OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Status Report: Parallel Programming
Support Environment Research at Oregon State University

W.G. Rudd
T.G. Lewis
Hesham El-Rewini
David V. Judge
Scott Handley
Inkyu Kim

Oregon Advanced Computing Institute (OACIS)
&
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902

89-80-1

Status Report: Parallel Programming Support Environment Research at Oregon State University

W.G. Rudd, T. G. Lewis, Hesham El-Rewini, David V. Judge, Scott Handley, Inkyu Kim
Oregon Advanced Computing Institute (OACIS) and
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902
(503)737-3273
August 17, 1989

Introduction

The most significant problem facing the parallel computing field is parallel programming [4]. Along with all the software problems associated with sequential programming, parallel programmers must deal with non-determinism, race conditions, and problems involving portability and compatibility between different parallel and sequential architectures. The Parallel Programming Support Environment (PPSE) is a set of software tools designed to help parallel programmers deal with reverse engineering and forward engineering aspects of parallel programs. Forward engineering deals with the task of writing a new parallel program from scratch. Reverse engineering involves retrofitting existing sequential programs onto parallel computers. PPSE research involves the following topics (from [4]):

- how to partition an application into parallel parts,
- how to map parallel parts onto multiple processors,
- how to optimally schedule and run parallel parts,
- how to reverse engineer existing serial source code,
- how to measure and analyze performance,
- how to distribute data over a multiprocessor network,
- how to coordinate design, coding, debugging and performance,
- which techniques work.

With the forward engineering part of the project, the research involves how to design, implement, test, and evaluate the performance of a parallel program. PPSE research at Oregon State University addresses a series of issues related to designing and writing software for parallel computers. A number of practical tools have been developed which allow a programmer to visually design an architecture independent program, specify a high-level description of architectures on which the parallel program might run, determine a schedule or map for assigning program segments to processors, and automatically generate source code for a specific parallel computer from code fragments, the graphical description of the machine and the graphical description of the software. The major areas of research address the following general problems:

- Developing a Graphical Notation for the Design and Description of Parallel Programs.
- Developing a Graphical Notation for the Description of Parallel Machines.
- Mapping the Parallel Software to the Parallel Machine.
- Automatic Generation of Machine Dependent Parallel Source Code.
- Developing Visual Methods of Inputting the Hardware and Software Design Details.

Extended Large Grain Data Flow (ELGDF)[1] is a graphical language for designing parallel programs. Ideally, parallel software should be designed independent of any specific hardware on which the developed code might eventually run. ELGDF allows the development of a high level, machine independent description of a parallel program. ELGDF also allows the design of parallel software without being bound to any particular programming language. To enter descriptions of parallel programs, an ELGDF design editor which runs on a Macintosh has been developed. The design editor provides a visual method of inputting software design details in ELGDF notation. The following features have been implemented into the design editor:

- ability to produce a hierarchical design for parallel software in ELGDF notation,
- ability to add detailed textual specification to graphic notation through dialog windows,
- easy manipulation of design by resizing, encapsulating, and expanding the graphical description,
- ability to assign source code fragments to specific graphical objects,
- graphics to text (and text to graphics) transformations for interface with other PPSE tools.

Once the program design has been entered, other PPSE tools permit the analysis of the design and transformation of the design into forms such as dependency graphs, flow graphs and source code. Before these steps can be taken, however, specific implementation details must be entered.

To enter descriptions of parallel machines, a target machine editor has been developed. The Target Machine Editor provides a graphical analog to the classical Processor-Memory-Switch hardware description notation developed by Siewiorek, Newell, and Bell [2]. The present implementation of the target machine editor runs on top of the Extend™ simulation package on a Macintosh. The following features are implemented:

- ability to graphically describe small irregular architectures or easily describe large regular architectures,
- graphically create shared memory, tightly coupled distributed memory or loosely coupled distributed memory architecture descriptions,
- describe system specific information by entering the information in dialog boxes which are logically attached to the graphical icons,
- graphics to text transformation for interface with other PPSE tools,
- ability to save and edit graphical descriptions of systems.

In general, a number of system level blocks (processor, memory, bus, and switch) are kept in a library. The user selects blocks from the library and enters specific information by double clicking on the block and keying in the block specific information (like processor speed or memory size) in the fields of the dialog window. A global information block, called the Topology File Generator must be present in all system descriptions. This block contains information which is global to the system. In the case of large regular architectures, the Topology File Generator block may be the only block necessary. All necessary information can be entered in its dialog.

Once the software and hardware descriptions have been gathered, the software designer should determine the optimal assignment of software processes to processors. MH (Mapping Heuristic) is a tool which performs an automated mapping of the software onto the hardware. MH maps program modules represented as nodes in a precedence task graph with communication (a transformation of the ELGDF design file) onto arbitrary machine topologies and gives an allocation and ordering of tasks onto processors. It produces as output a Gantt chart, providing easy visualization of the allocation of the program modules onto the target machine processing elements, and the execution order of tasks allocated to each processing element. The Gantt chart consists of a list of all processing elements in the target machine. For each processing element, the Gantt chart shows a list of all tasks allocated to that processing element, ordered by execution time, including task start and finish times. The mapping heuristic modifies Kruatrachue's [3] basic heuristic to handle communication delay between tasks assigned to heterogeneous processing elements in an arbitrary target machine topology.

A desirable output from the PPSE design is compilable source code. A glue code module has been developed which takes the PPSE software design, C code fragments, and a hardware description as input and produces C-Linda source code as output. The C-Linda code can then be compiled on parallel machines which support C-Linda such as the Intel IPSC2, Sequent Balance, and the Cogent Machine. One of the primary problems with manual generation of parallel programs is the lack of portability of the finished code due to the architecture and vendor specific parallel programming primitives. Parallel programs, like sequential programs, frequently need to be transported across architectures. The glue code module allows the parallel program designer to design parallel programs without specifying architecture specific synchronization and communication primitives (such as locks on a shared memory system or message passing primitives on a distributed system). The glue code module automatically adds these primitives to the code fragments according to the specified design in the ELGDF editor.

The PPSE project is an attempt at creating a unified approach toward parallelism. Each of the following sections of this paper describe the major areas of PPSE research, at Oregon State University, in more detail. We feel that it is necessary to provide tools which allow program designers the ability to experiment with the ELGDF paradigm as one possibility among many different parallel programming models. Only through real practice and experimentation will the ultimate solutions to the parallel software design problem emerge.

References

1. H. El-Rewini and T. Lewis ," Software Development in Parallax: The ELGDF Language," Technical Report (88-60-17), Dept. of Computer Science, Oregon State, University, July 1988.
2. D. Siewiorek, C. G. Bell, A. Newell, Computer Structures: Principles and Examples, Mcgraw-Hill Book Co, 1982.
3. B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processing Systems," Ph.D. Thesis, Oregon State University, Corvallis, Oregon, 1987.
4. T. Lewis, "Parallel Programming Support Environment Research", TR-PPSE-89-1, Oregon Advanced Computing Institute, Beaverton, Oregon, 1989.

ELGDF: Design Language for Parallel Programming

Abstract

ELGDF (Extended Large Grain Data Flow) is a graphical language for designing parallel programs. The goal of ELGDF is two-fold: 1) to provide a program design notation and computer-aided software engineering tool, and 2) to provide a software description notation for use by automated schedulers and performance analyzers. The syntax is hierarchical to allow construction and viewing of realistically sized applications. ELGDF is a program design language, and not a programming language, but an ELGDF design can be refined into Pascal, C, FORTRAN, etc. source code programs through simple transformations. ELGDF facilitates describing parallel programs in a natural way for both shared-memory and message-passing models using architecture-independent higher abstractions that allow program designers to express their algorithms in high level structures such as replicators, loops, pipes, branches, and fans without having to worry about details such as synchronization code. Arc overloading in current graphical languages is resolved in ELGDF by using different symbols and different attributes for different types of arcs.

1. Introduction

It seems clear that the next generation of computers will be based on the multiprocessor paradigm, but more effort is needed to help software engineers develop programs for parallel computers. Because humans tend to think sequentially rather than concurrently, program development is most naturally done in a sequential language [11]. Unfortunately sequential programming is incapable of directly making effective use of parallel computers.

If we look at the evolution of sequential programming, we find that sequential programming has evolved in the following way: at the beginning all the programs were written in architecture-specific low level languages. Then high level languages started to appear allowing programs to be written in architecture independent languages so the programmers didn't have to worry about the architectural details. Finally extensions have been made to high level languages to make them more structured and abstract leading to programs that are easier to develop, test, and maintain. We believe that parallel programming should evolve in the same direction. Developing hand-coded parallel programs is equivalent, in a sense, to programming in a low level sequential language, because hand-coded parallel programs are quite architecture dependent. For example synchronization is done using locks in a shared memory architecture, but synchronization is done via message passing in a distributed memory architecture.

In order to develop hand-coded programs for parallel systems, the programmer has to exploit the potential concurrency of the algorithm, write the parallel program for a given architecture using a language and synchronization constructs suitable for the given architecture, schedule tasks on the available processors using intuitive methods, execute the program, and finally debug the program if it doesn't give the expected results or if it goes into a deadlock situation. Programmers have a great deal of details to worry about at any time which makes parallel programming a very difficult process. In order to make parallel programming easy, we need to get the system to shoulder more of the burden.

It is not surprising that an architecture independent higher abstraction is needed so program designers can express their algorithms in high level structures without having to worry about the details like the synchronization code. High level parallel programs then can be analyzed and translated into schedulable units of computation that fit the target hardware architecture.

We describe the ELGDF design language that allows program designers to easily express parallel program designs in a graphical, hierarchical, and natural way for both shared-memory and message-passing models. The ELGDF provides design files that contain the information needed by different tools in the PPSE (Parallel Programming Support Environment) under development at Oregon Advanced Computing Institute (OACIS). For example an ELGDF design can be easily transformed into task graphs at different levels of granularity to be used by scheduling tools. Estimated execution time of tasks at different levels of granularity can also be used by performance evaluation tools. An ELGDF design also can be refined into Pascal, C, FORTRAN, etc. source code programs through simple transformations. We believe that the ELGDF design language will ease software development for parallel computers, help programmer comprehension and will produce parallel program designs in a form appropriate for analysis.

In our work a program is represented as a large grain data flow network. This work is related to a number of other studies [1,2,3,4,5,6,7,8,9,10], but extends LGDF [2,3,4] to facilitate the following: 1) The syntax poses high level structures such as replicators, loops, pipes, etc., 2) Branch and loop constructs are provided which give more information for scheduling and analysis purposes, 3) Parameterized constructs that can be expressed compactly are provided, 4) Arc overloading is resolved by providing different symbols and different attributes for different types of arcs, 5) Mutual exclusion for shared memory systems can be easily expressed, 6) Synchronized pipelining is provided through repeated arcs, and 7) ELGDF captures program designs that can be easily transformed into different forms appropriate for analysis before being refined into source code.

