Abstract

Data-parallel programming languages have many desirable features, such as deterministic semantics and the ability to express large amounts of fine-grained parallelism. However, it is challenging to implement such languages efficiently on conventional MIMD multiprocessors, because these machines incur a high overhead for such small grain sizes. This paper presents analysis techniques for data-parallel program graphs that reduce these overheads in two ways: by stepping up the grain size, and by relaxing the synchronous nature of the computation without altering the program semantics. The algorithms work by partitioning the program graph into clusters of nodes such that all nodes in a cluster have the same iteration space size, and further refining these clusters into epochs based on generation and consumption patterns of data vectors. This converts the fine-grain parallelism in the original program to medium-grain loop parallelism, which is better suited to MIMD machines. We present performance results for data-parallel kernels analyzed by these techniques and converted to single-program multiple-data (SPMD) code running on an Encore Multimax.

1 Introduction

Despite the advances made in parallel computer hardware, parallel programs remain difficult to write, debug and port. Recently, researchers have started investigating these issues, particularly that of portability. Research in portable parallel programming has largely taken one of two forms: those based on parallelism extraction [30] and those based on virtual machine emulation [14]. However, both approaches have their drawbacks. Parallelism extraction systems can only extract parallelism that exists in the original code, which may not be expressing a good parallel algorithm in the first place. Virtual machine emulators are usually limited in the machine topologies they can support, and are not always efficient. We believe that parallelism should be explicit in the source program, but that it should not be based on the notion of processes, as this requires the programmer to manage a great amount of difficult low-level detail like process creation, load balancing and synchronization. Instead, we have chosen a data-parallel style of programming, where parallelism is expressed as (parallel) operations over sets of data. A large fraction of the existing parallel algorithms for PRAM and other machine models are either data-parallel in nature or can be easily converted to such a form [26, 13].

Data-parallel languages have traditionally been linked with SIMD parallel computers, and researchers have largely avoided implementing such languages on MIMD parallel machines, stating that “fine grained computations incur excessive overhead when executed on conventional MIMD multiprocessors” [14]. This objection is valid if the MIMD implementation is a straightforward instruction-by-instruction emulation of...
a synchronous SIMD machine. Maintaining efficiency on MIMD machines requires aggregating the fine-grained operations into larger-grained tasks and relaxing the lock-step synchronization while maintaining semantic equivalence. The aggregation of multiple operations also allows various optimizations to be performed on the aggregate. Given a data-parallel program, the problem, then, is to gather information about the program variables and statements to obtain a good aggregation. In this paper we develop compile-time techniques called size and access inference that extract such information and perform the aggregation. Size inference is used to derive relations between symbolic “sizes” of program variables and “iteration space sizes” of program operations; these can then be used to partition the program graph into clusters based on iteration space size and schedulability constraints. These clusters serve as units of load balancing and work allocation. Access inference analyzes generation and consumption patterns of program variables within clusters and identifies conflicts requiring synchronization. These are used to subdivide clusters into epochs. The operations within an epoch can be performed in any order that maintains data dependences; no synchronization is needed. Epochs map fairly naturally to loops that can then be executed in parallel through loop distribution techniques [23, 30].

Another issue that arises in the context of data-parallel languages is that of storage. Frequently, large intermediate results are created if the language is naively implemented. Our storage allocation and code generation techniques tie in with the analysis techniques shown in this paper to remove such intermediate storage, or reduce it to storage for a small section of a vector, similar to “drag-through” transformations in APL or loop fusion in the Fortran world.

The analysis techniques and program transformations discussed in this paper apply to uniprocessor as well as multiprocessor systems. They provide the following benefits:

- The grain size of the output program is larger than that of the input program, making it suitable for execution on MIMD multiprocessors.
- The only synchronization in the output program is that required for maintaining semantic equivalence with the original program.
- Storage requirements are reduced, and storage is not required for many temporary vectors. Vector storage can be reduced to storage of sections of vectors. This can take advantage of scalar and vector registers.
- Locality of reference is improved, since multiple operations are combined into single loops. This can take advantage of chaining in vector machines, and registers and caches in general.

The work described here is related to compilation techniques for APL, FP and similar languages, the compilation of C* for multiprocessors, and optimizations performed by some Fortran compilers. It differs in the kinds of operations it can handle, and we shall discuss these matters further in Section 7.

The remainder of the paper is organized as follows. We introduce the language and machine models in Section 2. The technique of size inference is introduced in Section 3, and of access inference in Section 4. We discuss the storage management and code generation aspects of the compiler in Section 5, and present some results. Extensions to the control structure of the language are discussed in Section 6, and relationship to other work is discussed in Section 7, and conclusions and future directions are presented in Section 8.

## 2 Language and Machine Models

The compiler operates on data-parallel program graphs, where graph nodes are data-parallel operations and graph edges represent data precedence constraints. These are similar to dataflow graphs [2], except that operations compute on entire vectors at a time. The primitive data type is the vector. Vectors must be homogeneous. Scalars are treated as singleton vectors. The primitive operations all take vectors as arguments; they include traditional arithmetic and logical operations applied elementwise to vectors (such as \( A + B \)), as well as associative scans (+ \( A \) in APL), permutations (\( A[B] \) in APL) and distribute operations (similar to the spread intrinsic in Fortran 90). The choice of operations is not particularly important; the set we have chosen is based on the scan model of computation [4]. However, scans and permutes present problems related to data dependence that are usually not handled by existing compiler
| Operation | init | kernel | lo, hi |
|-----------|------|--------|--------|
| +        |      | out[i] = in1[i] + in2[i] | 0, in1.len |
| !        |      | out[i] = ! in1[i] | 0, in1.len |
| SELECT   |      | out[i] = in1[i] ? in2[i] : in3[i] | 0, in1.len |
| +\ (SCAN) | out[0] = 0 | out[i] = out[i-1]+in1[i-1] | 1, in1.len |
| +/ (REDUCE) | out = 0 | out = out+in1[i] | 0, in1.len |
| MIN/LEN  | out = INT_MAX | out = min(out, in1[i]) | 0, in1.len |
| DIST     | out = len(in1) | 0, 0 |
| DISTV    |      | out[i] = in1 | 0, in2 |
| PERM     | out[in2[i]] = in1[i] | 0, in1.len |
| BPERM    | out[i] = in1[in2[i]] | 0, in1.len |
| DPERM    | out[i] = in3[i] | 0, in1.len |
| INDEX    | out[i] = i | 0, in1 |
| GET      | out = in1[in2] | 0, 0 |

