COMPUTER ARCHITECTURE EVALUATION FOR STRUCTURAL DYNAMICS COMPUTATIONS

Final Technical Report

Project Summary

Principal Investigator:

Dr. Hilda M. Standley
Department of Computer Science

February 10, 1986 - August 7, 1989

The University of Toledo
Toledo, Ohio 43606

NASA Lewis Research Center

Grant Number: NAG 3-699

ABSTRACT

The intent of the proposed effort is the examination of the impact of the elements of parallel architectures on the performance realized in a parallel computation. To this end, three major projects are developed: a language for the expression of high-level parallelism, a statistical technique for the synthesis of multicomputer interconnection networks based upon performance prediction, and a queueing model for the analysis of shared memory hierarchies.

INTRODUCTION

Parallel computer architectures, both commercial and theoretical, are proliferating as the speed advantages of parallel computation are recognized. New architecture designs may be classified as fundamentally "traditional" in which an old design is given a slight modification or "radical" in which the standard approach is discarded in favor of an entirely new design. The fact remains that, because of the lack of a well developed, disciplined approach, computer architecture design today is very much trial and error. A design is produced and then evaluated to determine how good it is.

This study considers a variety of parallel architectures and selects two architectural elements having profound impact on performance, one from each of two diverse architectural classes. Models are developed by which the performance for these architectures may be predicted. In one case the performance of interconnection networks, described by graphical properties, may be predicted through a statistical analysis of the data collected about existing networks. In the other case, the performance of a shared memory multiprocessor with a memory hierarchy component is modeled analytically.

Relative to all large-grained parallel computation, a high-level parallel language, EASY-FLOW is developed to assist with the expression of parallel tasks in the context of traditional programming languages.

A HIGH-LEVEL PARALLEL LANGUAGE

Software for computers offering parallel computation must provide the level of parallelism specific for the target architecture, designating a point on the spectrum from a high-level multiple task model to low-level bit operations. The effort in this project is directed toward the former parallel model which is specific to the message passing, multicomputer architecture. A high-level parallel language is developed based upon the data flow schema of data-dependency directed execution, incorporating the three fundamental models of control directed execution: sequencing, branching, and looping.

Data flow computing is based upon the notion that the execution of a computation may be initiated by the availability of data, instead of by a sequence determined from the "flow of control." Data values "flow" between computations, triggering executions which consume input data and produce output data as results. Results that are produced at one computation may be consumed at a subsequent computation, establishing a data dependency between the two computations. Computations that are data dependent are constrained to execute in sequence. Other computations not so constrained may be executed in parallel.

The objectives of this language design project are to: (1) develop a language that requires little retraining of conventional language programmers, (2) provide for the reuse of existing software libraries, and (3) expose potential parallelism both implicitly and explicitly at varying levels of procedural computation. To this end the EASY-FLOW language is developed.

The basic unit of computation in EASY-FLOW is the atomic unit (atomic since it has no substructure) supplied by a subprogram written in a conventional high-level language (e.g. FORTRAN, C). The program notation provided by EASY-FLOW gives a superstructure located conceptually above the subprograms and relates them by explicitly expressed data dependencies. Units, other than atomic, may have a substructure consisting of other units related by data dependencies.

The EASY-FLOW notation provides information which may be used in scheduling the execution of units. Units which are not constrained by data dependencies may be scheduled to execute in parallel or overlapping in time. The data dependencies are made clear by the "single assignment" rule: any name in an EASY-FLOW program

is associated with only one value throughout execution. As an exception to this rule, the looping construct allows for the convenient update of a name used in iteration, but this may be done only in specifically isolated instances which are clearly marked in the program.

While the EASY-FLOW statements allow for the scheduling of units or tasks, the atomic units provide for the computation specified in the program. Data values as parameters are passed by assigning their values to actual parameter variables to be used in a subprogram call. Upon returning from the call, assignments are made from the returning parameters to EASY-FLOW variable names, thus shielding the EASY-FLOW variables from alteration within the subprogram.

