Supporting Tasks with Adaptive Groups in Data Parallel Programming

John O'Donnell

***Abstract*— A set of communication operations is defined which allows a form of task parallelism to be achieved in a data parallel architecture. The set of processors can be subdivided recursively into groups, and a communication operation inside a group never conflicts with communications taking place in other groups. The groups may be subdivided and recombined at any time, allowing the task structure to adapt to the needs of the data. The algorithms implementing the grouping and communications are defined using parallel scans and folds which can be executed efficiently in an abstract tree machine. This approach is best suited for massively parallel systems with fine grain processors.**

***Index Terms*— Data parallel, task parallel, adaptive algorithms, processor groups, parallel scan, parallel tree machine.**

I. Introduction

Two common models of parallel computing are *task parallelism* and *data parallelism* [1]. As the names suggest, task parallelism organizes a program into independent pieces of work, while data parallelism provides concurrent operations over large aggregate data structures. A pure task parallel system is often supported by a coarse grain MIMD multiprocessor with a relatively small number of powerful machines, such as a cluster of workstations. A pure data parallel system may be supported by a fine grain SIMD multiprocessor, with a large number of small computers. Such systems include vector machines, massively parallel processors, and application specific VLSI designs.

Some applications can be handled effectively using a mixture of task and data parallelism. Examples include scientific computations with irregular structure and digital circuit simulation. A variety of approaches have been developed for supporting such a mixture in one system [2].

At the level of software, it is possible to make a distinction between task and data parallelism while supporting both in one system. The programming model TwoL [3] allows both modes of parallelism in one application, and offers methods for developing programs as well as predicting their performance. There are also data parallel languages that allow nested data parallel operations [4], giving some of the capabilities of tasks.

At the level of hardware, parallel systems are usually tuned to give best support for one form of parallelism. Coarse grain systems comprising a few processors connected by a network with high latency can execute programs with a small number of long-running tasks, but are generally inefficient for data parallel execution. At the opposite extreme, fine grain SIMD computers (e.g. the MPP [5] and Connection Machine [6]) are optimized for data parallelism but their SIMD organization precludes efficient task parallelism. Some parallel systems are intended to support both task and data parallelism, but their coarse granularity makes them less effective for fine grain data parallel computations than for tasks. An example is the Connection Machine CM-5, which has relatively powerful processors that can execute ordinary tasks efficiently and also incorporates some support in the interconnection network for data parallel operations. Much of this previous work aims to adapt a coarse-grained architecture with excellent support for parallel tasks so that it also offers limited support for data parallel operations.

This paper takes the opposite approach, by describing a technique for embedding a limited form of task parallelism in a computational model that is well suited for data parallel architectures. Each task runs in a group of computers, which pool their resources—both memories and processors—for the execution of that task. The processor groups may be subdivided recursively, and subgroups may recombined into larger groups. The grouping can be changed quickly, and there are no constraints on the sizes of groups or the degree of subdivision; this enables the application to adapt quickly to the demands of problems with irregular or dynamically changing structure. Ordinary data parallelism can be used at any time, giving the approach the characteristics of both task and data parallelism.

The grouping technique presented here is implemented using an abstract tree architecture with a SIMD organization. A variety of concrete implementations are possible, ranging from massively parallel processors, to FPGAs, to custom VLSI hardware designs. This paper focuses on the theoretical properties of the grouping algorithms and the cost model, so the use of an abstract architecture model simplifies and clarifies the analysis. The suitability of the approach presented here for a particular parallel architecture depends on the range of operations required for a specific algorithm and the frequency of their use.

In order to gain insight into the inherent efficiency of the grouping operations, it is helpful to analyze the performance on the weakest architecture that can support them, not the richest one. There is a tradeoff between the flexibility of a system and the speed of the operations that it provides. The choice in this paper of an abstract SIMD tree architecture allows a fast implementation of the adaptable grouping and communication operations that are presented below, and it provides a cost model that gives an accurate measure of the inherent cost of those operations. The abstract SIMD tree also has some drawbacks: it prevents some commonly used operations, including some of the standard MPI collective communications, from being supported efficiently. Naturally, it is possible to use the algorithms presented in this paper on a

Author's address: Computing Science Department, University of Glasgow, Glasgow G12 8QQ, Great Britain. Email: jtod@dcs.gla.ac.uk Web: www.dcs.gla.ac.uk/~jtod/

system with a richer interconnection network, or with a MIMD organization, in order to support the entire range of standard collective communications at the expense of worse latency in the grouping algorithms.

The algorithms in this paper are expressed using Haskell [7], a standard nonstrict pure functional language. The communications operations are defined as combinators: higher order functions without free variables that can express general patterns of computation.

Section II defines the abstract architecture used throughout this paper, and Section III discusses two methods for programming it: direct use of the low level sweep operations, and the definition of an intermediate level family of fold and scan combinators. The main innovation of this paper, adaptive processor groups that can support intragroup communications without interference among groups, is presented in Section IV. There are several ways to build a practical implementation of these algorithms, which are discussed in Section V. The programming paradigm proposed in this paper is unusual, and Section VI gives some examples of how it can be used. Related work is summarized in Section VII, and Section VIII concludes.

II. An abstract parallel tree machine

The system used in this paper is described abstractly as a network of processors whose structure is defined by a tree data structure, and whose behavior is defined by a global state transition function.

A. Structure of the machine

The system consists of a network of leaf processors connected by a tree of node processors. Each leaf processor has a local memory, but the node processors do not. It is *not* necessary for the tree to be balanced, although communication operations are faster if it is balanced. The system has an input and output port at the root node. The state of a leaf processor has a generic type s, but each specific algorithm supplies a concrete type for s.

$$\mathbf{data}\ \mathit{Tree}\ s = \mathit{Leaf}\ s \mid \mathit{Node}\ (\mathit{Tree}\ s)\ (\mathit{Tree}\ s)$$

B. Computation and communication steps

The behavior of the system is defined abstractly as a global state transformation operation called *sweep*. During a sweep operation, every processor reads an input on all of its ports, performs a local computation, and sends an output on all of its ports. All processors participate in every sweep, which acts as a global synchronization. Since the entire system is synchronized at the end of a sweep, the global state of the system can be specified precisely as the set of leaf processor states; this is written as $[x_0, \ldots, x_{n-1}] :: [s]$, where x_i is the memory state of processor P_i.

For the sake of simplicity, there are some useful generalizations and extensions of the abstract tree machine that are not presented here. The state types can be made more general, and two useful special cases of the sweep operation are omitted; these are *upsweep* and *downsweep*, in which information flows only up or down the tree respectively. With some concrete implementations of the tree machine, these restricted operations offer minor technical advantages. However, they complicate the algorithms because a node state needs to be introduced to make downsweep useful, and they do not improve the theoretical efficiency of the algorithms.

The sweep operation is defined as a global state transition function that takes a system state (the states of all the leaf processors) and a singleton input at the root node processor and returns the final global state along with a singleton output from the root. The behavior is determined by two individual processor state transition functions, one for the leaves and the other for the nodes. Since all the leaves execute the same leaf transition function, and all the nodes execute the same node transition function, the system belongs to the SIMD class of parallel computers.

