MULTI- COMPUTER PARALLEL ARCHITECTURES FOR SOLVING COMBINATORIAL PROBLEMS [1]

W. M. McCormack [2]
F. Gail Gray
Joseph G. Tront
Robert M. Haralick
Glenn S. Fowler

Department of Computer Science
and
Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

I. INTRODUCTION

Combinatorial problem solving underlies numerous important problems in areas such as operations research, non-parametric statistics, graph theory, computer science, and artificial intelligence. Examples of specific combinatorial problems include, but are not limited to, various resource allocation problems, the traveling salesman problem, the relation homomorphism problem, the graph clique problem, the graph vertex cover problem, the graph independent set problem, the consistent labeling problem, and propositional logic problems [Hillier & Lieberman, 1979; Knuth, 1973; Kung, 1980; Lee, 1980]. These problems have the common feature that all known algorithms to solve them take, in the worst case, exponential time as problem size increases.

---

[1] This work was supported in part by the Office of Naval Research under Grant N00014-80-C-0689.

[2] The order of the authors was randomly chosen.

ISBN 0-12-564480-9

They belong to the problem class NP.

This paper describes a technique for the design of parallel computer architectures which most efficiently, or for the least cost, or for the smallest time to completion, execute parallel algorithms for solving these problems. The techniques we examine take into account the interaction between each specific algorithm and the parallel computer architecture. The class of architectures we consider are those which have inherent distributed control and whose connection structure is regular.

Combinatorial problems require solutions which do searching. In a very natural way, the algorithm for searching keeps track of what part of the search space has been examined so far and what part of the search is yet to be examined. The mechanism which represents the division between that which has been searched so far and that which is yet to be searched can also be used to partition the space which is yet to be searched into two or more mutually exclusive pieces. This is precisely the mechanism which can let a combinatorial problem be solved in an asynchronous parallel computer.

To help in describing the parallel combinatorial search, we associate with the space yet to be searched the term "the current problem". The representation mechanism which can represent a partition of the space yet to be searched can, therefore, divide the current problem into mutually exclusive subproblems.

Now suppose that one processor in a parallel computer is given a combinatorial problem. In order to get other processors involved, the processor divides the problem into mutually exclusive subproblems and gives one subproblem to each of the neighboring processors, keeping one subproblem itself. At any moment in time each of the processors in the parallel computer network may be busy solving a subproblem or may be idle after having finished the subproblem on which it was working. At suitable occasions in the processing, a busy processor may notice that one of its neighbors is idle. On such an occasion the busy processor divides its current problem into two subproblems, hands one off to the idle neighbor and keeps one itself.

The key points of this description are

(1) the capability of problem division

(2) the ability of every processor to solve the entire problem alone, if it had to

(3) the capability of a busy processor to transfer a subproblem to an idle neighbor.

The parallel computer architecture research issue is: to determine that way of problem subdivision which maximizes computation efficiency for each way of arranging a given number of processors and their bus communication links.

To precisely define this research issue requires

(1) that we have a systematic parametric way of describing processor/bus arrangements and

(2) that we have alternative problem subdivision techniques.

For the purpose of describing processor/bus arrangements, we use a labeled bipartite graph. The nodes are either labeled as being a processor or as being a bus. A link between a pair of nodes means that the processor node is connected to the bus node. We do not consider all possible such graphs but restrict our attention to regular ones. Regular means that the local neighborhood of any processor node is the same as that of any other processor node and the local neighborhood of any bus node is the same as that of any other bus node. As a consequence, each processor is connected to the same number of buses and each bus is connected to the same number of processors.

It may not be readily apparent why different problem subdivision techniques would influence computational efficiency. After all, the entire space needs to be searched one way or another. However, subdivision has an integral relation to efficiency. Processors which are not busy problem solving can be either idle or transferring subproblems. Too much time spent transferring subproblems will negatively affect efficiency. Excessive transferring of subproblems can occur because the subproblems chosen for transfer are too small. A good problem subdivision mechanism transfers large enough problems to minimize the number of times subproblems are transferred, but transfers enough subproblems to minimize the number of idle processors. The key variable of problem subdivision is, therefore, the expected number of operations it takes to solve the subproblem. This, of course, is a direct function of the size of the search space for the subproblem, the basic search algorithm, and the type of combinatorial problem being solved.

