A Reduction Architecture for the Optimal Scheduling of Binary Trees

K. RAVIKANTH, P.S. SASTRY, K.R. RAMAKRISHNAN and Y.V. VENKATESH
Department of Electrical Engineering, Indian Institute of Science, Bangalore, India

This paper addresses the problem of designing a parallel reduction architecture for applicative languages. An interconnection network that allows for scheduling of binary trees of arbitrary depth is presented. It is shown that using a static scheduling strategy the architecture achieves optimal performance while scheduling complete binary trees. Some issues related to the design of a machine based on this network are also discussed.

1. Introduction

Over the years, it has become increasingly evident that future solutions for high-speed computing cannot depend on technology alone. This realization has led to a restructuring of the notion of a computer, with the concept of parallelism emerging as the key to building faster machines. Thus, the design of parallel computers for either a special class of problems or general purpose applications has been the main thrust of research in architecture. The advent of VLSI technology has given these efforts a basis for practical realization.

For programming these parallel machines, many of the existing languages have, in fact, been extended to suit the parallel framework. But these efforts have not resulted in elegant parallel programming methods as these languages are closely tied to the Von Neumann computer model in their semantics. Backus [1] was apparently the first to realize this and he proposed the so called “applicative semantic model” as a basis for future language design as an alternative.

In the applicative semantic model, as is well known, a program unfolds recursively into subtasks giving it a tree structure. It follows from the Church–Rosser property [2] of these languages that the final result of the program is independent of the order in which these subtasks are evaluated. This property makes the applicative languages ideally suited to the domain of parallel computing since neither special techniques for extracting parallelism on the part of the compiler nor special efforts to express parallelism on the part of the programmer are required.

By virtue of this property of the applicative semantic model, several machines called reduction machines, have been proposed [10]. The main contribution of the present paper is the description of an interconnection network which may be employed to construct a reduction machine that efficiently executes applicative languages in parallel.

We first present an interconnection network which turns out to be a binary De Bruijn graph [9]. A static scheduling policy is presented using which equal load can be maintained on each processor when a complete binary tree task structure is distributed on this network. This is realized along with minimum possible interprocessor communication. We also present some results regarding the performance of the network under static scheduling for task structures which are not complete binary trees. Next, we explore certain issues related to the design of a machine based on the interconnection network presented. At the end, we make a comprehensive comparison of our work with related efforts in this area.

2. The Interconnection Network

Architecture design addresses itself to the task of configuring a physical structure that best meets the problem requirements. Since performance of a parallel architecture can be characterized mainly
by communication delays, distribution of load among processors and scheduling overheads, a close correspondence between the structure of the problem and the architecture is desired in order to minimize these overheads. Thus, in the context of parallel architectures, this essentially implies the need for the right choice of an interconnection network and a scheduling strategy which together can achieve the mapping of the problem on to the machine so as to minimize these overheads. When the problem structure is such that it recursively decomposes itself into identical subproblems, the architectural requirements can be formulated as follows:

Given a finite set of identical processors, what interconnection network and scheduling strategy give rise to a virtual tree structure of arbitrary depth? (*)

Before presenting the proposed interconnection network and the corresponding scheduling strategy, we identify certain ideal properties desired of any parallel architecture to be satisfied. An architecture with these properties minimizes overheads thus guaranteeing maximum possible speedup.

Properties for Ideal Performance

(P1) Minimum distance property: This property assures the minimization of communication in distributing subtasks and collecting partial results. This property may be formally stated as: If \( T \) and \( T' \) are two tasks from a task tree of a given problem such that \( T \) is the parent of \( T' \) and if \( P \) and \( P' \) are the processors \( T \) and \( T' \) are scheduled on respectively, then \( P \) should be directly connected to \( P' \) in the network.

(P2) Static scheduling: When a task spawns subtasks, there should be a simple static rule to decide which processors these subtasks may be executed on. This is desirable since it minimizes scheduling overheads and obviates the need for processors to exchange status information.

(P3) Minimum load imbalance: For a task structure that recursively unfolds as a tree, computation is active only at a single level—the level that is currently the lowest one. Since the tasks that are computationally active at any instant come from the same level, it is desirable that the tasks at every level of the task tree are distributed evenly among all processors in the network. This property may be formally stated as: If \( P' \) and \( P'' \) are any two processors belonging to the network, then \( |\text{load}_k(P') - \text{load}_k(P'')| \leq 1 \) (where \( \text{load}_k(P) \) denotes the number of tasks belonging to level \( k \) of the task structure that are scheduled to execute on \( P \)).

