Size and Access Inference for Data-Parallel Programs

Siddhartha Chatterjee Guy E. Blelloch Allan L. Fisher

March 1991

CMU-CS-91-118

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 26–28, 1991.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

Keywords: Data-parallel programming, size inference, access inference, compilers

Abstract

Data-parallel programming languages have many desirable features, such as single-thread semantics and the ability to express fine-grained parallelism. However, it is challenging to implement such languages efficiently on conventional MIMD multiprocessors, because these machines incur a high overhead for small grain sizes. This paper presents compile-time 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 partition the program graph into *clusters* of nodes such that all nodes in a cluster have the same loop structure, and further refine 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. A compiler has been implemented based on these ideas. We present performance results for data-parallel kernels analyzed by the compiler and converted to single-program multiple-data (SPMD) code running on an Encore Multimax.

1 Introduction

The diversity of parallel computer hardware makes it very difficult to port parallel programs without sacrificing execution efficiency. Research in portable parallel programming has largely taken one of two forms: those based on parallelism extraction [34] and those based on virtual machine emulation [17]. However, both approaches have their drawbacks. Parallelism extraction systems can only extract parallelism that exists in the original code. Virtual machine emulators are usually limited in the machine topologies they can support, and often have large overheads associated with mapping and emulation. They also usually rely on the user to solve the difficult problem of program and data partitioning.

We believe that parallelism should be explicit in the source program. However, 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 (large) 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 [5, 16, 31].

Data-parallel languages have historically been linked with SIMD parallel computers, and researchers have largely shied from implementing such languages on MIMD parallel machines. A naive implementation of data parallelism on a MIMD machine has the following performance bottlenecks, which affect both the asymptotic performance of the parallel program and the performance for small problem sizes.

- The parallelism of the source language is too fine-grained for the multiprocessor to handle. The startup overheads are too large. Since loop overhead scales with problem size, it limits the asymptotic performance of the parallel program. Serial overhead related to load balancing, on the other hand, depends on the machine, but is independent of the problem size, and therefore influences the small problem size performance but not the asymptotic performance.

- The implicit lock-step synchronization of the data-parallel language is expensive to implement on MIMD machines, whereas it comes for free on SIMD machines. As the cost of a barrier on a MIMD machine depends on the number of processors but is independent of problem size, this only affects the performance at small problem sizes.

- The intermediate results generated by the fine-grained parallelism cause problems for the memory organizations typically found in MIMD machines. In particular, locality of reference and its concomitant benefits are compromised. Loss of locality increases data access times; it scales with the problem size, and limits asymptotic performance.

This paper demonstrates how to solve these problems, and how to make data parallelism an appropriate programming model for both classes of machines. Maintaining efficiency on MIMD machines requires aggregating the fine-grained operations into larger-grained tasks (loops) and relaxing the lock-step synchronization while maintaining semantic equivalence. The aggregation of multiple operations also allows traditional code improvement techniques to be applied to the aggregate. Recently, Quinn and Hatcher [28] have demonstrated techniques for compiling the data-parallel language C* [30] for MIMD multiprocessors. However, the programs they can handle are limited by the restricted semantics of C*. Our work goes beyond this, showing how to handle loops with dependences, such as the scan primitive of APL [19].

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. The steps involved in this process are as follows:

Size inference: This technique is used to derive symbolic relations between "sizes" of vectors (the variables in the data-parallel program) based on the semantics of operations. This is completely symbolic; the user does not have to specify sizes of vectors. The relations derived for vector sizes are used to characterize the loop structures of the operation nodes.

Cluster formation: The size information produced by the previous step is used to partition the program graph into *clusters* based on loop structure and scheduling constraints. Operations that have provably different loop structures are put into different clusters. Clusters serve as units of work allocation.

Access inference: This technique analyzes generation and consumption patterns of vectors within clusters and identifies conflicts requiring synchronization.

Epoch formation: The conflict information produced by access inference is used to subdivide clusters into *epochs*. Operations within a single epoch do not require synchronization and can be performed in any order that respects data dependences.

Epochs map fairly naturally to loops, which can be executed in parallel by scheduling different iterations of the loop on different processors [34]. Our adoption of loop parallelism as our model of parallel execution has been guided by the Fortran experience, which suggests that this form of parallelism is well-matched to the capabilities of MIMD multiprocessors. In our model, some number of threads (specified as a command line parameter) are spawned when the program begins execution, and remain active until the program terminates. Other models of parallel execution, such as macro-actors [32] or functional pipelines [11], are beyond the scope of this paper.

A complementary issue is that of storage. A naive implementation often creates many unnecessary large intermediate results. Our storage allocation techniques tie in with our analysis to remove such intermediate storage, or reduce it to storage for a small section of a vector. This is similar to "drag-through" transformations in APL or loop fusion in Fortran.

The work described here is related to compilation techniques for APL, FP [3] and similar languages, the compilation of C* for multiprocessors, and optimizations performed by Fortran compilers. It differs in the kinds of operations it can handle, and we will return to these differences further in Section 10.

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

Granularity: The grain size of the output program is larger than that of the input program, making it suitable for execution on MIMD multiprocessors.

Synchronization: The only synchronization in the output program is that required for maintaining semantic equivalence with the original program.

Storage: Storage requirements are reduced, and many temporary vectors are eliminated. Vector storage can be reduced to storage for small sections of vectors, and can therefore take advantage of scalar and vector registers.

Locality: Combining multiple operations into single loops improves locality of reference. This can take advantage of chaining in vector machines, and of registers and caches in general.

Name	Symbol	init	kernel	lo, hi
plus	+		`out[i] = in1[i] + in2[i]`	`0, in1_siz`
times	*		`out[i] = in1[i] * in2[i]`	`0, in1_siz`
not	!		`out[i] = ! in1[i]`	`0, in1_siz`
select	SEL		`out[i] = in1[i] ? in2[i] : in3[i]`	`0, in1_siz`
plus-scan	+\	`out[0] = 0`	`out[i] = out[i-1] + in1[i-1]`	`1, in1_siz`
plus-reduce	+/	`out = 0`	`out = out + in1[i]`	`0, in1_siz`
min-reduce	MIN/	`out = INT_MAX`	`out = min(out, in1[i])`	`0, in1_siz`
length	LEN	`out = in1_siz`		`0, 0`
dist	DIST		`out[i] = in1`	`0, in2`
distv	DISTV		`out[i] = in1`	`0, in2_siz`
permute	PERM		`out[in2[i]] = in1[i]`	`0, in1_siz`
bpermute	BPERM		`out[i] = in1[in2[i]]`	`0, in1_siz`
dpermute	DPERM	`out[i] = in3[i]`	`out[in2[i]] = in1[i]`	`0, in1_siz`
index	INDEX		`out[i] = i`	`0, in1`
get	GET	`out = in1[in2]`		`0, 0`

Table 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`. All operations except the permutes can be done in-place.

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 inputs and outputs of the operations. These are similar to dataflow graphs [25], except that operations compute on entire vectors at a time. The primitive data type is the homogeneous vector of atomic types. Scalars are treated as singleton vectors. The primitive operations are shown in Table 1 and 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 [2]). Any set of primitives can be used as long as each primitive has an efficient parallel implementation. This excludes operations such as nonassociative scans as primitives. The set we have chosen is based on the scan vector model of computation [5], and is part of an experimental data-parallel language called VCODE [6]. It is the ability to handle scans and permutations that sets our work apart from other work in this area. The code can be equivalently represented in single-assignment form, which we also use (with a LISP-like syntax) for ease of understanding.

