniform across the processors and the interconnection network. While this often simplifies the analysis of the system and algorithms, it may only lead to specific solutions and implementations which are not be able to be generalized.

Due to the high cost of parallel processor arrays or multicomputers, the difficulty of parallel programming, and immense popularity and low cost of high speed interconnection networks available, many computer users are forming powerful distributed systems from existing computer resources. Large networks of heterogeneous computer systems will become more prevalent, and with them, efficient methods of estimating communication costs for program partitioning will become extremely important.

5.1. Guidelines and Assumptions

An interprocess communication cost formula should accurately define all parameters that affect the transmission of a message from one process to another. Likewise, the formula should be general enough to be able to describe a message transmission in one multiprocess computer or a complete heterogeneous distributed system. It should also contain terms specific enough to describe, with a high degree of accuracy, the real cost of traffic on any given link within any part of a complex distributed system. While all of the terms described in the following descriptions may not be available or practical to use in making allocation or load balancing decisions, they are essential to an accurate interprocess communication cost function.

For this derivation I will assume that the program to be distributed has been segmented into tasks, and that the communication patterns between the various tasks are known. I further assume that the distributed system can be represented by a connected graph, such that there are no isolated processors. I also assume that the program to be distributed across the processors is represented by a directed graph and that only one program is present in the system.

5.2. Terms and Definitions

Let the program graph contain P different program modules and P_n be the designation for process n in the system. Let the edge weights of the directed graph represent the number of messages that must be sent between P_j and P_k denoted as W_{jk}. Note that W_{jk} implies a logical connection between processes j and k and that this may be established by using zero or more physical links in the system.

I will derive a general cost formula for representation of communication overhead by defining the communication cost between pairs of processes, then summing these to obtain the overall communication cost of a program partitioning. For the derivation, I will assume only one bidirectional communication link between neighboring processors and that given two processes, j and k, there is only one possible path for routing messages from process j to process k. It is not assumed that the routing path from process k to process j is the same as the routing path from process j to process k. I also assume, for the derivation, that only one program will be partitioned across the distributed system at one time although the formula developed will work for multiple sets of processes. After the general formula has been derived, I will discuss the ramifications of relaxing this assumption and allowing multiple links between neighboring processors as well as parallel communication paths between processes.

The analysis of general communication cost function should originate from the point of view of two independent processes, residing on two separate host processors, connected by some communications network, one of which wants to send a message to the other. The case of two communicating processes on one host processor becomes a simplified case of the independent processor scenario. In a message sent from process P_j to process P_k, P_j issues a send command to its host operating system and expects that the message will be delivered to P_k on the remote system. The analysis that follows details the delays in performing this operation with the assumption that neither the sending nor the receiving processes participate in the transmission of the message in any way other than the send and receive primitive instructions to their respective local operating systems.

In any communications system, there are several major factors that incur delays in the transmission of a message. The delay associated with message transmission is largely a function of the hardware used as an interconnection mechanism and its electrical characteristics. The bandwidth of a link in the network determines the quantity of information that can be transmitted using that link in a specified period of time. Assuming that the system is not homogeneous, each link in the system could potentially have a different bandwidth. I will use L as the total number of links in the system and L_y to define link number y in the system network connecting a pair of processors.

The operating system of the sending processor has some required overhead in preparation of a message for transmission. Likewise, the receiving processor's operating system must incur some overhead, usually to buffer the message before presenting it to the receiving process. In between, the message must traverse a path through the network of the distributed system before arriving at the destination. The sending operating system is usually responsible for establishing the connections requested in addition to transmitting the message. I will refer to the operating system overhead time as $S_j + R_k$, where S_j is the local operating system setup overhead time prior to sending a message from process j, and R_k is the local operating system collection overhead time while receiving a message at process k.

If a message is being broken into packets, this requires more overhead at both the sending and receiving nodes to segment the message into packets and reassemble the message at the receiving end, but does not change the overhead term definitions; it only increases their magnitude. This overhead will only be incurred for the preparation and reception of the message as a whole, not on an individual packet basis. In the case of a packetized message, the transmission delay will include the sum of the network delays of all packets sent, plus the operating system overhead of the sending and receiving systems counted only once. Since it is assumed in this thesis that the message routing path is the same for all packets of a message, and that all packets of a message are sent immediately following the initial packet of a message, then the cost of sending a message from process j to process k is proportional to W_{jk}, and does not require the number of packets. The case of multiple routing paths is briefly mentioned in section 6.1 but is not explored in this thesis.

Regardless of whether packets are used, the communication time over a given communication link is directly proportional to the number of bytes of information being transmitted. I will define C_y to be the amount of time required to send one byte of information over link L_y of the distributed system and M_{ijk} to be the magnitude in bytes of the ith message sent from process j to process k. I will use X_{jk} to represent the sum, in bytes, of all data transmitted from process j to process k. I designate T_y to represent the total traffic density on link y of the system as the sum of all data, in bytes, transferred between any process pairs using link y. Note that the operating system, when sending a message, or packets of a message, usually includes extra bytes for error detection and message reassembly. These extra bytes must be included in the M_{ijk} magnitude values.