A network structure and scheduling strategy that can satisfy properties (P1)–(P3) for a given class of problems give us the best possible solution for the problem (*). It is not always possible to arrive at such a solution for every class of problems. For example, consider the problems that assume the structure of an arbitrary tree. It is impossible to configure a network that can satisfy properties (P1)–(P3). However, if we restrict ourselves to complete trees, it is possible to find solutions that satisfy all the three properties.

In what follows, we first present a network structure and a scheduling strategy satisfying properties (P1)–(P3) for task structures which assume the form of a complete binary tree and then generalize them for complete \( k \)-ary trees.

2.1. The Interconnection Scheme for Binary Trees

The interconnection network we present turns out to be a binary De Bruijn graph [9]. It has the property of a binary tree folding onto itself giving the appearance of infinite depth.

Let \( Q \) be a set of \( N \) identical processors. We can then define the interconnection network in terms of two functions \( L \) and \( R \), which map \( Q \) into \( Q \), as follows:

\[
\text{If } Q = \{ P_0, P_1, \ldots, P_{N-1} \} \text{ then,}
\]

\[
L(P_i) = P_{2i \mod N} \quad \text{and} \quad R(P_i) = P_{(2i+1) \mod N} \quad \text{for all } P_i \text{ in } Q.
\] (1)

The functions \( L \) and \( R \) establish links from the processor \( P \) to the processors \( L(P) \) and \( R(P) \), for each processor \( P \) in \( Q \). It turns out, as a result of the above definitions, that each processor has a connectivity of four in this network. Figure 1 shows, for instance, the interconnection network for eight processors.

The functions \( L \) and \( R \) are equivalent to the left-child and right-child of a processor respectively for any processor. By composing the functions \( L \) and \( R \), we can refer to processors that are not directly connected to a processor. For example, \( LRR(P) \) would refer to the left-child of the
right-child of the right child of P. By associating a direction with the arcs from P to L(P) and R(P), as shown in Fig. 1, we can evolve a simple static scheduling strategy which is described below:

**Scheduling strategy:** Let a task T on a processor P from Q spawn two subtasks T' and T'' then, a simple static scheduling (2) amounts to assigning T' to processor L(P) and T'' to processor R(P).

Figure 2 illustrates how an oversized problem, a complete binary tree with 31 tasks, gets distributed onto a network of eight processors using the static scheduling mechanism just described.

We now demonstrate that complete binary trees can be scheduled on this network rooted at any processor, satisfying the properties (P1)–(P3). (P1) follows from the fact that every task downloads the two subtasks it spawns on its immediate neighbours. (P2) is evident from the fact that scheduling is done using the strategy (2). (P3) follows from the theorem given below.

**Theorem.** The scheduling strategy (2) distributes tasks at every level of the complete binary tree in a round robin fashion among the processors in the network (1).

**Proof.** We shall prove by induction, for any level k of the task tree, that the tasks at that level are distributed in a round robin fashion among the processors in the network. Let us assume that the task at the root node is scheduled on processor P, for some P in Q. For notational simplicity we shall henceforth refer to a processor by its index.

Since there is only one task at level zero, it follows trivially that tasks at this level are distributed in a round robin fashion. Now, assume that the tasks at level k – 1 of the task structure are
distributed in a round robin fashion. Let \( j \) and \( (j + 1) \mod N \) be two consecutive processors in \( Q \) on which two consecutive tasks at level \( k - 1 \) have been scheduled. The tasks at level \( k \) spawned by these two tasks, according to the scheduling strategy, are assigned to the following processors:

\[
2j \mod N, \quad (2j + 1) \mod N, \quad (2j + 2) \mod N, \quad (2j + 3) \mod N
\]

From this it follows that tasks at level \( k \) are assigned in a round robin fashion since two consecutive tasks at level \( k \) are assigned to consecutive processors. □

**Remark.** The scheduling strategy adopted distributes tasks at every level of the complete binary tree in a round robin fashion among the processors in the network with the left most task at the \( k \)th level being assigned to the processor \( L^k(P) \) (\( P \) being the processor to which the task at the root is assigned). This implies that no two processors in the network can have a difference of load which is greater than one, thus leading to the minimum load imbalance property (P3).