As the sweep operation proceeds, messages of type u are transmitted up the tree, and messages of type d are transmitted down. The computation performed by a leaf processor depends on its state, with type s, and a message of type d received from the node above it. The processor computes a new state and a message of type u to send up the tree. This computation is specified by a leaf transition function of type *LTF*. Each node processor receives an input on each port, and computes an output to send on each port; the node computation is specified by a node transition function of type *NTF*.

$$\mathbf{type}\ \mathit{LTF} = s \to d \to (s, u)$$
$$\mathbf{type}\ \mathit{NTF} = d \to u \to u \to (u, d, d)$$

The most common operation performed by the tree machine is a *sweep*. During a sweep, each leaf processor function takes initial messages of type s from the processors, constructs up-packets of type u and sends them up the tree, combining them in the nodes. A singleton down-packet of type d is received from the root. Packets of type d then move down the tree, and at the interface to the processors are converted to data of type s which is placed in the local stores.

$$\mathit{sweep} :: \mathit{LTF} \to \mathit{NTF} \to d \to \mathit{Tree}\ s \to (u, \mathit{Tree}\ s)$$

The sweep function is defined recursively. The base case is just a leaf, in which the processor and network exchange messages, while the recursive case defines the behavior of a node as it communicates with its parent and two children.

$$\begin{aligned}
&\mathit{sweep\ leaf\ node\ a}\ (\mathit{Leaf}\ x) = \\
&\quad \mathbf{let}\ (x', a') = \mathit{leaf}\ x\ a \\
&\quad \mathbf{in}\ (a', \mathit{Leaf}\ x') \\
&\mathit{sweep\ leaf\ node\ a}\ (\mathit{Node}\ x\ y) = \\
&\quad \mathbf{let}\ (a', p', q') = \mathit{node}\ a\ p\ q \\
&\qquad (p, x') = \mathit{sweep\ leaf\ node}\ p'\ x \\
&\qquad (q, y') = \mathit{sweep\ leaf\ node}\ q'\ y \\
&\quad \mathbf{in}\ (a', \mathit{Node}\ x'\ y')
\end{aligned}$$

The definition of *sweep* is general, and allows arbitrary communication up and down the tree. In particular, for some specific node functions the *sweep* can deadlock. However, the algorithms presented later in this paper consist of an upsweep followed by a downsweep, and they do not deadlock.

III. Programming the tree machine

It is possible to program the tree network directly, by manually writing suitable *leaf* and *node* functions to solve a specific problem. A number of techniques can be used to write such programs, but this paper follows an alternative approach: we develop a set of combinators describing operations on lists (i.e. the set of processor states), and then seek efficient implementations of these combinators using the primitive tree operations. The main tools are the standard fold and scan functions, which can be used to express communications within processor groups. These functions are commonly used in parallel programming [8], [9].

With a SIMD organization, we can express the algorithm using *map* $:: (a \to b) \to [a] \to [b]$ to express a computation step. Let $f :: S \to S$ be a function that takes a processor state, performs a computation on it, and updates the state. The *map* combinator applies a function to each element of a list, so *map* $f\ [x_0,\ \ldots,\ x_{n-1}] = [f\ x_0,\ \ldots,\ f\ x_{n-1}]$. This describes the effect of a computation step at the level of the entire system, and it also captures the single-instruction constraint of the architecture. With an MIMD organization, it may be more convenient to express the algorithm in SPMD style, at the level of an individual processor.

A standard technique in SIMD algorithms is the use of conditional operations. Each instruction may have a field indicating that it should be executed by a processor only if a specified Boolean value is True (or False). Several algorithms that appear later in the paper use this technique.

The *foldl* and *foldr* combinators take a function f of two arguments, and use it to combine the elements of a list in order to produce a singleton result. An initial accumulator argument provides a default value in case the list is empty.

$$\textit{foldl} :: (a \to b \to a) \to a \to [b] \to a$$
$$\textit{foldr} :: (b \to a \to a) \to a \to [b] \to a$$

The result of the fold is a single value which is computed by n applications of f, where n is the length of the list. The difference between *foldl* and *foldr* is the order in which they iterate over the list, which leads to the following results:

$$\textit{foldl}\ f\ a\ [x_0, x_1, x_2, x_3] = f\ (f\ (f\ (f\ a\ x_0)\ x_1)\ x_2)\ x_3$$
$$\textit{foldr}\ f\ a\ [x_0, x_1, x_2, x_3] = f\ x_0\ (f\ x_1\ (f\ x_2\ (f\ x_3\ a)))$$

An alternative notation for these equations uses an infix operator $\oplus$ in place of the function:

$$\textit{foldl}\ (\oplus)\ a\ [x_0, x_1, x_2, x_3] = ((((a \oplus x_0) \oplus x_1) \oplus x_2) \oplus x_3)$$
$$\textit{foldr}\ (\oplus)\ a\ [x_0, x_1, x_2, x_3] = x_0 \oplus (x_1 \oplus (x_2 \oplus (x_3 \oplus a)))$$

The folds are specified as linear recursions over the list, so they are both sequential. However, if f is associative then the folds can be performed in logarithmic time by the tree machine.

$$\textit{foldl}\ f\ a\ [\,] = a$$
$$\textit{foldl}\ f\ a\ (x : xs) = \textit{foldl}\ f\ (f\ a\ x)\ xs$$

$$\textit{foldr}\ f\ a\ [\,] = a$$
$$\textit{foldr}\ f\ a\ (x : xs) = f\ x\ (\textit{foldr}\ f\ a\ xs)$$

The fold functions are useful because there is an extensive set of programming techniques that express a variety of computations as folds. Many of the standard iteration constructs from imperative languages, such as **for** and **while** loops, are often expressed in functional languages using the family of fold functions. Folds are commonly used for ordinary reductions of data, such as computing the sum of a list of numbers. In this paper they are used with much more complex auxiliary functions to perform operations on data structures and group communication.

A fold describes the behavior of the communications network as it takes inputs from a sequence of processors, and sends a result back to an individual processor P_i. Since the network actually sends a result to every processor during each communication operation, we usually need to compute a separate fold value for every processor in parallel. This set of folds is called a *scan*. A *scanl* is defined as a list of folds over all the prefixes of a list, while the *scanr* is the list of folds over the suffixes.

$$\textit{prefixes}\ [x_0, x_1, x_2, x_3] = [[\,],\ [x_0],\ [x_0, x_1],\ [x_0, x_1, x_2]]$$
$$\textit{suffixes}\ [x_0, x_1, x_2, x_3] = [[x_1, x_2, x_3],\ [x_2, x_3],\ [x_3],\ [\,]]$$

For example,

$$\textit{scanl}\ (\oplus)\ a\ [x_0, x_1, x_2, x_3] = \left[a,\ a \oplus x_0,\ (a \oplus x_0) \oplus x_1,\ ((a \oplus x_0) \oplus x_1) \oplus x_2\right]$$

The scans have similar types to their corresponding folds; the only difference is in the return type, which is a singleton for fold and a list (of fold results) for scan.