The rest of this paper is organized as follows. Section 2 contains the definition and details of the proposed design language while the implementation is briefly described in section 3. Section 4 shows ELGDF designs for analysis. An example is given in section 5. We give our conclusion in section 6.

2. Definition of ELGDF

ELGDF is rich enough to express the common structures found in parallel programs. An ELGDF design takes the form of a directed network consisting of nodes, storage constructs, parameterized constructs, structures, and arcs. Figure 1 shows an ELGDF design network at some level in the hierarchy.

2.1 Basic Constructs

Nodes

A node, as shown in Figure 1, is represented by a "bubble", and can represent either a simple or a compound node. A simple node consists of sequentially executed code and is carried out by at most one processor. A compound node

is a decomposable high level abstraction of a subnetwork of the program design network.

Storage Constructs

A storage construct is represented by a rectangle, and can represent either a storage cell or a collection of storage cells. A storage cell represents the data structure to be read or written by a simple node. A node connected to the top of a storage construct has access to it before any node connected to its bottom. Nodes connected to a storage construct on the same side (top/bottom) compete to gain access to that storage construct in any order. A shared storage cell X is used in Figure 1. A compound node connected to the left or the right sides of a rectangle representing a collection of storage cells means that the compound node accesses the constituents of the storage collection, but the details are given in a lower level description.

Arcs

An arc in ELGDF can express either data dependency, sequencing, transfer of control, or read and/or write access to a storage construct. A set of attributes is associated with each arc to provide information about the arc type, data to be passed through the arc, storage access policy, and communication strategy. An arc can be either a simple arc which cannot be decomposed or a compound arc which is decomposable into a set of other simple and/or compound arcs.

Simple arcs can be classified into control and data arcs. A control arc, as shown in Figure 1 (dotted line) expresses sequencing or transfer of control among nodes. A data arc carries data from one node to another or can connect a node to a storage construct. A data arc connecting a node and a storage construct can represent READ, WRITE, or READ/WRITE access according to the direction of the arc. A data arc can be used to carry data once or repeated times per activation. One of the arc's attributes is used to indicate the number of times the data will be passed through. If the value of that attribute is greater than one then the arc is considered a repeated arc. The repeated arc is used basically in pipelines. It can carry data (repeated times) from a simple node to another in a synchronized fashion. Also it can express synchronized writing and reading to or from a storage cell.

Split and Merge

Split and merge, as in Figure 1, are special purpose simple nodes for representing conditional branching. Split has two output control-arcs; one for T = True, and the other for F = False. According to the truth or the falsehood of the condition associated with the split node one of its two output control arcs is activated. Merge has N input control arcs and one output control arc. Merge activates its output arc when it gets activated by any one of its N inputs.

Replicators

A replicator, as used in Figure 1, is one of the parameterized constructs in ELGDF that allows program designers to represent concurrent loop iterations compactly. A set of attributes is associated with the replicator such as the control variable, initial value, step, and replicator bound. Replication of a node N times produces N concurrent instances of that node. An arc connected to a replicator is expanded as a set of identical arcs each of which is connected to one of the replicated instances.

Pipes

A pipe, as in Figure 1, is a high level abstraction that allows program designers to compactly represent a set of N nodes forming a pipeline . The pipe consists of N simple nodes and N-1 m-repeated arcs. The nodes forming the pipeline are replications of the same simple node. A pipe has several attributes associated with it such as number of stages in the pipeline (N), number of times the data will be passed through repeated arcs in the pipe (m) and others.

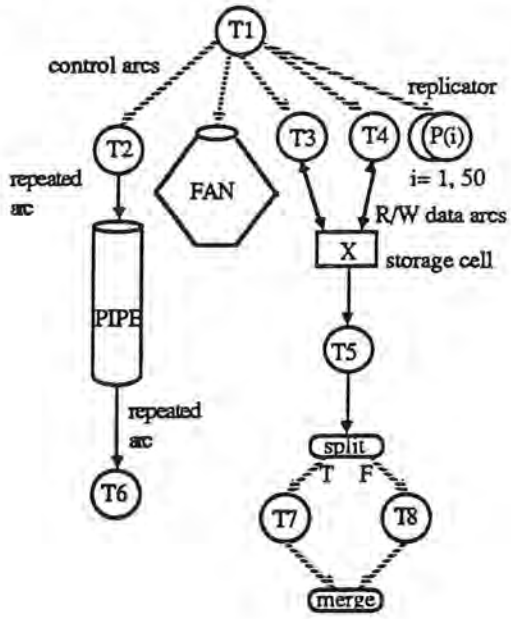


Figure 1

Loops

A loop can represent For, While, or Repeat structures. ELGDF allows program designers to express loops compactly without using cycles in the graph. This is made possible by describing only the node (simple or compound) that forms the loop body, and then specify a set of attributes such as the control variable, initial value, step, and loop bound in case of "For" or the termination condition in case of While (Repeat). A For loop iterated N times over a node can be automatically unrolled as a sequence of N instances of that node connected by N-1 arcs. Similarly, a While (Repeat) structure can automatically be represented in terms of split, merge, node and While (Repeat) constructs. Figure 2 shows a For loop construct and its unrolling with data flow from one iteration to another.

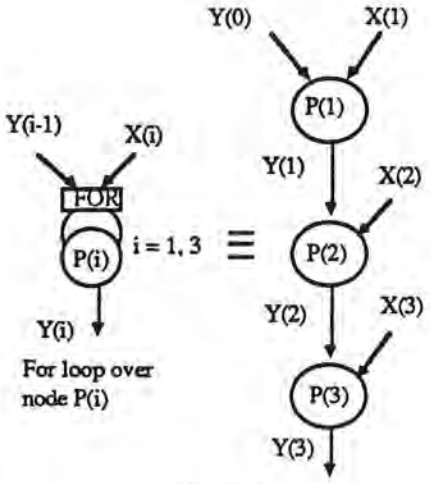


Figure 2

2.2 Common Structures

ELGDF also supports many of the common structures in parallel programs that can be synthesized using the constructs given in section 2.1 [12]. It automatically provides them for program designer convenience. Complete trees, meshes, branches, and fans are examples of common structures. The system can prepare skeletons for various types of structures per designer request. Using these structures reduces the drawing time, helps design readability and comprehension, gives more information for analysis tools (regularity of trees for instance). For example, a fan of size n is composed of a start node S, n parallel nodes P_i, i = [1..n], 2n control arcs a_j, j = [1..2n], and an end node (E). Arc a_j connects S to P_j, j = [1..n]. Arc a_k connects P_{k-n} to E, k = [n+1..2n]. The start node activates the parallel nodes and when they all finish E gets activated. Compound arcs that are connected to a fan carry data to or from its constituents.

2.3 Mutual Exclusion

ELGDF helps designers to easily express mutual exclusive access to shared variables by having an attribute associated with each arc connecting a node to a storage construct. If the exclusion attribute is set, then mutual exclusion is guaranteed. Figure 3 shows three simple nodes A, B, and C and a storage cell X forming an ELGDF network. Nodes A, B, and C share the variable X, yet A and B have access to X before C because A and B are connected to the top of X and C is connected to the bottom. A and B can access X in any order since they are both in the top side of X. Both A and B want to update X through a READ/WRITE arc and that might produce an incorrect result unless we set the mutual exclusion attribute (exclusion) associated with those READ/WRITE arcs to guarantee mutual exclusive access to X as shown.

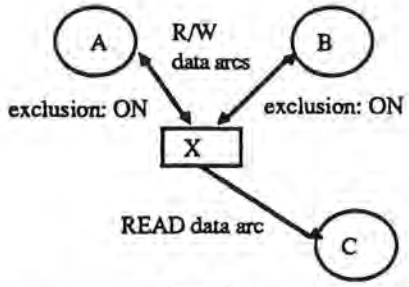


Figure 3

3. ELGDF Implementation

The ELGDF design language is being implemented in Lightspeed Pascal on Macintosh II. A user-friendly graphical Design Editor is provided as user interface. The Design Editor is a computer-assisted software engineering tool for parallel program design and implementation. It takes ELGDF designs as input, and source code fragments for each simple node in the ELGDF design, and produces source code that can be compiled and run on a parallel computer.

A parallel program designer can use the Design Editor to draw ELGDF graphs, and from these graphs produce a PP design file that contains the design primitives with additional information needed by various tools in the PPSE.

The Design editor is designed to have a menu bar as well as a palette of language symbols and tools by which a user can easily synthesize parallel programs. It also supports easy drawing and graph manipulation facilities such as dragging, resizing, encapsulation, expansion, etc. It also provides multi-window system to show parallel programs at different levels in the hierarchy. A program designer can define the attributes associated with each construct in the program using fill-in-the-spaces type of dialogues. Source code fragments for each simple node in the design are specified in FORTRAN 77 either using text editing windows or from external files. The Design Editor automatically generates some of the common parallel program structures such as trees, meshes, fans, etc. for parallel program designer convenience. The Design Editor also supports syntax checking that catches illegal connections in the ELGDF design network.

4. ELGDF Designs for Analysis

ELGDF provides the information needed by different analysis tools in the PPSE, so that program designers can get feedback and try different forms of their designs before code is written. Scheduling tools, for instance, can use a very large grain task graphs automatically obtained from ELGDF designs that hide loops, branches, and other details. Alternately, a small grain task graph that shows some or all of the branches, loops, and other details can be automatically generated.

ELGDF designs can provide information concerning the regularity of the algorithm by generating a task graph containing unrolled loops or common structures like trees and meshes. Scheduling as well as performance estimation tools are given important information such as the estimated execution time at each node at different levels of granularity, and the amount of data to be passed among nodes. For instance the operations in a simple node might be used to estimate the execution time of the node. The estimated execution time of a compound node that contains branches or

loops can be calculated from the estimated probabilities of taking different branches.

Glue code tools are provided with the information needed for code generation – for example, the files that contain the sequential code at each of the simple nodes, the precedence relations among nodes, the data/control flow in the program, the shared variables in a shared-memory system, communication protocols among communicating nodes in message passing system, and others.

5. Example

In this section ELGDF is demonstrated by means of an example that shows the top-down program construction for the solution of AX = B, where A is a lower triangular matrix. The computation, suggested by J. Dongarra and D. Sorensen of Argonne National Laboratories, is the solution of AX=B, where A is an N * N lower triangular matrix, X and B are N-vectors [5]. The tasks used in the algorithm are:

1) S(sol#)

This task solves for the triangular diagonal block sol#. It computes:

$$X(sol\#) = B(sol\#)/A(sol\#,sol\#)$$

This can be done only after all (T) tasks for row sol# have completed. Notice that S(1) can start without any preconditions.