Using graphs as the internal representation of programs allows us to formulate the analysis problems in graph-theoretic terms, and also gives a good handle on storage optimizations. It also allows us to handle various input languages without much effort. We now give a formal definition of a computation graph for a function. This is similar to the IF1 graph representation for SISAL functions [25].

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** $\in T = \{$*SIMPLE, IF, FNCALL, INPUT, OUTPUT*$\}$.

– **op** *is: the* **operation** *performed by n, if n.t = SIMPLE; the function called by n, if n.t = FNCALL; 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 inputs and outputs of the nodes. An edge $e \in E$ is the pair $((n_s, p_s), (n_d, p_d))$, 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;* $1 \leq p_s \leq n_s.out$.

  – $n_d \in N$ *is the node at which the edge terminates.*

  – p_d *is the input port of* n_d *at which the edge terminates;* $1 \leq p_d \leq n_d.in$.

We abbreviate an edge to (n_s, n_d) *if ports are unimportant, or can be inferred from context.*

There are two control flow mechanisms: an if-then-else form, and recursion (for iteration). The `IF` node contains a `THEN` subgraph and an `ELSE` subgraph, and thus introduces hierarchy in the computation graph. One or the other of the subgraphs is executed, depending on the value of a scalar Boolean control input. The two subgraphs must produce the same number of results of matching types. It would be easy to add other structured looping constructs to the language.

The complete language model [6] supports additional data types, segmented versions of the operations [5], and function definition/call as encapsulation mechanisms. For the present, we ignore segmentation and assume that all nonrecursive function calls are inlined, and defer a discussion of these issues to Section 9.

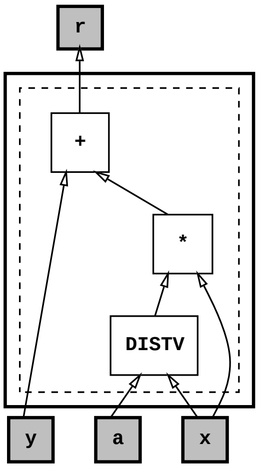
The parallel execution model is based on data partitioning, *i.e.*, each processor is responsible for a contiguous portion of the vector. The output of the compiler is single-program multiple-data (SPMD) loop code [21] suitable for execution on MIMD multiprocessors. Storage is of two types: *vector* storage, which holds a complete vector, and *buffer* storage, which holds an iteration's worth of computation on a vector.

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 have the same loop structure. 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, allocating vector and buffer storage where needed. Finally, we generate code in a straightforward manner for each epoch.

3 A Few Examples

Before getting to the details of the analysis techniques, we present a few simple examples to give the reader a feel for the end-to-end effect of the techniques. For each example, we show the Lisp-like code for the original program, the original program graph, the (uniprocessor) C code generated by the compiler, and running times for: (a) efficient serial code for the problem, (b) uniprocessor code generated by the compiler, and (c) multiprocessor code generated by the compiler. All times are on the Encore Multimax [14]. The data size is 64000 elements, and the multiprocessor times are on eight processors.

The first example (Figure 1) is the SAXPY operation. This computation takes two vectors $\vec{X}$ and $\vec{Y}$ and a scalar A, and returns $A.\vec{X} + \vec{Y}$. This routine is one of the Basic Linear Algebra Subprograms [24]. The straightforward implementation requires three loops (one each for distribution, multiplication, and addition), and storage for the distributed scalar and the intermediate result of the multiplication. The




```
(defun saxpy (a x y)
    (+ y
       (* (distv a x)
          x)))
```

```
for (i = 0; i < x_siz; i++) {
    r[i] = y[i] + (a * x[i]);
}
```

```
Serial:                   833.8 ms
Uniprocessor:             882.6 ms
Mulitprocessor (8 way):108.7 ms
```

Figure 1: The SAXPY operation. The figure shows the original program, the computation graph, and the final C code produced by the compiler. The dashed rectangles show the grouping of nodes into clusters and epochs. Running times are also shown for an input size of 64000 elements.

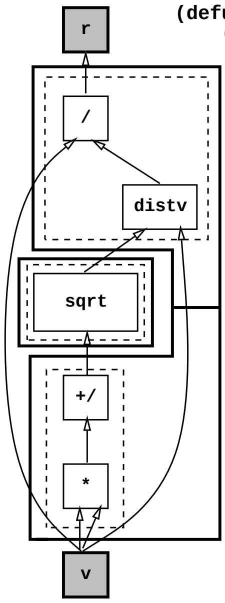
analysis techniques produce one cluster containing one epoch, and the C code produced by the compiler has the desired single loop with the distribution of the constant folded into the multiplication, and no intermediate storage.

The second example (Figure 2) is normalizing a vector, *i.e.*, dividing each element of the vector by its norm. The `sqrt` operation in the middle of the graph is a scalar operation separating two loops over the vector. A naive transcription produces four loops, a scalar operation, and storage for two additional vectors. Size analysis infers that the `sqrt` operation is scalar, and the other nodes have the same loop structure. However, such a partitioning cannot be scheduled, and the vector cluster must be split into two as indicated. Each cluster has a single epoch. The C code produced by the compiler contains two loops and no additional vector storage.

The final example (Figure 3) is the `split` operation, which 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, maintaining order within each part, and returns the resulting vector. This operation forms the core of sorting algorithms such as radix sort and stable quicksort [22]. The computation involves two plus-scans, a plus-reduction, three elementwise operations, a distribute, and a final permutation. The unoptimized code has eight loops and six intermediate vectors. Size inference infers that all the nodes can be put into a single cluster. However, the scalar output of the `+/` node causes an access conflict, which results in the cluster being split into two epochs as shown. In the multiprocessor case shown on the right, the scans must be split to allow multiprocessing. (The reason for this is explained in Section 7.) In either case, the C code produced by the compiler has two loops and one intermediate vector.

4 Size Inference

Our intent is to partition the program graph into chunks that can be executed in parallel, with synchronization and loop partitioning occurring only between chunks. Since all our primitives can be expressed as 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. In this section, we 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. In Section 6, we show




```
(defun normalize (v)
    (/ v
       (distv
            (sqrt (plus-reduce (* v v)))
            v)))
```

```
sum = 0.0;
for (i = 0; i < v_siz; i++) {
    sum += v[i] * v[i];
}
sum = sqrt((double) sum);
for (i = 0; i < v_siz; i++) {
    r[i] = v[i] / sum;
}
```

```
Serial:                   1784.4 ms
Uniprocessor:             1901.6 ms
Multiprocessor (8 way): 248.9 ms
```

Figure 2: Normalizing a vector. The figure shows the original program, the computation graph, and the final C code produced by the compiler. The dashed rectangles show the grouping of nodes into clusters and epochs. Running times are also shown for an input size of 64000 elements.

how to use this information to partition the graph into clusters, such that all nodes within a cluster have the same iteration space size.