$$\textit{scanl} :: (a \to b \to a) \to a \to [b] \to [a]$$
$$\textit{scanl}\ f\ a = \textit{map}\ (\textit{foldl}\ f\ a) \circ \textit{prefixes}$$

$$\textit{scanr} :: (b \to a \to a) \to a \to [b] \to [a]$$
$$\textit{scanr}\ f\ a = \textit{map}\ (\textit{foldr}\ f\ a) \circ \textit{suffixes}$$

Ladner and Fischer presented a parallel algorithm that computes a *scan* f in one primitive step, provided that f is associative [10]. The tree architecture can be programmed to do this using a suitable sweep operation, with simple tree processor programs *leaf* and *node*:

$$tscanl, tscanr :: (a \to a \to a) \to a \to \mathit{Tree}\ a \to (a, \mathit{Tree}\ a)$$
$$tscanl\ f\ a = sweep\ leaf\ node\ a$$
$$\quad \textbf{where}\ leaf\ x\ a = (a, x)$$
$$\quad\quad node\ a\ p\ q = (f\ p\ q, a, f\ a\ p)$$
$$tscanr\ f\ a = sweep\ leaf\ node\ a$$
$$\quad \textbf{where}\ leaf\ x\ a = (a, x)$$
$$\quad\quad node\ a\ p\ q = (f\ p\ q, f\ q\ a, a)$$

The tree network provides more general communications than are required by either *scanl* or *scanr*. In particular, these scans are unidirectional, but the network can transmit information both left to right and right to left across the sequence of processors. The bidirectional *tscan* operation performs a *scanl* and *scanr* in parallel, and can be used to transmit messages between processors in both directions at the same time in the interconnection network.

For practical parallel programming, it is convenient to use slightly generalized versions of the parallel scans. The familiar list notation is used to express the global state of the system as a list of the leaf processor states; thus the state type is written as $[s]$ rather than *Tree s*. In addition, auxiliary functions are used to initialize a scan by extracting an up-message from a processor state, and to complete the scan by updating the final processor state using the down-message. The *pscanl* function has type

$$pscanl :: (s \to a) \to (a \to a \to a) \to (s \to a \to s) \to a \to [s] \to (a, [s])$$

An application has the general form *pscanl* $f\ g\ h\ x$ *xs*, where the arguments are:

- $f :: s \to a$ extracts a message of type a from each leaf processor with state of type s.
- $g :: a \to a \to a$ combines the messages during the scan. The function g must be associative.
- $h :: s \to a \to s$ updates the processor states using the results of the scan.
- $x :: a$ is the initial singleton data value; it can be thought of as a "horizontal" message available at the left side of the row of leaf processors, but it actually is provided by the control processor at the root of the tree.
- *xs* $:: [a]$ is the global state, represented as a list of the leaf processor states.

A logarithmic time operation on the tree network can compute the scans and folds of a function f over a list of processors, provided that f is associative. The singleton fold results computed by the tree do not include the extra accumulator value a which is used in *foldl* and *foldr*; these slightly modified combinators are called *foldl1* and *foldr1*. A proof appears in [11].

IV. Processor groups

The abstract tree architecture presented in the previous sections is well suited for conventional data parallel computation. For example, the leaf processors can perform arithmetic operations simultaneously on all the elements of a vector, while the tree network can compute reductions and scans. In this section, the programming model is extended to support a restricted form of task parallelism. The approach is to partition the set of leaf processors into a set of contiguous processor groups, and to provide a set of collective communication operations that allow the processors within a group to cooperate on a task.

The collective communications allow independent communications to occur within each processor group. They are implemented using sweeps and scans on the abstract tree architecture, and they are used by application programs. The collective communication operations described in this paper share an essential characteristic: the same operation must be performed simultaneously within all the processor groups (this is the SIMD restriction), but latency is good because there are no communication conflicts between distinct groups. However, more general collective communication operations could be supported by an abstract machine with an enhanced interconnection network, at the cost of longer communication latencies. These issues are discussed in more detail below.

A. Representing processor groups

The n leaf processors of the system are indexed to form a sequence $[P_0, \ldots, P_{n-1}]$. This set of processors can be subdivided into any number of *contiguous processor groups*; each group must consist of $[P_i, P_{i+1}, \ldots, P_j]$ such that $0 \leqslant i \leqslant j < n$. A group may comprise just one leaf processor, or it could consist of the entire set of processors in the full system.

The group representation is distributed. That is, no processor in the system has a global picture of the partitioning into groups; instead, each processor contains just enough information to know how to communicate with the other members of its current group. Each processor's local memory contains a *distance pair* that describes its position within the grouping structure. The current grouping status of the entire system is determined by the set of distance pairs of all the processors.

The elements of a distance pair give the distance from a processor to the leftmost and rightmost members of its group. Therefore a processor that is in the ith position (counting from 0) of a group with k members has a distance pair $(i, k-i-1)$. In particular, the leftmost processor has distance pair $(0, k-1)$ and the rightmost one has $(k-1, 0)$. If the current distance pair for P_i is (j, k), then P_i is currently a member of the group comprising processors $[P_{i-j}, \ldots, P_i, \ldots, P_{i+k}]$. For example, suppose there are 8 processors with two groups of sizes 3 and 5. The set of distance pairs would then be

$$[(0,2), (1,1), (2,0), (0,4), (1,3), (2,2), (3,1), (4,0)]$$

The elements of this list are distributed across the processors. The system is initialized as one group containing all the processors, using $[(0, n-1), (1, n-2), \ldots, (n-1, 0)]$.

The distance pairs are used to control the collective communication operations. If at any time the distance pairs in a group are corrupted, then the group representation is inconsistent and the communication algorithms will not function properly. A processor with distance pair (j, k) can send a message to the left by a distance of j and to the right by a distance of k, with the message remaining within the group.

To allow the grouping structure to adapt to the needs of the application, each processor maintains a stack of distance pairs, where the top element of the stack describes the current grouping. A group can be split into two or more subgroups. This requires each processor to compute a new distance pair for the subgroup it will belong to, and then to push this onto its stack. To recombine the subgroups into the original group, the processors pop their stacks. These operations are parallel local computations, although they require preparation by collective communication operations.

The following sections describe some of the collective operations that are supported by the abstract tree machine. To illustrate how the tree can perform these operations, the implementation of one of them—*leftmost*—is shown in detail.

B. Local computation

In a local computation, each processor updates its state using only data held in its own memory. Since there is no interprocessor communication, the grouping structure and distance pairs are irrelevant.

With the SIMD system organization presented earlier, all the processors are constrained to execute the same local computation. Conditionals are still possible, but the only conditional action a processor may take is to omit the operation being executed; the processor cannot do a different operation instead. Naturally, it is possible to generalize the system to an MIMD organization in order to allow the processors to perform different instructions during local computation steps.

In general, each processor applies a function $f :: S \to S$ to its state $x :: S$ in order to compute its new state $x' = f\ x$. The effect on the entire system is to compute a new global state *xs' = map* f *xs*, where the original state was *xs*.

C. Global operations

A separate control processor can communicate with the tree network through its root. This can serve as an input/output port for the system. The control processor may be a workstation that treats the tree network as a dedicated parallel processing system.

The control processor can broadcast data to all the leaf processors. Alternatively, it can broadcast a message (a, d) that tells P_a to store d. This requires each processor to know its index. As the system is initialized, the processors can all cooperate to compute their indices using the collective communication operations described later.