This paper addresses the interaction between the processor/bus graph and problem size subdivision transfer mechanism. Once the relationships are determined and expressed mathematically, the parallel computer architecture design problem becomes less of an art and more of a mathematical optimization. In addition, this paper examines the effects of interconnection graph regularity on the physical implementation of the system. The problem of finding a mathematical basis for a system partitioning which produces a cost-effective VLSI implementation is examined.

Our ultimate goal is to allow computer engineers to begin with the combinatorial problems of interest and determine via a mathematical optimization, the optimal parallel computer architecture to solve the problems assuming that the associated combinatorial algorithms, number of processors, number of buses, and costs are given.

II. PROCESSOR-BUS MODEL

In this section we discuss a processor-bus model which can be used to model all known regular parallel architectures [Anderson & Jensen, 1975; Benes, 1964;Batcher, 1968; Despain & Patterson, 1979; Finkel & Solomon, 1980; Goke & Gipouski, 1974; Rogerson, 1979; Siegel & McMillan & Mueller, 1979; Stone, 1971; Sullivan and Bashkow, 1977; Thompson, 1978; Wulf & Bell, 1972]. The model does not currently include the general interconnection and shuffle type networks.

The graphical basis for the model is a connected regular bipartite graph. A graph is bipartite if its nodes can be partitioned into two disjoint subsets in such a way that all edges connect a node in one subset with a node in the second subset. A graph is connected if there is a path between every pair of nodes in the graph. A bipartite graph is regular if every node in the first set has the same degree and every node in the second set has the same degree. One subset of nodes represents the processor nodes and one subset represents the communication nodes in the parallel processing system. Every edge in the graph then connects a processing node to a communication node.

At this time we are not certain exactly how to compare the costs of various parallel architectures. Certainly the number of processors (n_p) and the number of communication nodes (n_c) will affect the costs. It is generally believed that design and manufacturing costs can be reduced by building the global architecture using a systematic interconnection of identical modules. If the modules must be identical, then each module must have the same number of neighboring modules. In graphical terms, this means that the bipartite graph must be regular. Let d_p be the degree of the processor nodes. This parameter defines the number of buses which the processor may directly access. Let d_c be the degree of the communication nodes (buses). This parameter defines the number of processors that a communication node must service. If $d_c>2$, then either the communication nodes or the attached processors must possess arbitration logic to determine which processors have current access to the bus.

Any regular bipartite graph can be used to design a parallel computer structure by assigning the nodes in one set to be processors and the nodes in the other set to be communication links (or buses). Notice that theoretically either set of the bipartite graph could be the processor set. Therefore, each unlabeled bipartite graph would represent two distinctly different computer architectures depending upon which set is considered to be the processors and which set is considered to be the buses.

In systems for which $d_c=2$, (i.e., each communiations link is reserved for transferring information between two specific processors), it is customary to model the system as a simple graph in which each processor is represented by a node and each communications link by an edge. For example, the Boolean n-cube (shown in Fig. 1) has been studied by many investigators.

Our model would require that each edge in Fig. 1 be replaced by a communication node and two edges as shown in Fig. 2. Figure 2 clearly conveys the alternative node assignments. We may make the dark nodes processors and the light nodes buses, producing the Boolean 3-cube or we may make the light nodes processors and the dark nodes buses [Armstrong & Gray, 1980]. In this case, each processor has access to two buses and each bus services three processors.

This second architecture cannot be adequately represented by a graph in which each node is a processor and each edge a communiations link. If one considers the graph constructed by replacing every edge of the standard drawing of the Boolean 3-cube, by a vertex and connecting two vertices if and only if their corresponding edges were connected to a common vertex in the original graph, then the fact that communication among the three processors is restricted to a common bus is obscured by the resulting triangle structure which implies three independent communication paths.