4.1 Node and edge attributes

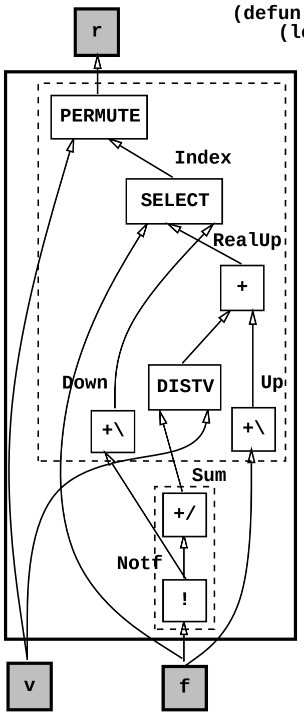
We associate a vector with each output port of every node in the graph, and map the vector associated with (n_s, p_s) to each edge *e* = $((n_s, p_s), (n_d, p_d))$. The *size* of a vector is the number of elements it contains. Given this map between vectors and edges, we define the following attributes for a graph edge *e*:

- *size(e)*, the *size* of the vector associated with that edge.
- *gen(e)*, the *generation pattern* of the vector; this refers to the order in which the elements of the vector are generated by the source node of the edge.
- *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 at the destination node of the edge.

The *gen(e)* and *use(e)* attributes are required only for access inference (Section 7), and will not be used further in this section.

Analogously, an operation node *n* (*i.e.*, one of type `SIMPLE`) has the following attributes:

- A set *In(n)* of constraints on the sizes of its input vectors of the form $l_i = l_j$ or $l_i = k$, where $1 \leq i, j \leq n.in$, l_i is the size of the vector associated with the edge *e* = $((n_s, p_s), (n, i))$, and *k* is a constant; these constraints define the vector sizes for which the computation is well-formed. The only value of *k* we currently use is 1.
- A set of transfer functions *Out*(n, i) ($1 \leq i \leq n.out$) that compute the sizes of the output vectors of the node in terms of the *sizes* of its input vectors.




```
(defun split (v f)
    (let* ((NotF (not f))
           (Up (plus-scan f))
           (Down (plus-scan NotF))
           (Sum (plus-reduce NotF))
           (RealUp (+ (distv Sum v) Up))
           (Index (select f RealUp Down)))
  (permute v Index)))

Example:

    v        [5 4 8 1 0 9 2]
    f        [0 1 1 0 1 0 0]
    Notf     [1 0 0 1 0 1 1]
    Up       [0 0 1 2 2 3 3]
    Down     [0 1 1 1 2 2 3]
    Sum       4
    RealUp   [4 4 5 6 6 7 7]
    Index    [0 4 5 1 6 2 3]
    r        [5 1 9 2 4 8 0]
```

```
sum = 0;
for (i = 0; i < v_siz; i++) {
    sum += (Notf[i] = ! f[i]);
}
up = down = 0;
for (i = 0; i < v_siz; i++) {
    r[f[i]?sum+up:down] = v[i];
    up += f[i];
    down += Notf[i];
}
```

```
Serial:                      825.9 ms
Uniprocessor:               1402.4 ms
Multiprocessor (8 way):      299.9 ms
```

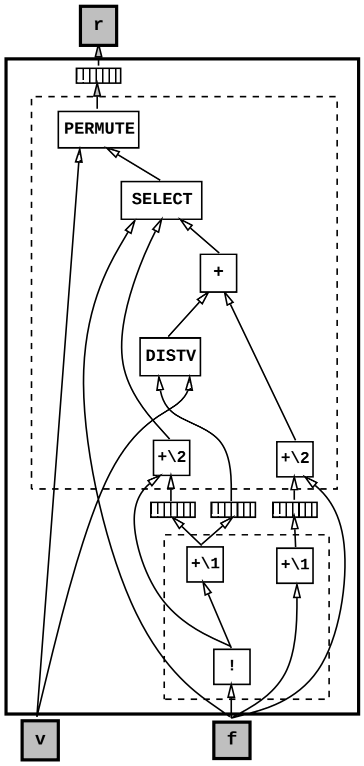


Figure 3: The `split` operation. The figure shows the original program, the computation graph, an example computation, and the final C code produced by the compiler. The graph on the left (respectively, right) is the uniprocessor (respectively, multiprocessor) version. The dashed rectangles show the grouping of nodes into clusters and epochs. Running times are also shown for an input size of 64000 elements.

- An iteration size function *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 is important as nodes can take inputs of several sizes, or produce outputs whose sizes differ from those of the input.

The node and edge attributes of the operations shown in Table 1 are shown in Table 2. Note that the size of the output may be different from the iteration space size, as for the `+/` operation.

4.2 The basic algorithm

The goal of size inference is to symbolically determine which nodes of a computation graph have the same loop structure. To this end, we first symbolically determine the *size(e)* attributes for the edges, and then use the formula for *Loop(n)* to determine the loop structures of the nodes. With perfect information, we could 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 always had the same size at runtime in order for the computation to be well-formed. Such an assignment 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 lack of information shows up as 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 approximation to this ideal; if

n.op	*In*(n)	*Out*(n, i)	*Loop*(n)	*size*	*gen*	*use*
`+`	$\{l_1 = l_2\}$	l_1	l_1	l_1	ind	ind, ind
`*`	$\{l_1 = l_2\}$	l_1	l_1	l_1	ind	ind, ind
`!`	$\emptyset$	l_1	l_1	l_1	ind	ind
`SEL`	$\{l_1 = l_2 , l_1 = l_3\}$	l_1	l_1	l_1	ind	ind, ind, ind
`+\`	$\emptyset$	l_1	l_1	l_1	ind	ind
`+/`	$\emptyset$	1	l_1	1	acc	ind
`MIN/`	$\emptyset$	1	l_1	1	acc	ind
`LEN`	$\emptyset$	1	1	1	ind	ind
`DIST`	$\{l_1 = 1 , l_2 = 1\}$	$\perp$	l_0	l_0	ind	ind, ind
`DISTV`	$\{l_1 = 1\}$	l_2	l_2	l_2	ind	ind, unused
`PERM`	$\{l_1 = l_2\}$	l_1	l_1	l_1	arb	ind, ind
`BPERM`	$\emptyset$	l_2	l_2	l_2	ind	arb, ind
`DPERM`	$\{l_1 = l_2\}$	l_3	l_1	l_3	arb	ind, ind, ind
`INDEX`	$\{l_1 = 1\}$	$\perp$	l_0	l_0	ind	ind
`GET`	$\{l_1 = 1 , l_2 = 1\}$	1	1	1	ind	ind, ind

Table 2: Node and edge attributes for the data-parallel operations defined in Table 1. l_0 is the size of the output vector. Each node shown here has exactly one output. A *Out*(n, i) value of $\perp$ indicates that the output sizes cannot be computed in terms of the input sizes. The edge attributes are explained in Section 7.

two edges are assigned the same size label, then the vectors associated with them are guaranteed to always have the same size at runtime if the function call is well-formed. In order to make this approximation, we restrict *Out*(n, i) so that it returns either the size of one of the input vectors, a constant, or the reserved value $\perp$.

In general, the following cases may arise in the course of analysis:

- 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, making some operation ill-formed. This is a program error, and the program is rejected.

- The compiler infers that two vectors must (in general) have different sizes, but cannot determine a functional relation between the sizes due to lack of knowledge in the compiler (about symbolic arithmetic on vector sizes, for instance).

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

- The compiler determines that the operation is well-formed provided two vectors have the same size, but can neither prove nor disprove this equality, since it may depend on runtime values. The compiler proceeds on the assumption that the sizes are equal, but inserts code to check for equality at runtime.