Figure 1: Primitive data-parallel operations, and corresponding C code. The input vectors to an operation are called in1, in2 and so on. The template for the C code is as follows: init; for (i = lo; i < hi; i++) {kernel;}. The output vector is called out.

techniques. A complete list of the operations that we use in this paper is given in Figure 1. These can be represented equivalently as single-assignment code, which we will also use (with a LISP-like syntax) for ease of understanding.

The complete language model [5] is more complex, supporting additional data types and segmented versions of the operations [4], if-then-else forms and recursion for control flow, and function definition and call as encapsulation mechanisms. To simplify the exposition, we will defer treatment of these features until Section 6.

As an example, we present the computation graph for the split operation in Figure 2, along with an example of its action. split takes a vector of values (integers) and a vector of flags (boolean, encoded as 0/1 integers), packs the values corresponding to the 0 flags to the bottom and those to the 1 flags to the top, and returns the resulting vector. This operation forms the core of sorting algorithms such as radix sort and quicksort [18]. The representation of programs as graphs allows us to formulate the analysis problems in graph-theoretic terms, and also gives a good handle on storage optimizations. We now give a formal definition of a computation graph for a function. This is essentially similar to the IF1 graph representation for SISAL functions [21].

**Definition 1 (Computation graph)** The computation graph $G(F)$ for a function $F$ is a directed acyclic graph $(N, E)$, where

- $N$ is a set of nodes. A node $n \in N$ is the tuple $(t, op, in, out)$, where
  - $t$ is the type of $n$; $t$ is chosen from the set $T = \{\text{SIMPLE, FCALL, INPUT, OUTPUT}\}$.
  - $op$ is the operation performed by $n$, if $n.t = \text{SIMPLE}$; if $n.t = \text{FCALL}$, it is the name of the function; undefined otherwise.
  - $in$ is the number of input ports of $n$, numbered 1...in.
  - $out$ is the number of output ports of $n$, numbered 1...out.
- $E$ is a set of edges representing data precedence constraints. An edge $e \in E$ is the tuple $((n_s, p_s), (n_d, p_d))$, where
  - $n_s \in N$ is the node from which the edge originates.
  - $p_s$ is the output port of $n_s$ from which the edge originates; $1 \leq p_s \leq n_s.out$.  

3
(defun split (v f)
  (with ((NotF (not f))
         (Op (plus-scan f))
         (Down (plus-scan NotF))
         (Sum (plus-reduce NotF)))
    (permute v
      (select f
        (* (distv Sum v) Op)
        Down))))

Figure 2: Computation graph of the split operation, and an example of its action. The labels on the edges are used in size analysis.

- $n_d \in N$ is the node at which the edge terminates.
- $p_d$ is the input port of $n_d$ at which the edge terminates; $1 \leq p_d \leq n_d$.

The parallel execution model is based on data partitioning, i.e., each processor is responsible for a portion of the vector. The output of the compiler is single-program multiple-data (SPMD) loop code [17] suitable for execution on MIMD multiprocessors. Storage is of two types: vector storage, which holds a complete vector, and buffer storage, which holds an iteration’s worth of computation on a vector.\(^1\)

Given a computation graph on which initial transformations such as function call inlining and common subexpression elimination have been performed, we first use size inference to identify nodes that can be executed together in a single loop. We use this information to partition the graph into clusters, and allocate vector storage to cross-cluster arcs. We then examine clusters and further refine them into epochs based on information provided by access inference, and allocate vector and buffer storage where needed. Code is then generated in a straightforward manner for each epoch.

### 3 Size Inference and Graph Partitioning

Our intent is to partition the program graph into chunks that can be executed in parallel, with synchronization and loop distribution occurring only between chunks. Since all our primitives can be expressed as normalized loops in C or Fortran, we use the size of the iteration space (the difference of the loop bounds) of the nodes as the basis for partitioning. We first introduce size inference, which uses the semantics of the primitives to infer relations among the sizes of variables and the iteration space sizes of nodes. Then we show how to use this information to partition the graph into clusters, such that all nodes within a cluster have the same iteration space size.

\(^1\)We are deliberately vague here because this can mean different things on different machines. On a Cray Y-MP, a processor computes on 64 elements per iteration, while on an Encore Multimax, it processes 1 element.
3.1 Size inference

The size of a vector is the number of elements it contains. As vectors are naturally mapped to the edges of the computation graph, we characterize an edge $e$ of the graph with the following attributes:

- $\text{size}(e)$, the size of the vector associated with that edge.
- $\text{gen}(e)$, the generation pattern of the vector; this refers to the order in which the elements of the vector are generated by the node from which the edge emanates. For the present, we choose $\text{gen}(e)$ from the following set of stylized patterns:
  - index: generated in index order.
  - accumulated: generated by accumulation. In the context of loops, this means that the value is available only after all iterations of the loop have completed.
  - arbitrary: generated in some data-dependent order that cannot be predicted at compile time.
- $\text{use}(e)$, the consumption pattern for the vector on that edge; this refers to the order in which the elements of the vector are used in the node at which the edge terminates. As for $\text{gen}(e)$, this is chosen from the following set of stylized patterns:
  - unused: not used. (This means that the data values are not required for the operation. DISTV is an example, where only the length of the second input is needed.)
  - index: used in index order.
  - arbitrary: used in some data-dependent order not predictable at compile-time.