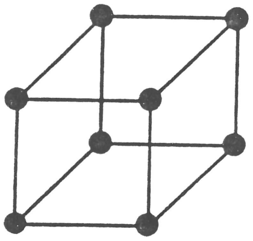

Figure 1
Common Representation
of Boolean 3-cube
processor Array

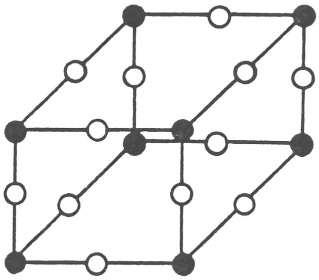

Figure 2
Bi-partite Graph
Representation of
Boolean 3-cube
Processor Array

The notation $B(n_p, d_p, n_c, d_c)$ will be used to denote a regular bipartite graph which represents an architecture with n_p processors (each connected to d_p communication nodes) and n_c communication nodes (each servicing d_c processors). The Boolean 3-cube will then be represented by a graph B(8,3,12,2). In general, the Boolean n-cube will be represented by a graph $B(2^n, n, n2^{n-1}, 2)$. Reversing the assignment of nodes to processors and buses produces the B(12, 2, 8, 3) graph of Fig. 2. This graph is called the p-cube by some investigators.

Other common architectures also have representations as bipartite graphs. For example, a planar array of size x^2 connected in the Von Neumann manner is modeled as a $B(x^2,4,2x^2,2)$ graph, the Moore connection results in a $B(x^2, 8, 4x^2, 2)$ graph, the common bus architecture (or star) with x processors is a B(x,1,1,x) graph, and a ring architecture with x processors is a B(x,2,x,2) graph. All existing architectures with regular local neighborhood interconnections can be modeled as a $B(n_p,d_p,n_c,d_c)$.

In addition to modeling existing architectures, we want to be able to generate new architectures. For example, all hardware architectures $B(n_p, d_p, n_c, d_c)$ with the same four parameters will probably have similar costs. Planarity considerations will have an effect of course, but we are not certain exactly how to quantify that cost. Neglecting planarity effects for now, we assume that two graphs with the same parameters will have similar costs. Several interesting questions arise immediately. How many bipartite graphs are there with a given set of parameters and how do we generate them? Assuming we can generate all or at least many graphs with a given set of parameters, are there other graph properties that will relate to performance evaluation?

We are currently doing simulations on similar graphs to determine whether any differences in performance occur and what the magnitude of these differences might be. The results to date are described in section IV. We plan to use simulation data to look for graphical properties that are useful in the prediction of performance. Certainly, graphs that are isomorphic must have similar cost and performance; however determining whether two graphs are isomorphic cannot be done in polynomial time. This problem is known to be equivalent to the problem of computing the automorphism partition for a given graph. We have been trying to develop efficient calculations for the automorphism partition of a graph because this partition has other performance implications.

For a given regular bipartite graph, all vertices will not necessarily be equivalent relative to algorithm performance. In particular, nodes that are more central (have more near neighbors) will perform differently than nodes that are more remote (have fewer near neighbors). The automorphism partition classifies vertices into similar performing sets. We have some simulation results that indicate the performance differences.

Another natural question asks whether a coarser partition than the automorphism partition might be used to classify performance. If so, this partition could be easier to compute. We have been experimenting with several such classification schemes based on distance properties. Regular bipartite graphs are said to be distance-1 regular since all vertices have the same number of similar vertices distance-1 away. If in a distance-1 regular graph, vertices also have the same number of vertices distance-2 away, the graph is distance-2 regular. Similarly, a graph is distance-K regular if it is distance(K-1) regular and all vertices have the same number of vertices distance -K away. We suspect that the question of how to optimally overlay masks to mass produce large chips is related to distance-K regularity as well. We are beginning to perform simulation experiments to collect data in an attempt to relate system performance to distance properties.

III. TREE SEARCHING IN MULTIPROCESSOR SYSTEMS

A. Introduction to Tree Searching

In order to make effective use of a multiple asynchronous processor system for any problem, a major concern is how to distribute the work among the processors with a minimum of interprocessor communication. Kung [Kung,1980] defines _module granularity_ as the maximal amount of computational time a module can process without having to communicate. Large module granularity is better because it reduces the contention for the buses and reduces the amount of time a processor is either idle or sending or receiving work. Also, large granularity is usually better because of the typically fixed overhead associated with the synchronization of the multiple processors.

In the combinatorial tree search problems we are considering, module granularity as defined by Kung is not as meaningful because each processor could in fact solve the entire problem by itself without communicating to anybody. For our problem a more appropriate definition of module granularity might be the expected amount of processing time or the minimum amount of processing time before a processor splits its problem into two subproblems, one of which is given to an idle neighboring processor and one of which is kept itself.