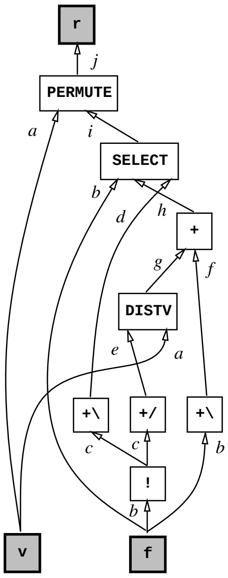


Figure 4: How size inference works on the `split` operation. The figure shows the initial assignment of size labels to the graph edges.

The *Loop* attribute of nodes is trivial to compute given the *size* attribute of edges. The key component of this evaluation, therefore, is the assignment of size labels to edges of the graph. The method 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. Arbitrarily assign distinct symbolic sizes to each vector of the computation graph, and assign this size to each edge the vector maps to.
2. Form a system of equations $\mathcal{E}$ representing the constraints on symbolic sizes as follows.
   (a) For each non-`IF` node *n* in the graph, instantiate each element of *In*(n) with the sizes assigned to its input edges and add the constraint to $\mathcal{E}$.
   (b) For each node *n* in the graph whose transfer function does not return $\perp$, instantiate *Out*(n, i) with the sizes assigned to its input edges, equate it to the size assigned to the output vector, and add it to $\mathcal{E}$.
   (c) For each `IF` node in the graph:
      i. Equate the size of the boolean input to 1 and add it to $\mathcal{E}$.
      ii. Recursively label the then- and else-subgraphs.
      iii. Merge the output sizes from the two subgraphs. This point is elaborated in Section 4.3.
3. Solve $\mathcal{E}$. Given the forms of the constituent equations, the solution partitions 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.

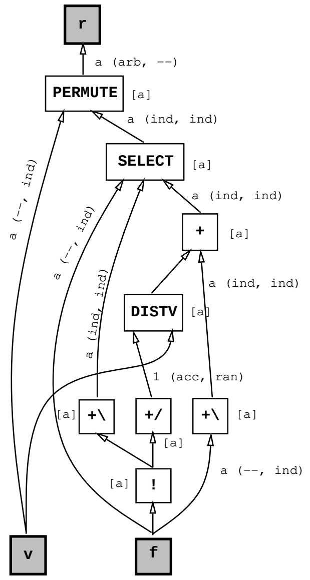


Figure 5: Computation graph of `split` operation, with node and edge attributes shown. The edge label is the size, with generation and use patterns in parentheses. The node label (in square brackets) gives the size of the iteration space.

To illustrate how the above algorithm works, consider the `split` operation. Given the initial labeling of the edges as shown in Figure 4, and using the node characteristics shown in Table 2, we get the following system of equations:

$$\mathcal{E} = \{b = c, c = d, e = 1, b = f, g = a, g = f, h = g, b = d,$$

$$b = h, i = b, a = i, j = a\}$$

which upon solution yields the following two equivalence classes:

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

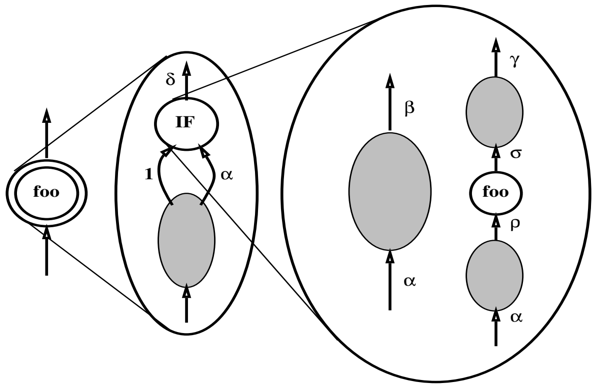
This gives the edge labeling shown in Figure 5. The other node and edge attributes are easily computed given the size labels on the edges and are also shown in Figure 5.

4.3 Refinements to the basic algorithm

A complication arises in Algorithm 1 concerning `IF` nodes. In general, it is not necessary for corresponding results from the `THEN` and `ELSE` subgraphs to have the same size. Therefore, if the result vectors have different symbolic sizes, we must take a conservative view and assign a new symbolic size after merging. This, however, is not as discriminating as we would like. In particular, it works poorly in the presence of recursive function calls. There are some important special cases where we can do better.

We classify `IF` nodes into two categories: simple and recursive. We only consider self-recursion for the moment. Simple `IF` nodes do not have recursive calls in either branch, while a recursive `IF` node has a recursive call in at least one branch. If a function contains a recursive `IF` node, wellformedness requires that there be at least one source-to-sink path in the graph free of recursive calls.

As simple `IF` nodes do not present any problems for the size inference algorithm, we will only consider recursive `IF` nodes. The general form of a recursive `IF` node with a recursive call in one branch is shown in



$$\begin{array}{rcl} true & \Rightarrow & (\delta =< \beta, \gamma >) \\ (\sigma = \gamma) & \Rightarrow & (\sigma = \beta) \wedge (\delta = \beta) \\ (\alpha = \beta) \wedge (\sigma = \gamma) & \Rightarrow & (\delta =< \rho, \alpha >) \\ (\alpha = \beta) \wedge (\alpha = \rho) \wedge (\sigma = \gamma) & \Rightarrow & (\delta = \alpha) \\ (\beta = 1) \wedge (\sigma = \gamma) & \Rightarrow & (\delta = 1) \wedge (\sigma = 1) \end{array}$$

Figure 6: Size possibilities for a recursive `IF` node with a recursive call in one branch. The Greek symbols indicate vector sizes, and the right hand sides of the equations indicate inferences that may be made if the conditions on the left hand side are fulfilled. The shaded ovals represent arbitrary nonrecursive computation graphs.

Figure 6. Without loss of generality, we assume that the `IF` node is the last node in the function definition. In the absence of additional information, we can only infer that the output size δ is either β or γ. If we know that the computation following the recursive call is size-preserving, then we can conclude that the final output size will be β. Note that β may not be simply related to α. If, in addition, the nonrecursive branch is size-preserving, all we can conclude is that the final output size is either ρ or α. Now, if we also know that the computation preceding the recursive call is size-preserving, we can infer that the entire computation is size-preserving. The last line deals with the special case where the output variable is a scalar.

The compiler incorporates these refinements, choosing a new symbolic size when there are multiple possibilities for a size.

4.4 Relationship to other work

This work is related in a general way to research in abstract interpretation; Algorithm 1 is superficially similar to the type inference algorithm of ML [26]. However, the two differ in the following fundamental ways:

- Size inference still requires runtime checks of some equalities. Type inference is completely a compile-time operation.
- The output sizes of a function can be unrelated to the input sizes; output types must be related to input types.
- Sizes need not match in the two branches of an IF statements to guarantee well-formedness. Types must match.

The `domain` construct in C* essentially provides the information that size inferencing computes. However, the C* model is more restrictive, since domains cannot be created dynamically within a program, and domain sizes must be known at compile time.

5 Scheduling

In the remainder of the paper, we will be replacing the partial order of the computation DAG with a linear order in various ways. The problem may be described as follows. We are given the DAG with some subset of its edges *marked* (represented as a mapping $\delta : E \rightarrow \{0, 1\}$), and are asked to pack the nodes into a chain of buckets, each of infinite capacity. If edge *(m, n)* is marked, we must place node *m* in a bucket that occurs earlier in the chain than the bucket in which we place node *n*. We also want to minimize the number of buckets used. The **length** of a packing is the number of buckets it consumes. A **schedule** of *G* under δ is a packing of minimal length, and its length is denoted $\mathcal{L}(G, \delta)$. Define δ on a path as the sum of the δ values of its constituent edges. Then the following lemma is obvious.