### 2.2. The Interconnection Scheme for \( k \)-ary Trees

Though the interconnection network and scheduling strategy are specific to the case of the complete binary tree, they can be generalized to optimally schedule complete \( k \)-ary trees. The interconnection network can be understood in terms of \( k \) functions which map \( Q \) into \( Q \). The \( j \)th function of these, for \( 1 \leq j \leq k \), is defined as:

\[
\text{Child}_j(P_i) = P((k + j - 1) \mod N) \quad \text{for every } P_i \in Q.
\]

Every processor in \( Q \) has a connectivity of \( 2 \times k \) in this network. A static scheduling policy, on similar lines to the one described for the complete binary tree case can be evolved. According to this policy when a task on processor \( P \) spawns \( k \) subtasks, they get assigned to the following processors:

\[
\text{Child}_1(P), \text{Child}_2(P), \ldots, \text{Child}_k(P).
\]

A complete \( k \)-ary task tree can be distributed on the network described above using the static scheduling policy presented. Such a distribution is optimal since it can be shown, proceeding along the lines taken to prove for the case of the binary tree, that the properties (P1)–(P3) are satisfied.

### 3. Some Issues Related to Machine Design

The interconnection network presented in the previous section is the basis of our architecture which, as a consequence of the theorem, is ideal for evaluating complete binary tree structures. In this section we present some results regarding the performance of the network on a larger class of tree structured problems. We also present a method of reducing communication overheads in our architecture through the use of shared memories.

#### 3.1. Expected Speedup

We derive an expected speedup for complete binary tree task structures under the assumption that no structured data is passed between the processors. The task tree is assumed to unfold to a depth of \( n \).

For a task tree that unfolds recursively, each task has associated with it a distribution time \( t_D \) and an execution time \( t_E \). The time taken to execute the task tree then would be:

\[
T = (t_D + t_E) \ast (\text{load}_0 + \text{load}_1 + \cdots + \text{load}_n).
\]

Load on a processor due to the \( k \)th level of the task tree is given by

\[
\text{load}_k = \left\lfloor \frac{2^k}{2^m} \right\rfloor \quad \text{where} \quad m = \log_2 N
\]

Thus, it follows that:

\[
T = (t_D + t_E) \ast (n + 1) \quad \text{for} \quad n < m,
\]

\[
T = (t_D + t_E) \ast (m + 2^{n-m+1} - 1) \quad \text{for} \quad n \geq m.
\]

The distribution and execution steps exist even when the task tree is executed on a uniprocessor machine. The distribution step, in this case, corresponds to the creation of an activation record on a function call. Let \( t'_D \) and \( t'_E \) be the respective times taken for the distribution and execution steps. The time taken to execute the task tree in the uniprocessor case then would be:

\[
T' = (t'_D + t'_E) \ast (2^{n+1} - 1).
\]

Since \( t_D \) will be fractionally greater than \( t'_D \) in our architecture (as the extra information a task packet has to contain is only that regarding the destination processor) and \( t_E \) is equal to \( t'_E \), we get a speedup solely in terms \( n \) and \( m \) as follows:

\[
T' / T = (2^{n+1} - 1) / (n + 1) \quad \text{if} \quad n < m,
\]

\[
T' / T = (2^{n+1} - 1) / (m + 2^{n-m+1} - 1) \quad \text{if} \quad n \geq m.
\]
The speedup achieved is always greater than one and becomes asymptotic to $N$ as $n$ grows large when compared to $m$, that is, when the problem wraps around the network several times. Though this speedup is ideal, we expect the actual speedup to be quite close to this for problems that do not handle structured data since communication between processors can be made efficient by the use of shared memories as described later in this section.

3.2. Scheduling Arbitrary Tree Structures

Since our network is ideally suited for scheduling complete binary trees, a valid question that has to be answered is: “What is the basis for the choice of a binary tree, as an abstraction of the dynamic structure of an applicative program?”

Recursive algorithms having exploitable parallelism generally unfold as binary trees since they have a “divide and conquer” structure. Thus we believe, an architecture designed to efficiently execute binary trees is on a sound footing. This belief has been strengthened by the simulation studies conducted to investigate the performance of the network for other tree structures as well. These studies, though incomplete, indicate that the network has a tendency to smoothen out any imbalances in load among individual processors as the task tree wraps around it. The task structures considered for this investigation include cases of incomplete binary and ternary trees and the complete ternary tree.