To properly account for the bandwidth of the individual links in the system, it is necessary to look at the amount of data transferred over what period of time. I will use E_y to denote the total expected execution time between the first and last message between any processes that use link y in the distributed system. Theoretically it is possible to figure out this term, although practically, it generally is not feasible. The ratio of T_y over E_y will then yield an approximation to the amount of bandwidth used on link y. The amount of bandwidth used may be directly related to the performance of the link under different load conditions.

To the sending and receiving operating system overhead, another term must be added to include the cost of acquiring a communication link between the two systems. I separate the message transmission and reception overhead costs from the acquisition of a communication channel because the acquisition time required to obtain a free channel is not generally a function of either the sending or receiving operating systems. I will define $B_y(T_y/E_y)$ as the waiting time required in trying to obtain the desired connection due to a busy link, for each link, L_y, in the system. The delay time of obtaining a communication channel is a function of the communication network used in the system and its bandwidth. The density of traffic on the links may play a significant role in the time spent waiting due to busy communication resources, especially as the traffic density increases. I will use Q_{jk} to denote the delay incurred waiting for busy links in sending all packets from process j to process k.

If message paths between processes involve intermediate connections with other processors, the delay time increases by the incurred delay at each intermediate point in the network. I will define N to be the number of processor nodes in the distributed system, and N_x to denote node x of N nodes in the system. I will use A_x as the delay associated with node N_x in the system when used as an intermediate processor in the transmission of a message. In a store and forward scenario, this delay includes S_x and R_x of the intermediate node, which is due to the processor's operating system software. In a cut through type of routing, the A_x delay term corresponds to the time required to establish a hardware connection from one link to another, bypassing the node (not including the time required to obtain the next link in the routing path).

In a good mapping of a program graph to the system graph, communication densities on the various links are minimized as much as possible also minimizing the overall communication time. To define the total traffic density associated with a specific link in the system, I need to define a link usage function U_{jk}, such that $U_{jk}(L_y) = 1$ if the connection L_y is used in the communication between processes j and k, and $U_{jk}(L_y) = 0$ otherwise. This usage function can also be used with intermediate processor nodes as arguments to determine if the node is used in communication between processes j and k. I will define two other intermediate result terms, G_{jk} and H_{jk}, to represent the total delay due to the number of messages sent and the total delay due to the number of bytes sent respectively, in all messages from process j to process k. I also need to define D_{jk} to be the total communication delay between processes j and k.

Before deriving the set of equations to calculate D_{jk}, all defined terminology is summarized in the following table:

Table I - Definition of Terms

P_n:	Process n of P processes in the program being partitioned across the distributed system.
W_{jk}:	Weight of directed program graph edge between processes j and k. (Number of messages)
L_y:	Link y of L links in the distributed system.
S_j:	Local sending operating system overhead where process j resides.
R_k:	Local receiving operating system overhead where process k resides.
C_y:	The communication time for one byte of information to be transmitted over link y of the distributed system.
M_{ijk}:	The magnitude, in bytes, of the ith message sent from process j to process k.
X_{jk}:	The magnitude, in bytes, of the amount of data sent from process j to process k.
T_y:	Traffic density in terms of messages or packets on link y of the distributed system.
E_y:	Expected execution time between first and last message to use link y of the distributed system.
$B_y(z)$:	Delay while waiting for link y of the distributed system to become free where z is the bandwidth used on link y.
Q_{jk}:	Total busy waiting time for sending all packets from process j to process k.
N_x:	Node x of N nodes in the distributed system.
A_x:	The delay associated with intermediate node x.
$U_{jk}(z)$:	Usage function for item z (link or node) in communication between processes j and k.
G_{jk}:	The total delay due to the number of messages sent from process j to process k.

H_{jk}:	The total delay due to the number of bytes sent from process j to process k.
D_{jk}:	The total communication delay between processes j and k.

5.3. A General IPC Cost Formula

To begin the derivation of the cost formula, the total traffic sent (in number of bytes) from process j to process k is calculated as follows:

$$X_{jk}=\sum_{i=1}^{W_{jk}} M_{ijk} \tag{5}$$

To calculate the traffic densities, T_y, one must sum over all process pairs, the magnitudes of the messages that must be transmitted for all messages if link L_y is used in the transmission of the messages. The formula for computing the link traffic densities is as follows:

$$T_y=\sum_{j=1}^{P}\sum_{k=1}^{P}\sum_{i=1}^{W_{jk}} M_{ijk}\cdot U_{jk}(L_y) \tag{6}$$

The total time spent waiting on busy links while sending all packets from process j to process k can be written as:

$$Q_{jk}=W_{jk}\cdot\sum_{y=1}^{L} B_y\left(\frac{T_y}{E_y}\right)\cdot U_{jk}(L_y) \tag{7}$$