Lemma 1 $\mathcal{L}(G, \delta)$ = *1* + max$\{$*δ(P): P is a source-to-sink path in G*$\}$.

5.1 Lower and upper bounds

The scheduling problem has the characteristic that nodes have upper and lower bounds on their position in any schedule. We separate policy from mechanism by splitting the scheduling process in two: finding these bounds, and then actually choosing a schedule. The method for finding the bounds is given in Algorithm 2.

Algorithm 2 (l-h labeling of a graph.)

Input: A graph $G = (N, E)$, and a function $\delta : E \rightarrow \{0, 1\}$.

Output: Functions *l* and *h*: $N \rightarrow \{0, .., \mathcal{L}(G, \delta) - 1\}$ that give the lower and upper bounds on the position of a node in any schedule of *N* under δ.

Method:

1. Sort the node set *N* into list F such that all ancestors of a node precede it on the list. This can be done, for instance, by a breadth-first traversal of the DAG.

2. Sort the node set *N* into list B such that all descendants of a node precede it on the list. This can be done, for instance, by a breadth-first traversal of the graph with the edges reversed.

3. For each node *n* on F, in order:

$$\begin{aligned} l(n) &= 0, \text{if } n \text{ is a source node} \\ &= \min_{e=(m,n)} \{l(m) + \delta(e)\}), \text{otherwise.} \end{aligned}$$

4. Compute L = $\max_n \{l(n)\}$.

5. For each node *n* on B, in order:

$$\begin{aligned} h(n) &= L, \text{if } n \text{ is a sink node} \\ &= \max_{e=(n,m)} \{h(m) - \delta(e)\}, \text{otherwise.} \end{aligned}$$

5.2 Scheduling policies

Two obvious schedules are $S = l$ and $S = h$, where l and h are the labelings calculated by Algorithm 2. These schedules are derived from purely local considerations.

The next natural step is to determine a "good" schedule based on some notion of cost. In our context, a useful cost measure is the intermediate storage required by the schedule. We will see in later sections that storage is needed only for edges that cross between buckets, and the size of the vector associated with that edge gives the storage requirement for the edge. Let *Opt* be a schedule that minimizes *storage lifetime*, which we define as the product of the size of the storage and its extent. In the absence of additional information, we assume that all vectors have the same size, and normalize this to 1. We can formulate this as the following integer programming problem:

Minimize $\sum_{(m,n)\in E}(Opt(n) - Opt(m))$ subject to

$$\begin{array}{rcll} l(n) & \leq & Opt(n) & \text{(lower bound)} \\ Opt(n) & \leq & h(n) & \text{(upper bound)} \\ Opt(m) & \leq & Opt(n) - \delta((m,n)) & \text{(marked edge)} \end{array}$$

It is easy to verify that the corresponding linear programming problem has integer solutions, and hence it is sufficient to solve the linear programming problem to determine *Opt*.

The compiler provides switches for all these scheduling mechanisms. The default scheduling is $S = h$, since this is inexpensive to compute and also tends to reduce storage requirements.

5.3 Comparison with other work

It is interesting to compare our scheduling problem with previous work in the field, such as microcode compaction [15] and scheduling of macro-dataflow graphs [32]. Those techniques use integer weights on nodes and edges to represent computation and communication costs respectively. As our interest is in grouping nodes together rather than forming an actual timing schedule, we use 0/1 weights for edges and no weights at all for nodes. Further, the critical resource in our problem is related not to nodes (as in microcode compaction) but to edges.

6 Cluster Formation and Refinement

Size inference provides information about the iteration space sizes of the graph nodes. We want to group together computation nodes that have the same *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. As all nodes in a cluster have the same iteration space size, the data needs to be divided among the physical processors only at cluster entry. Thus, clusters serve as natural units of loop partitioning (load balancing). We are interested in making clusters as large as possible, since the optimizations that follow do not go beyond cluster boundaries. We now present a framework for achieving these ends.

Definition 2 (Critical edge) *A graph edge* (n_s, n_d) *is said to be* **critical** *iff Loop*(n_s) $\neq$ *Loop*(n_d).

Definition 3 (Cluster) *A* **cluster** $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ *of a computation graph G is a connected subgraph of G containing no critical edges.*

Definition 4 (Clustering) *A* **clustering** *P* = $\{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ *is a node partition of G into clusters.*

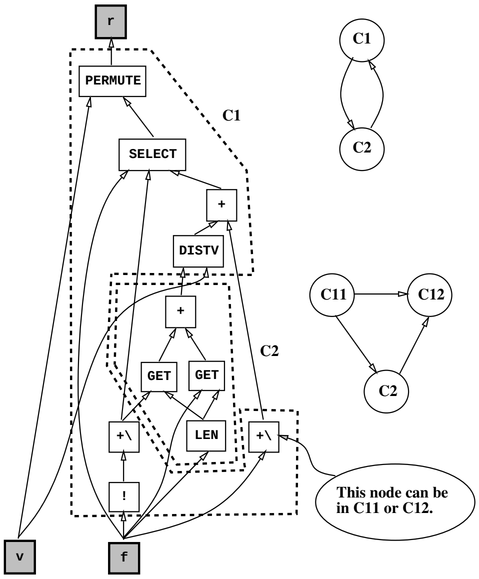


Figure 7: A program graph G, $M(G)$, and the condensation graph $C_G^{M(G)}$. The graph is that of the `split` operation where `+-REDUCE` is not considered a primitive operation. Instead, it is simulated by performing a `+-SCAN` on the vector, and adding the final values of the original and the scanned vectors. The maximal clusters of the graph are indicated by dashed lines, and condensation graphs are shown on the right. The upper graph is $C_G^{M(G)}$, which shows that $M(G)$ is not viable. The lower condensation graph is acyclic, but the mapping of nodes to clusters is not unique, as shown by the indicated `+-SCAN` node.

Definition 5 (Refinement) *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 $M(G)$. It is clear that $M(G)$ is unique. Define the *condensation graph* of a computation graph under a clustering as follows:

Definition 6 (Condensation graph) *Let P =* $\{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ *be a clustering of a computation graph G = (N, E), where* $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{E}_i)$*. The* **condensation graph** *of G under P is the graph* C_G^P *= (P,* E_P*), where* $(\mathcal{G}_i, \mathcal{G}_j) \in E_P$ *iff there exist nodes* $n_s \in \mathcal{N}_i$ *and* $n_d \in \mathcal{N}_j$ *(i ≠ j) with* $(n_s, n_d) \in E$*.*

Definition 7 (Viability) *A clustering P of* G *is said to be* **viable** *if* C_G^P *is acyclic.*

The condensation graph collapses each cluster to a vertex and captures the data dependences between clusters of a computation graph. 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 7 shows. However, the following property holds of any clustering (hence, of any viable clustering), due to the maximality of $M(G)$.