When a processor has finished searching that portion of the tree required to solve its subproblem, it must wait for new work to be transferred from another processor. The amount of time a processor must wait before transmission begins and until transmission is completed is time wasted in the parallel environment that would not be lost in a single processor system. Thus, one must expect improvement in the time to completion to solve a problem in the multiple processor environment to be less than proportional to the number of processors. The factors that can affect the performance by either reducing the average transmission time or reducing the required number of transmissions include choice of algorithm, choice of search strategy, and choice of subproblems that busy processors transfer to idle processors.

B. Choice of Algorithm

In the single processor case, various algorithms have been proposed and studied to efficiently solve problems requiring tree searches. These usually involve investing an additional amount of computation at one node in the tree in order to prune the tree early and avoid needless backtracking. In work on constraint satisfaction [Haralicck & Elliott, 1980], the forward checking pruning algorithm was found to perform the best of the six tested and backtracking the worst.

For the same reasons, it seems clear that pruning the tree early should be carried over to a multiple processor system to reduce the amount of computation necessary to solve the problem. There are other reasons as well. Failure to prune the tree early may later result in transfers to idle processors of problems which will be very quickly completed. Since a transfer ties up, to some extent, both the sending and receiving processor, time is lost doing the communication and the processor receiving the problem would shortly become idle.

We would, therefore, expect that in the multiple processor environment the forward checking pruning algorithm for constraint satisfaction would work much better than backtracking. However, in the uniprocessor environment Haralick and Elliott also showed that too much look ahead computation at a node in the search could actually increase the problem completion time. It is also not clear why the best algorithm for the single processor case would be the best for the multiple processor system. Doing additional testing, as some of the other algorithms do, may be better in the multiple processor case because it may eliminate more nodes in the tree earlier and result in less communication overhead and delay. Thus, it may be best to do as much testing early in order to eliminate future transfers in contrast to the single processor case where only some extra testing has been found to be worthwhile.

A second consideration in the selection of a search algorithm is the amount of information that must be transferred to an idle processor to specify a subproblem and any associated lookahead information already obtained pertinent to the subproblem. In most cases this is proportional (or inversely proportional) to the complexity of the problem remaining to be solved. Thus the transmission time will be a function of the problem complexity. Backtracking requires very little information to be passed while, for forward checking, a table of labels yet to be eliminated must be sent.

C. Search Strategy

Search strategy is a second factor of importance to the multiple processor environment. When a problem involves finding all solutions, like the consistent labeling problem, the entire tree must be searched. Thus, in a uniprocessor system the particular order in which the search is conducted, i.e., depth first or breadth first, has no effect. In a multiple processor system, however, this is a critical factor because it directly affects the complexity of the problems remaining in the tree to be solved and available to be sent to idle processors from busy processors.

A depth first search will leave high complexity problems to be solved later (that is, problems near the root of the tree). This would seem to be desirable in the multiple processor environment because passing such a problem to an idle processor would increase the length of time the processor could work before going idle and thereby reduce the need for communication. On the other hand, a breadth first search would tend to produce problems of approximately the same size. Since the problem is not completed until all processors are finished, the breadth first strategy might be preferable if it results in all processors finishing at about the same time. It might be that the best approach could be some combination of the two; for example, one might follow a depth first strategy for a certain number of levels, then go breadth first to a certain depth, and then continue depth first again.

D. Problem Passing Strategy

A factor closely related to the search strategy occurs when a processor has a number of problems of various complexities to send to an idle processor. The optimization question is how many should be sent and of what complexity(ies). Further complicating this is a situation where the processor is aware of more than one idle processor. In such a situation, how should the available work be divided and still leave a significant amount for the sending processor?

Further complicating this question is the fact that the overhead involved in synchronizing the various processors and transmitting problems to idle ones will eventually reach a point where it will be more than the amount of work left be done. An analogous situation exists in sorting; fast versions of QUICKSORT eventually resort to a simple sort when the amount remaining to be sorted is small [Knuth, 1973].

In this case, it would appear that a point will eventually be reached where it is more effective for a processor to simply complete the problem itself rather than transmit parts of it to others. Determination of this point will depend on the depth in the tree of the problem to be solved and the amount of information that must be passed (which depends on the lookahead algorithm being used).

E. Processor Intercommunication

One decision that has to be made is how the need to transfer work is recognized. Specifically, does a processor which has no further work interrupt a busy processor, or does a processor with extra work <u>poll</u> its neighboring processors to see if they are idle.