The tree network can perform reductions, or folds, using an associative function to combine values provided from all the leaf processors.

The control processor may need to fetch data from individual processors, for example to output results or to take a snapshot of the distributed memories. To read a value x from P_k, the control processor first broadcasts k. Each processor P_i then computes locally the value of $x' = (\ \textbf{if}\ i == k\ \textbf{then}\ x\ \textbf{else}\ 0)$. The set of x'_i is then reduced in the tree by folding it with a logical or, and the result x_k is produced at the root.

D. Unique local responders

The most critical collective communication operations are the ones that perform a computation involving communication among all the processors within a group. To illustrate the techniques required, a characteristic operation called *leftmost* is described in detail, along with its implementation using parallel map and scan. Briefer introductions to some of the other operations appear below. The *leftmost*, and its parallel implementation using a tree, was first used in APSA, an architecture for supporting list processing [12]. APSA supports a recursive decomposition of processors into groups, although the representation is based on the standard Lisp technique of "cdr-coding" rather than the distance pairs used in this paper.

Suppose that all the processors in a group have computed some local data values, and it is necessary to do further work on just those "responders" that satisfy some predicate. A function *rsp* $:: s \to$ *Bool* determines whether a processor is a responder, and the aim of the operation is to find the leftmost processor where *rsp* gives True. The situation can be depicted by showing a symbol $\circledast$ for the processors that are responders, and _ for the ones that do not. For example, the status of the processors might be

$$[\ \_\ \_\ \circledast\ \_\ \circledast\ \circledast\ \_\ \_\ \_\ \circledast\ \_\ ]$$

Since the flagged data values must be processed one at a time, a collective communication operation is needed to choose an arbitrary one. The *leftmost* operation can be used to select just the first processor in the group that has the flag set. Using the symbol $\boxtimes$ to denote this choice, the result of executing *leftmost* using the example above would be

$$[\ \_\ \_\ \boxtimes\ \_\ \circledast\ \circledast\ \_\ \_\ \_\ \circledast\ \_\ ]$$

The operation performs same work *simultaneously on all groups*. For example, if the system state is

$$[\ \_\ \_\ \circledast\ ]\ [\ \_\ \circledast\ \circledast\ \circledast\ ]\ [\ \_\ \_\ \_\ ]\ [\circledast\ \circledast\ ]\ [\ \_\ \circledast\ \_\ \circledast\ ]$$

then the result of a single *leftmost* operation is

$$[\ \_\ \_\ \boxtimes\ ]\ [\ \_\ \boxtimes\ \circledast\ \circledast\ ]\ [\ \_\ \_\ \_\ ]\ [\boxtimes\ \circledast\ ]\ [\ \_\ \boxtimes\ \_\ \circledast\ ]$$

There is a corresponding *rightmost* operation, and it is also possible to locate both the leftmost and rightmost flagged processor in each group in a single tree sweep operation.

The implementation of *leftmost* is presented in order to illustrate the programming style for the abstract tree architecture, as well as to provide a foundation for a cost analysis.

Since the aim is to use a *scanl* to implement *leftmost*, the starting point is to consider the auxiliary functions that are required. Two are needed to obtain information from each processor's local memory (of type s) to prepare the scan: $grp :: Groupings$ obtains the distance pair for a processor, while $rsp :: s \to Bool$ determines whether a processor satisfies the $\circledast$ condition. In effect, the scan sends a message of type a from the left side of the row of processors to the right side, and the processors update their state and modify the message as needed. This requires an associative function $combine :: a \to a \to a$ for the communication, and a function $upd :: s \to a \to s$ to update each processor's state, using the information provided by the message it receives during the scan.

It now remains to choose the message type a, the initial message value, and to define the *combine* function. To appreciate the subtleties involved in the implementation, it may be helpful to consider a simple but incorrect first attempt before giving the correct (but more complex) solution.

It is natural (but, as will become clear shortly, misleading) to view a *scanl* as an iteration across a list of elements, with each processor in turn receiving a message from the left and sending an updated message to the right. This suggests using a Boolean message to indicate whether a responder has been located yet within the current group: an initial message of False would be provided as input to the leftmost processor, and each processor would send on a message of True if it is marked, or False if

it is the leftmost processor in a new group (determined by the *leftmost* predicate). The first processor within a group (if any) that is marked and also receives an incoming False would set its ⊠ flag to True, and this happens at most once in each group. To achieve this, the initial message must be False, and the combine function must satisfy the following informal specification:

$$
\begin{array}{l}
\mathit{combine}\ a\ x = \\
\quad \textbf{if}\ \mathit{rsp}\ x \\
\qquad \textbf{then}\ \mathit{True} \\
\qquad \textbf{else if}\ \mathit{leftbound}\ x\ \textbf{then}\ \mathit{False}\ \textbf{else}\ a
\end{array}
$$

This leads to a well-formed function definition:

$$\mathit{combine}\ a\ x = \mathit{rsp}\ x \vee (a \wedge \neg\ \mathit{leftbound}\ x)$$

Unfortunately this approach is fatally flawed, because *combine* is not associative. This means that the logarithmic time parallel tree scan does not produce the same result as the linear time iteration across the row of leaf processors, which was the motivation behind this erroneous algorithm.

It is unproductive to cast around for an associative *combine* that might work; the best way forward is to observe that the tree sweep requires that *combine* must be able to compute (by folding) a summary of all the relevant information pertaining to a span of contiguous processors. Consider a span of contiguous processors consisting of two parts, *span1* ++ *span2*. A responder should be reported for the combined span if one was found in *span2*, or if one was found in *span1* and a new group did not begin in *span2*.

To implement this new idea, the message for the scan is defined to be a pair of Booleans (b, r), where b is True if and only if a group left boundary occurs in the span, and r is True if and only if a responder has already been found in the active (rightmost) group. A processor with state x computes these values as $(b, r) = (\mathit{leftbound}\ x,\ \mathit{rsp}\ x)$. The combine function is therefore defined as follows:

$$
\begin{array}{l}
\mathit{combine}\ (b_1, r_1)\ (b_2, r_2)\ =\ (b, r)\ \text{where} \\
\quad r\ =\ r_2 \vee (r_1 \wedge \neg b_2) \\
\quad b\ =\ b_1 \vee b_2
\end{array}
$$

It is necessary to verify that combine is associative; if it were nonassociative then the parallel scan computation in the tree would produce an incorrect result.

$$
\begin{array}{lll}
 & \mathit{combine}\ (b_1, r_1)\ (\mathit{combine}\ (b_2, r_2)(b_3, r_3)) \\
= & \mathit{combine}\ (b_1, r_1)\ (b_2 \vee b_3, r_3 \vee (r_2 \wedge \neg b_3)) \\
= & (b_1 \vee (b_2 \vee b_3), \\
 & \qquad (r_3 \vee (r_2 \wedge \neg b_3)) \vee (r_1 \wedge \neg (b_2 \vee b_3)))
\end{array}
$$

$$
\begin{array}{lll}
 & \mathit{combine}\ (\mathit{combine}\ (b_1, r_1)(b_2, r_2))\ (b_3, r_3) \\
= & \mathit{combine}\ (b_1 \vee b_2, r_2 \vee (r_1 \wedge \neg b_2))(b_3, r_3) \\
= & ((b_1 \vee b_2) \vee b_3, \\
 & \qquad r_3 \vee ((r_2 \vee (r_1 \wedge \neg b_2)) \wedge \neg b_3))
\end{array}
$$

Since $\vee$ is associative, $b_1 \vee (b_2 \vee b_3) = (b_1 \vee b_2) \vee b_3$ and the first parts of the pairs are equal. The second parts of the pairs can be verified algebraically or by model checking.