The patterns for $\text{gen}(e)$ and $\text{use}(e)$ are meant to be representative, not exhaustive. A particular choice is motivated by the set of primitives. For instance, a reverse-index pattern would be appropriate for a backwards scan operation. Also note that since a vector may be mapped to multiple edges with the same source (corresponding to fanout), the consumption pattern for a vector is defined as the “most constrained” consumption pattern among all the edges to which the vector is mapped, arbitrary being more constrained than index, which is more constrained than unused.

Analogously, a computation node $n$ has the following attributes:

- A set $\text{Input}(n)$ of constraints on the sizes of its input vectors of the form $l_i = l_j$ or $l_i = k$, where $i$ and $j$ range over the input ports of the node, $l_i$ is the size of the vector associated with the edge ending at input port $i$ of the node, and $k$ is a constant; these constraints define the vector sizes for which the computation is meaningful (well-formed).

- A transfer function $\text{Output}(n)$ that computes the sizes of its output vectors in terms of the sizes of its input vectors.

- An iteration size function $\text{Loop}(n)$ that computes the size of the iteration space of the operation performed by the node in terms of the sizes of its input and/or output vectors. This becomes important for nodes that take inputs of several sizes, or produce outputs whose sizes are different from the input sizes.

The attributes of certain representative nodes and their output edges are shown in Table 1. Note that the size of the output may be different from the iteration space size, as for the $+/ \text{}$ operation.

The goal of size inference is to symbolically compute these attributes for the nodes and edges of a given computation graph. With perfect information, it would assign symbolic size labels to the edges of the graph such that two edges would have the same label if and only if the vectors associated with them were required to have the same size at runtime in order for the computation to be well-formed. This is not always possible with compile-time information, because the sizes of vectors can depend on values contained in other vectors, information that is not available until runtime. This manifests itself in an imprecise transfer function. The DIST operation is an example. Its transfer function requires the value of the second argument in order to compute the size of the result. Hence, the analysis performed by the compiler makes a conservative
Table 1: Node and output edge attributes for a variety of operations.

| n  | Input(n) | Output(n) | Loop(n) | size | gen |
|----|----------|-----------|---------|------|-----|
| +  | $l_1 = l_2$ | $l_1$ | $l_1$ | $l_1$ | index |
| SELECT | $l_1 = l_2, l_1 = l_3$ | $l_1$ | $l_1$ | $l_1$ | index |
| +\ | $\emptyset$ | $l_1$ | $l_1$ | $l_1$ | index |
| +/ | $\emptyset$ | 1 | $l_1$ | 1 | accumulated |
| DIST | $l_1 = l_2, l_2 = l_1$ | UNKNOWN | UNKNOWN | UNKNOWN | index |
| DISTV | $l_1 = l_1$ | $l_2$ | $l_2$ | $l_2$ | index |
| DPERV | $l_1 = l_2$ | $l_3$ | $l_1$ | $l_3$ | arbitrary |

approximation to this ideal; if two edges are assigned the same label, then the vectors associated with them are guaranteed to have the same size at runtime if the function call is well-formed. In order to make this approximation, we restrict the transfer function so that it returns either the size of one of the input vectors, or the reserved value UNKNOWN. In general, the following cases may arise:

- The compiler infers that two vectors are guaranteed to have the same size. No runtime checks are required.
- The compiler infers that two vectors cannot have the same size, and this makes some operation ill-formed. This is a program error, and the program can be rejected.
- The compiler infers that the operation will be well-formed provided two vectors have the same size, but cannot guarantee this equality, since it may depend on runtime values. The compiler proceeds on the assumption that this is so, but inserts code to perform a check for this equality at runtime.
- The compiler infers that two vectors must (in general) have different sizes, but cannot determine a functional relation between the sizes. This is due to lack of knowledge in the compiler (about arithmetic, for instance).

Two vectors determined to be of unequal size may, in fact, have the same size in a certain run of the program (due to the data values). The compiler cannot take advantage of this fact.

The gen(e) and use(e) attributes of edges depend on the node operation, while the Loop(n) attribute of nodes depends on the sizes of the edges. The key component of this evaluation, therefore, is the assignment of size labels to edges of the graph. The method for this is shown in Algorithm 1.

Algorithm 1 (Size inference of a data-parallel computation graph.)

Input: A data-parallel computation graph G.
Output: Size labels for the edges of G.
Method:

1. Assign distinct symbolic sizes from some set $S$ to each vector of the computation graph, and assign to each edge the size of the vector that maps to it.

2. Form a system of equations $E$ representing the constraints on symbolic sizes.

   (a) For each node $n$ in the graph, instantiate each element of Input($n$) with the sizes assigned to its input edges and add it to $E$.

   (b) For each node $n$ in the graph whose transfer function does not return UNKNOWN, instantiate Output($n$) with the sizes assigned to its input edges, equate it to the size assigned to the output vector, and add it to $E$.

3. Solve $E$. Given the forms of the constituent equations, this will partition the set of sizes into equivalence classes. Assign distinct labels to each class.
4. Replace the size label on each edge of the graph with the label of the equivalence class to which the size belongs.

To illustrate the working of this algorithm, consider the split operation. Given the initial labeling of the edges as shown in Figure 2, we get the following system of equations:

| Equation | How derived |
|----------|-------------|
| \( b = c \) | \( O_{output}(\text{NOT}) \) |
| \( c = d \) | \( O_{output}(+ \_\text{SCAN}) \) |
| \( e = 1 \) | \( O_{output}(+ \_\text{REDUCE}) \) |
| \( b = f \) | \( O_{output}(+ \_\text{SCAN}) \) |
| \( g = a \) | \( O_{output}(\text{DISTV}) \) |
| \( g = f \) | \( I_{nput}(+) \) |
| \( h = g \) | \( O_{output}(+) \) |
| \( b = d \) | \( I_{nput}(\text{SELECT}) \) |
| \( b = h \) | \( I_{nput}(\text{SELECT}) \) |
| \( i = b \) | \( O_{output}(\text{SELECT}) \) |
| \( a = i \) | \( I_{nput}(\text{PERMUTE}) \) |
| \( j = a \) | \( O_{output}(\text{PERMUTE}) \) |

which upon solution yields the following two equivalence classes:

\[
\mathcal{EC}_1 = \{ e \} \\
\mathcal{EC}_2 = \{ a, b, c, d, f, g, h, i, j \}
\]

resulting in the edge labeling shown in Figure 3. The other node and edge attributes are easily computed given the size labels on the edges and are also shown in Figure 3.