2) Task T(i,j)

This task executes the transformation:

$$B(i) = B(i) - A(i,j)*X(j)$$

on the ith block in column j. This step can only be executed if S(j) has been completed.

To express this program in ELGDF, we first give the abstract top level design network of the program that shows the program and its input/output interaction. Then we define every construct in the top level by giving the subnetwork describing its function. We keep going down in the hierarchy defining the network constructs until we reach the lowest level in the hierarchy when we specify the source code with each simple node. Figure 4 shows the ELGDF top-down construction of the program.

As shown in Figure 4a, we give the very high level (top level) description of the program which consists of a compound node (AX=B) connected, through a READ/WRITE compound arc to a storage construct representing the data structure to be used in the program. Now we define each construct in the top level. We can decompose the compound node (AX=B) into two separate concurrent subnetworks: 1) solves for the first triangular diagonal block, and 2) solves for triangular diagonal blocks [2 ... N].

The first subnetwork consists of the compound node solve(1) connected to the storage collection representing the data structure it accesses. The second subnetwork consists of a replicator over a compound node solve(sol#) for sol# = 2, N and the storage collection representing the data structure it accesses. The replication over the compound node solve(sol#) gives (N-1) concurrent nodes (solve(2), solve(3), ..., solve(N)). Figure 4b shows the two subnetworks describing the compound node (AX=B).

Figure 4c shows the subnetwork describing the compound node solve(1). The task S(1) can start without having to wait for any other tasks. It takes B(1) and A(1,1) as input and it produces X(1). Once S(1) finishes, all non-diagonal (T) tasks in the first column can start in parallel. These parallel tasks are represented using a replicator over the simple node T(arow,1) for arow = 2, N. The replicator is connected to the bottom of the storage cell X(1) so the replicated tasks cannot start until S(1) which is connected to the top of X(1) finishes.

The subnetwork defining the compound node solve(sol#), for sol# = 2 to N, is given in Figure 4d. Since S(sol#) can start only after all (T) tasks in row (sol#) have updated B(sol#), a replicator over the simple node T(sol#,k) for k = 1 , sol#-1 is connected to the top of the storage cell B(sol#) and S(sol#) is connected to its bottom. Once S(sol#) which is connected to the top of X(sol#) finishes, all non-diagonal (T) tasks in the column (sol#) can start in parallel. These parallel tasks are represented using a replicator over the simple node T(j,sol#) for j = sol#+1, N. The replicator is connected to the bottom of the storage cell X(sol#) so the replicated tasks cannot start until S(sol#) writes into X(sol#). Notice that the READ/WRITE arcs connecting the nodes representing the (T) tasks to the elements of B vector have their exclusion attribute set so mutual exclusion is guaranteed when concurrent (T) tasks try to update an element in vector B at the same time.

Figure 4e shows the FORTRAN code associated with simple nodes S(i) for a given i and T(i,j) for a given i and j. At this point the program designer has finished the program description and the system now can generate the expanded network and the analysis files for any N.

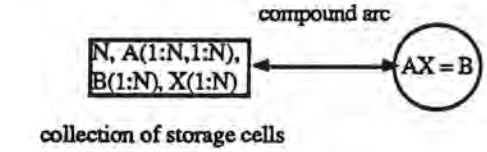


Figure 4a

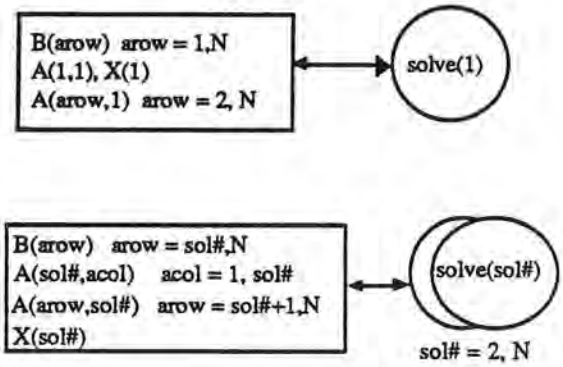


Figure 4b

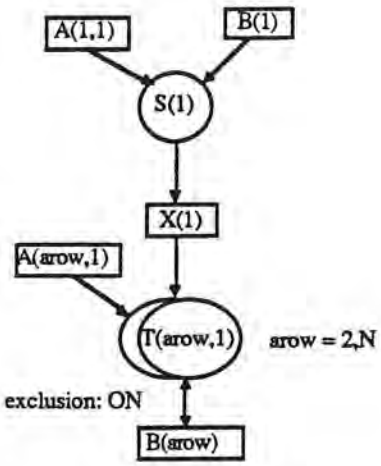


Figure 4c

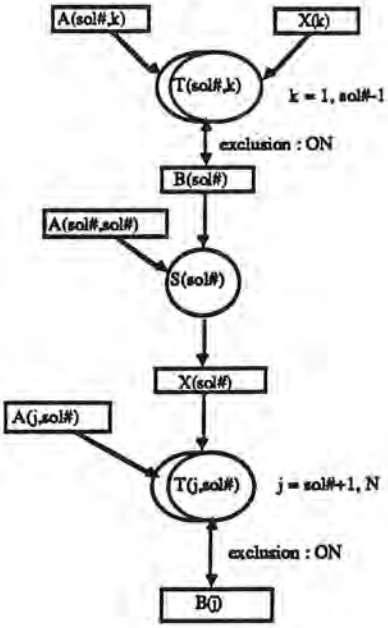


Figure 4d

$X(i) = B(i)/A(i,i)$ $\qquad$ $B(i) = B(i) - A(i,j)*X(j)$

Code at S(i) $\qquad$ Code at T(i,j)

Figure 4e

6. Conclusion

In this paper, we have presented a graphical design language for parallel programming. The complete syntax of ELGDF helps program designers to deal with parallelism in the manner most natural to the problem at hand. It allows the expression of the common structures in parallel programs easily and compactly. For example the replication mechanism used in ELGDF leads to a compact, flexible, and powerful representation of dynamic graph structures.

In addition to expressing parallel programs in a natural way for both shared memory and message passing systems, ELGDF provides a vehicle for studying parallel programs. It helps as a way to capture parallel program designs for the purpose of analysis. ELGDF provides design files that contain information needed by different tools in the PPSE.

7 References

1. P. Stotts, "The PFG Environment: Parallel programming with Petri net semantics," proceedings of the HICSS conference, 1988.

2. D. DiNucci and R. Babb, "Practical support for parallel programming," proceedings of the HICSS conference, 1988.

3. R. Babb and D. DiNucci, "Design and implementation of parallel programs with large-grain data flow," in Characteristics of parallel algorithms, Cambridge, MA: MIT Press, pp. 335-349, 1987.

4. R. Babb, "parallel processing with large-grain data flow techniques," IEEE Computer, pp. 55-61, July 1984.

5. A. Adiga and J. Browne, "A graph model for parallel computations expressed in the computation structures language," proceedings of ICPP, 1986.

6. A. Davis and R. Keller, "Data flow program graphs," IEEE Computer, pp. 26-41, February 1982.

7. J. Dennis, " Data Flow Supercomputers," IEEE Computer, pp. 48-56, November 1980.

8. T. Kimura, "Visual programming by transaction network," proceedings of the HICSS conference, 1988.

9. G. Raeder, "A survey of current graphical programming techniques," IEEE Computer, pp. 11-25, August 1985.

10. W. Ackerman, "Data flow languages," IEEE Computer, pp. 15-25, February 1982.

11. J. Allen, and K. Kennedy, " A Parallel Programming Environment," IEEE Software, pp. 21-29, July 1985.

12. H. El-Rewini and T. Lewis ," Software Development in Parallax: The ELGDF Language," Technical Report (88-60-17), Dept. of Computer Science, Oregon State, University, July 1988.

Graphical Descriptions of Parallel Machines: The PPSE Target Machine Editor

Abstract

A parallel program must run on a parallel machine. While it is desirable to produce architecture independent software to achieve portability goals, the inclusion of architecture specific details in the software design and development phase will often provide a means of gaining much needed efficiency in the performance of the software on a specific machine. Ideally, independent hardware and software descriptions should be optimally fit together through an automated process (mapping or scheduling). This paper describes current work involving the graphical description of parallel machines. The first part of this section discusses several issues related to the needed level of abstraction for the description of parallel machines and the necessary information to include in the description. The second part describes several examples that were created with a demonstration version of the Parallel Programming Support Environment (PPSE) Target Machine Editor. The Target Machine Editor provides a graphical analog to the classical Processor-Memory-Switch hardware description notation developed by Newell, Siewiorek, and Bell [6]. The third part of the paper describes the information format of the resulting text file produced by a graphics to text transformation on the graphical machine description.

Part 1 - Target Machine Description

Introduction

A target machine, in the scope of the Parallel Programming Support Environment (PPSE), is defined as the machine on which a designated parallel program will run. Essentially, three categories of parallel machines can be defined:

- machines that have globally shared memory,
- machines that have no shared memory and in which processors communicate by sending messages, and
- machines that are composed of loosely coupled "clusters" of processors.

The last category is a composition of the first two. Shared memory exists within each cluster while message passing takes place between clusters.

The target machine editor provides a visual method of entering a description of the target machine. The editor provides a library of system level blocks which can be constructed into a representation of the target machine. Each block also contains a dialog window in which system specific information (like processor speed, word size, communication bandwidth, etc.) can be entered. The graphical diagram with its associated information can be transformed into a topology (text) file containing information which uniquely describes the target machine. The information contained in the topology file can be directly fed into the PPSE database (developed at PSU).

The topology files should contain a textual description of the target machine architecture. Three different groups/modules within the PPSE project need to access the topology files: schedulers, performance analysis, and program transform/glue code. Our primary goal is to determine which architectural aspects of the target machine need to be included in the topology file according to the current and future needs of the three groups.

The Target Machine Editor

The present implementation of the target machine editor runs on the Extend™ simulation package on a Macintosh. The following features are implemented:

- ability to graphically construct small irregular architectures or easily describe large regular architectures,
- graphically create shared memory, tightly coupled distributed memory or loosely coupled distributed memory architectures,
- describe system specific information by entering the information in dialog boxes which are logically attached to the graphical icons,
- perform a graphics to text transformation in order to save the system specific information in a text file,
- ability to save and edit graphical descriptions of systems,

In general, a number of system level blocks (processor, memory, bus, switch, etc) are kept in a library. The user selects blocks from the library and enters specific information by double clicking on the block and keying in the block specific information (like processor speed, memory size, etc.) in the fields of the dialog window. A global information block, called the Topology File Generator must be present in all system descriptions. This block contains information which is global to the system. In the case of large regular architectures, the Topology File Generator block may be the only block necessary. All necessary information can be entered in its dialog.