The intermediate term, G_{jk}, representing the total cost due to the number of messages sent from process j to process k, can be calculated as follows:

$$G_{jk}=W_{jk}\cdot\left(S_j+R_k+\sum_{z=1}^{N} A_z\cdot U_{jk}(N_z)\right) \tag{8}$$

The intermediate term, H_{jk}, representing the total cost due to the number of bytes sent from process j to process k, can be calculated as follows:

$$H_{jk}=X_{jk}\cdot\sum_{y=1}^{L} C_y\cdot U_{jk}(L_y) \tag{9}$$

The general form of an accurate interprocess communication cost function, D_{jk}, representing the communication delay between processes P_j and P_k can now be derived. The following equation combines all appropriate terms to yield the cost of sending all messages from any process j to any process k.

$$D_{jk}=G_{jk}+Q_{jk}+H_{jk} \tag{10}$$

The first term is computed by adding the cost of sending a message from process j to the cost of receiving a message at process k and any intermediate node delay based on the number of messages sent, then multiplying the result by the number of messages sent between process j and process k. This term represents the total operating system costs of sending all messages between process j and process k. The second term of the equation represents the total time spent waiting for busy communication resources in sending all packets of information from process j to process k. The third and final term, represents the time required to send all bytes of data from process P_j to process P_k. This value is calculated as the time it takes to send one byte of data over each link used plus the time it takes to pass through each intermediate node that is used along the way.

In evaluating the cost of a particular partitioning of a program across the distributed system, the total communication cost of the partitioning is needed. One method of calculating the total communication cost of a specific segmentation of the program to be distributed is the following equation. It simply evaluates the sum of all individual process to process communication costs.

$$\textit{Total Communication Cost} = \sum_{j=1}^{P} \sum_{k=1}^{P} D_{jk} \tag{11}$$

While this can be used as a measure of a particular program partitioning, it does not reflect the actual delay that would be observed on a real system, since all of the D_{jk} terms are in fact independent and occur in some parallel overlapping fashion. In the next chapter, I will discuss the ramifications of relaxing the assumption of single routing paths and analyze the derived cost function in terms of several interconnection topologies.

Chapter 6. Analysis of Topologies

After discussing the results of relaxing the assumption that all messages passed from one process to another must be routed along the same path, I will demonstrate the flexible nature of the above cost function, by showing how the equations can be used to analyze several of the popular bus, static and dynamic network topologies. In all cases, the networks shown are assumed to be homogeneous, but this restriction does not apply to the processors. In so much as it is possible to examine the cost function in cases where several different networks are combined in one system, in the interests of simplicity and brevity, I will not examine any such examples in this text.

Due to the abundance of data formats, communication protocols, and routing algorithms, I will not relate the discussion of the following sections to specific hardware or software products, keeping the discussion at the theoretical level only. A thorough discussion of all different permutations of the various network types would fill up too much space and detract from the main idea of analyzing the communication costs. Specific networks could easily be analyzed in the same fashion as the following general theoretical discussion.

6.1. Parallel Links and Message Paths

While one of the assumptions of the cost function derived in section 5.3 is that messages sent from a process P_j to a process P_k always take the same route, utilizing the same links and passing through the same intermediate nodes during communication, I also assumed that there is only one bidirectional connection between neighboring processors. The cost formula does not have to change if multiple communication links between neighboring processors are present as long as the messages sent from process j to process k can only be routed along one path. If parallel message routing paths are allowed between processes, then the cost function must change to accommodate the fact that the communication load between the processes is now distributed over several possible sets of links with potentially different traffic densities.

The major effect of allowing parallel routing paths between processes is that the communication time is spread over multiple paths, and thus is much harder to calculate accurately. If a packetized message is transferred from process j executing on processor s to process k residing on a processor t over several parallel message paths, the communication cost of sending the message can be analyzed as follows. The communication time starts when the first packet is sent from processor s, and ends when the last packet is successfully received at processor t. No assumptions about the ordering of the packets during transmission or the communication links used may be made without knowledge of the routing algorithms used at each node in the distributed system that played a part in one or more of the messages transferred.

If the communication paths of all packets, and their sizes are known, one can attempt a calculation of the communication cost in the parallel scenario. By analyzing the cost of sending one message at a time from process j to process k, then summing the costs of all messages sent from P_j to P_k, a total cost D_{jk} may be obtained. For each parallel packet routing path used between process j and k, a cost can be calculated based on the amount of information sent and the delays along each path. Since all of the paths are in parallel, the obvious choice for the message cost would be to take the maximum delay of all of the paths, however, this is not accurate. While it is true that the communication time of some packets on different paths may overlap, thus reducing the overall delay, it is not easily discernable which packets are sent in parallel, how many overlap, or when the time of overlap between packets begins or ends. If we assume that one or more of the terms in the new equations will have to be some sort of statistical approximation, we can continue with the discussion of the new cost equations.