An EASY-FLOW compiler has been written that produces sequential FORTRAN code (in order to determine feasibility) for use with FORTRAN subprograms. The data flow graph produced by the compiler is made sequential through application of a topological sort. A compiler to produce parallel FORTRAN code for a Transputer system is currently in progress.

MULTICOMPUTER NETWORK SYNTHESIS

Inter-task communication in a multiprocess computation may dominate processing time and determine in large part the performance realized. In a multicomputer system, the interconnection network linking the processing elements provides the pathways for messages passed between tasks residing on separate processors. An interconnection network that closely fits the pattern of interprocess communication will clearly assist in alleviating the communications overhead. The alternative situation, one in which the communications requirements of the application must be mapped to a dissimilar interconnection network by mapping multiple edges to single physical communication links or mapping single edges to paths passing through multiple processing elemnets, may cause delays due to resulting bottlenecks.

Previous interconnection network designs have incorporated a regular network which matches to a degree the pattern of intertask communications. The selection of the network structure has been an intuitive decision based upon the experience of the designer. As an alternative, this project examines the use of statistical and optimization techniques used in the modeling and synthesis of interconnection networks. This approach represents a way to compare elements of diverse interconnection network designs in a way that allows the synthesis of networks by selection of the best elements of existing designs and other, perhaps hybrid, networks that may offer better performance.

A multidimensional solution space is constructed by considering the performance (the dependent variable) of existing networks along with both quantitative and qualitative characteristics (the independent variables) of graphs. Such characteristics may include graph size, average degree, diameter, radius, girth, node-connectivity, edge-connectivity, minimum dominating set size, and maximum number of prime node and edge cutsets. Network performance may be described by the average message delay or the ratio of message completion rate to network connection cost. By using the method of stepwise linear regression, a polynomial surface is developed in the solution space. Optimization techniques such as response surface methodology or steepest ascent path may then be used to optimize the performance variable from the polynomial surface.

Screening of the relatively large number of independent variables may eliminate those that contribute little to the dependent variable value. An optimization technique is used to determine local or global points of "optimum" network performance. An "optimum" point is an indication of an "ideal" interconnection network, based upon the values of the various independent variables. The gradient vector for an optimum point which does not have corresponding realistically-valued independent variable values may indicate general trends or direction(s) of greatest increase in the value of the dependent (performance) variable.

The optimization process produces a ranking of desirable characteristics and their suitable levels. "Optimal" network synthesis will not follow directly from this. The information in the ranking will assist the designer in the design process, perhaps indicating unconventional directions in the choice of network elements.

QUEUEING MODEL FOR SHARED MEMORY HIERARCHIES

Interference between processors issuing requests to a shared memory may be a major factor in limiting performance in a shared memory multiprocessor system. Simultaneous requests to a single memory module cannot be serviced simultaneously. Only one request may be served, requiring the others to wait under some queueing scheme. Memory requests waiting in a queue translate to processors blocked from computation and a consequential degradation in achieved performance.

The queueing model presented is one for a hierarchy of memory modules. A hierarchy represents a realistic view of shared memory organization, with relatively small, high speed memories at the direct access level and larger, slower response memories organized at more remote levels of access.

An analytical model is developed, based upon a general queueing model. The mean waiting time for a request from a processor to be served at a memory module is calculated, including the time spent in a queue awaiting service and the time required to retrieve the data from the memory module. Queueing delay is based on an estimate of queue length and the average service time for a memory module access. From this the expected number of busy memories is computed and used as the measure of system performance. Analytic results are compared with simulated results for several systems differing in the relative numbers of processors and memory modules and the correlation found to be high.

APPENDIX A--EASY-FLOW GRAMMAR

Modified 2/1/89

1)	<program>	::=	<unit>
2)	<unit>	::=	unit <id> : <possible declarations> input : <list> <body of unit> output : <list> endunit <id>

Note: input and output are important enough, it was decided to require them even if no explicit I/O is called for.