Lemma 2 *Any clustering of G is a refinement of* $M(G)$*.*

As we are interested in finding large clusters, the only candidates for refinement are the clusters that form some non-trivial strongly connected component (SCC) of $C_G^{M(G)}$. We need a way to break the cycle

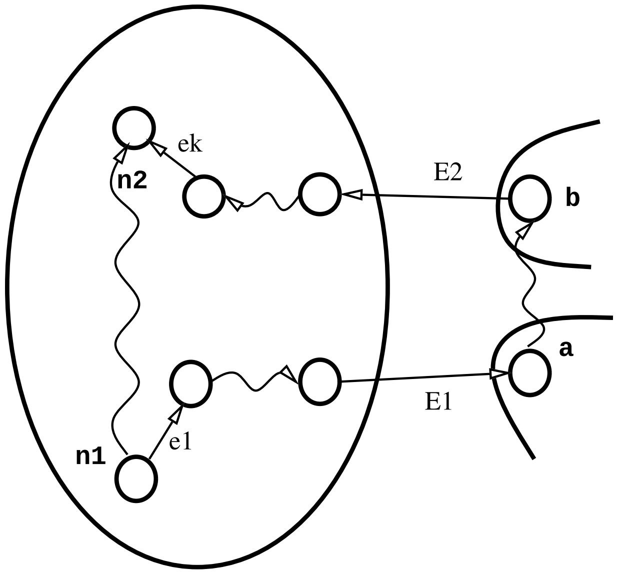


Figure 8: Proof of the viability lemma. Straight lines represent edges, and wavy lines represent paths in the graph.

in an SCC in order to achieve viability. While the minimum number of breaks required for this is unique, there can, however, be several ways of breaking the cycle with that number of breaks. For instance, in Figure 7, 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 3 *Two nodes n_1 and n_2 cannot be in the same cluster of a viable clustering if there exists a path from n_1 to n_2 containing a critical edge.*

Proof: By contradiction. By definition, the two nodes cannot be in the same cluster if $Loop(n_1) \neq Loop(n_2)$. We therefore only consider the case where *Loop(n_1) = Loop(n_2).* Refer to Figure 8 and suppose that n_1 and n_2 satisfying the given condition are in the same cluster $\mathcal{G}_n$ of a viable clustering P. Observe that since n_1 and n_2 are in the same cluster, there must be at least two critical edges on some path between them, one leaving the cluster and another entering it. Call these critical edges E_1 and E_2. Now consider the path p = $(e_1, \ldots, E_1, \ldots, E_2, \ldots, e_k)$ between n_1 and n_2. Since E_1 and E_2 are critical edges, *Loop(a) $\neq$ Loop(n_1)* and *Loop(b) $\neq$ Loop(n_2).* Let nodes *a* and *b* be in clusters $\mathcal{G}_a$ and $\mathcal{G}_b$ (not necessarily distinct) of P. But then C_G^P contains the cycle $\mathcal{G}_n \rightarrow \mathcal{G}_a \rightarrow \mathcal{G}_b \rightarrow \mathcal{G}_n$, and hence P is not viable. □

We call (n_1, n_2) above a **critical pair** if a path between them begins and ends with critical edges. To separate the elements of critical pairs, we add *separator edges* between them, producing the separator graph $G_S = (N, E \cup E_S)$, 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. As explained in Section 5, the number of refinements required to make a cluster viable is equal to the maximum number of separator edges on any path within the cluster, but there is some flexibility in the actual assignments of nodes to refined clusters. Intuitively, separator edges must span refined clusters. A viable partition of a graph is generated by Algorithm 3.

Algorithm 3 (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 $C_G^{M(G)}$ [1, p. 193].

3. For each $l \in L$:

   (a) Identify critical pairs and create the separator graph l_S.

   (b) Apply Algorithm 2 with G = l_S and δ defined as follows:

$$\delta((i,j)) = \begin{cases} 1 & \text{if } (i,j) \in E_S \text{ of } l_S \\ 0 & \text{otherwise.} \end{cases}$$

4. Assign refined cluster numbers to nodes *n* from the range $[l(n), h(n)]$, using one of the scheduling policies mentioned in Section 5.2.

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

7 Access Inference

Each cluster of a viable clustering 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 in C.

Recall that we defined *gen(e)* and *use(e)* attributes for edges in Section 4. We now define the possible values of these attributes. We choose *gen(e)* and *use(e)* from the following set of stylized patterns:

1. *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.

2. *ind*: generated/consumed in index order.

3. *acc*: generated by accumulation. This means that the value is available only after all iterations of the loop have completed.

4. *arb*: generated/consumed in some data-dependent order that cannot be predicted at compile time, *e.g.*, in a `PERMUTE` operation.

These patterns are representative, not exhaustive. Also, because 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, *arb* being more constrained than *ind*, which is more constrained than *unused*. Table 2 shows the edge attributes for various nodes.

The language primitives can be divided into the following groups: elementwise operations, structure accessors (such as `LENGTH`), permutes and distributes, and scans and reductions. The first three groups

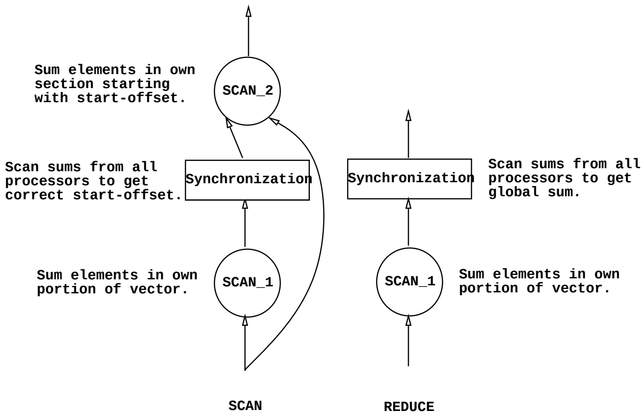


Figure 9: Templates for SCAN and REDUCE operations that expose their microstructure.

can be implemented on a multiprocessor in a single loop, as there are no loop-carried dependences [34] in these operations. This is not true for scans and reductions, however, where processors need to communicate state information. 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.

The decomposition follows from the standard parallel algorithm for scans [23], where each processor performs the following three phases:

- Sum the elements in its partition of the data. 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 synchronization step. These details are relegated to 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 9. We replace occurrences of scan and reduce nodes with the corresponding templates, and perform common subexpression elimination to remove any redundant nodes. For the `split` operation, this results in the graph shown in Figure 10.

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 *ind* and *ind*), then storage 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 repartitioned, as we are still within a single cluster.) We capture this notion by defining an incompatibility relation $\mathcal{R}(gen(e), use(e))$ between *gen(e)* and *use(e)*, as shown in Table 3. We define the *access weight* of an edge as follows:

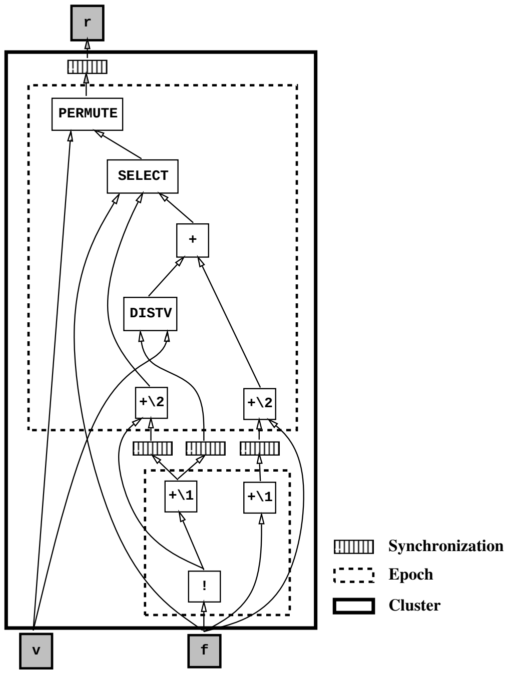


Figure 10: The `split` operation after template expansion and CSE. Cluster and epoch boundaries are shown, along with computation nodes and synchronization events.

		use(e)		
		unused	ind	arb
gen(e)	ind	0	0	1
	arb	0	1	1
	acc	0	1	1

Table 3: Incompatibility relation $\mathcal{R}(gen(e), use(e))$ between generation and usage patterns. A 0 entry indicates that the two patterns are compatible, while a 1 indicates an incompatibility.