To properly account for the parallel nature of multiple routing paths between two processes, it is necessary to start with a different foundation for the new cost equations. X_{jk}, the total amount of data, in bytes, sent from process j to process k, was used to calculate the amount of delay for each byte transferred along the single routing path. As all of the data will not be sent over one path, each message or packet of a message must be treated on an individual basis, according to the route taken, when computing its transfer time.

Likewise, the busy time encountered along the message path, will vary depending on the traffic density of the links on the route taken by each packet of the message. The traffic densities of the links must be computed using the magnitude of the packets sent, rather than the magnitude of the messages sent. The Q_{jk} term, total busy waiting time, must be computed based on the path taken by each packet transferred from process j to process k and the traffic densities of the links used based on the packet sizes.

Using the new X_{jk} and Q_{jk} terms, each message may be analyzed among all of its parallel paths. Due to the uncertainty of the parallel overlap of packets, one must statistically estimate the percentage overlap, based on the number of parallel paths, the versatility of the routing algorithm, and the bandwidths of the links used on the various parallel paths between the two processes. Using a statistical parallelization function, one can derive the total cost of sending all messages between processes j and k, denoted D_{jk}, by summing the delays associated with each message sent. The D_{jk}'s may then be summed for all process pairs in the system, yielding the overall communication cost for a given program partitioning.

When this communication cost is used in conjunction with other parameters, such as expected execution time, an objective function can be formed. By minimizing this objective function, one can find near optimal solutions to the NP complete partitioning problem.

Lee and Aggarwal[19] show and discuss the results of optimization problems using several different objective functions based on the communication cost between two processes. When assigning an appropriate objective function for use in determining the performance of a particular mapping of tasks to processors, the objective function must accurately represent the quantities being minimized. Additionally, when utilizing a communication cost function to analyze the problems of task allocation or load balancing, certain modules of a program may need to reside at specific processors in the distributed system in order to take advantage of special resources. This means that finding near optimal solutions in these cases may not be probable, or even possible. Further analysis of parallel message routing paths and objective functions is not discussed in this thesis.

6.2. Bus Topologies

In the case of parallel computers, multicomputers, or distributed systems, the processors must be connected with some high speed interconnection network. There are three classifications of interconnection networks used in modern multicomputers, bus, static and dynamic. Bus architectures communicate with resources by sharing one common communications medium with all resources in the system. The next two sections will present serial and parallel bus architectures and the analysis of them with respect to the derived cost function.

6.2.1. Ethernet Networks

One of the most popular and widely used local area networks today is Ethernet. The concept of Ethernet was developed at the Xerox Palo Alto Research Center in the early 1970's, and standardized by Xerox Corporation, Intel Corporation, and Digital Equipment Corporation in 1978[25]. Consisting of simple coaxial cable, twisted pair, or fiber optic connections between participating machines, Ethernet provides a low cost way of sharing computer resources.

From an architectural point of view, an Ethernet is one long serial bus over which many devices communicate by sending messages. The bandwidth of a typical Ethernet is 10 megabits per second and has a maximum length of 2.5 kilometers. The network itself is completely passive and, as such, relies on each resource attached to the Ethernet to be able to communicate without assistance from the network. It was designed to be easily expandable by simply adding resources to the cable, readily accomplished by tapping into the main Ethernet cable as required[26]. Bridges could be used to join individual

Ethernets to one another, forming larger Ethernets, so long as only one routing path between any two resources on the network is maintained.

A message passed over an Ethernet is broken into packets which include a sending and receiving address, CRC, protocol information, and the data to be sent. The basic protocol for transmission of message packets is to "listen" to the Ethernet for current traffic, and if none, transmit the packet to the destination. If traffic is present, wait until the end of the current packet and then transmit. Because more than one device may be waiting for the current packet to finish, or more than one processor decides that the network is free, multiple packets may be sent at the same time causing a collision. In the case of a collision, all resources involved wait a random period of time before retrying the entire process. If the send of a packet is started successfully, then the entire packet will be placed onto the network without interruption.

Since the Ethernet is one long bus, and all packets are seen by all resources on the Ethernet, any device may receive the packet. Although each message has a delivery address, which could be to all resources, there is no extra overhead for broadcast messages. Unfortunately, packet ordering and delivery over an Ethernet are not guaranteed by the network, therefore, whatever protocol is used for transmission of messages must handle packets received out of order and retransmission of lost packets. In most systems, a reliable and unreliable, but faster, protocol are available for user applications.

For the case of TCP/IP, a common reliable delivery protocol which is machine independent, the theoretical maximum is almost 12 megabits per second with a maximum packet size of 1530 bytes[27]. This performance level is generally not reachable because of layered operating system delays and protocol constraints. When the Ethernet is heavily loaded, many collisions occur because of multiple waiting senders, causing the overall performance of the network to degrade making the theoretical maximum impossible to reach. Improvement of the capacity and efficiency of Ethernet has been an active area of research. One interesting approach presented by Dobosiewski and Gburzynski[28] is the concept of using segmented carrier and dual cables for directional message passing.