Now that the auxiliary functions have all been worked out, the *leftmost* operation can be defined as an executable function.

$$
\begin{array}{l}
\mathit{leftmost} :: \mathit{Grouping}\ s \to (s \to (\mathit{Bool}, \mathit{Bool})) \\
\quad \to (s \to (\mathit{Bool}, \mathit{Bool}) \to s) \to [s] \to [s] \\
\mathit{leftmost}\ \mathit{grp}\ \mathit{rsp}\ \mathit{upd}\ \mathit{xs} = \\
\quad \textbf{let}\ (\_, \mathit{xs}') = \mathit{pscanl}\ \mathit{rsp}\ g\ \mathit{upd}\ (\mathit{False}, \mathit{False})\ \mathit{xs} \\
\qquad g\ (b1, r1)\ (b2, r2) = \\
\qquad\quad (b1 \vee b2, r2 \vee (r1 \wedge \neg\ b2)) \\
\quad \textbf{in}\ \mathit{xs}'
\end{array}
$$

The *leftmost* computation has low latency because it is implemented by just one scan that takes logarithmic time using the tree network, as well as a small amount of local computation carried out in parallel by the leaf processors. The time to execute *leftmost* is independent of the number of groups, and there is never any time lost due to conflicts in the tree interconnection network. Regardless of the number, sizes, and locations of the groups, there is never any interference among groups that would require more than one scan to be performed in order to mark all the groups in parallel, and there is never any interference that slows down the scan. In particular, there is no need for the grouping structure to match the tree structure.

E. Finding predecessors and successors

A common data parallel programming technique is to identify a specific processor within a group, and then to perform some operation on all the processors within the group that lie to the right (or to the left) of the selected one. For example, suppose there are two subgroups, each containing one selected cell, which is determined by a Boolean indicated below with ⊠ .

$$[\ \_\ \_\ \_\ \boxtimes\ \_\ \_\ \_\ \_\ ][\ \_\ \_\ \_\ \boxtimes\ \_\ \_\ ]$$

The `mark_right` operation sets a second boolean variable, indicated with a ⊛ , in the processors that are to the right but still within the same group. The result is:

$$[\ \_\ \_\ \_\ \boxtimes\ \circledast\ \circledast\ \circledast\ \circledast\ ][\ \_\ \_\ \_\ \boxtimes\ \circledast\ \circledast\ ]$$

Similarly, if `mark_left` is performed on the original state, the result would be

$$[\circledast\ \circledast\ \circledast\ \boxtimes\ \_\ \_\ \_\ \_\ ][\circledast\ \circledast\ \circledast\ \boxtimes\ \_\ \_\ ]$$

These marking operations are useful in algorithms that use relative positions of data to represent significant information. One example is quicksort, where the location of a data item relative to the splitter is important. By marking all the processors in a group to the left (or right) of a selected member of the group, it is possible to operate in parallel on all the predecessors or successors of the selected processor.

F. Broadcasting within groups

The most complex of the group data parallel communication operations is broadcasting. A Boolean condition is used to identify a unique processor within a group which is requesting to send a message, and the network delivers the message to all the other processors within the group. The entire communication takes one machine step, and an independent broadcast may take place in any or all of the groups, with no conflict among groups. There are three variants of broadcast; the bidirectional broadcast sends the message to all other processors in the group, while *send_right* and *send_left* transmits the message only to the processors within the group which lie to the right or left of the sender respectively.

The implementation of broadcasting is similar to that of *leftmost*; the essential point is that a message carrying the data value and the remaining distance within the group is transmitted by the scan. The techniques used to derive an associative function that does this are similar to the ones used for *leftmost*.

G. Shifting within a group

In general, many-to-many communications operations require an interconnection network with a large number of independent paths. The tree network requires many steps in order to perform such an operation, and is generally better for one-to-many communications such as multi-broadcast. However, there is one many-to-many communication that can be performed efficiently in the tree: shifting. These operations allow all the processors in a group to communication in parallel, but only with their neighbors.

The shifting operations proceed in three steps. First, a computation step causes each processor to perform a fetch operation (called *fet*) on its local memory. The fetch returns either a value v to be transmitted to the neighbor (represented as *Just* v), or no value (represented as *Nothing*). The fetching step results in a list of messages, one for each processor, which is sent into the leaf nodes of the interconnection network. The actual communication is performed by a parallel scan, resulting in a sequence of messages to be returned to the processors. Finally, a computation step is performed where each processor uses a function *sto* that stores the incoming message into the processor.

$$\begin{aligned} &shift\_left, shift\_right :: (a \to Maybe\ b) \\ &\quad \to (a \to Maybe\ b \to a) \to [a] \to [a] \end{aligned}$$

The shift operations are straightforward to implement with a *pscanl* or *pscanr*. Some parallel tree machines, such as the FFP machine [13], provide direct links between neighboring leaf processors, so that the shift operations do not need to use the tree.

H. Group subdivision

Some of the intra-group communication operations have been introduced in the previous sections. These operations allow the processors within a group to cooperate on a computation, giving some of the characteristics of task parallelism within a data parallel framework.

The processor groups can be changed dynamically as a program is running. This allows the sizes of groups to adapt to the needs of the application. There is no need to restrict the group sizes; for example, there is no performance penalty if the groups do not match the sizes of subtrees in the system network. Furthermore, the latency required to change the grouping structure is very small.

A group may be subdivided around a selected pivot processor, which is determined by a predicate function. The `split` operation creates three subgroups comprising the cells to the left of the pivot, a group consisting of just the pivot cell, and a group consisting of the cells to its right. For example, the following system state contains a group with one processor selected by the predicate:

$$[(0,7),(1,6),(2,5),\boxtimes\ (3,4),(4,3),(5,2),(6,1),(7,0)]$$

Given this state, a *split* produces three groups:

$$[(0,2),(1,1),(2,0)],\ [(0,0)],\ [(0,3),(1,2),(2,1),(3,0)]$$

The subdivision operation requires a parallel scan to communicate information about the location of the pivot to all the members of the group, followed by a computation step in which each processor calculates its new distance pair and pushes it onto the stack.

There are minor variations on the `split` operation, in which the pivot cell does not form its own subgroup, but is included instead in either the left or right subgroup. It is also possible to partition a group into a larger number of subgroups, by selecting more than one pivot cell in advance.

I. Group recombination

After a group has been subdivided into several subgroups, these subgroups can be recombined back into the original group. This is performed as a local computation that must be performed in every cell belonging to the original group. Each of these cells removes the top element of its distance pair stack, reverting to the previous configuration. The algorithm must take care to ensure that all the cells of all the original group perform this operation; otherwise the distance pairs will no longer be consistent.