### 3.2 Graph partitioning

Size inference provides information about the iteration space sizes of the various nodes. We want to group together computation nodes that have the same \( \text{Loop}(n) \) value into larger clusters. Conversely, nodes
with different Loop(n) values must be placed in separate clusters. We must also ensure that clusters can be scheduled, i.e., a cluster can run to completion once all its inputs are available. Since all nodes in a cluster have the same iteration space size, the data needs to be divided up among the physical processors only at cluster entry. Thus, clusters serve as natural units of loop distribution (load balancing). We are interested in making clusters as large as possible, since the optimizations that follow will not go beyond cluster boundaries. We now present a framework for achieving these ends.

**Definition 2 (Critical edge)** An edge ((n1, p1), (n2, p2)) of a computation graph is a critical edge iff Loop(n1) ≠ Loop(n2).

**Definition 3 (Clustering)** A cluster of a computation graph G = (N, E) is a connected subgraph of G containing no critical edges. A clustering P = {G1, ..., Gm} is a collection of pairwise disjoint clusters covering G. A clustering P is a refinement of another clustering Q iff every cluster of P is a subgraph of some cluster of Q.

The maximal clusters of the computation graph can be found by removing all critical edges of the graph and finding the connected components of the resulting graph. The set of maximal clusters form a clustering of the computation graph, which we will refer to as M_G. Define the condensation graph of a computation graph under a clustering as follows:

**Definition 4 (Condensation graph)** Let P = {G1, ..., Gm} be a clustering of a computation graph G = (N, E). The condensation graph of G under P is the graph C^P_G = (P, E^P), where (G_i, G_j) ∈ E^P iff there exist nodes i, j ∈ N (i ≠ j) with ((m, p), (n, q)) ∈ E.

The condensation graph captures the data dependences between clusters of a computation graph. A clustering P of G is said to be viable if C^P_G is acyclic. Intuitively, the clusters of a viable clustering can be scheduled (linearized) based on data dependences, and a cluster can run to completion once all its input data are available. M_G is not necessarily viable, as Figure 4 shows. However, the following property holds of any clustering (hence, of any viable clustering).

**Lemma 1** Any clustering of G is a refinement of M_G.

**Proof:** Follows from the maximality of M_G. □

As we are interested in finding large clusters, the only candidates for refinement are the non-trivial strongly connected components (SCCs) of M_G. We need a way to break the cycle in an SCC in order to achieve viability. While the minimum number of breaks required for this is unique, there can, however, be a number of partitions of the SCC having this cardinality. For instance, in Figure 4, the cluster C1 must be refined into two clusters to make the resulting clustering viable. However, the rightmost + node can be scheduled in either of the two refinements. The following lemma captures the property that nodes must satisfy in a viable clustering.

**Lemma 2** Two nodes n1 and n2 with Loop(n1) = Loop(n2) cannot be in the same cluster of a viable clustering if there exists a path from n1 to n2 containing a critical edge.

**Proof:** By contradiction. Refer to Figure 5 and suppose that n1 and n2 satisfying the above conditions are in the same cluster G_s of a viable clustering P. Observe that since n1 and n2 are in the same cluster, there must be at least two critical edges on the path between them, one leaving the cluster and another entering it. Call these critical edges E_1 and E_2. Then the path between n1 and n2 is p = (e_1, ..., E_1, ..., E_2, ..., e_k). Since E_1 and E_2 are critical edges, Loop(a) ≠ Loop(n1) and Loop(b) ≠ Loop(n2). Let nodes a and b be in clusters G_a and G_b (not necessarily distinct) of P. But then C^P_G contains the cycle G_a → G_b → G_n, and hence P is not viable. □

We call (n1, n2) above a critical pair if the path between them begins and ends with critical edges. To separate the members of critical pairs, we add separator edges between them, producing the separator graph G_S = (N, E ∪ E_S), where E_S is the set of separator edges. The separator edges encapsulate the interactions between clusters, so that we can now examine and refine each cluster in isolation. The number of refinements required to make a cluster viable is equal to the maximum number of separator edges on any
Figure 4: A program graph $G$, $M_G$, and the condensation graph $G^M_C$. This shows that $M_G$ is not always a viable clustering. The graph is that of the split operation where ++REDUCE is not considered a primitive operation. Instead, it is simulated by performing a ++SCAN on the vector, and adding the final values of the original and the scanned vectors. The maximal clusters of the graph are indicated by shading, and condensation graphs is shown on the right. The upper graph is $G^M_C$, which is not viable. The lower condensation graph is viable, but the mapping of nodes to clusters is not unique, as shown by the indicated ++SCAN node.

Figure 5: Proof of the viability lemma.
path within the cluster, but there is some flexibility in the actual assignments of nodes to refined clusters. We capture this in the form of lower and upper bounds for refined cluster numbers. Intuitively, separator edges must span refined clusters. The lower and upper bounds for nodes in a cluster can be established with two passes over the graph, as shown in Algorithm 2.