Using the cost function derived in section 5.3, an Ethernet network with no bridges can be analyzed as follows. The term in the equations that deals with delays incurred at intermediate nodes can be ignored because there are no intermediate nodes on an Ethernet. Since there is only one link, and it must be used for any communication between processes, the delay per byte on the link is simply C_y, and $U_{jk}(L_i)$ always equals 1, yielding the following equations:

$$T_y=\sum_{j=1}^{P}\sum_{k=1}^{P}\sum_{i=1}^{W_{jk}} M_{ijk} \qquad (12)$$

$$Q_{jk}=W_{jk}\cdot\sum_{y=1}^{L} B_y\left(\frac{T_y}{E_y}\right) \qquad (13)$$

$$G_{jk}=W_{jk}\cdot(S_j+R_k) \qquad (14)$$

$$H_{jk}=X_{jk}\cdot C_y \qquad (15)$$

If the network has bridges, the situation changes as follows: the intermediate delay term in the equations, A_z, must account for the delay due to the bridge(s) traversed. This leaves all of the original equations in tact, where the U_{jk} function is used to select only those Ethernets and bridges used in the system.

6.2.2. Single and Multiple Parallel Bus

The single parallel bus has been the heart of uni-processor computer systems for decades, so in the development of parallel computer systems, a natural extension of this well understood concept was to place several processors on one single bus. The Ethernet, as described in the previous section, is essentially a single bus architecture, with the main difference being the serial nature of the communication protocol. A bus architecture, has a fixed bandwidth which can easily become saturated as the number of devices using the bus increases. This is especially true if the arbitration scheme controlling the bus is asynchronous, allowing any device to attempt to obtain the bus at any time and the data flow is continuously heavy. By synchronizing the bus and the devices that communicate over it, use of the available bus bandwidth may be maximized, however, maintaining clocks without skew mandates short length busses. This makes single parallel busses practical only in multicomputers and dedicated parallel machines with few numbers of processors where the length of busses may be controlled. Synchronizing the bus also introduces a performance penalty if data patterns of the program are bursty or not evenly distributed across program modules.

The next logical step to improve system performance is to provide multiple busses between system resources to improve the communication bandwidth. While this tends to increase resource availability, it also drives the cost up geometrically with the gain in bandwidth. Direct interconnection of all resources to one another becomes impossible for larger systems, and the problems associated with a single bus are still present. The busses must be kept relatively short, and to maximize the available bandwidth, the processors and busses must be synchronized and programs partitioned in such a manner to maintain data flow as evenly as possible.

To alleviate some of the cost burden of highly interconnected resources, several researchers have adopted partial multiple bus models. Sheu and Chen[29] propose a method which prioritizes the connections to available busses in order of access probabilities and fault tolerance. This method, a slight derivation of the fully connected scheme, attaches important resources to more busses and less critical ones to fewer busses in the system. Nanda, DeGroot, and Stenger[30] discuss the task allocation problem using Texas Instruments' Tapestry architecture, which uses bus couplers to interconnect resource pools in the system. The resource pools are processors and peripherals that are connected to a single bus. The single busses are then cross coupled in a chordal fashion so as to provide multiple routing paths from one resource pool to another. This type of architecture is well suited to problems that can be broken into sets of tasks, allocated to the resource pools, which communicate among themselves more frequently than across groups. Although the number of processors in the multiple bus model can be increased over the single parallel bus model, multiple parallel busses are not well suited to large distributed systems either.

In evaluating the multiple parallel bus configuration, one must be careful to observe the fact that all messages sent from one process to another are routed along the same path. The use of multiple busses is designed to allow multiple routing paths between resources in the hope that traffic may be split among them to gain bandwidth. In these situations, the resources connected to more than one bus usually take the first available bus that can reach the destination, not a specific bus all the time. If one send-receive path cannot be guaranteed between process pairs, the formula will not yield accurate delay information and the parallel paths discussion above would apply.

From the communication cost function point of view, the single parallel bus performs similarly to the single Ethernet example. The term in the equations that deals with delays incurred at intermediate nodes can be ignored because there are no intermediate nodes on a single bus. The number of links sum becomes simply one term, C_y, because there is only one bus and it must be used for all communication between processes. Thus, the equations for the single parallel bus are identical to the ones for the single Ethernet example shown in the previous section. Although the equations are the same, the magnitude of C_y for the parallel bus is considerably smaller than the C_y value in the single Ethernet case. This occurs because the parallel bus, which transfers bytes or words in parallel, has a much higher bandwidth than its serial Ethernet counterpart.

For the multiple bus case, with the busses in parallel with one another, the system can be analyzed using the original Q_{jk} and H_{jk} equations, but with G_{jk} as follows:

$$G_{jk}=W_{jk}\cdot(S_j+R_k) \tag{16}$$

If there are bus couplers in the system, these act as intermediate nodes and A_z must correspond to the delay associated with a bus coupler. One can then use the original equations as was the case for multiple Ethernet with bridge nodes. Again, the value of C_y is much smaller for the multiple parallel bus example than for the Ethernet case due to the larger bandwidth of the bus topology.