An incomplete task tree is built using two random variables—one to decide whether a task should spawn subtasks or not, and another to decide the number of subtasks that should be spawned. By associating different probability distributions with these two random variables different kinds of trees can be generated.

When a task on a processor $P$ spawns subtasks, these subtasks are assigned alternately to the processors $L(P)$ and $R(P)$, the left-child and the right-child respectively of $P$. Such a scheduling mechanism is simple but it does not satisfy the condition (P3). The effectiveness of this scheduling mechanism can be understood by studying how the maximum load on a processor deviates from the ideal load, for every level of the task tree.

To this end we introduce the term “load-imbalance-factor” (lif) which is defined for every level of the task tree. It represents percentage deviation of the maximum load on a processor from the

![Fig. 3(a). lif vs. level no. for the complete ternary tree on the network sizes 8–256. lif peaks at (4, 700) and (5, 1100) (occurring outside the figure) for the network sizes 128 and 256 respectively.](image-url)
ideal load. The load-imbalance-factor for the $k$th level of the task tree is defined as:

$$\text{lif}_k = 100 \times \left( \frac{\max_i \{ \text{load}_k(P_i) \} - \text{ideal-load}_k}{\text{ideal-load}_k} \right)$$

where

$$\text{ideal-load}_k = \left( \text{load}_k(P_0) + \text{load}_k(P_1) + \cdots + \text{load}_k(P_{N-1}) \right)/N.$$
Here, $\lambda_k(P)$ stands for the load on processor $P$ due to the $k$th level of the task tree.

The simulation studies conducted on the tree structures mentioned above indicate that the load-imbalance-factor approaches zero asymptotically as one goes down the levels of the task tree. When the size of the task tree is relatively larger than the size of the network, it has been observed that the load-imbalance-factor approaches zero much faster.

Figures 3(a)–(d) represent the average behaviour of the load-imbalance-factor with respect to the level in the task tree for different randomly generated tree structures on various network sizes. It has been observed that the load-imbalance-factor shows a similar behaviour in all these cases, rising initially from zero to a peak and then reducing to zero asymptotically. The number of tasks, in those levels of the graphs where the load-imbalance-factor shows a rising trend, is less than the number of processors in the network. The load-imbalance-factor starts falling in these graphs once sufficient number of tasks are available in the network. It is also observed that for problems having a high degree of parallelism the load-imbalance-factor approaches zero quickly. This may be confirmed from the differences in behaviour which the graphs for the complete and incomplete ternary trees exhibit.

A dynamic load balancing scheme may be employed to reduce differences of load between processors which arise as a consequence of the static scheduling mechanism. Each processor looks at the load on its neighbours and downloads a task to a neighbour whenever the neighbour has a lesser load, thereby amounting to a distributed load diffusion process. The load diffusion process and the task execution process may be realized either on two different hardware units in the processor or may be multiplexed on the same hardware unit. From the available data, it is difficult to say conclusively whether such a dynamic scheduling scheme is necessary or not. It is conjectured that the extra complexity involved in implementing the dynamic scheduling scheme may not justify the increment in speedup that may be achieved through it. This aspect is currently being investigated.

3.3. Use of Shared Memories

In a multiprocessor system two steps are involved in the distribution of tasks. They are, first, the creation of a task packet, and next, the communication of this task packet over the network to the destination processor. The time required for distribution can be further reduced through the use of shared memories between neighbouring processors in our network. The use of shared memories is possible only because every new task that is created on a processor is scheduled to execute on an immediate neighbour. If such an
organization is employed, when a subtask is spawned, the task packet may be directly created in the memory of the destination processor.

Since each processor in the network has four neighbours, two connected to its incoming arcs and two to its outgoing arcs, it may appear at first sight that the contention problem would make the choice of shared memories undesirable. This problem, however, can be minimized by dividing the memory of each processor into three banks: left-memory, right-memory and local-memory. Here, the bank local-memory is local to each processor and contains all programs, relevant tables, etc. including a copy of the interpreter. Processors communicate through the memory banks they share. Figure 4 shows the interconnection of neighbouring processors through the memory banks they share.