The advantage of interrupts is that as soon as a processor needs work, it can notify another processor instead of waiting to be polled. This assumes, however, that a processor would service the interrupt immediately instead of waiting until it had finished its current work. Furthermore, when a processor goes idle, it cannot know which of its neighbors to interrupt. This is related to handling multiple servers with a single queue which performs better than using one queue per server. Using polling, an idle processor can be sent work by any available neighboring processor instead of being forced to choose and interrupt one. In addition, although an interrupted processor may be working or transmitting (a logical and necessary condition) when interrupted, it may not have a problem to pass when it is time to pass work to the interrupting processor. In fact, the interrupted processor could itself go idle. For these reasons the simulation we discuss in section IV uses polling. Whenever a processor completes a node in the tree, and as long as it has work it could transfer, it checks each neighboring CPU and the connecting bus. If both are idle, a transfer is made.

IV. SIMULATION EXPERIMENTS

In order to better understand the behavior of the tightly coupled asynchronous parallel computer, we have conducted simulation experiments using the consistent labeling constraint satisfaction problem. The program SIMULA [Birtwistle, Myhrhaug & Nygaard, 1973] was used to perform these experiments. Let U and L be finite sets with the same cardinality. Let $R \subseteq (U \times L)^2$ with $\#R/\#(UxL)^2 = 0.65$. We use the simulated parallel computer to find all functions $f: U \longrightarrow L$ satisfying that for all $(u,v) \in UxU$, $(u,f(u),v, f(v)) \in R$.

The goals of the experiments were to investigate the effect of problem size (the cardinality of U and L), the algorithm, the problem passing strategy, and the number of processors. In the set of experiments we describe here, each processor is connected via buses to six other processors in a regular manner. Specifically processor i is connected to processor i-1, i+1, i-7, i+7, i-11, and processor i+11, all taken modulo the number of processors.

In the above regular architecture we varied the number of processors and measured for each execution of the same problem the average number of processors working. (where working means working on the problem and not sending or receiving problems or being idle). We compared the forward checking and backtracking algorithms for various problem passing and tree searching strategies.

Our results indicate the following using the forward checking algorithm with #U=#L. As we increased problem complexity, by increasing the cardinality of U and L, and while keeping the number of processors constant, the average number of working processors got closer and closer to the number of processors. This indicated that for small problem sizes the problem at a depth of three or four in the tree soon became so easy to solve that it was not passed to processors far away from the processor having the original problem. A greater processor interconnection could solve this but at a greater parallel computer cost.

In [Haralick & Elliott, 1980], the superiority of forward checking to backtracking is clearly shown with respect to the number of checks needed to solve a problem. This superiority grows as the size of the problem increases. For example, when the problem size is #U=#L=12, then backtracking requires 7.5 times as many checks, but by size 24 it is over 14 times as many. Table I shows a comparison of these two algorithms for a small problem (size =12) and 25 processors. Because forward checking requires fewer tests, the time the problem was completed is, as expected, much less. The magnitude of the difference is reduced because, on the average, more processors are working with backtracking. The problem is too small for forward checking to keep all processors busy.

Also of significance is the disparity in the number of problems transferred. Since forward checking, by looking ahead, eliminates impossible solutions earlier, there are fewer problems to be transferred. This reduces the dependency of the problem on the processor interconnection scheme and reduces overhead associated with transferring work. Similar results were obtained in testing the two algorithms on other small problems. Tests were limited because the backtracking experiments took considerably longer to run; for example, in the case above, backtracking took over thirty times longer.

TABLE I. Comparison of the backtracking algorithm to the forward checking algorithm with 25 processors and #U=#L=12

	Time Done	#Problem Sent	Ave. # working
Backtracking	140298	1666	23.998
Forward checking	37710	252	19.058

In addition to the number of processors or algorithm used, we tested other factors to study their effect. For example, when a processor has a choice of subproblems to transfer to another, intuitively the problem requiring the most work to finish should be transferred because the second processor will be longer which reduces overall communication time. In one test to confirm this, the subproblem requiring the least work instead of the most was always transferred. It took 70% longer to complete the problem and there was a 250% increase in the number of subproblems transferred. There was a corresponding drop in the average number of working processors.