What is Needed in the Topology Files

Schedulers:

Most scheduling algorithms being considered within PPSE need to know at most the number of processors, interconnection network, latency and contention information. Latency is a function of the distance a message must travel, the time it takes for a message to travel one hop, the size of the message and the number of packets into which the message must be split. Contention occurs when several processors contend for a common resource. When contention occurs, requests must be serialized.

To deal with the mapping problem , Berman [1] has proposed performing a series of transformations (contraction, placement, routing) which can be applied to the algorithm communication graph which result in a mapping of the algorithm into the multiprocessor. The researchers at the University of Oregon have proposed similar methods. Their

mapping algorithms calculate a near optimum process->processor mapping by approximation methods (like simulated annealing or neighborhood search). These mapping algorithms attempt to minimize a cost function which may consider conflicting goals. The real problem then becomes determining the information that must be captured in the cost function for a general class of target algorithms and architectures. The inclusion of pertinent architectural characteristics in the cost function may enhance the performance of these types of mapping algorithms while adding little to the time complexity of the mapping algorithms.

According to Jamieson [2], the following architectural characteristics all affect algorithm performance: number of PEs, memory organization, memory size, mode(SIMD/MIMD/Pipelined), network, synchronization, processor capacity, data types, addressing modes, data structures, I/O. Each of these characteristics include many sub-characteristics or metrics. For example, the network information could include such metrics as diameter, bandwidth, average distance between processors, etc. Including these characteristics in the topology files would be beneficial only if the modules accessing the information can use this information.

Performance Analysis:

From the performance analysis standpoint, the inclusion of actual metrics and behavioral characteristics in the form of statistical values from real machines, and the ability to tweek these values to simulate not-yet-built machines may be an important addition to the topology files. According to Levitan [3] there are a number of metrics which can be used to evaluate the effectiveness of certain communication structures. Additional metrics could be formulated to measure processor and memory access capabilities. Real values could be acquired from target machines by running a set of tasks and experiments on the target machine. Rough simulations of the performance of the parallel code running on machine models could then be accomplished by running different statistical values through PPSE (if the different modules take these values into consideration) and seeing if different code and/or schedules are generated.

Program Transform

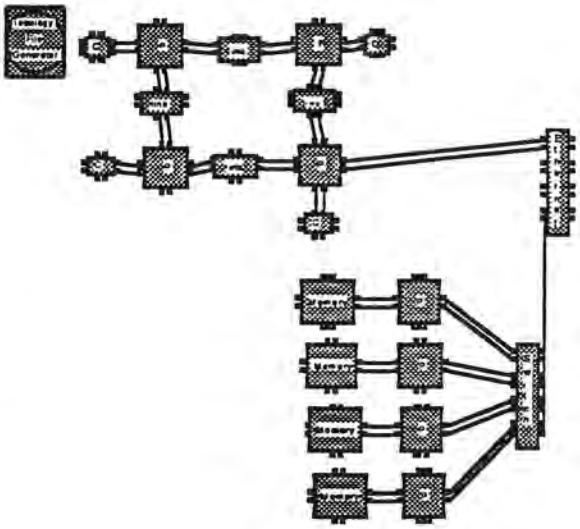
The program transform module needs to know which set (package) of glue code or macros to grab from the glue code files. Presumably, each machine will need a different set of macros based on language, version, compiler and architecture. For example, if the glue code needed is for version 1.09 F77 compiler, FORTRAN Language on the Sequent Balance (B21), we should specify this in the topology files.

Part 2 - Target Machine Editor - Examples

The Target Machine Editor is built on top of the Extend Simulation package. To describe computer systems, a library of system level blocks has been created. The blocks can be classified into four categories:

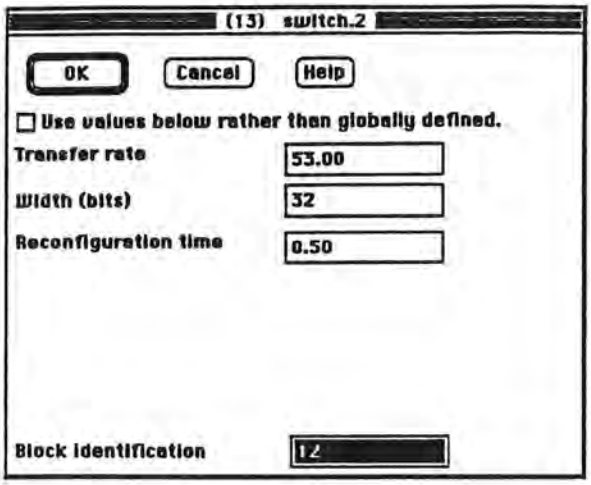
- Processing Elements
- Memory Elements
- Communication Elements
- Global Information Elements

Example 1 - Random Network of processors



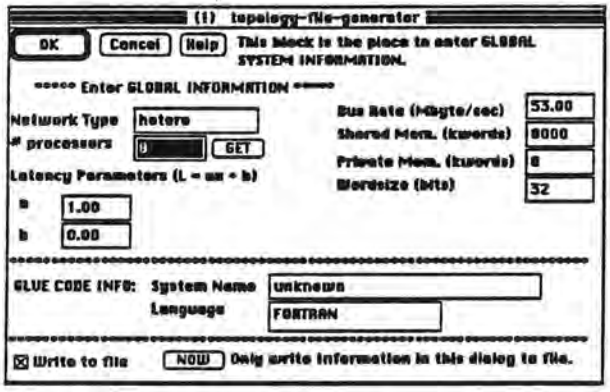
Each block in this example was selected from a library which can be accessed from the menu bar. The Topology-File-Generator (TFG) block must be present for all target machine descriptions and must always be placed in the upper lefthand corner. (Extend executes the code for each block according to the placement of the block on the screen. The TFG block must always execute first. No other placement or ordering constraints exist for the other blocks.)

- All blocks have an associated dialog box - to see the box for any block, double click on the block's icon. The following dialog box is for a switch.

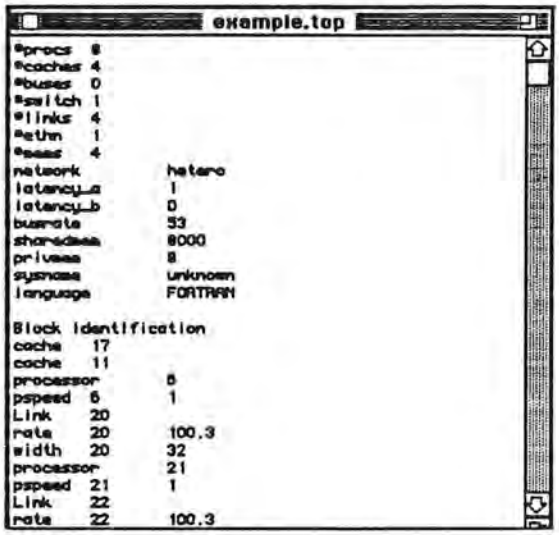


The topology file is a textual analog to the information contained in the graphical and dialog-entered system description.

- To generate a topology file, double click on the TFG block and check that the "Write to file" box, in the lower left corner of the dialog box, is checked (and click the OK button to record the selection).



- Select Run Simulation from the Run menu to actually create the file. The following is part of the generated topology file. The full file format is listed in Part 3 of this paper.



Example 2 - Hypercube

Describing a 512 node hypercube using a graphically entered description would be difficult with the current version of the target machine editor. However, a topology file can still be generated for completely regular architectures like hypercubes. The information in the Topology File Generator dialog box would look as follows:

- Make sure the Write to file box is checked and then click the "NOW" button on the bottom center of the dialog box. This will create a topology file consisting of only the information in the TFG dialog box.

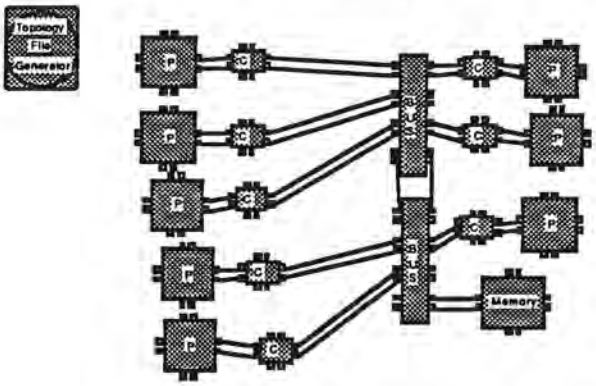
Example 3 - Shared Memory System

To graphically enter an eight processor shared-memory system:

- Select New from the File menu to get a new worksheet.
- Select Topology-File-Generator from the TM-Lib in the Libraries menu. A TFG block should appear on the worksheet.

- Select processor, cache, bus and then memory from the TM-Lib in the Libraries menu and make the correct number of copies of each block.

- Enter information into the TFG block. Enter information into the other blocks. The visual description might look as follows:



- To generate a topology file, double click on the TFG block and check that the "Write to file" box, in the lower left corner of the dialog box, is checked (and click the OK button to record the selection).

- Select Run Simulation from the Run menu to generate the topology file.

Part 3 - Topology File Format

The topology file consists of the following sections:

- Block counts - how many of each block type is represented. Format:

  <count-description> <count>

- Global data - information which pertains to the entire system or is default for any block which doesn't over-ride the information. Format:

  <data-description> <data>

- Block identification and block information - each block type is associated with a unique number for identification as follows:

  <block-type> <id>

  Several blocks contain information of the form:

  <type-info> <block-number> <data>

- Input Adjacency List - each line consists of a block identifying itself and all blocks which provide input to the identifying block. Format:

  <me> <input-1> <input-2> ...

- Output Adjacency List - each line consists of a block identifying itself and all blocks which the identifying block provides output to. Format:

  <me> <output-1> <output-2> ...

Future Work

The current version of the Target Machine Editor is capable of interfacing with the other PPSE modules and has served as a useful exercise in determining the proper level of abstraction for machine description. The next steps should be to customize the tool so that hierarchical descriptions can be generated, add the capability of text to graphics transformation, and work on the problem of representing large, possible irregular architectures. These steps will create a more useful and more robust tool for describing parallel architectures.

References

1. F. Berman, "Experience with an Automatic Solution to the Mapping Problem", in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, R. Douglas (Editors), MIT Press, (1988).

2. Jamieson, "Characterizing Parallel Algorithms", in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, R. Douglas (Editors), MIT Press, (1988).