Semantics: Make note of the unit id. Record declarations in symbol table (associate them with this unit). Record input list and output list, associated with this unit.

3)	<possible declarations>	::=	declare : <declarations list> | nil
4)	<list>	::=	<id> <more list> | nil
5)	<more list>	::=	,<id><more list>|nil
6)	<body of unit>	::=	<subprogram> | <if unit> | <iter unit> | <distribute unit> | <unit set>
7)	<subprogram>	::=	into : <pairs> <subprogram call> outof:<pairs>
8)	<if unit>	::=	if <boolean exp> then <unit set> else <unit set>

Note: <boolean exp> is treated as a subprogram call for now (see grammar). "Unitsets" used here because unit would require another input/output pair and this is already provided by the enclosing unit.

9)	<iter unit>	::=	iter <boolean exp> do <unit set> reassign <pairs>

Semantics: Process boolean expression same as above (see <if unit>).

10)	<distribute unit>	::=	distribute <id> = <range> <unit set>
11)	<unit set>	::=	<unit> <unit set> | nil

Note: a <unit set> may be nil.

12)	<boolean expression>	::=	<subprogram>
13)	<pairs>	::=	<match><pairs>| nil

Note: <pairs> may be nil.

14) <range> ::= <const> .. <const>

15) <match> ::= <variable id> => <variable id>

16) <subprogram call> ::= subprogram <id> <optional parameters>

17) <optional parameters> ::= (<svariable list>) | nil

18) <declarations list> ::= real <dvariable list> <declarations list>|
integer <dvariable list><declarations list>|
boolean <dvariable list><declarations list>|
double precision <dvariable list><declarations list>|nil

Note: The nil above allows declare: <nil>. This is OK to emphasize no declarations!

19) <dvariable list> ::= <dvariable id><more dvariable list>

20) <dvariable id> ::= <id><optional dimension list>

21) <more dvariable list> ::= ,<dvariable id><more dvariable list>|nil

---

22) <svariable list> ::= <variable id><more svariable list> |nil

23) <more svariable list> ::= ,<variable id><more svariable list>|nil

24) <variable id> ::= <id><optional subscript list>

25) <optional subscript list> ::= (<subscript list>)|nil

26) <subscript list> ::= <cid> <more subscript list>

27) <more subscript list> ::= ,<cid><more subscript list>|nil

28) <cid> ::= <variable id>|<const>

---

29) <optional dimension list> ::= (<dimension list>)|nil

30) <dimension list> ::= <const><more dimension list>

31) <more dimension list> ::= ,<const><more dimension list>|nil

---

Note: Three kinds of variable list are provided for:

1. Used in declarations and allow only constant dimensions.<dvariable list>
2. Used in input/output lists in units. For now, no subscripts are allowed.<list>
3. Other places allow subscripts.<svariable list> (most general).

Note: <variable id> allows any subscripts (not only constants).

<id> is any unsubscripted id (or simple id).

APPENDIX B--BIBLIOGRAPHIC REFERENCES

(Copies attached.)

"A General Model for Memory Interference in A Multiprocessor System with Memory Hierarchy," Badie A. Taha, Hilda M. Standley, 1989 International Conference on Parallel Processing, pp. I-225--I-232, August 8-12, 1989.

"Adapting High-Level Language Programs for Parallel Processing Using Data Flow," Lewis Structures Technology--1988, NASA Conference Publication 3003, Vol. 1, pp. 103--111, May 24-25, 1988.

"Modeling and Synthesis of Multicomputer Interconnection Networks," Hilda M. Standley and D. Steve Auxter, Technical Report, Dept. of Computer Science and Engineering, 1988.

"Multiprocessor Architecture: Synthesis and Evaluation," NASA Langley Workshop on Computational Mechanics, November 1987.

"A Very High Level Language for Large-Grained Data Flow," 1987 ACM Fifteenth Annual Computer Science Conference, pp. 191-195, February 1987.