Algorithm 2 (Viable partitioning of a data-parallel program graph.)
Input: A data-parallel computation graph G on which size inference has been performed.
Output: An assignment of cluster numbers to nodes of G such that the resulting clustering is viable.
Method:
1. Compute $M_G$ by finding the connected components of the graph $(N, E - E_C)$, where $E_C$ is the set of critical edges of G.
2. Find $L$, the set of non-trivial strongly connected components of $M_G$ [1, p. 193].
3. For each $l \in L$:
   (a) Identify critical pairs and create the separator graph $l_S$.
   (b) Sort the node set of $l_S$ into list F so that each ancestor of a node precedes it on F.
   (c) Sort the node set of $l_S$ into list B so that each successor of a node precedes it on B.
   (d) For each node $n$ on B:
      
      \[ ih(n) = \min(0, \max_{e=([n,p],[m,q])} \{ ih(m) - \delta(n,m) \}) \]
      
      where \( \delta(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E_S \text{ of } l_S \\ 0 & \text{otherwise.} \end{cases} \)
   (e) Compute $S = -\min_{n \in B} \{ ih(n) \}$. This gives the maximum number of separator edges on any path in the cluster.
   (f) For each node $n$ on F:
      
      \[ l(n) = \max(0, \min_{e=([m,p],[n,q])} \{ l(m) + \delta(m,n) \}) \]
      \[ h(n) = ih(n) + S \]
4. Assign refined cluster numbers to nodes from the range $[l(n), h(n)]$. We currently choose the refined cluster number by the as-late-as-possible policy (i.e., the refined cluster number is chosen to be $h(n)$), but other choices are possible. The particular assignment policy affects the number of edges that go between clusters; this is important, as storage must be allocated for the vectors corresponding to these edges.

The clusters of the resulting viable clustering can now be scheduled in any way that maintains the inter-cluster data dependences.

4 Access Inference

Each cluster of a viable clustering of a computation graph represents a section of code that could potentially be fused into a single loop. It does not necessarily guarantee that this is always possible, as operations within a cluster may conflict in the order in which they produce and use vectors, or may require other kinds of synchronization. Access inference identifies these conflicts and further refines clusters into epochs, where each epoch corresponds to a single loop.

The language primitives can be divided into the following groups:

- Elementwise operations,
Figure 6: Templates for SCAN and REDUCE operations that expose their microstructure.

- Structure accessors (such as LENGTH),
- Permutes and distributes, and
- Scans and reductions.

The first three groups can be implemented on a multiprocessor in a single loop, as there are no loop-carried dependences in these operations. This is not true for scans and reductions, however, as there is state information that has to be communicated. In order for access inference to effectively handle SCAN and REDUCE operations, we must expose their microstructure. The idea is to decompose these operations into more primitive loops with inter-processor communication isolated between the loops. This decomposition follows from the standard parallel algorithm for scans [19], where each processor performs the following three phases:

- Sum the elements in its section. This is an elementwise loop requiring no interaction with other processors.
- Scan the sums from all the processors to obtain the correct start-offset. This involves inter-processor communication (synchronization).
- Sum the elements of its section starting with the start-offset derived from the previous step. This is another elementwise loop that requires no interaction with other processors.

Note that we do not have to know the exact number of processors, or the algorithm used for the scan in the middle of the algorithm. It suffices to know that they will participate in the synchronization operation, the details of which are managed by the runtime system. A reduction is treated in much the same way, except that the third phase is not required, and the synchronization step returns to each processor the global sum instead of the start-offset. The templates for these two operations are shown in Figure 6. We replace occurrences of scan and reduce nodes with the corresponding templates, and perform common subexpression elimination to remove any redundant nodes. This results in the graph for the split operation shown in Figure 7.

The identification of edges requiring synchronization is based on def-use conflicts for the vectors corresponding to those edges, and a special case for scan operations. Intuitively, the idea is as follows. If a vector is produced and consumed in ways that are compatible (such as index and index), then storage
Figure 7: The split operation after template expansion and CSE. Cluster and epoch boundaries are shown, along with computation nodes and synchronization events.

| gen(e)         | unused | index | arbitrary |
|----------------|--------|-------|-----------|
| index          | 0      | 0     | 1         |
| arbitrary      | 0      | 1     | 1         |
| accumulated    | 0      | 1     | 1         |

Table 2: Incompatibility relation $R(gen(e), use(e))$ between generation and usage patterns. A 0 in an entry indicates that the two patterns are compatible, while a 1 indicates an incompatibility.

is only required for a small section of the vector at any given time, the producer and consumer nodes can be executed in a single loop, and no synchronization is needed. Conversely, for incompatible patterns, the entire vector must be generated before any of it can be used, the producer and consumer nodes must be in different loops, and synchronization is required between the loops. (However, loops need not be redistributed as this is all within a single cluster.) We capture this notion by defining an incompatibility relation $R(gen(e), use(e))$ between $gen(e)$ and $use(e)$, as shown in Table 2. Using this, we can define the synchronization weight of an edge as follows:

**Definition 5 (Synchronization weight)** The synchronization weight $\delta_s(e)$ of an edge $e = ((m, p), (n, q))$ is equal to $R(gen(e), use(e))$ if $e$ is not an edge between a SCAN$_1$ and a SCAN$_2$ node, and is 1 otherwise.

Again, while the number of epochs that a given cluster must be broken into is well-defined, there is some flexibility in the mapping of nodes to epochs. If epochs are numbered so that all data dependences are from lower-numbered epochs to higher-numbered ones, there is a range of epoch numbers $[l(n), h(n)]$ that a node $n$ can be mapped to without violating the dataflow constraints. These numbers can be computed for each node of a cluster by Algorithm 3, which is similar to Algorithm 2.

**Algorithm 3 (Access inference of a cluster of a data-parallel computation graph.)**

*Input:* A cluster from a viable clustering of a data-parallel program graph.

*Output:* An epoch numbering of the nodes of the cluster.

*Method:*
1. Sort the node set into list F such that all ancestors of a node precede it on the list.