3. Levitan, "Measuring Communication Structures in Parallel Architectures and Algorithms",in *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, R. Douglas (Editors), MIT Press, (1988).

4. Kramer, Magee, Sloman, "Configuration Support for System Description, Construction and Evolution", Communications of the ACM, 1989.

5. E. Lusk, R. Overbeek, et al., Portable Programs for Parallel Processors, Holt, Rinehart & Winston, Inc., 1987.

6. D. Siewiorek, C. G. Bell, A. Newell, Computer Structures: Principles and Examples, Mcgraw-Hill Book Co, 1982.

Static Mapping of Task Graphs with Communication onto Arbitrary Target Machines - Case Study: Hypercube

Abstract

A new scheduling heuristic is introduced with the following characteristics: 1) inputs to the scheduler are a) an arbitrary labelled task graph representing a parallel program with estimated task size, estimated message size, and b) a target machine description which includes the speed of the processors, the initialization time, the transmission rate, and the interconnection topology, and 2) schedules are computed by an adapted highest-level-first heuristic. The results for scheduling simulated task graphs on ring, star, mesh, hypercube, and fully connected networks are introduced. On hypercubes these simulations suggest that 1) communication delays should be considered in task selection when scheduling communication intensive applications, 2) priority scheduling is insensitive to the communication delays of computation intensive applications, 3) performance is inversely proportional to the ratio between average communication and average task execution time, and 4) the effect of increasing the task graph average degree increases as the number of processing elements increases.

1. Introduction

The problem of scheduling parallel program modules onto multi-processor computers has received considerable attention in recent years. This problem is known to be NP-complete in its most general form [10]. Regardless, many researchers have studied restricted forms of the problem by constraining the task graph representing the parallel program or the parallel system model [1,4,6,11,14]. For example when communication between tasks is not considered, a polynomial time algorithm can be found for scheduling tree-structured task graphs wherein all tasks execute in one time unit [5].

It is well known that linear speedup generally does not occur in a multi-processor system because adding additional processors to the system also increases inter-processor communication [13]. In order to be more realistic we need to consider communication delay in scheduling tasks onto multi-processor system. Prastein [12] proved that by taking communication into consideration, the problem of scheduling an arbitrary precedence program graph onto two processors is NP-Complete and scheduling a tree-structured program onto arbitrary many processors is also NP-Complete. Kruatrachue [2] introduced a new heuristic based on the so called list algorithms that considers the time delay imposed by message transmission among concurrently running tasks by assuming a homogeneous fully connected parallel system.

Task allocation is not the same as task scheduling. The goal of task allocation is to minimize the communication delay between processors and to balance the load among processors [7,8,9]. Kruatrachue [2] showed that task allocation is not sufficient to obtain minimum run time since there is a significant difference in performance when the order of execution is changed among allocated tasks on a certain processing element. Other work has been done in task allocation when the program is represented as an undirected task graph [15].

Kruatrachue [2] suggested some directions for future work in relaxing restrictions in the program task graph and the parallel system model. In this paper we extend the parallel system model used by Kruatrachue to accommodate arbitrary parallel systems. We introduce a mapping heuristic (MH) that maps program modules represented as nodes in a precedence task graph with communication onto arbitrary machine topology. MH gives an allocation and ordering of tasks onto processors. We then apply MH to the problem of mapping task graphs with precedence and communication delay onto cube-connected multiprocessors.

The rest of this paper is organized as follows. Section 2 contains the formulation of the problem. List scheduling is briefly described in section 3. Section 4 shows the proposed mapping heuristic. Experimental results that show the effect of changing the policy used in MH to select a task and changing two parameters of the task graph representing the parallel program on the performance when hypercube is used as the target machine are given in section 5. We give our conclusions in section 6.

2. Formulation of the Problem

Our goal is to devise an efficient heuristic scheduler to statically map parallel program modules onto a finite number of processing elements in a pattern that minimizes final completion time as determined by actual task computation time and communication between processors.

Program Graph

A parallel program consists of M separate cooperating and communicating modules called tasks. Its behavior is represented by an acyclic directed graph called a *task graph*. A directed edge (i,j) between two tasks i and j exists if there is a data dependency between the two tasks which means that task j cannot start execution until it gets some input from task i after its completion. Once a task begins execution, it executes until its completion (non-preemption). The task graph is assumed to be static which means it remains unchanged during execution.

Target Machine

A target machine is assumed to be made up of an arbitrary number N of heterogeneous processing elements that run a single application program at a time. These processing elements are assumed to be interconnected in an arbitrary way. A message sent from a task running on processing element P_i to another task running on processing element P_j takes the shortest path between the two processing elements through one or more hops. Communication time between two tasks located on the same processing element is assumed to be zero time units. A processing element can execute a task and communicate with another processing element at the same time.

System Parameters

Parameters are required to represent the computational costs and communication costs incurred by a parallel program on a specific parallel processing system. The costs are as follows:

1) E(m,n): the execution time of task m when executed on processing element n, m = 1, ..., M; n = 0, ..., N-1.

2) $C(m_1,m_2,n_1,n_2)$: the communication delay between tasks m_1 and m_2 when they are executed on processing elements n_1 and n_2, respectively, m_1, m_2 = 1, ..., M; n_1, n_2 = 0, ..., N - 1.

The parameter E(*) reflects the speed of the processing elements and the size of the tasks. E(m,n) = INS(m)/S(n) where INS(m) gives the number of instructions to be executed in task m and S(n) gives the speed of processing element n; m = 1, ..., M; n = 0, ..., N-1.

The parameter C(*) reflects the target machine performance parameters as well as the size of the data to be transmitted. $C(m_1,m_2,n_1,n_2) = (D(m_1,m_2)/ R + I)*H(n_1,n_2)$ where $D(m_1,m_2)$ gives the size of the data to be sent from m_1 to m_2, $H(n_1,n_2)$ gives the number of hops between n_1 and n_2, I represents the time to initiate message passing on each processing element, and R represents the transmission rate, m_1,m_2 = 1, ..., M; n_1, n_2 = 0, ..., N-1. The model studied by Kruatrachue [2] can be easily generated as a special case of our model.

3. List scheduling

One class of scheduling heuristics, in which many parallel processing schedulers are classified, is list scheduling. In list scheduling each task is assigned a priority. Whenever a processor is available, a task with the highest priority is selected from the list and assigned to that processor. The schedulers in this class differ only in the way that each scheduler assigns priorities to nodes. Priority assignment results in different schedules because tasks are selected in different order. A comparison between different task priorities has been studied in [3].

The insertion scheduling heuristic (ISH) introduced by Kruatrachue [2] is essentially a list scheduler that considers communication with an improvement to the communication delay problem by plugging in an insertion routine that inserts tasks in available communication delay time slots.

4. The Mapping Heuristic (MH)

A mapper is an algorithm that takes two inputs: 1) a description of the parallel program modules and their interactions, and 2) description of the target machine. It produces as output a Gantt chart that shows the allocation of the program modules onto the target machine processing elements and the execution order of tasks allocated to each processing elements. A Gantt chart consists of a list of all processing elements in the target machine and for each processing element a list of all tasks allocated to that processing element ordered by their execution time, including task start and finish times.

Our mapping heuristic modifies Kruatrachue's basic heuristic so it can handle communication delay between tasks assigned to heterogeneous processing elements in an arbitrary target machine topology. The insertion routine used in ISH can be easily plugged in MH. The time complexity of MH is $O(n^2)$ for a constant number of processing elements. We study the effect of interconnection topology on schedule, and in turn, on the performance of the parallel program on a specific parallel processing architecture.

Definitions

The length of a path in a task graph is the summation of all node execution times and edge communication delays along the path. The level of a node is defined as the length of the longest path from the node to the exit node.

Adam et al. [3] compared 5 different ways of assigning priorities: HLFET (Highest Level First with Estimated Times), HLFNET (Highest Level First with No Estimated Times), RANDOM, SCFET (Smallest C0-level First with Estimated Times), and SCFNET (Smallest C0-level First with No Estimated Times). He showed that among all priority schedulers, level priority schedulers are the best at getting close to the optimal schedule. Following the advice of Adam et al., we use the level at each node as its priority. However, after adding communication delay, the node level is not static and may change according to the mapping. Some researchers simply ignore communication delays in calculating the level at each node. We have studied both strategies for calculating the level: 1) with, and 2) without communication. The results of our study using hypercube target machines is given in section 5.

The ready time of a processing element P (ready_time[P]) is the time when processing element P has finished its assigned task and is ready to execute a new one. The message ready time of a task (Time_message_ready) is the time when all messages to the task have been received by the processing element containing the task. The speed up is defined as the program execution time when it runs on one processing element divided by its execution time when it runs on a multi-processor system.

MH Algorithm

The algorithm can be explained in the following three steps:

I. The level of each node in the task graph is calculated and used as each node's priority. (In case of a tie we break it by selecting the one with the largest number of immediate successors. If this does not break the tie, we select one randomly). A ready queue is initialized by inserting all nodes that don't have immediate predecessors. The ready queue is sorted according to their priorities, yielding the highest priority node, first, followed by lower priority nodes. Also, an event list is needed in step II, so an event list is initialized.

II. Then, as long as the ready queue is not empty: 1) a task is selected (dequeued from the front of the ready queue), 2) a processing element is selected to run the task. A processing element is selected in such a way that the task cannot finish on any other processing element earlier, 3) the selected task is allocated to the selected processing element, and 4) the time when the selected task will finish running on the selected processing element is added to the event list. Once the ready queue becomes empty, the event list is used to modify the status of the immediate successors of the finished tasks. So when a task finishes execution, the number of conditions that prevent any of its immediate successors from being run is decreased by one. When the number of conditions associated with a particular successor becomes zero then that successor node can be inserted into the ready queue.