Definition 8 (Access weight) *The access weight* $\delta_a(e)$ *of an edge* e = (n_s, n_d) *is 1 if* $n_s.op$ = `SCAN_1` *and* $n_d.op$ = `SCAN_2`*, and is equal to* $\mathcal{R}(gen(e), use(e))$ *otherwise.*

As explained in Section 5, 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. The upper and lower bounds can be computed for each node of a cluster by Algorithm 4, which is similar to Algorithm 3.

Algorithm 4 (Access inference of a cluster of a data-parallel computation graph.)
Input: A cluster C from a viable clustering of a data-parallel program graph.
Output: An epoch numbering of the nodes of the cluster.
Method:

1. Apply Algorithm 2 with G = C and $\delta = \delta_a$.

2. Choose an assignment of nodes to epochs, using one of the policies in Section 5.2.

The epoch assignments for the `split` operation are shown in Figure 10. 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.

8 Implementation and Results

We have implemented a compiler for data-parallel computation graphs incorporating the above ideas. The compiler produces C Threads code [13] suitable for execution on a shared-memory multiprocessor. In this section, we briefly discuss storage management and code generation, and then present some preliminary results on the Encore Multimax.

8.1 Storage management

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 three categories based on the persistence of the vector, as follows:

- Vectors that persist across epochs within a single cluster.
- Vectors that persist across clusters of a single function.
- Vectors that persist across functions.

This distinction is important because of the high overhead of parallel memory allocation on a multiprocessor. Heap storage is required for vectors that persist across functions, while stack storage is sufficient for the other two categories. The runtime system uses separate vector stacks for the first two categories. 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 easily augmented to optimize reuse of storage. The extension consists of taking into account the size of a memory block allocated for a vector when considering it for reuse.

8.2 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. This use of C as a universal assembly language allows the use of a single back end in the compiler. The compiler does, however, perform source-level transformations such as strength reducing array indexing to pointer incrementing, and unrolling loops to reduce loop overheads, as native C compilers are not very good at such optimizations. Code generation is done by traversing backwards through epochs; buffer storage associated with nodes with fanout is used to avoid recomputation of results.

8.3 Results

In Figure 11, we show preliminary performance numbers for eight test kernels, for various data sizes and number of processors. Note that the speedup is calculated with respect to a good serial algorithm for the problem, *not the parallel algorithm running on one processor*. In some cases the serial code uses completely different data structures and algorithms from those used by the parallel code. Source level tuning [4] was done for the serial programs, including loop invariant motion and converting array

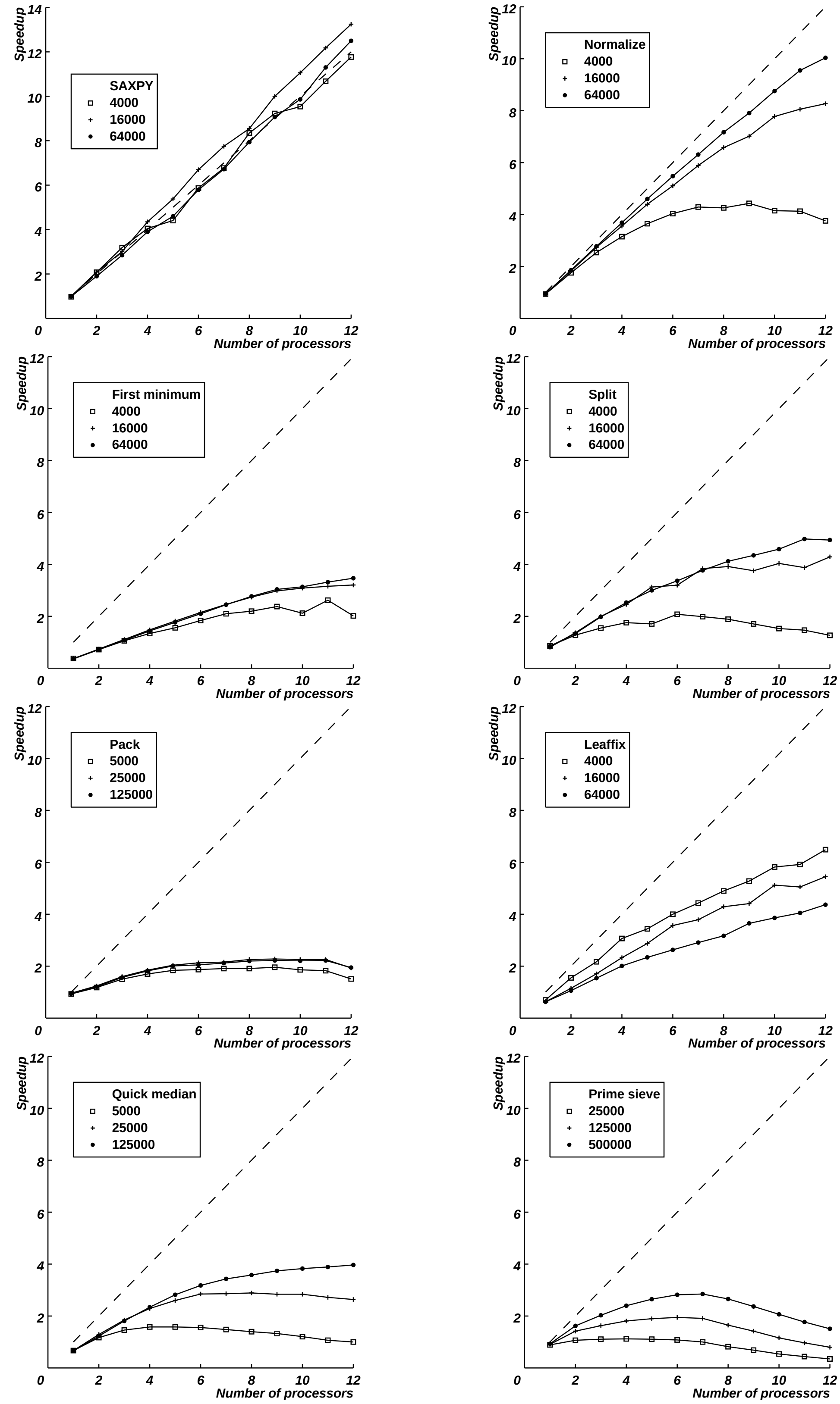


Figure 11: Speedup curves for the eight benchmarks kernels discussed in Section 8.3. Speedups are with respect to the best serial C code for the problem. The dashed line in each graph shows the ideal speedup curve.

indexing to pointer incrementing. The measurements were taken on a 16-processor Encore Multimax (with NS32332 processors and 96 Mbytes of main memory) running the Mach operating system [29]. 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. We now analyze the results in greater detail. We emphasize once again that the results are preliminary. We are currently adding more optimizations to the compiler, and we expect the numbers to improve in future.

Example 1: (SAXPY) This computation was discussed in Section 3. It is a straightforward elementwise loop, and shows the expected linear speedup. The superlinear speedup for moderate data sizes is a cache effect: the data overflows the cache on a single processor, but not on multiple processors.