V. Implementation

A variety of methods exist for building an executable implementation of the abstract tree machine. This section describes a few of them, contrasting several choices between alternatives: control (SIMD or MIMD); network (physical tree or embedded tree); processors (general purpose, reconfigurable logic, or ASIC); and signaling (I/O between processors or direct combinational tree circuit).

a) Control.: During a sweep operation, the abstract tree requires all the leaf processors to perform one operation, and all the node processors to perform another. Thus the abstract architecture needs only a SIMD organization. However, a SIMD system places a severe constraint on the tasks that are running in separate groups. If an application has a heterogeneous structure, so that different tasks need to be able to run different code, then it would be best to embed the abstract system into a general purpose MIMD architecture. In effect, the grouping operations defined here would then just constitute a new set of collective communication operations. However, MIMD architectures generally have far more hardware per processor, resulting in a smaller total number of processors. It may be necessary to represent a large set of virtual processors on one physical processor, and this would cause a serious slowdown in the intra group communications. Furthermore, the latency of interprocessor communications is often higher in MIMD systems. To conclude, then, the algorithms presented in this paper could be used in either SIMD or MIMD systems, but they are probably better suited to fine grain architectures.

b) Network.: A significant result of this paper is that a tree architecture can be used to support intra-group communication *where the group structure does not match the tree structure*. Much past work on parallel tree algorithms has assumed that the work has to be divided into separate tasks to run on the left and right subtrees, in order to avoid a bottleneck near the root. Notwithstanding the good performance of the group operations presented here, it remains true that many applications require richer collective communication operations, such as scatter and gather, which cannot be implemented efficiently on a tree. In such cases, the algorithm should be run on a system with a more general interconnection network, and the tree communications used in this paper can then be embedded in this.

c) Processors.: The system could use conventional processor chips, but this is not the only alternative. Many SIMD architectures use special purpose VLSI processors so that many can be packed into one chip. Also, some data parallel algorithms are bit serial in nature, and do not need all the complex operations provided by standard microprocessors. A dedicated VLSI processor—even a bit serial one—can perform the group communication operations presented in this paper efficiently. A particularly interesting target is programming for FPGAs, which are fine grain SIMD machines. FPGAs are often described as programmable digital circuits, because their granularity is so fine. They offer easy support for data parallel computations, and allow efficient implementation of the fast tree sweeps, but FPGAs lack good tools for adaptive and task-based algorithms. These characteristics make them a good potential target for the algorithms presented in this paper. However, performance will normally be better if ad hoc communication algorithms are targeted for the FPGA's specific topology. The potential benefit in the group communication algorithms presented in this paper, on an FPGA host, is more likely to be portability rather than performance.

d) Signaling.: If conventional processors are used for the tree nodes, then the nodes will communicate with each other via input/output mechanisms. The time for a sweep is proportional to the height of the tree, and the latency for each level will be many microseconds. However, another alternative is feasible. The abstract tree can be built as a physical tree using VLSI combinational logic for the nodes [14]. This approach is especially appropriate if the leaf processors are also custom VLSI or configurable hardware. The latency would now be only a few gate delays in each level of the tree, and it would be reasonable to view the system as a "smart memory" rather than a conventional multiprocessor. In effect, the tree then corresponds to the address decoder tree in a single processor system, with one difference: the addressing hardware performs a significant amount of useful computation as well as simply decoding addresses, and the increase in latency is only a small constant factor. The situation is similar if the abstract tree is embedded in reconfigurable hardware, such as FPGAs. The interconnection topology of typical FPGAs provides long distance paths controlled by pass transistors, but these may require buffering to overcome the high capacitive load of long paths [15]. The tree nodes used in this paper could provide buffering while performing some useful computation with little extra overhead.

VI. Applications

The grouping techniques described in this paper constitute an unusual programming environment, and there is not yet a large amount of experience with it. There are some applications that have the characteristics that might make them suitable for this system, including active data structures for list processing, combinator reduction, and digital circuit simulation. In order to illustrate what an application for the system might look like, one example will be presented in some detail: a variation on the quicksort algorithm that uses the ability to split groups dynamically in order to adapt to the irregularities in real data.

A. Data parallel quicksort

This section illustrates the use of intragroup communication operations in a data parallel implementation of quicksort, assuming an SIMD system organization. A similar algorithm appeared in [16], although it used operations specially designed for the quicksort rather than a general set of group communication operations. A survey of parallel sorting algorithms targeted for a several architectures, along with analyses of their performance, is given by Akl [17].

The algorithm begins with a sequence of unsorted numbers in a group. It arbitrarily selects the leftmost element as the splitter and broadcasts it to the rest of the processors in the group, which compare their own value with the splitter. Each element that is less than the splitter, and therefore out of place, is marked with a Boolean flag. These values are then rotated to the leftmost position in the group, ensuring that they appear to the left of the splitter. The rotations involve sending the misplaced value to the left, while simultaneously shifting the intervening elements to the right. An important point is that a single scan can perform a rotation in parallel on every group that needs one (and leaves other groups unaffected). After all the data are on the correct side of the splitter, in every group,

the groups are then partitioned into subgroups and the whole process is repeated.

To show how the quicksort algorithm proceeds, an execution is sketched using the sample data $[13, 15, 10, 12, 14, 11]$.

The algorithm begins by initializing the group indices. To save space, the distance pair (i, j) is written below as $\frac{i}{j}$. Each processor (or "cell") holds several Boolean flags, the distance pair stack, and two data values in its local memory. The input data is read into the first value fields of the cells, the second data value is initialized to 0, and the leftmost cell (the one with left index 0) is marked as the splitter (denoted \$).

$$\begin{array}{lll} \textbf{[}\$\frac{0}{5}13, 0 & \frac{1}{4}15, 0 & \frac{2}{3}10, 0 \\ \frac{3}{2}12, 0 & \frac{4}{1}14, 0 & \frac{5}{0}11, 0\textbf{]} \end{array}$$

The splitter is broadcast to the rest of the group using the *send_right* operation. The cell selected by the the \$ flag is the sender, and all the cells in the group to its right receive the message and store it in their second data field.

$$\begin{array}{lll} \textbf{[}\$\frac{0}{5}13, 13 & \frac{1}{4}15, 13 & \frac{2}{3}10, 13 \\ \frac{3}{2}12, 13 & \frac{4}{1}14, 13 & \frac{5}{0}11, 13\textbf{]} \end{array}$$

The cells next perform a local computation: if their data value is less than the splitter (which is stored locally), they set a flag indicating that the value must be moved to the left of the splitter. This flag is indicated with a $\boxtimes$.

$$\begin{array}{lll} \textbf{[}\$\frac{0}{5}13, 13 & \frac{1}{4}15, 13 & \boxtimes\ \frac{2}{3}10, 13 \\ \boxtimes\ \frac{3}{2}12, 13 & \frac{4}{1}14, 13 & \boxtimes\ \frac{5}{0}11, 13\textbf{]} \end{array}$$