III. Step II is repeated until all the nodes of the task graph are allocated to a processing element. (Fig. 1. gives the detailed algorithm)

```
Load the program task graph.
Load the target machine.
Compute the level of each task.
Initialize the ready_queue (Q).
Initialize the event_list (E).
repeat
        while Q is not empty do
                begin
                        get task (T) from Q.
                        locate_processor(T,P).
                        update the event list E
                end
        while E is not empty do
                begin
                        get event (event) from E.
                        process_event(event)
                end
until all tasks are allocated
```

Fig. 1a

```
procedure locate_processor(T,P)
begin
        P ← k.
        where finish_time(T,k) ≤ finish_time(T,i), i = 0, ..., N-1
end.
```

Fig. 1b

```
function finish_time(T,P)
begin
Let IMP be the set of all immediate predecessors of T.
        if IMP is empty then
                finish_time ← ready_time[P] + E(T,P)
        else
                Let IMP = { t1, t2, ..., tm }
                where ti is assigned to processor pi.
                Time_message_ready← max(ready_time[pi]
                + C(ti, T, pi, P)), i = 1, ..., m.
                start_time←max(Time_message_ready,
                                ready_time[P])
                finish_time ← start_time + E(T,P)
end.
```

Fig. 1c

```
procedure process_event(event);
begin
Let "task T is done" be the event to be processed.
Let IMS be the set of all immediate successors of T.
Let IMS = { t1, t2, ..., tm } where ti has ci associated with, where ci is the
number of conditions that prevents ti from starting execution (initially ci =
number of immediate predecessors of task ti);
        if IMS is not empty then
                for i := 1 to m do
                        begin
                                ci ← ci - 1;
                                if ci = 0 then
                                        insert ti into Q.
                        end
end.
```

Fig. 1d

Example

Fig. 2a shows a task graph consisting of 8 nodes (M = 8), where each node represents a task. The number shown inside each node represents its task number, the number to the left of a node i represents the parameter INS(i), and the number to the right of an edge (i,j) represents the parameter D(i,j). For example INS(1) = 5, D(4,7) = 5. Fig. 2b shows a target machine consisting of 4 processing elements (N = 4) forming a cube of dimension = 2. Notice that H(0,3) = 2, because a message sent from node 0 to node 3 takes two hops. Fig. 2c shows the Gantt chart that results from scheduling the task graph given in Fig. 2a on the hypercube given in Fig. 2b.

Fig. 3 shows the average speed up curves that resulted from scheduling 25 random task graphs, with average number of nodes = 60, average number of edges in the range (25 - 100), average execution time in the range (10 - 100) time units, and average amount of data at each arc in the range (10 - 100) data units, on the following target machine topologies: 1) fully connected, 2) hypercube, 3) mesh, 4) star, and 5) ring with transfer rate = 1 and number of similar processing elements in the range (2 - 64).

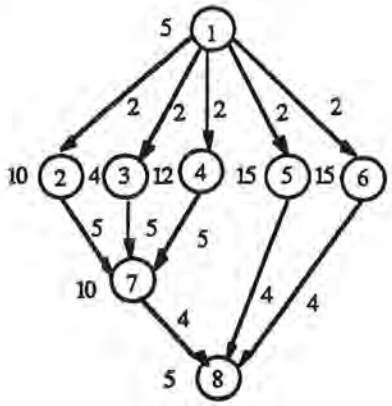


Fig. 2a (program task graph)

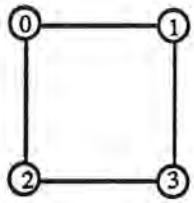


Fig. 2b (target machine)

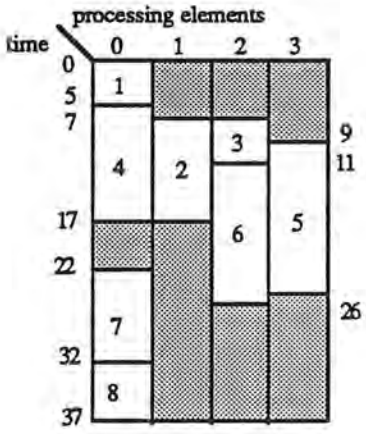


Fig. 2c (Gantt chart)

5. Case Study: Hypercube

In this section we show the results of some experiments we conducted using homogeneous hypercube architecture as a target machine. We generated 400 random graphs with average number of nodes = 50, average number of edges in the range (25 - 100), average task execution time in the range (10 - 100) time units, and average size of data at each arc in the range (10 - 100) data units. We ran each graph on hypercubes of size 2, 4, 8, 16 , 32, and 64 similar processing elements.

The first experiment shows the effect of using communication delays in calculating the level at each node in the MH algorithm.We found that 52.6% of the time using communication delays in calculating the level is better than not using them and the improvement is in the range (0.04% - 7.72%). In the 52.6% of improvements, most task graphs were "communication intensive". The performance was observed to be the same 10.8% of the time. We also found that 36.6% of the time not using the communication delays in calculating the level is better with improvement in the range (0.05% - 4.58%). Not all of these 36.6% were "execution intensive" task graphs.

The second experiment shows the effect of varying two program task graph parameters on the performance. We chose the parameters: 1) the average degree and 2) the C/E_ratio, defined as follows:

Average degree = number of edges / number of nodes.

Average C/E ratio = average communication delay between nodes / node average execution time.

We ran 80 random graphs at each average C/E_ratio value in {0.1, 0.5, 1.0, 2.0, 10.0} and 100 random graphs at each average degree value in {0.5, 1.0, 1.5, 2.0}. Fig. 4 shows the speed up curves at each C/E_ratio and Fig. 5 shows the speed up curves at each degree. The two sets of curves show the degradation in performance when the average communication delay between program tasks begins to dominate the average task execution time or when the number of edges in the task graph which represents the number of interactions among tasks dominates the number of tasks in the program. The curves also show that as the number of processing elements in the hypercube increases, the degradation of performance due to the increase in the C/E_ratio or the graph average degree is steeper.

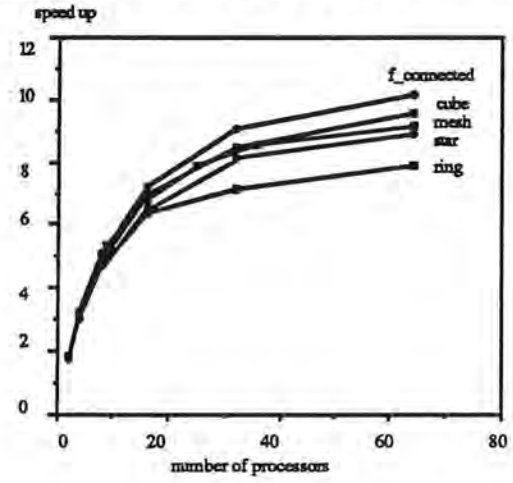


Fig 3

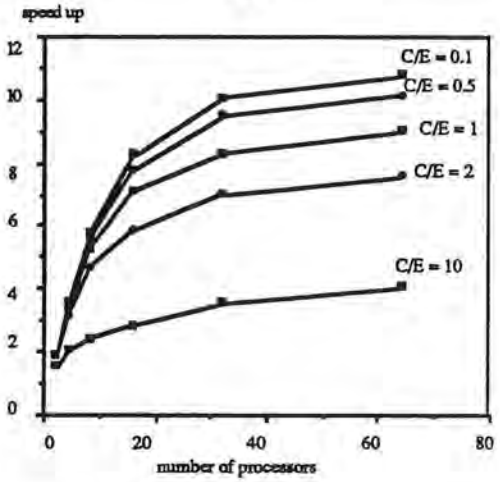


Fig 4

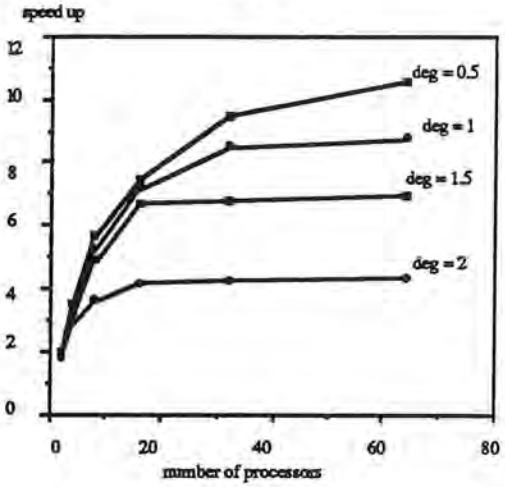


Fig 5

6. Conclusions

The results of this experiment are preliminary, but they suggest the following:

1. For communication intensive applications, the scheduler should consider communication delay in the scheduling algorithm's priority,

2. For computation intensive applications, priority scheduling is insensitive to the communication delays of the application.

3. Hypercubes perform better than meshes, star, and ring networks, but are not as high performance as fully connected networks. This is not surprising, and indeed, the performance of a hypercube as compared with a fully connected network is comparable.

4. Degree of nodes is a good indicator of the "amount of communication" in the task graph.

More work needs to be done to characterize parallel algorithms as task graphs, and to quantify the performance that can be expected from certain kinds of task graphs running on specific

network topologies. The significance of this work is that we can now begin to schedule task graphs onto multiprocessor systems in an optimal way by considering the target machine, communication delay, and the balance between computation and communication. MH is recommended for communication intensive task graph scheduling. MH does not consider network contention. This is left as an open research problem.

7. References

1. M. Gonzalez, "Deterministic Processor Scheduling," Computing Surveys, vol. 9, no. 3, September 1977.

2. B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processing Systems," Ph.D. Thesis, Oregon State University, Corvallis, Oregon, 1987.

3. T. Adam, K. Chandy, and J. Dickson, " A Comparison of list Schedulers for Parallel Processing Systems," Comm. ACM, vol. 17, pp. 685-690, December 1974.

4. T. Casavant and J. Kuhl, " A Taxonomy of Scheduling in General Purpose Distributed Computing Systems," IEEE Transaction on Software Engineering, vol. SE-14, no. 2, February 1988.

5. T. Hu, "Parallel Sequencing and Assembly Line Problems," Operation Research, vol.9, pp. 841-848, 1961.

6 E. Coffman and R. Graham, "Optimal Scheduling for Two-Processor Systems," Acta Informatica, vol. 1, pp. 200-213, 1972.

7. S. Bokhari, " A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in Distributed Processor System," IEEE Transaction on Software Engineering, vol. SE-7, no. 6, November 1981.

8. T. Chou and J. Abraham, "Load Balancing in Distributed Systems," IEEE Transaction on Software Engineering, vol. SE-8, no. 4, July 1981.

9. D. Towsley, "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," IEEE Transaction on Software Engineering, vol. SE-12, no. 10, October 1986.

10. J. Ullman, "NP-Complete Scheduling Problems," Journal of Computer and System Sciences, vol. 10, pp. 384-393, 1975.

11. V. Linnemann, "Deterministic Processor Scheduling with Communication Cost," Fachedaling Informatik Universitat, Frankfurt.

12. M. Prastein, "Precedence-Constrained Scheduling with Minimum Time and Communication," MS. Thesis, University of Illinois at Urbana-Champaign, 1987.

13. W. Chu, L. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," IEEE Computer, pp. 57-69, November 1980.

14. M. Chen, and K. Shin, "Embedment of Interesting Task Modules into a Hypercube Multiprocessor," Proc. Second Hypercube Conf., pp. 121-129, Oct. 1986.

15. V. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," proc. 4th Int. Conf. Distr. Comput. Syst., pp.30-39, May 1984.