Given that it is better to pass more complex problems, we conducted tests to compare doing the tree search breadth first or depth first. The results clearly show the advantage of a depth first search. The average completion time was typically at least 25% less and one fourth as many problems were transferred. Usually the differences were even greater. This improvement is achieved because depth first leaves larger problems available to transfer; thus processors sent work can work longer before becoming idle again.

A final issue in problem passing is whether or not very low complexity subproblems should be transferred at all. Table II shows the effect of not passing to other processors problems at or below a particular depth in the tree, using the forward checking algorithm.

TABLE II. The Effect of Not Sending Problems below depth d.

Depth d	Time done	#Prob. sent	Ave. working	Ave. sleeping	Expected # of consisdency checks to complete problem at this depth
3	144568	121	34.7	64.1	166.0
4	86059	671	56.7	32.3	81.7
5	85652	1730	56.3	16.5	39.6
6	89213	2238	53.3	13.8	22.2
7	93269	2426	51.4	15.0	15.0

There are 100 processors and #U=#L=16 with the forward checking algorithm, depth first search, and always passing the biggest problem.

For comparison the expected number of tests necessary to complete a problem at that depth, based on [Haralick & Elliott,1980] is included. The results clearly show that restricting problem passing for small problems can improve the performance of the system. Of course, too great a restriction will degrade performance because not all processors will be busy. It is not clear how the optimum value should be determined; although further comparisons or analytical and simulation results should provide insight.

V. EFFECTS OF REGULARITY ON SYSTEM IMPLEMENTATION

Advances in integrated circuit technology have led to VLSI circuits which have ten times the number of devices possible in LSI circuits [Foster & Kung, 1980; Mead & Conway, 1980]. This increase in the number of possible devices on a chip makes it more feasible to implement multiprocessor or multicomputer architectures [Sieworek & Thomas & Scharfetter, 1978]. The most significant of the recent advances in IC technology has been in the area of X-ray lithography. Laboratory circuits fabricated using X-ray lithography have been shown to be two orders of magnitude better than those built using optical techniques. X-ray lithography has also been shown to be ten times better than electron lithography and is in addition less expensive. Thus, depending on the circuit complexity, it may be possible to put as many as ten computing systems on a single chip.

Maintaining high speeds in a multiprocessor/multicomputer system is one of the prime reasons that it is necessay to use a higher density IC technology like VLSI rather than using multiple LSI chips. The inter-chip capacitance encountered in multiple LSI chip systems would prove to be extremely detrimental in a system where the number of processors is greater than ten.

In order to take the best advantage of VLSI technology and in order to be cost effective, it is necessary to design multiprocessor/multicomputer systems to be as geometrically regular as possible [Mead & Rem, 1979]. Regularity improves the ease with which a system may be designed and also leads to higher chip yield. Additionally, regularity in the hardware makes the system easier to expand, allowing the basic design to be useable in a broader set of applications.

Many different types of regularity can be incorporated in multiprocessor/ multicomputer interconnection networks. These types of regularity range from the global level, wherein the entire architechture is completely regular, (e.g., the Von Neumann or Moore array) to networks wherein the regularity is quite local(e. g. snowflake architectures). In a network which is globally regular, each element in the network will have an identical structure in terms of its capabilities and its interconnection facilities. Local or sub-global regularity implies that there may be more than one type of network element, each having different capabilities or interconnection facilities.

The degree of regularity is linked to the robustness of the interconnection network as well as to the overall system performance and cost constraints. Tradeoffs are therefore possible, which will produce the design of a system that meets the basic design objectives and is physically realizable.

Realization of a VLSI implementation of a large parallel processing system mandates much interplay between the system architects and the integrated circuit designers. A systematic approach to the problem is to divide the overall network up into small hardware subsections. This subsection division is best made at regularity boundaries. The ideal situation would be to have each hardware subsection identical to every other hardware subsection.

In the case where the subsections are not identical, the implementation details of one subsection should not strongly influence the details of other subsections. Some interaction between subsections may occur, especially if an entire system is to be implemented on a single chip. Examples of typical interaction include layout area overlap, crossing wires, and power consumption restriction. To maintain the integrity of a modular approach to the implementation, these interactions should be minimized as much as possible.