When a task running on a processor \( P \) spawns two subtasks, the task packets for these subtasks are created in the left-memory and right-memory banks of the processor and are later directly accessed by the processors \( L(P) \) and \( R(P) \). Results of computation are also returned from the processors \( L(P) \) and \( R(P) \) to the processor \( P \) in a similar manner. When a processor’s memory is divided in this manner, no contention arises amongst the four neighbours of a processor. Contention arises only when two processors sharing a common memory make a simultaneous access on it. However, since each processor divides its attention between five memory units, and every shared memory bank can be accessed by only two processors, this contention is not significant.

4. Comparisons with Related Work

Minimizing interprocessor communication and maintaining an even distribution of load among processors are primary goals in any multiprocessor design, which are generally realized by judicious choice of an interconnection network and an efficient scheduling mechanism.

For programs that unfold as trees, we have shown that by using the interconnection network presented in this paper we are able to achieve a fairly good load distribution, along with minimal communication and no scheduling overheads. Though applicative programs having sufficient parallelism do unfold as tree structures, this property is not restricted to applicative programs alone but holds for any program written in a ‘divide and conquer’ fashion, irrespective of the underlying programming paradigm. Since our work has been motivated by the desire to configure architectural elements suitable to applicative languages, we compare our work mainly with other similar efforts.

Amongst parallel reduction machines Mago’s cellular tree architecture [3] that executes Backus’ FP is one of the pioneering works. The program is present in the leaf cells (occupying one symbol per cell) and the execution takes place in two phases. First comes an upsweep in which all reducible expressions are recognized, which is followed by a downsweep in which these expressions are rewritten by their values. In our architecture, unlike in Mago’s machine, computation first unfolds by building a tree structure in a top-down fashion followed by the reduction of the tree in a bottom-up manner.

AMPS is another tree structured machine which was proposed by Keller et al. [6]. In this machine the processors at the leaves perform computation and the processors at intermediate node positions perform routing and load balancing. Maintaining a uniform load in the leaf processors leads to an increase in communication which can get severely affected by the bottleneck at the root.
The REDIFLOW project [5] was a follow up of AMPS which incorporated a distributed load balancing scheme. In this scheme diffusion of load takes place along “pressure gradients” existing between neighbouring processors. The “pressure” at a processor is computed as a function of load on the processor, available memory, etc. Simulation results indicate that a speedup of up to 30 can be obtained with programs that use the “divide and conquer” principle on a system consisting of 128 processors. In our architecture, with a network of 128 processors alone in size, the same speedup can be obtained on a problem that unfolds to eight levels. We get increasingly better results as the problem size grows larger. In addition, unlike in REDIFLOW, the scheduling is static in our case.

The Zero Assignment Parallel Processor ZAPP, designed by Sleep and Burton [8], comes closest to our architecture in performance. The designers have chosen an indirect binary n-cube as the interconnection network in their machine. A static scheduling algorithm is employed to distribute a binary tree task structure on this network. This scheme spreads the nodes of the binary tree fairly uniformly among the processors in the network but fails to achieve optimal distribution in contrast with our network.

There have been other efforts, though unconnected to reduction architectures, to configure interconnection networks to execute binary tree task structures. Martin [4] proposed the twisted torus, which is a modification of the hypertorus. Sequin [7] further modified it to obtain the doubly twisted torus. These networks, through a static scheduling strategy, distribute binary trees assigning tasks at level k of the tree unevenly among k + 1 processors. Since the number of tasks at a level are related exponentially to the level number, these networks do not achieve optimal performance.

5. Conclusions

In this paper we have presented an architecture for implementing applicative languages efficiently. Our architecture minimizes communication overheads by the choice of an interconnection network which closely matches the problem structure. This network turns out to be a binary de Bruijn graph. For a class of applicative programs that unfold as complete binary trees, we are able to show that the network achieves an ideal performance using a simple static scheduling mechanism. This class of problems is significant because many “divide and conquer” algorithms would result in such binary tree structures. However, for problems that assume the form of arbitrary tree structures, the network is able to achieve a fairly good distribution of tasks using static scheduling. In fact, it is seen that the performance of the network improves as the problem size grows larger. It is shown that by use of shared memories between neighbouring processors one can reduce the interprocessor communication. Whether or not dynamic scheduling can be employed to improve the performance in a significant manner is currently being investigated.

References