SuperGlue: Integrating the Tools

Abstract

A desirable output from the Parallel Programming Support Environment (PPSE) tools is compilable source code. SuperGlue takes a program flow file produced from the program design, an architecture description, and C code fragments from other PPSE tools, and produces C-Linda code that can be compiled and executed on machines which support the C-Linda environment. This paper contains a discussion of the reasons for developing SuperGlue, along with an explanation of its functionality. We will also present a demonstration of the utility of C-Linda and examine possible future directions of research.

Introduction

The ultimate goal of this research is to create a system which can be fed disjoint, abstract descriptions of software and hardware and automatically merge the two into machine specific source code. SuperGlue takes a flow file (dataflow representation) of a parallel program (which is translated from the output of the Extended Large Grain Data Flow (ELGDF) design tool[6]), along with a target machine file, a gantt chart file (which represents task scheduling), and code fragments and generates a parallel application able to run on the described architecture.

Several difficulties inherent to this research are:

- handling communication between tasks,
- handling task synchronization,
- handling the scheduling of tasks,
- providing portability between architectures,
- dealing with variable scoping.

Currently, parallel programmers are forced to use vendor and architecture-specific parallel programming primitives to deal with communications between tasks and synchronization of tasks. This undesirable situation causes programmers to create code which is neither robust nor portable across a range of architectures. The job of coding task communication and synchronization is left to the programmer, who has to juggle low-level system calls with inadequate higher-level facilities.

The programmer must also determine the proper order to schedule tasks. Frequently, this schedule will be incorporated permanently into the code - no changes in task scheduling can take place unless the code is completely rewritten. The problem is not so much in the difficulty of translating an idea into working code, but in transforming the code to experiment with different possible execution courses.

Finally, the problem of variable scoping must be dealt with. This difficulty is easily remedied by adhering to a basic rule of software engineering: Do not use global variables. If you must use global variables, ELGDF has facilities to allow you to do so. Otherwise, all variables are local to a specific code fragment (procedure) unless needed by another fragment, and those variables needed by other fragments are passed by a

communication link (as specified by ELGDF). This may sound restrictive but as will be explained, just a few changes in programming habit, allows us to do some pretty neat things.

These problem are met head-on in SuperGlue research. The objective is to abstract the nuts and bolts of parallel machines and parallel programming primitives so that software designers need only worry about their applications and their parallel design. The specifics are dealt with in an automated source code generation phase.

C-Linda was chosen as the initial SuperGlue output source language because it provides a portable high-level approach to the problem of architecture-independent parallel programming by providing simple yet powerful commands. The hardware layer is abstracted off, providing the user with a stable platform that is available on many architectures. With little or no effort, a parallel program generated from the PPSE tools will be able to run on a variety of architectures.

PPSE Overview

Most of the forward engineering aspects of PPSE are shown in the following figure:

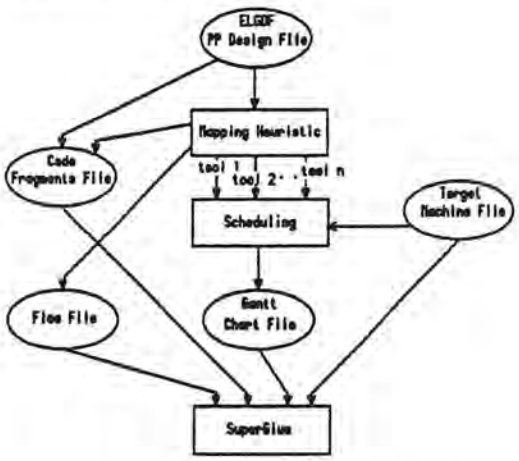


figure 1
Forward engineering portion of PPSE.

A user would use a program design tool such as ELGDF to design and describe their parallel program. When finished, ELGDF would save the users program in a large ASCII file called the PP design file (see table 1). Also, all code fragments that make up the users parallel program would be saved in a code fragments file. Next the PP design file would be input to the Mapping Heuristic tool (MH) which would take the users parallel program and break the code into pi blocks[1], expanding structures (i.e. unroll loops) where possible and limit the decomposition of the program by selecting a specific depth in the design. In addition, the MH will generate a flow file (see table 2) which contains the code fragment (task) connectivity and the variables to be passed between fragments. Also, the code fragment file must be updated with all changes made to the re-organization of code. Next the output of MH is passed to a scheduler. Which then schedules the code fragments onto the available processors using a user specified heuristic or algorithm. The end result of the scheduler is a gantt chart file (see table 3). At some point the user must use the target machine editor to describe the architecture that the newly designed parallel program is to run on. The resulting file is called the target machine file. Now the users parallel program is described in four different files: code fragment, flow, gantt chart, and target machine.

SuperGlue will take these four files and generate the users parallel program, which has been mapped onto the desired architecture, and place the new program into an output file called the SuperGlue file. The resulting SuperGlue file will then be FTP'ed (transferred via ethernet) to the desired target machine, where the application will be compiled, linked, and executed.

SuperGlue Overview

SuperGlue is not an interface design language like MatchMaker[8], but is a PPSE tool integrator that generates all necessary linkage between code fragments in the designed parallel program as per the specifications of the PPSE tools.

For the purposes of clarification, a code fragment is a procedure or program that can execute independently of other code fragments using local variables only except for those variables that are shared (i.e. passed from one task to another via a communication link), and since we are approaching the parallel programming from a large-grain dataflow point-of-view, all code fragments will be at the procedural level, not at the instruction level.

SuperGlue will not analyze the parallel code as an optimizing compiler would. We are relying on the efforts of the Mapping Heuristics (see figure 1), and the schedulers to have done all necessary validation and optimization.

Currently SuperGlue assumes that there are no cycles in the flow file and all data/control arcs point to code fragments of a larger number (nodes lower in graph), with the flow file used to determine the connectivity and the variables to be passed between the fragments. Because MH is not functional, SuperGlue takes the output of ELGDF (PP design file) and generates both a flow, and a code fragment file. At a later date MH will do this automatically (see figure 1). The gantt chart file is used to determine how the tasks should be bundled (grouped) onto separate processors.

The shared variables come in two flavors: input and output. A code fragment will need all input variables before executing and will output all output variables before terminating.

SuperGlue

SuperGlue uses the gantt chart, flow, target machine, and code fragment files to piece the parallel application together. The gantt chart file describes the order in which code fragments are to be executed on each processor and which code fragments are to be bundled onto which processors. The flow file determines code fragment connectivity and shared

variables. The target machine file specifies the target architecture, the number of available processors, and the base language (C or FORTRAN). The code fragment file contains the code pieces of the users parallel program.

With this information, a base language, and the inherent abilities of Linda for handling fragment communication and synchronization, SuperGlue generates a complete parallel application. If the user wants the parallel application to run on a different architecture, all that is needed is a change of the target machine (using the target machine editor) and re-running SuperGlue. A new parallel application will then be built for the desired architecture.

SuperGlue works by constructing a program with the users code fragments as procedures, and inserting the correct Linda statements at the beginning and end of each procedure. SuperGlue is different from other systems that append code into existing routines, in that SuperGlue builds from the ground up, using many pieces, to form a complete parallel application.

The nicest feature of Linda is that the hardware is totally abstracted away from the user, and by using PPSE tools including SuperGlue, even task communication and synchronization are abstracted away. The user is then left to only understand how to program parallel applications and how to use the tools made available through PPSE. All of the complexities are handled automatically.

However, a draw-back to using Linda as an integral part of SuperGlue is that there must be a Linda implementation for the hardware the user wishes to use.

Linda

The Linda parallel programming environment consists of a small number of operations that may be integrated into a conventional programming language, yielding a parallel programming dialect. A host programming language is extended with four basic operations (and two variant forms) that provide the programmer with simple mechanisms for accessing logically-shared object memory.

The common currency within the Linda environment is the tuple. A tuple is simply an ordered set of data, such as ("Any_number", 6, 13.89). Tuples are added to and removed from logically-shared memory, called tuple space, using the Linda tuple operations. The basic tuple operations on shared tuple space provide the necessary mechanisms for inter-process communication, process creation, and inter-process synchronization.

Communication is handled via operations that allow tuples to be added to and removed from tuple space. Tuple space provides a "bulletin board" style repository for data. Tuples are persistent objects, remaining in tuple space until removed. This approach provides communication free of the complexity associated with address and time based schemes.

Synchronization concerns are common in many parallel programming systems. The tuple removal and read operation implicitly handle synchronization concerns by blocking until an appropriate tuple becomes available. Predicate forms of these operations provide non-blocking behavior.

Tuple Space

The Linda parallel programming model is based on a logically-shared associative memory called tuple space (TS). Tuple space can be supported efficiently regardless of whether the underlying hardware includes physically-shared memory. Successful prototype implementations exist on networks of conventional uniprocessors, disjoint-memory multicomputers, and shared-memory multiprocessors.

Tuple space is an associative memory; there are no tuple addresses. Tuple lookup is similar to the select operation in relational databases. Tuples are selected by in() or rd() on the basis of any combination of their field values. Tuples are inserted into TS by using either out() or eval(). For example to place a three element tuple into TS we could execute out("data", a, b, c). To select or read this particular tuple from TS we could execute in("data", ?a, ?b, ?c).

Simple Example

As a first-cut for testing PPSE's usefulness, we want to examine the task of parallel programming by using programs that are made up of large grained, loosely coupled processes (each process can be developed independently of the others).

The first programming paradigm we wish to tackle is that of distributed data structures[5], where a group of identical worker processes access data structures simultaneously. The sample problem for this paper is to approximate pi using the rectangle rule[9] (see listing 1). When parallelized, this sequential program becomes basically a broadcast, calculate, aggregate (BCA) problem where we send (broadcast) separate intervals to a bunch of worker tasks who after doing their calculations, send the results back to a task who collects (aggregate) them. Using ELGDF the user might design the parallel version as follows:

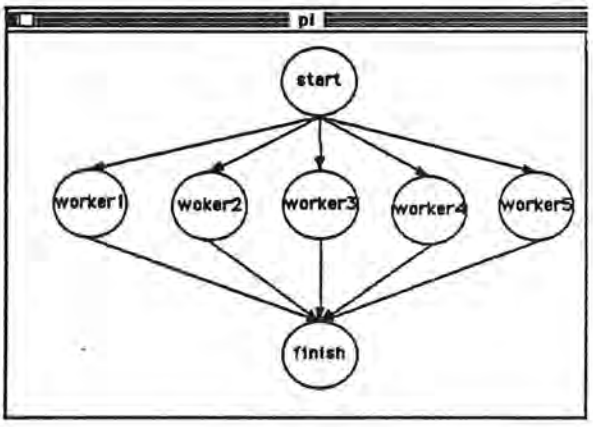


figure 2.