6.3. Static Network Topologies

Static network topologies, or point-to-point networks, allow processors to share information with one another over dedicated network links that allow communication between only specific processor pairs. In contrast to the Ethernet and bus architectures, where messages on the bus are seen by all resources, messages in a point-to-point network are seen by only the intermediate nodes along the routing path between the source and destination. The communication links are usually arranged in structures that attempt to minimize the latency of the network and maximize the fault tolerance capabilities of the system. The next two sections analyze standard packet switched networks and those with virtual cut through capability using the derived IPC equations.

6.3.1. Store and Forward Networks

Store and forward, or packet switched networks, route packets of a message from the source to the destination by hopping from one processor to another along the selected routing path. Each intermediate node along the routing path stores all incoming packets as they arrive and then forwards them to the next node. While different packets of the same message may generally be routed along different paths in the network to increase bandwidth and fault tolerance, I will only be examining the case where all packets are routed along the same path.

Packet switched networks come in many varieties, and account for many of the popular multiprocessor configurations in research today. These include mesh, ring, tree, star, chordal, hypercube, and cube connected cycle arrangements. In all of these configurations, messages sent from one processor to another may have to pass through intermediate nodes, where the message must be stored and then forwarded using some routing algorithm.

The advantages of a packet switched network are that communication resources needed by another process are only tied up for the length of one packet, rather than for the whole message. This means that, while the overhead is higher at each node because packets are stored and forwarded at each intermediate node, the time spent waiting for busy links is smaller. As a result, packet switched networks may be advantageous in a system where some messages sent between processors are large so that smaller messages are not kept waiting for long periods of time. One drawback of packet switched networks is that storing and forwarding of message packets places increased load on the processors used as intermediate nodes.

As long as the routing algorithm always chooses the same path for all messages, the derived cost formula can be applied directly to the network with no modifications to the equations. As stated in the derivation, the intermediate delay term, A_z, represents the operating system delay at intermediate node N_z associated with temporarily storing the message and then forwarding it to the next node.

6.3.2. Cut Through

Cut through, or wormhole routing, is not a new type of network, but rather a way of looking at the packet switched network as a circuit switched variation. Although dedicated circuit switched networks and their properties are discussed in the section on dynamic network topologies, the concepts are used to form a circuit switched type routing within the packet switched topology in an attempt to improve the data transfer rate.

The overhead of packet switching is substantial when using the store and forward method, and while packet switched networks have some advantages, the circuit switched method substantially reduces the transfer time of large messages between processors. The concept of cut through or wormhole routing, is that a connection through the network, including intermediate nodes, is established allowing all data to be transferred without involving the intermediate nodes. To establish the connections, there must be additional hardware added to each node in the distributed system to handle the bypass routing. This type of network can be analyzed as shown in the next section under Circuit Switched Networks.

Virtual cut through is an adaptation of the pure cut through routing where both packet switched and circuit switched types of routing can be supported in the same system for maximum communication resource utilization. This hybrid algorithm is designed to maximize the throughput of a packet switched network by adding some hardware to the intermediate nodes that has the capability of establishing circuit switched type of message paths. The combination of the two approaches, especially in a general purpose machine, may substantially reduce the communication time between parallel programs exhibiting different communication patterns.

Kandlur and Shin[31] discuss the routing path selection problem in networks with cut through routing capabilities. They assume that a system can use either packet switching, cut through, or a combination of the two in routing messages. Using probabilities of establishing cut through routes in the network, the authors attempt to maximize the use of cut through in order to limit the number of nodes at which messages are buffered. In this hybrid of packet switching and circuit switching, messages are routed using cut through until a link needed is unavailable, then buffered at that node until the link is not busy. The authors evaluate their routing methodology for the hypercube and C-wrapped hexagonal mesh topologies.

The virtual cut through routing scheme may be evaluated using the derived cost function by looking at the analysis of the packet switched and circuit switched cases. If a circuit switched path can be established from source to destination, the communication cost may be evaluated as in the above circuit switched case. Likewise, if only packet switching is used, the packet switching analysis applies. If a combination of the two is used, the portion of the path that uses each type of routing can be analyzed separately using the proper equations.

6.4. Dynamic Network Topologies

Dynamic network topologies allow the communication paths to change dynamically via switches and provide connections between multiple processor pairs at different times. The last point to point network type is circuit switched or the multiple stage interconnection network, in which a routing path is established from the source, through all intermediate nodes to the destination before any data is transmitted. The data is then sent along the established path in much the same way a telephone call is dialed, a connection is established and a then a conversation takes place. Circuit switched networks can further be divided into two types, one where the intermediate nodes in the connection are actually switches in the routing path, and the other in which the intermediate nodes are processors capable of establishing bypass paths through the node without storing the messages, called cut through.