Once the individual subsections have been identified and their implementation details specified, the IC designer must decide on the type and number of subsection which can be placed on a single chip. This decision is of course highly dependent upon the limitation of the IC technology being used. Not only must the needed chip area and power dissipation be considered, but also, and most importantly, the amount of information which will need to be conveyed off-chip must be taken into account. (The need for finding an efficient means of transferring sufficient information to/from a VLSI circuit is a difficult problem which will require solutions different from those used with today's LSI technology.)

As an example of the types of subsection placement that might be found in a system, consider the Von Neumann array of computers. One VLSI realization might be as a two chip-type set. The first chip-type might be one which has five computers and their four associated interconnecting bus controllers on a single chip. The second chip-type would then consist of a single computer and four bus controllers. The Von Neumann interconnection would be formed by appropriately wiring each of the two different chip-types to form the complete array. See Fig. 3

Certain problems are obvious in this example. Probably the foremost is that the first chip-type will need to have the facility for connecting to twelve off chip buses. Although this may seem to be an unrealistic burden to have been placed on a single chip, it may be that the design objectives necessitated a more powerful computer in the array positions occupied by the second chip-type. Assuming that the more powerful computer takes up more semiconductor real estate, the tradeoff by the IC designer may be justifiable.

It can be seen in the above example that the regularity of the system contributes heavily to the ability to make subsection realization positioning tradeoffs at IC design time. In addition, system regularity generally reduces the problem of programming a large parallel processing system and greatly improves the overall understandability of the architecture.

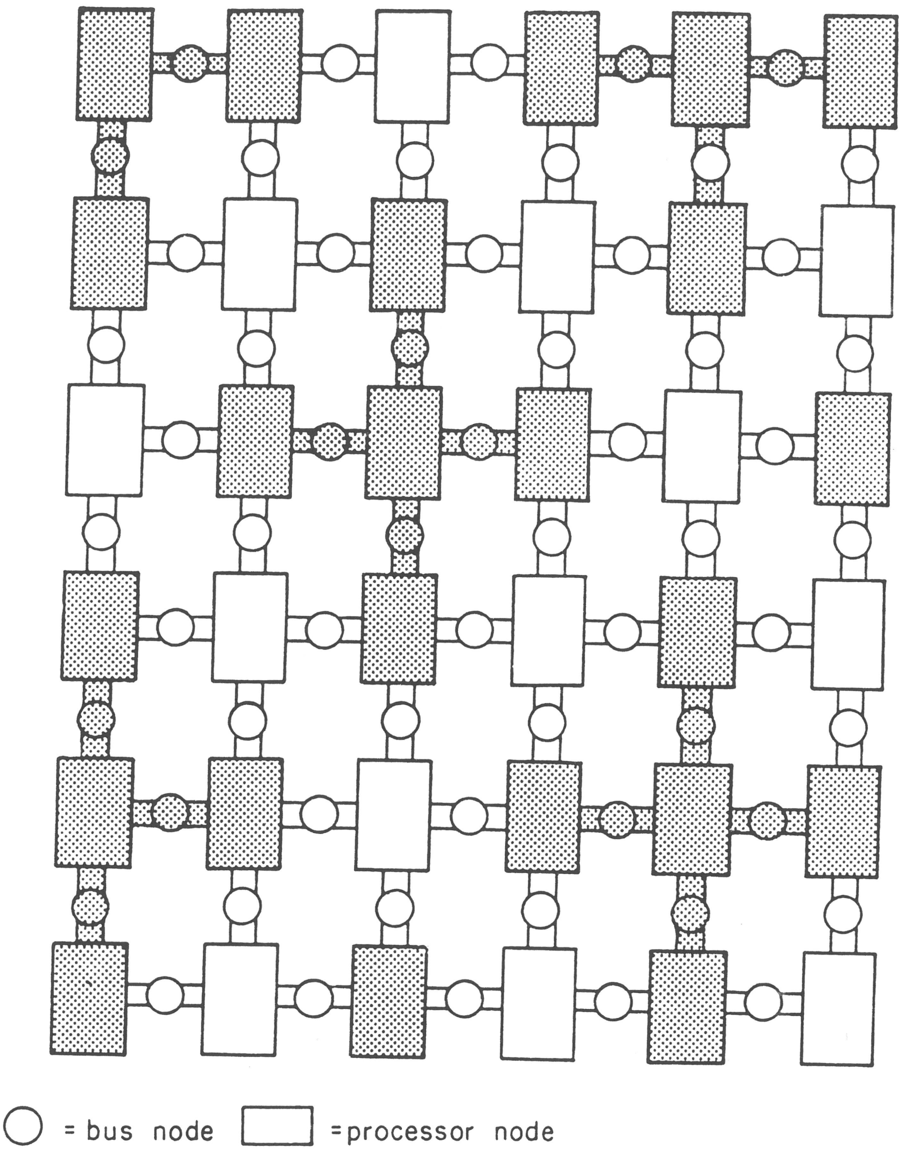

Fig. 3. Regular layout, VonNewmann Array.