Example 2: (Normalize) This computation was discussed in Section 3. The reduction at the end of the first loop causes a serial bottleneck. This, however, becomes less significant for larger data sizes, where the speedup shows a linear trend.

Example 3: (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 data-parallel algorithm involves two min-reductions, two distributes, a comparison, and the `INDEX` operation. Our analysis coalesces this graph into two loops with two synchronization points. The speedup is limited to about 3 because of the extra work done by the parallel algorithm. This is a result of the limited set of reduction operators we allow, and could be improved by augmenting the language model to allow reduction with user-defined functions.

Example 4: (Split) This computation was discussed in Section 3. Note that the parallel version of scan performs twice as many operations as the serial version. This is inherent to the parallel algorithm for scan and is not an artifact of the compiler. Any parallel algorithm for split requires this extra work. This limits the speedup we can expect. The additional memory traffic due to the permute gives a maximum speedup of about 5.

Example 5: (Pack) This takes a vector of values and a vector of flags, and returns only those values whose corresponding flags are 1. The serial algorithm can take advantage of the fact that the size of the result is less than that of the input to simplify computing the size of the output. The parallel version requires two loops, one to count the number of flags that are 1, and the second to actually permute the values into the result. This extra work and bus contention limits the speedup to about 2. The compiler does not yet generate optimal code for this benchmark.

Example 6: (Leaffix) This operation 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 as an Euler tour, and the details of the representation and algorithm may be found in [5]. This computation can be used as a kernel for many tree operations, such as determining the number of descendants for each vertex. 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. A speedup of 6 is achieved on this benchmark, which is close to the bound based on operation counts.

Example 7: (Quick-median) This computes the median of a vector of values using an algorithm of $O(n)$ average-case complexity. It is quite similar to quicksort. It chooses a pivot, and finds the number of elements whose value is less than the pivot value. Depending on this value, it either packs those elements or those that are greater than the pivot, and calls itself recursively on that packed vector. The graph contains nested recursive `IF` nodes. The synchronization required around recursive calls limits speedup to about 4 over the serial program. This could be improved with further optimizations of function calls, as explained in Section 9.1.

Example 8: (Prime sieve) This finds all prime numbers less than a given value using the sieve of Eratosthenes. A Boolean array distinguishes prime and composite numbers in the desired range. In successive steps, the next prime is located, and its multiples are marked as composite. The program generates a large number of writes, which show up as bus transactions because the caches are write-through. We therefore expect the performance to be limited by bus performance. We determined in a separate experiment that a steady write traffic causes bus saturation at seven processors, and the peaking of the speedup curve at that value confirms our hypothesis.

9 Extensions

9.1 Function calls

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

However, inlining may not always be possible 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.

If the compiler cannot access the function definition, the function call must be treated as a black box that is placed in a cluster by itself. This means that *gen(e)* and *use(e)* patterns for input and output edges must be considered to be *arb*, requiring storage and synchronization at those edges.

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:

- We can derive an end-to-end *Out*(n, i) for the function call by performing size inference on the definition and deriving the size of each output of the function in terms of the input sizes. 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. 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. This requires compiling a single-element version of the function to allow fusion 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. Partial inlining can be 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 the first of these optimizations, and we are working on incorporating the others.

9.2 Nested parallelism

As mentioned in Section 2, we can augment the language with segmented vectors and segmented versions of the primitive operations. These augmentations allow the implementation of nested parallelism [7]. The analysis techniques extend quite naturally to handle segmented operations. Segmented operations simply introduce more levels of loops, and the *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.

9.3 Alternative traversal orders

One drawback of the current analysis is the limited choice of generation and use patterns of vectors. In particular, this restricts our ability to analyze structured permutations such as reverse and rotate. We are working on refinements to access inference similar to ideas in [33] that would allow us to identify such permutations and use this information to choose alternative and more efficient traversal orders through the iteration space.

10 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 Fortran, and compilation of C* for multiprocessors. We now discuss how our work stands with respect to each of these.

Applicative languages: APL 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 [18], 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 [8, 9]. 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. Ching [12] presents results for compiling APL into System/370 assembly code. His type-shape analyzer is very similar to size inference, but his primary aim is to get tight bounds on the types, ranks and sizes of variables. He also does not treat multiprocessor issues such as load balancing and synchronization. Ju and Ching [20] present similar results. While they are aware of the benefits of loop fusion, their compiler does not perform this transformation automatically. Our techniques provide a systematic way of doing this.

Similar work has also been reported for FP by Budd [10], and more recently by Walinsky and Banerjee [33]. 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 [32] 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 clusters.

Fortran: Loop fusion [34] is a well-known optimization technique in Fortran. The idea there is to fuse adjacent loop bodies, thereby reducing loop overheads, and allowing for further interstatement optimizations. Its use is again limited 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 [27].

C*: Quinn and Hatcher have worked on compiling C* for MIMD machines [28]. Their work has some of the same goals as ours. It 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. They also do not attempt to perform source-to-source optimizations such as loop fusion. The `domain` construct in C* essentially provides the information that our size inferencing computes. However, the C* model is more restrictive, since domains cannot be created dynamically within a program, and domain sizes must be known at compile time. It is not clear how their techniques would handle scans, or extend to nested parallelism.

11 Conclusions

This paper has introduced two techniques for the analysis of data-parallel program graphs. The first, size inference, derives symbolic relations between the sizes of program vectors, and uses this information to partition the program graph into regions (called clusters) that have different loop sizes. The second technique, access inference, analyzes generation and usage patterns of vectors, and uses conflicts between these patterns to further refine clusters into epochs. These techniques are used to step up the grain size, reduce storage and synchronization requirements, and improve locality of data-parallel programs, making it viable to run them on traditional MIMD multiprocessors. The major contribution of this paper lies in demonstrating 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 results have been presented for several benchmarks.

Acknowledgments

We would like to thank Eric Cooper, Allan Heydon, Peter Lee, Margaret Reid-Miller, Jay Sipelstein, Peter Steenkiste, Jaspal Subhlok, and Marco Zagha for their comments on early drafts of this paper.

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] American National Standards Institute. *American National Standard for Information Systems Programming Language Fortran: S8(X3.9-198x)*, March 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] Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

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

[6] 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.

[7] 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, February 1990.

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

[9] 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.

[10] 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.

[11] Siddhartha Chatterjee and Prathima Agrawal. Connected Speech Recognition on a Multiple Processor Pipeline. In *Proceedings of the 1989 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 774–777, Glasgow, Scotland, May 1989.

[12] Wai-Mee Ching. Program Analysis and Code Generation in an APL/370 Compiler. *IBM Journal of Research and Development*, 30(6):594–602, November 1986.

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

[14] Encore Computer Corporation. *Multimax Technical Summary*. Encore Computer Corporation, 1988.

[15] 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.

[16] 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, Volume II*, pages 897–955, Pasadena, CA, January 1988.

[17] 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.

[18] 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.

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

[20] Dz-Ching Ju and Wai-Mee Ching. Exploitation of APL Data Parallelism on a Shared-memory MIMD Machine. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 61–72, Williamsburg, VA, April 1991.

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

[22] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1973.

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

[24] 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.

[25] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, 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.

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

[27] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In *Conference Record of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 79–92, Orlando, FL, January 1991.

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

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

[30] J. R. Rose and G. L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing, Vol. 2*, pages 2–16, San Francisco, CA, May 1987.

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

[32] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1987.

[33] 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.

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