The user would then specify the data to be passed on the arcs. Figure 3 shows the variables being entered for the arc between the start node and the worker1 node. Likewise the user would fill in the data to be passed on the other arcs. Once the user had completed filling in the data arcs, the code for each node should then be entered into the simple nodes (see listing 2). Once all code is entered the user should save their current work using the facilities of ELGDF. Figure 4 contains a small portion of the ASCII file generated by ELGDF for the pi example.

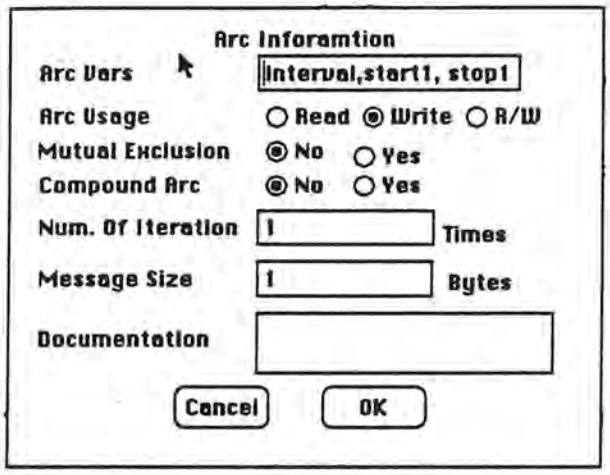


figure 3.

After saving, the user will want to schedule the new program using one of the scheduling tools. For this example lets use Kruatrachue's Insertion heuristic with two processors, which will produce the gantt chart in figure 5. The produced gantt chart should then be saved by the user for later reference and for use by SuperGlue.

Lets assume that the target machine file specifies a Sequent Balance as the target machine. Now that we have all necessary files, SuperGlue is executed with the result being similar to listing 2. This code can then be FTP'ed to the desired architecture, where it can be compiled, linked and executed.

For comparison, the sequential version took on average for an interval of 10000, 0.91 seconds. While the PPSE designed and generated version for the same interval with two processors took on average 0.63 seconds.

This small example with all its complexities has shown that it is possible to achieve a speedup by using parallel programming tools such as those found in PPSE, which sufficiently aid the user in dealing with the complexities of parallel programming.

```
$**Window Start**$
$SYMBOL START$
            Symbol Name: start
              Symbol ID:          0
            Symbol Kind: Node
       Symbol CompOrSimp: Simple
          Execution Time:          0
          Num Of Iteration0
         Symbol Rect Top:         11
        Symbol Rect Left:        197
      Symbol Rect Bottom:         61
       Symbol Rect Right:        251
   Symbol Documentation:
(Pre Object Name Start )
(Pre Object Name End)
$$ START    ARC INFORAMTION $$
(One Arc Information Start)
          Arc Variables: interval,start1, stop1
                 Arc  ID:          0
               Arc Kind: dataArc
        Arc Message Size:          1
          Arc Direction: PutData
         Arc CompOrSimp: Simple
    Arc Mutual Exclusion: FALSE
              Arc Count:          1
      Arc Documentation:
```

figure 4.
Portion of PP design file.

Current and Future Status

Currently we have several early versions of SuperGlue running on the Sequent Balance (written in C). However, since all tools for PPSE are Macintosh applications, we decided to do the same with SuperGlue. This will allow, in the future, a seamless design environment on the Macintosh.

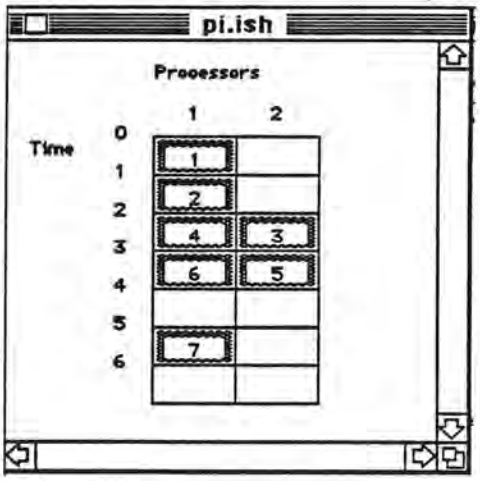


Figure 5.

Currently SuperGlue is under development on a Macintosh which generates code able to run on a Sequent Balance. Hopefully in the near future, we will be generating code for Intel Hyper-cube and Cogent architectures.

We need to test the concepts with actual parallel programs to gain further insight into potential methods and techniques.

References

[1] Allen, R., and Kennedy, K. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Prog. Lang. Syst.* 9, 4 (1988), pp. 491-542.

[2] Babb, R., and DiNucci, D. Design and Implementation of Parallel Programs with Large-Grain Data Flow. In *The Characteristics of Parallel Algorithms*, L. Jamieson, D. Gannon, R. Douglas (Editors), MIT Press, (1988).

[3] Berndt, D. C-Linda Reference Manual, Scientific Computing Associates, inc. New Haven, CT. Jan. 1989.

[4] Bjornson, R., Carriero, N., Gelernter, D., and Leichter, J. Linda the Portable Parallel, Yale University Department of Computer Science Research Report 520, (Feb. 1987).

[5] Carriero, N., Gelernter, D., and Leichter, J. Distributed data structures in Linda. In *Proceedings of the ACM Symposium Principles of Programming Languages* (St. Petersburg, Fla., Jan. 13-15, 1986).

[6] El-Rewini, H., and Lewis, T. Software Development in Parallax: The ELGDF Language. Technical Report 88-60-17, Oregon State University, Corvallis, (1988).

[7] Gelernter, D. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.* 7, 1 (1985), pp. 80-112.

[8] Jones, M., Rashid, R., and Thompson, M. Matchmaker: An interface specification language for distributed processing. In *Proceedings of the ACM Symposium Principles of Programming Languages* (New Orleans, La., Jan. 14-16, 1985).

[9] Karp, A. and Babb, R. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*, Sept. 1988, pp. 52-67.

APPENDIX - Program Listings

Listing 1. (sequential version)

```
/************************************/
/*          PI.CL                   */
/*  Sequential version  6/27/89  */
/************************************/
real_main()
{
 int i, start=1,interval;
 double scale, pi_approx = 0.0, rectangle_rule();

 interval = 10000;
 start_timer();
 scale = 1.0 / interval;  /* per calculate 1/n */

/* collect the partial sum */
 pi_approx = rectangle_rule(start,interval, scale);

 pi_approx = pi_approx * scale; /* scale result */
 timer_split("approx. calculated.");
 print_times();
 printf("pi approximation %20.15lf\n", pi_approx);
}

double rectangle_rule(start,stop,scale)
 int start, stop;
 double scale;
{
 int i;
 double x, rr_sum = 0.0;

/* do the summation over the given interval */
 for (i = start; i <= stop; ++i)
  {
   x = (i - 0.5) * scale;
   rr_sum += 4.0 / (1.0 + x * x);
  }
 return(rr_sum);
}
```

Listing 2. (parallel version)

```
/************************************/
/*          PI.CL                   */
/*  Parallel version  6/27/89  */
/************************************/

#include <linda.h>
#include <stdio.h>
real_main()
{
  int df0;
  int df1;
  int df3;
  int df5;
  int df6;
  int df2;
  int df4;
  int i;
  start_timer();
  scheduler();
  timer_split("All tasks completed");
  print_times();
} /* End of MainLine!! */
df0()
{
  int i,l=1,m,n, workers,processors;
  int start1,start2,start3,start4,start5;
  int stop1,stop2,stop3,stop4,stop5;
  double interval,h;

  n = 10000;
  interval = 1.0/n;

  start1=1;
  start2=2000;
  start3=4000;
  start4=6000;
  start5=8000;
  stop1=1999;
  stop2=3999;
  stop3=5999;
  stop4=7999;
  stop5=10000;
  out("F06", interval);
  out("F05", interval,start5,stop5);
  out("F04", interval,start4,stop4);
  out("F03", interval,start3,stop3);
  out("F02", interval,start2,stop2);
  out("F01", interval,start1,stop1);
} /* End of function df0*/
df1()
{
  int start1,stop1;
  int i;
  double interval,x,result1=0.0;

  in("F01", ?interval, ?start1, ?stop1);
  for(i=start1;i<=stop1;++i)
   {
    x=(i-0.5)*interval;
    result1+=4.0/(1.0+x*x);
   }
  out("F16", result1);
} /* End of function df1*/
df2()
{
  int start2,stop2;
  int i;
  double interval,x,result2=0.0;

  in("F02", ?interval, ?start2, ?stop2);
  for(i=start2;i<=stop2;++i)
   {
    x=(i-0.5)*interval;
    result2+=4.0/(1.0+x*x);
   }
  out("F26", result2);
} /* End of function df2*/
df3()
{
  int start3,stop3;
  int i;
  double interval,x,result3=0.0;
```

```
    in("F03", ?interval, ?start3, ?stop3);
    for(i=start3;i<=stop3;++i)
     {
      x=(i-0.5)*interval;
      result3+=4.0/(1.0+x*x);
     }
    out("F36", result3);
} /* End of function df3*/
df4()
{
    int start4,stop4;
    int i;
    double interval,x,result4=0.0;

    in("F04", ?interval, ?start4, ?stop4);
    for(i=start4;i<=stop4;++i)
     {
      x=(i-0.5)*interval;
      result4+=4.0/(1.0+x*x);
     }
    out("F46", result4);
} /* End of function df4*/
df5()
{
    int start5,stop5;
    int i;
    double interval,x,result5=0.0;

    in("F05", ?interval, ?start5, ?stop5);
    for(i=start5;i<=stop5;++i)
     {
      x=(i-0.5)*interval;
      result5+=4.0/(1.0+x*x);
     }
    out("F56", result5);
} /* End of function df5*/
df6()
{
    double result1,result2,result3,result4,result5;
    double h,pi_approx=0.0, interval;

    in("F56", ?result5);
    in("F46", ?result4);
    in("F36", ?result3);
    in("F26", ?result2);
    in("F16", ?result1);
    in("F06", ?interval);
    pi_approx=result1+result2+result3+result4+result5;
    pi_approx=pi_approx*interval;
    timer_split("approx. calculated.");
    printf("pi approximation %20.15lf\n",pi_approx);
} /* End of function df6*/
gant1() {
    df0();
    df1();
    df3();
    df5();
    df6();
    out("gant_done");
     } /*end of gantt chart 1 */
gant2() {
    df2();
    df4();
    out("gant_done");
     } /*end of gantt chart 2 */
scheduler()
{
    timer_split("gant1");
    eval(gant1());
    timer_split("gant2");
    eval(gant2());
    in("gant_done");
    in("gant_done");
}
```