2. Sort the node set into list B such that all descendants of a node precede it on the list.

3. For each node $n$ on B:
   \[
   ih(n) = \min(0, \max_{e=([n, p], [m, q])} \{ ih(m) - \delta_e(e) \}).
   \]

4. Compute $S = -\min_{n \in B} \{ ih(n) \}$.

5. For each node $n$ on F:
   \[
   l(n) = \max(0, \min_{e=([m, p], [n, q])} \{ l(m) + \delta_e(e) \})
   \]
   \[
   h(n) = ih(n) + S
   \]

Given the valid epoch ranges for nodes, we must choose an assignment of nodes to epochs. One consideration for an assignment is the number of edges that cross epoch boundaries, as storage must be allocated for the vectors corresponding to these edges. This is a list scheduling problem, which is considered solved [12]. Currently we use the as-late-as-possible policy as this tends to reduce edge crossings, but other assignment algorithms can certainly be used. The epoch assignments for the split operation are shown in Figure 7. Note that multiple synchronization events at an epoch boundary can be combined, as in the case of the two scans and the reduce at the end of the first epoch.

5 Implementation and Results

We have implemented a compiler for data-parallel computation graphs incorporating the above ideas. The compiler produces C Threads code [10] suitable for execution on a shared-memory multiprocessor. In this section, we briefly discuss storage management and code generation, and then show some results.

5.1 Storage management and optimization

Intermediate storage required for the computation is of two types: vector storage and buffer storage. The former is needed for those edges that cross cluster or epoch boundaries, while the latter is needed for intra-epoch edges with fanout. Vector storage can be further subdivided into stack and heap storage. This distinction is important because of the high overhead of memory allocation on a multiprocessor. Heap storage is required for vectors associated with critical edges, as these must persist across multiple clusters, while stack storage is sufficient for inter-epoch edges. Allocation and reclamation is much cheaper for stack storage than for heap storage, since it can be done without interprocessor communication. Standard liveness analysis techniques can be used to optimize reuse of storage. The only extension required is that the size of a memory block allocated for a vector must be taken into account when considering it for reuse.

5.2 Code generation

Code generation is fairly straightforward, since at this point the compiler is dealing with well-structured loops. The compiler generates C code and relies on the native C compiler to perform machine-specific optimizations. It does, however, perform source-level transformations such as converting array indexing to pointer dereferencing, and unrolling loops to reduce loop overheads, as native C compilers are not very good at these. Code generation is done by traversing backwards through epochs; the buffer storage associated with nodes with fanout is used to avoid recomputation of results.
5.3 Results

We discuss the following (small) examples.

**Example 1:** (SAXPY) This computation takes two vectors \( \vec{X} \) and \( \vec{Y} \) and a scalar \( A \), and computes \( A \cdot \vec{X} + \vec{Y} \). This routine is one of the Basic Linear Algebra Subprograms [20]. The computation graph for this example is shown in Figure 8. If implemented naively, this would require three loops, one each for the distribute, multiplication and addition. It would also require storage for the distributed scalar and the intermediate result of the multiplication, which are not needed. The analysis techniques transform the computation into the desired single loop with the distribution of the constant folded into the multiplication.

**Example 2:** (Dot product) The computation graph for computing the dot product of two vectors is shown in Figure 9. This involves an elementwise multiplication followed by a plus-reduce. A naive implementation would require two loops: one for the multiplication, and one for the reduction. These two loops are fused into one by our analysis techniques. It also does not require any intermediate storage.

**Example 3:** (Split) The computation graph for the split operation is shown in Figure 2. It contains eight operations: two plus-scans, a plus-reduce, a distribute, two elementwise operations, and a final permute. Implemented naively, the operation would need ten loops, three synchronizations, and the storage of seven intermediate vectors. Our analysis techniques reduce this to two loops, one synchronization, and one intermediate vector.

**Example 4:** (First minimum) This computes the first location of the minimum value of a vector. The computation can be written in a trivial serial loop (Livermore Loop 24), which unfortunately does not parallelize well. The computation graph of a parallel algorithm for this problem is shown in Figure 10. It involves two min-reductions, two distributes, a comparison, and the index function (\( \#13 \) in APL). Our analysis coalesces this graph into two loops with two synchronization points.

**Example 5:** (Leaffix) The computation graph for this operation is shown in Figure 11. This takes a tree with a value at each node, and returns to each node the sum of the values at all its descendants. The tree is represented in an Euler tour representation, and the details of the representation and algorithm may be found in [4]. This computation can be used as a kernel for many tree operations, such as determining the number of descendants for each vertex. This graph differs from the previous ones in that the size of the iteration space changes during the computation (the scan operates on a vector that is twice the length of the original vectors). The analysis techniques transform this graph to four loops with three synchronization points.

In Table 3, we show the running times of the serial and parallel programs for each of these kernels.
(defun dotproduct (x y)
    (plus-reduce (* x y)))

x [1 2 1 4 3]
y [3 1 7 6 2]
r 42

r = 0;
for (i = 0; i < x.len; i++) {
    r += x[i]*y[i];
}

Figure 9: Computation graph for dot product.

(defun minindex (v)
    (with ((L (length v)))
        (min-reduce
            (select
                (= v
                    (dist
                        (min-reduce v)
                        L))
                (index L)
                (dist L L))))

v [4 7 2 1 3 1]
L 6
a 1
b [6 6 6 3 6 5]
r 3

Figure 10: Computation graph for first minimum.
for various data sizes and number of processors. Note that in some cases the serial code uses completely
different data structures and algorithms that are very different from those used by the parallel code. The
measurements were taken on an Encore Multimax with NS32332 processors [11] running the Mach operating
system [25]. Timing was done using the memory-mapped free-running microsecond timer, with averaging
over multiple trials. The processor allocation facility of the Mach kernel was used to gain exclusive access
to the appropriate number of processors.

6 Extensions

In this section, we briefly explain how the techniques mentioned above can be extended to handle function
calls and control flow primitives.

6.1 Function calls

Function calls can sometimes be inlined into the calling context. This is simplified by the single-assignment
nature of the language. This is desirable if it can be done, as the compiler can then use information from
the call site, and potentially generate larger clusters and epochs.