The algorithm performs a global or to determine whether there exists (in any group) a cell marked $\boxtimes$. There is such a value, so the algorithm rotates the first out-of-place value to the leftmost cell of the group. First, all the cells up to the leftmost starred value are marked with a flag, denoted $\circledast$ (using the `mark_left` operation):

$$\begin{array}{lll} \textbf{[}\circledast\ \$\frac{0}{5}13, 13 & \circledast\ \frac{1}{4}15, 13 & \boxtimes\ \frac{2}{3}10, 13 \\ \boxtimes\ \frac{3}{2}12, 13 & \frac{4}{1}14, 13 & \boxtimes\ \frac{5}{0}11, 13\textbf{]} \end{array}$$

Now the cells marked with $\circledast$ perform a rotation. The left endpoint of this span of cells is the one whose left index is 0; the right endpoint is the first cell with the $\boxtimes$ flag set. The rotation is performed by shifting the values (including the splitter flag \$) to the right by one position, while the rightmost value is sent back to the left and stored in the leftmost cell. The result of the first rotation is:

$$\begin{array}{lll} \textbf{[}\circledast\ \frac{0}{5}10, 13 & \circledast\ \$\frac{1}{4}13, 13 & \frac{2}{3}15, 13 \\ \boxtimes\ \frac{3}{2}12, 13 & \frac{4}{1}14, 13 & \boxtimes\ \frac{5}{0}11, 13\textbf{]} \end{array}$$

Such rotations take place independently, in parallel, in all subgroups that have a starred value. The rotations are repeated until no misplaced elements remain, and the result is:

$$\begin{array}{lll} \textbf{[}\frac{0}{5}11, 13 & \frac{1}{4}12, 13 & \frac{2}{3}10, 13 \\ \$\frac{3}{2}13, 13 & \frac{4}{1}15, 13 & \frac{5}{0}14, 13\textbf{]} \end{array}$$

At this point all values less than the splitter lie to its left, while all larger values lie to its right. The group is now subdivided using a `split` operation, which pushes a new index pair onto the index stack. Only the top pair in the index stack is shown, and the groups are emphasized with brackets, although the only representation of the group structure is the set of distance pairs.

$$\begin{array}{lll} \textbf{[}\frac{0}{2}11, 13 & \frac{1}{1}12, 13 & \frac{2}{0}10, 13\textbf{]} \\ \textbf{[}\frac{0}{0}13, 13\textbf{]} & \textbf{[}\frac{0}{1}15, 13 & \frac{1}{0}14, 13\textbf{]} \end{array}$$

The main step of the quicksort is now repeated. The crucial point about this algorithm is that each operation *acts on all subgroups in parallel*. For example, the next two operations are to identify the leftmost element of each group as its splitter and to broadcast the splitter to the rest of the group. These two operations produce the following result:

$$\begin{array}{lll} \textbf{[}\$\frac{0}{2}11, 11 & \frac{1}{1}12, 11 & \frac{2}{0}10, 11\textbf{]} \\ \textbf{[}\frac{0}{0}13, 13\textbf{]} & \textbf{[}\$\frac{0}{1}15, 15 & \frac{1}{0}14, 15\textbf{]} \end{array}$$

This algorithm requires an average time of $O(n)$ to sort an array of n numbers. The reason it is not faster is that the partitioning steps require a sequence of rotations; if these were performed in parallel a better complexity could be achieved. However, if this algorithm is implemented in a suitable technology, such as FPGAs or VLSI, the individual communication steps have extremely low latency and the overall performance may be good.

VII. Related work

Tree machines have appeared frequently in the literature on parallel architectures and programming. Introductions to parallel systems often depict a family of abstract interconnection networks, typically including a star, ring, tree, mesh, and cube. General discussions of tree machines may describe advantages such as a fixed number of ports per processor, and disadvantages including a potential bandwidth bottleneck near the root [18]. At this general introductory level, it may be assumed that the interconnection network is intended to provide arbitrary point to point communication between processors, and there is not necessarily an explicit connection between the network and the algorithms that run on it. Akl uses abstract tree architectures to analyze the complexity of several algorithm problems [19].

Tree networks are well suited for efficient fabrication, especially when the node processors are small enough to allow a large number of nodes per integrated circuit. The standard H-tree layout [20] gives an efficient usage of chip area, and each chip has a fixed small number of ports. This allows a system of arbitrary size to be constructed from a standard H-tree chip.

The main problem with a tree network is that the number of processors in each subtree increases toward the root, but the bandwidth does not. This can lead to a communication bottleneck between the two halves of the tree. Consequently, tree networks are poorly suited for general parallel computing where processors communicate randomly with each other.

One solution to the bandwidth problem is to choose problems whose communication needs match the architecture. A

common example of this is in computing reductions [21] and divide and conquer algorithms, where each problem is split into two subproblems of roughly equal size. In these applications, the links in the tree network are used primarily to communicate subproblem instances down the tree, and subproblem results back up, making efficient use of the available bandwidth.

Sometimes an algorithm for a parallel tree architecture can be mapped directly into hardware, rather than being programmed on a general purpose parallel system. In this case, there is no point in supplying an interconnection network that is richer than necessary. Any algorithm that happens to fit well in a tree structure—which, as pointed out before, is well suited for inclusion in a VLSI chip—is more cost effective with the tree network than with a "better" network. Two examples of this are associative processors and fast adders.

One of the first applications of a physical tree machine, where the network structure fits exactly the needs of the problem domain, is in the design of responder networks for associative processors. An associative processor [22] (also called content addressable parallel processor) is a data parallel system that provides operations for searching an aggregate data structure according to values, rather than addresses. In a typical application, the associative memory contains a list of tuples, with each tuple held in a separate local memory. A search operation would be issued by a single control processor, with a constant value for one element of the tuple specified as argument. All the processors then compare the argument with the corresponding tuple element in their local memory. Since many processors may find a match, the system needs to provide a fast mechanism for choosing a unique responder. This is best done with a binary tree circuit, and associative processors are among the earliest ancestors of the abstract tree machine described in this paper.

Another solution to the bandwidth problem is to enrich the network. The X-tree [23] is one example of this approach, and another idea is simply to increase the bandwidth on the links that are higher in the tree, resulting in the "fat tree" network which was used successfully in the network of the CM5 [24], a general purpose parallel computer intended to support both task and data parallelism.

Another class of architectures uses the tree to implement a specific set of higher level collective communication operations. The intention here is that the application programmer is concerned only with the higher level operations, and tailors the program in terms of the collective communication library rather than in terms of the network architecture, which remains hidden.

One example of this approach is a parallel string reduction machine [13], [25], intended for implementing the functional language FFP. A related system is APSA [14], which uses parallelism within the tree to provide data structure operations useful in the implementation of list processing languages such as Lisp and Scheme.

Another way to use a parallel tree architecture is to serve as an abstract host for implementing the family of parallel fold and scan functions. The primary purpose here is to use the tree as a vehicle for defining the communication patterns needed for the folds and scans, but this could be just a tree *pattern* of communication that is actually embedded in a richer interconnection network, which provides other collective communication operations that would not be supported efficiently by the tree. There are two advantages of using an abstract tree network in this way, rather than just defining the implementations of folds and scans directly in the richer network: the algorithms are clarified and made more portable, and the inherent costs of folds and scans is measured more accurately using a minimal network rather than one with arbitrarily chosen additional bandwidth that is unused. A parallel scan can be used to implement a "segmented scan" [26] which yields a form of processor grouping similar to the operations described in Section IV.