6.4.1. Crossbar Networks

Crossbar networks are usually presented in the context of N processors communicating with M memory modules, but could equally apply to interprocessor communication. In a crossbar network, each processor has one I/O port through which it may communicate with any of the other processors in the system via complex network switches. The dual of this configuration is a static topology where each processor has N I/O ports to communicate with each other processor, where N is the number of processors in the system. Since the complexity of the switches used in the network, or the number of I/O ports on the processors, are directly proportional to the number of processors in the system, this type of network is only used for a small, localized group of processors and usually is prohibitively expensive.

By using the derived cost function the crossbar interconnection structure can be analyzed using the original equations as written. The value of A_z represents the delay due to the switch network and is usually small enough that it can be ignored entirely. Since this thesis does not concern itself with the analysis of degraded system performance, and since a fully connected system with a fault looks like a packet or circuit switched network, I will not discuss it further. The fully connected network can be grouped under the category of point-to-point networks and analyzed as such.

6.4.2. Circuit Switched Networks

Circuit switched networks, like packet switched networks, come in many flavors depending on the arrangement of switches and level of connectivity provided. Some examples of dynamic switching networks are omega, shuffle-exchange, delta, benes, banyan, gamma, augmented data manipulator, and inverse augmented data manipulator configurations. While each of these networks possess certain inherent communication bandwidth, fault tolerance, cost, and complexity characteristics, all conform to the idea of establishing a consistent connection between source and destination for the duration of the data transmission. As previously stated, the case under consideration in this thesis is that where only one routing path may be chosen for sending all messages from one process to another.

One advantage of a circuit switched network is that the delay penalty for setting up the message path is only paid once for each message sent, as opposed to one at each intermediate node in the routing path. No additional load is placed on the processors used as intermediate nodes because the intermediate nodes do not look at the message as it passes. While the waiting time for free links may be higher, all data of a message is sent without the delay of store and forward, minimizing the communication time between processors. Circuit switched networks may be advantageous in a system where most messages sent between processors are about the same length so that the waiting time is evenly distributed, and data throughput is maximized while the links are in use.

An analysis using the derived cost function must account for the initial penalty of establishing the communication path, as it is one of the largest delays in the formula. In fact, the data of the message, following the first byte, is delivered in a pipeline fashion, so the delay paid per byte is only the maximum delay on any of the links used between the source and destination. The H_{jk} equation, which accounts for link delays, can be rewritten to account for this as follows:

$$H_{jk}=\sum_{y=1}^{L} C_y \cdot U_{jk}(L_y) + (X_{jk}-1)\underset{y=1}{\overset{L}{MAX}}(C_y \cdot U_{jk}(L_y)) \qquad (17)$$

The rest of the terms in the original equations are correct noting that A_z represents the hardware delay at an intermediate node or switch, a much smaller delay, than its packet switched counterpart.

Chapter 7. Conclusions and Future Work

Although very few products have been developed yet, using the software tools and parallelization techniques available today, fairly efficient parallelization of sequential programs is theoretically possible. By extracting as much information as possible from the source code, intended applications of the program, and the user environment, good estimates of performance improvements can be estimated and user programs may be segmented into parallel tasks. After accurately analyzing the interprocess communication costs for a program's parallel tasks, a good solution to the static allocation of program modules can be achieved, minimizing the system throughput time for each program. Integrating all of these elements into a new generation of software development tools will allow the programmers of today to develop the applications of tomorrow.

The parallel and distributed software industry is still in its infancy, and considerable research is needed before parallel computing environments can be utilized to their full potential. Further research is needed in determining parallelism in programs and how to optimally partition programs into tasks, while taking advantage of as much parallelism as possible. Although the static task allocation problem has been studied in depth by many researchers, no optimal solution to the problem has been found. Additional research into models that accurately represent parallel message paths in communication cost functions is also necessary for distributing program modules on machines with these capabilities. As new parallel languages, parallelization tools and distributed operating systems mature, the problems that can only be solved with today's most sophisticated computers will become tomorrow's building blocks.

List Of References

[1] L. C. Lu and M. C. Chen, "Parallelizing Loops with Indirect Array References or Pointers," *4th International Workshop on Languages and Compilers for Parallel Computing*, 1991, pp 201-217.

[2] Z. Li, P. C. Yew, and C. Q. Zhu, "An efficient data dependence analysis for parallelizing compilers," *IEEE Transactions on Parallel and Distributed Systems*, Jan. 1990, pp. 25-34.

[3] M. Girkar and C. Polychronopoulos, "Optimization of Data/Control Conditions in Task Graphs," *4th International Workshop on Languages and Compilers for Parallel Computing*, 1991, pp. 152-168.

[4] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, 1989.

[5] D. Bustard, J. Elder, and J. Welsh, *Concurrent Program Structures*, Prentice-Hall, Inc., 1988.

[6] G. R. Andrews, *Concurrent Programming - Priciples and Practices*, The Benjamin/Cummings Publishing Company, 1991.

[7] D. Padua, D. Kuck, and D. Lawrie, "High-speed multiprocessors and compilation techniques," *IEEE Transactions on Computers*, Vol. 29, 1980, pp. 763-776.