REFERENCES

Anderson, G. A., and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," Computing Surveys, Vol. 7, Dec. 1975, pp. 197-213.

Armstrong, J. R. and F. G. Gray, "Some Fault Tolerant Properties of the Boolean n-Cube," Proceedings of the 1980 Conference on Information Sciences and Systems, Princeton, NJ, March 26-28, 1980, pp. 541-544.

Benes, V. E., "Optimal Rearrangeable Multistage Connecting Networks," Bell System Technical Journal, July 1964, pp. 1641-1656.

Batcher, K. E., "Sorting Networks and Their Applications," Spring Joint Computer Conference, 1968, pp. 307-314.

Birtwistle, G. M., Dahl, O. J., B. Myhrhaug, and K. Nygaard, SIMULA Begin, Auerbach Publishers Inc. Philadelphia, PA, 1973.

Despain, A. M. and D. A. Patterson, "X-Tree: A Tree Structured Multi-processor Computer Architecture," 5th Annual Symposium on Computer Architetura,architecture, 1978, pp. 144-151.

Finkel, R. A. and M. A. Solomon, "Processor Interconnection Strategies," IEEE Transactions on Computers, Vol. C-29, May 1980, pp. 360-370.

Foster, M.J. and H.T. Kung, "The Design of Special Purpose VLSI Chips", Computer, Jan. 1980.

Goke, R. L. and B. S. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," Proceedings of First Conference on Computer Architecture, 1974, pp. 21-28.

Haralick, Robert M. and G. Elliott, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", Artificial Intelligence, Vol. 14, 1980, pp. 263-313

Hillier, F. S. and G. S. Lieberman, Operations Research, Holden Day, Inc., San Francisco, 1979.

Knuth, D. E. The Art of Computer Programming, Sorting and Searching, Addison-Wesley Publishing, Reading, MA, 1973.

Kung, H. T., "The Structure of Parallel Algorithms," in Advances in Computers, Vol. 19, edited by M. D. Yovits, Academic Press, 1980.

Lee, R. B., "Empirical Results on the Speed, Redundancy, and and Quality of Parallel Computations," Proceedings of 1980 International Conference on Parallel Processing, 1980.

Mead, C. A. and M. Rem, "Cost and Performance of VLSI Computing Structures," IEEE J. Solid State Circuits, sc-14(2), pp. 455-462, 1979.

Mead, C. A. and L. A. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass. 1980.

Mirza, J. H., "Performance Evaluation of Pipeline Architectures," Proceedings of 1980 International Conference on Parallel Processing, 1980.

Rogerson, P.C. Fault Tolerant Networks of Microprocessors, Master Thesis, Virginia Polytechnic Institute and State University, March 1979.

Siegel, H. J., R. J. McMillan and P. T. Mueller, Jr., "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems," National Computer Conference, June 1979. Siewiorek, D. P., D. E. Thomas and D. L. Scharfetter, "The Use of LSI Modular in jComputer Structures: Trends and Limitations", Computer, July 1978.

Stone, H. S., "Parallel Processing with the Perfect Shuffle," IEEE Transactions on Computers, Vol. C-20, Feb. 1971, pp. 153-161.

Sullivan and Bashkov, "A Large Scale, Homogenous, Fully Distributed Parallel Machine," Proceedings of IEEE Computer Architecture Conference, 1977, pp. 105-124.

Thompson, C. O., "Generalized Connection Networks for Parallel Processor Intercommunication," IEEE Transactions on Computers, Vol. C-27, Dec. 1978, pp. 1119-1125.

Wulf W. A., and C. G. Bell, "C.mmp- A multi-mini-processor," Fall Joint Computer Conference, 1972, pp. 765-777.