However, such inlining may not always be possible. This may be due to various reasons:

- the function definition may not be available to the compiler (library functions, separate compilation);
- inlining the function call may cause an unreasonable expansion in code size;
- the function may be recursive.

In the absence of other information, the function call must be treated as a black box that must be placed
within its own cluster. This means that *gent(e)* and *use(e)* patterns for input and output edges must be
considered to be arbitrary, enforcing storage and synchronization at those edges. This is the only possibility
if the compiler cannot access the function definition.

We can do much better if the definition is available, or some properties of the function are known. In
this case, we can do the following things, in increasing order of sophistication:
Table 3: Performance results for the benchmark kernels.

| Kernel     | Data size | Serial time (ms) | Parallel time (ms) |
|------------|-----------|------------------|--------------------|
|            |           |                  | 1 proc. 3 proc. 6 proc. 9 proc. 12 proc. |
| SAXPY      | 1000      | 9.5              | 9.6 3.4 1.8 1.3 0.9 |
|            | 4000      | 41.3             | 42.3 12.9 7.0 4.5 3.5 |
|            | 16000     | 177.9            | 178.2 59.1 26.5 17.7 13.4 |
|            | 64000     | 713.8            | 717.8 250.2 123.3 78.7 56.9 |
| dotproduct | 1000      | 8.9              | 9.1 3.8 2.6 2.6 2.7 |
|            | 4000      | 34.9             | 38.2 12.9 7.2 5.9 6.9 |
|            | 16000     | 140.0            | 143.8 49.3 25.6 18.1 14.4 |
|            | 64000     | 560.6            | 585.9 200.3 101.7 67.8 54.2 |
| split      | 1000      | 11.5             | 14.0 6.3 4.1 4.1 4.6 |
|            | 4000      | 53.7             | 62.3 34.7 25.7 31.4 42.2 |
|            | 16000     | 254.0            | 313.2 126.8 79.4 67.6 59.2 |
|            | 64000     | 1037.1           | 1218.2 524.2 308.1 238.6 210.0 |
| minindex   | 1000      | 4.6              | 12.4 5.3 3.8 4.0 4.3 |
|            | 4000      | 18.4             | 48.4 17.4 10.0 7.7 9.1 |
|            | 16000     | 74.2             | 198.8 66.8 34.4 24.8 23.1 |
|            | 64000     | 296.4            | 813.6 274.1 141.2 97.5 85.3 |
| leaffix    | 1000      | 18.9             | 20.8 7.9 4.6 3.7 3.1 |
|            | 4000      | 74.6             | 107.1 34.3 18.6 14.1 11.5 |
|            | 16000     | 299.3            | 472.6 174.2 96.3 67.8 54.9 |
|            | 64000     | 1196.2           | 1900.6 774.8 454.6 327.7 273.6 |

- We can derive an end-to-end $Output(n)$ for the function call by performing size analysis on the definition and deriving the size of each output of the function in terms of the sizes of its inputs. This allows information to pass through the function call in the calling context.

- We can tag each function to indicate whether it has a synchronization point (equivalently, whether it contains multiple clusters or epochs). For functions that do not have a synchronization point, we can associate a $Loop(n)$ with the call, and treat it as a user-defined operator. We would have to compile a single-element version of the function so that it could be fused with adjacent loops.

- We can partially inline the function call. If the function has multiple clusters, or multiple epochs within a single cluster, we can inline the first and last ones and treat the interior as a black box. This allows optimization of the inlined clusters and does not introduce any additional synchronization. This is particularly useful for recursive functions that are not tail-recursive.

- Tail-recursive functions can be converted to an equivalent iterative form.

Our current compiler performs some of these optimizations, and we are working on incorporating some of the others.

### 6.2 Control flow primitives

If-then-else forms introduce hierarchy in the program graph, as they contain a then-subgraph and an else-subgraph. An $IF$ node introduces the following additional constraints in the size inference process:

- The condition input of the node must be a scalar.
- The then- and else-subgraphs must return the same number of results.
- Corresponding results of the then- and else-subgraphs must have the same size.
Recursion (either self or mutual) does not change the basic structure of the size inference algorithm. It only requires that the $E$ sets of all functions involved in the cycle of recursion be solved simultaneously. This leads to the correct solution if the functions are well-formed.

IF nodes and recursive function calls must be placed in their own clusters due to the nature of the computation. For an IF node, control goes either to the then-subgraph or to the else-subgraph depending on the condition. In a recursive call, the recursion can unravel for some unknown depth. A restricted amount of unraveling can be performed at the graph level for recursive functions to produce larger grain sizes.

7 Relationship to Other Work

The work described in this paper is related to research in compiling functional and applicative languages such as FP, APL, and SISAL, loop fusion techniques in the Fortran world, compilation of $C^*$ for multiprocessors, abstract interpretation techniques for functional languages, and type recovery in Scheme. We now discuss how our work stands with respect to each of these.

APL [16] has a long history of compilation efforts, and some of the techniques in this paper can be traced back to ideas presented by Guibas and Wyatt [15], such as stylized access modes, the compilation of streams, and slicing. However, they were investigating these issues for uniprocessors, and therefore did not consider multiprocessor issues such as synchronization. They also did not handle scans. More recently, Budd has looked at generating vector code from APL [7, 8]. Again, he confines his “drag-through” transformation to elementwise sections of code, and in particular does not deal with pipelining scan and reduction operations. (This is partly due to the fact that APL allows nonassociative scan operators.) Similar work has also been reported for FP [3] by Budd [9], and more recently by Walinsky and Banerjee [29]. The goal of the latter work was to treat permutation computations as index manipulations. These suffer from the same limitation of being effective only in sections of code containing only insert and apply-to-all functionals. Our work shows how to decompose scans and allow size and access information to flow through them. Compilation of SISAL by Sarkar [27] uses a similar approach to partitioning and scheduling. However, his work requires estimates of execution times for the nodes of the graph, and does not explore the epoch structure within the clusters.