Many problems in high performance computing require a more general programming environment than the abstract tree machine described in this paper, yet they might also benefit from the data parallel techniques. This suggests embedding the abstract machine and algorithms given in this paper within a richer architecture. Hatcher and Quinn discuss the use of general data parallel programming techniques on architectures with an MIMD organization [27].

An interesting idea for generalizing SIMD architectures in order to support tasks, while still retaining some of the efficiency of SIMD for data parallelism, is the "multiple SIMD" or MSIMD architecture [28]. Since MSIMD architectures are intended to support efficient data parallelism, it is natural to base them on a tree network, or at least to embed a tree within a richer network. Several examples of this genre are surveyed in [29], including a hierarchical MSIMD tree machine [30]; DADO, a multiprocessor for parallel rule processing in expert systems [31]; the NonVon tree machine, and the FFP machine of Mago which was cited earlier. An architecture combining data parallelism with general MIMD hardware can also benefit from an MSIMD organization.

The trend in high performance computers has been away from fine grain architectures with large numbers of small processors, and toward systems based on powerful processors with large local memories. It is easier to exploit task parallelism using such architectures, and they appear to have a wider range of applicability than the fine grain data parallel architectures. However, fine grain parallel systems are making a comeback in the form of reconfigurable hardware, especially FPGAs [32]. These are often described in marketing literature as hardware that can be modified to suit an application, but in reality they are fine grain parallel SIMD systems. FPGAs often have a flexible interconnection network with a mesh organization containing pass transistors in the nodes that allow the programmer to establish relatively fast links that are tailored to the problem at hand. The abstract tree machine used in this paper could be embedded in an FPGA.

VIII. Conclusion

The main result of this paper is that a fine grain architecture with a tree interconnection network can perform algorithms that combine some of the characteristics of task and data parallelism. The central result is a representation of subgroups,

along with a set of communication algorithms that allow simultaneous data parallel communications within each group with absolutely no interference from other groups.

Many tree algorithms require load balancing for efficient performance. This leads to good performance when a problem can be decomposed into subtasks whose sizes match the subtree sizes, but performance deteriorates on irregular data or adaptive algorithms. The approach to group communication presented here does not suffer from that defect: the set of processors can be decomposed recursively into subgroups of any size; all the group communication operations take a constant number of fast primitive tree instructions; and a communication within one group never interferes with any other group.

The algorithms presented here have been expressed using the weakest possible host architecture, a SIMD organization with a tree interconnection network. This does not mean that the results are applicable only to such a limited architecture. On the contrary, the results are made stronger by starting with weaker assumptions about the capabilities of the host machine. If these algorithms are used on more powerful architectures, then they can be enhanced by additional techniques. Two such generalizations are particularly important. If the parallel system has MIMD control rather than SIMD control, then the local computation steps can vary among the processors, leading to much more flexible algorithms. Also, the set of communication operations can be expanded, including standard operations such as scatter and gather, if the interconnection network allows this.

ACKNWLEDGEMENT

I would like to thank the anonymous referees and the editors for several helpful suggestions.

REFERENCES

[1] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, no. 2, pp. 123–169, 1998.

[2] H. Bal and M. Haines, "Approaches for integrating task and data parallelism," *IEEE Concurrency*, vol. 6, no. 3, pp. 74–84, July-August 1998.

[3] T. Rauber and G. Rünger, "A transformation approach to derive efficient parallel implementations," *IEEE Transactions on Software Engineering*, vol. 26, no. 4, pp. 315–339, 2000.

[4] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, March 1996.

[5] J. L. Potter, *The Massively Parallel Processor*. MIT Press, 1985.

[6] W. D. Hillis, *The Connection Machine*. MIT Press, 1985.

[7] S. P. J. (ed.), "Haskell 98 language and libraries: the revised report," *Journal of Functional Programming*, vol. 13, no. 1, pp. 1–255, January 2003.

[8] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, November 1989.

[9] C. P. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix," in *Proc. International Conference on Parallel Processing*. IEEE, August 1985, pp. 180–185.

[10] R. Ladner and M. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 4, October 1980.

[11] J. O'Donnell, "A correctness proof of parallel scan," *Parallel Processing Letters*, vol. 4, no. 3, pp. 329–338, September 1994.

[12] ——, "A systolic associative LISP computer architecture with incremental parallel storage management," Ph.D. dissertation, University of Iowa, Iowa City, 1981, technical Report 81-5.

[13] G. Mago, "Copying operands versus copying results: A solution to the problem of large operands," in *Functional Programming Languages and Computer Architecture*. ACM, 1981, pp. 93–98.

[14] J. O'Donnell, T. Bridges, and S. Kitchel, "A VLSI implementation of an architecture for applicative programming," *Future Generation Computer Systems*, vol. 4, no. 3, pp. 245–254, October 1988.

[15] M. Sheng and J. Rose, "Mixing buffers and pass transistors in fpga routing architectures," in *FPGA 2001*, 2001, pp. 75–84, monterey, California.

[16] J. O'Donnell, "Functional microprogramming for a data parallel architecture," in *Proceedings of the 1988 Glasgow Workshop on Functional Programming*. Computing Science Department, University of Glasgow, 1988, pp. 124–145.

[17] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press, 1985.

[18] D. Crawley, "An analysis of mimd processor interconnection networks for nanoelectronic systems," University College London, Tech. Rep., 1998, image Processing Group, Department of Physics and Astronomy.

[19] S. G. Akl, *The Design and Analysis of Parallel Algorithms*. Prentice Hall, 1989.

[20] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1998.

[21] G. V. Wilson, *Practical Parallel Programming*. MIT Press, 1995.

[22] J. L. Potter, *Associative Computing: A Programming Paradigm for Massively Parallel Computers*. Plenum Press, 1992.

[23] A. M. Despain and D. A. Patterson, "X-Tree: A tree structured multiprocessor computer architecture," in *ISCA'78*, 1978, pp. 144–151.

[24] C. E. Leiserson and et. al., "The network architecture of the connection machine cm-5," in *ACM Symposium on Parallel Algorithms and Architectures*. ACM, 1992, pp. 272–285.

[25] G. Mago, "Data sharing in an FFP machine," in *Conference on Lisp and Functional Programming*. ACM, 1982, pp. 201–207.

[26] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[27] P. J. Hatcher and M. J. Quinn, *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[28] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. McGraw-Hill, 1986.

[29] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*. Benjamin/Cummings Publishing Co., 1989.

[30] A. L. DeCegama, *The Technology of Parallel Processing*. Prentice Hall, 1989, vol. 1.

[31] S. Stolfo, H. Dewan, D. Ohsie, and M. Hernandez, "A parallel and distributed environment for database rule processing, open problems and future directions," in *Emerging Trends in Database and Knowledge-based Machines*. IEEE Press, 1995, iSBN: 0-8186-6552-1.

[32] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, June 2002.

John O'Donnell received a B.S in astronomy from the California Institute of Technology (1974) and a Ph.D. in computer science from the University of Iowa (1981). His research is centered on the application of pure functional programming languages to the design of algorithms and systems, and he has developed the Hydra computer hardware description language. Other research interests include computer music applications.