[8] Z. Shen *et. al.*, "An empirical study on array subscripts and data dependencies" *Proceedings 1989 International Conference on Parallel Processing*, 1989.

[9] H. S. Stone, *High-Performance Computer Architecture, 2nd Edition*, Addison-Wesley Publishing Company, 1990.

[10] P. Siarry, L. Bergonzi, and G. Dreyfus, "Thermodynamic optimization of block placement," *IEEE Transactions on Computer Aided Design*, Vol. 6, Mar. 1987, pp. 211-221.

[11] J. S. Rose, D. R. Blythe, W. M. Snelgrove, and Z. G. Vranesic, "Fast, high quality VLSI placement on an MIMD multiprocessor," *Proceedings of the IEEE International Conference on Computer Aided Design*, 1986, pp. 42-45.

[12] J. Lam and J. M. Delosme, "Logic minimization using simulated annealing," *Proceedings of the IEEE International Conference on Computer Aided Design*, 1986, pp. 348-351.

[13] Special Issue on Computer Architectures for Image Processing, *IEEE Computer*, Vol. 16, Jan. 1983.

[14] S. Y. Lee and J. K. Aggarwal, "A problem-driven approach to parallel image processing: System design and Scheduling," *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 1985, pp. 680-686.

[15] S. Y. Lee and J. K. Aggarwal, "Exploitation of image parallelism via the hypercube," *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, 1986.

[16] K. Hwang and J. Xu, "Efficient Allocation of Partitioned Program Modules in A Message Passing Multicomputer," *Proceedings of the ISMM International Conference on Parallel and Distributed Computing and Systems*, Oct. 1990, pp. 226-230.

[17] A. K. Ezzat, R. D. Bergeron, and J. L. Pokoski, "Task Allocation Heuristics for Distributed Computing Systems," *Proceedings of the IEEE 6th International Conference On Distributed Systems*, 1986, pp. 337-346.

[18] S. W. Bollinger and S. F. Midkiff, "Heuristic Technique for Processor and Link Assignment in Multicomputers," *IEEE Transactions on Computers*, Vol. 40, No. 3, Mar. 1991, pp. 325-333.

[19] S. Y. Lee and J. K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Transactions on Computers*, Vol. 36, No. 4, Apr. 1987, pp. 433-442.

[20] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Jun. 1982, pp. 50-56.

[21] V. M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers*, Vol. 37, No. 11, Nov. 1988, pp. 1384-1397.

[22] D. T. Peng and K. G. Shin, "Static Allocation of Periodic Tasks With Precedence Constraints In Distributed Real-Time Systems," *Proceedings of the IEEE 9th International Conference On Distributed Systems*, 1989, pp. 190-198.

[23] A. N. Tantawi, "Optimal Static Load Balancing in Distributed Computer Systems," *Journal of the Association for Computing Machinery*, Vol. 32, No. 4, Apr. 1985, pp. 445-465.

[24] B. W. Kernighan and D. M. Ritchie, *The C Programming Language 2nd. Ed.*, Prentice-Hall, Inc., 1988.

[25] D. E. Comer, *Internetworking With TCP/IP, Second Edition, Volume I*, Prentice-Hall, Inc., 1991.

[26] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Area Networks," *Communications of the ACM*, Vol. 19, No. 7, Jul. 1976, pp. 395-404.

[27] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., 1990.

[28] W. Dobosiewicz and P. Gburzynski, "Ethernet with Segmented Carrier," *Proceedings of the IEEE Computer Networking Symposium*, 1988, pp. 72-78.

[29] J. P. Sheu and W. T. Chen, "Performance Analysis of Multiple Bus Interconnection Networks with Hierarchical Requesting Model," *Proceedings of the IEEE 8th International Conference On Distributed Systems*, 1988, pp. 138-144.

[30] A. K. Nanda, D. DeGroot, and D. L. Stenger, "Scheduling Directed Task Graphs on Multiprocessors using Simulated Annealing," *Proceedings of the IEEE 12th International Conference On Distributed Systems*, 1992, pp. 20-27.

[31] D. D. Kandlur and K. G. Shin, "Traffic Routing For Multi-Computer Networks With Virtual Cut-Through Capability," *Proceedings of the IEEE 10th International Conference On Distributed Systems*, 1990, pp. 398-405.

Vita

Andrew William List was born in Winfield, Illinois, U.S.A. on January 1, 1967 to William Howard Stillman List and Caroline Mulford Freed. He earned his Bachelor of Science degree in Computer Engineering from Lehigh University in June of 1989. After working for two years as a Design Engineer with the Aerospace Division of General Electric in Syracuse, New York, he accepted a teaching assistant position with the Department of Computer Science and Electrical Engineering at Lehigh University (August, 1991 to June, 1992). In June of 1992, he accepted employment with Paralogic, Incorporated as a parallel software programmer. Andrew is a member of Eta Kappa Nu.

END

OF

TITLE