Loop fusion has been around in the Fortran world for a long time. This is again limited in use to elementwise sections, and cannot work through operations such as scans due to data-dependence considerations. Permutes present a major obstacle because they are impervious to dependence analysis. Recently, there has been some work on identifying idioms such as scans and reductions in Fortran programs (Pinter and Pinter).

Quinn and Hatcher have worked on compiling $C^*$ for MIMD machines [24]. These have some of the same goals as our work. Their work differs from ours in two main ways: their runtime model involves virtual processor emulation by the physical processors, and they do not attempt any inter-statement storage optimizations. The domain construct in $C^*$ essentially provides the information that our size inferencing computes. It is not clear how their techniques would handle scans, or extend to nested parallelism.

This work is related in a general way to research in abstract interpretation; the size inference algorithm is very similar to the type inference algorithm of ML [22]. In a related vein, Shivers has analyzed the flow of control in Scheme programs to recover type information [28]. Some of the motivations for this work are similar to ours, although it is not directly related to parallelism.

8 Conclusions and Future Work

This paper has introduced two techniques for the analysis of data-parallel program graphs. The first, size inference, is used to partition the program graph into regions (called clusters) that have different loop sizes. The second technique, access inference, further refines these clusters into epochs based on conflicts between generation and usage patterns of vectors. These techniques are used to step up the grain size and reduce the synchronization requirements of data-parallel programs, making it viable to run them on traditional MIMD multiprocessors. The major contribution of this paper is to show how to make the techniques work
in the presence of scan, reduction, distribute and permutation operations. A compiler based on these ideas has been implemented and run on small benchmarks.

The examples used in the paper were simple in that they contained no control flow operations at the computation graph level. In Section 6, we explained how these techniques extend for a language with an if-then-else form and recursion as control flow primitives, and function definition and call as encapsulation mechanisms. More importantly, we can augment the language with segmented vectors and segmented versions of the primitive operations. This allows the implementation of nested parallelism [6]. The analysis techniques extend naturally to handle segmented operations. This simply introduces more levels of loops, and the $\text{Loop}(n)$ function must now return a tuple instead of a single value. There is also the question of runtime models for segmented vectors based on the data signature. These issues are beyond the scope of this paper.

One drawback of the current analysis is the limited choice of generation and use patterns of vectors. In particular, this affects our ability to analyze structured permutations such as reverse and rotate. We are working on refinements to access inference similar to ideas in [29] that would allow us to identify such permutations and use this information to choose alternative and more efficient traversal orders through the iteration space.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, Reading, MA, 1974.

[2] Arvind and Rishiyur S. Nikhil. A Dataflow Approach to General-purpose Parallel Computing. Computation Structures Group Memo 302, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July 1989.

[3] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM, 21(8):613–641, August 1978.

[4] Guy E. Blelloch. Vector Models for Data-Parallel Computing. The MIT Press, Cambridge, MA, 1990.

[5] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A Data-Parallel Intermediate Language. In Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, pages 471–480, College Park, MD, October 1990. Also available as technical report CMU-CS-90-146.

[6] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. Journal of Parallel and Distributed Computing, 8(2):119–134, Feb 1990.

[7] Timothy A. Budd. An APL Compiler for a Vector Processor. ACM Transactions on Programming Languages and Systems, 6(3):297–313, July 1984.

[8] Timothy A. Budd. A New Approach to Vector Code Generation for Applicative Languages. Technical Report 88-60-18, Department of Computer Science, Oregon State University, Corvallis, OR, August 1988.

[9] Timothy A. Budd. Composition and Compilation in Functional Programming Languages. Technical Report 88-60-14, Department of Computer Science, Oregon State University, Corvallis, OR, June 1988.

[10] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[11] Encore Computer Corporation. Multimax Technical Summary. Encore Computer Corporation, 1988.

[12] Joseph A. Fisher. The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources. PhD thesis, New York University, New York, NY, 1979.
[13] Geoffrey C. Fox. What Have We Learnt from Using Real Parallel Machines to Solve Real Problems? In Geoffrey Fox, editor, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Volume II, pages 897–955, New York, NY, January 1988. ACM Press.

[14] Eran Gabber. VMMP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1(3):304–317, July 1990.

[15] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, pages 2–8, Tuscon, AZ, January 1978.

[16] Kenneth E. Iverson. A Programming Language. Wiley, New York, NY, 1962.

[17] Alan H. Karp. Programming for Parallelism. Computer, 20(5):43–57, May 1987.

[18] Donald E. Knuth. Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[19] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. Journal of the ACM, 27(4):831–838, October 1980.

[20] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software, 5(3):308–323, September 1979.

[21] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauer, Chris Kirkham, Bill Noyce, and Robert Thomas. SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2. Lawrence Livermore National Laboratory, March 1985.

[22] Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17:348–375, 1978.

[23] Constantine D. Polychronopoulos. Parallel Programming and Compilers. Kluwer Academic Publishers, Norwell, MA, 1988.

[24] Michael J. Quinn and Philip J. Hatcher. Data-Parallel Programming on Multicomputers. IEEE Software, 7(5):69–76, September 1990.

[25] Richard F. Rashid. Threads of a New System. Unix Review, 4(8):37–49, August 1986.

[26] Gary W. Sabot. The Paralation Model: Architecture-Independent Parallel Programming. The MIT Press, Cambridge, MA, 1988.

[27] Vivek Sarkar. Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1987. Technical Report No. CSL-TR-87-328.

[28] Olin Shivers. Data-Flow Analysis and Type Recovery in Scheme. Technical Report CMU-CS-90-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1990.

[29] Clifford Walinsky and Deb Banerjee. A Functional Programming Language Compiler for Massively Parallel Computers. In Proceedings of the ACM Conference on Lisp and Functional Programming, pages 131–138, Nice, France, June 1990.

[30] Michael Wolfe. Optimizing Supercompilers for Supercomputers. The MIT Press, Cambridge, MA, 1989.