The Planguages

L. Ridgway Scott

This talk is based on joint work with Ernesto Gomez, Terry Clark, Babak Bagheri and a host of others [15].
Figure 1: Death of the dinosaur supercomputers.
1 Structure of scientific codes

Figure 2: Parallel scientific computing has no natural parallelism. You must decide where to break dependences and incur communication and synchronization overhead.

The scalability challenge was set in the following e-mail message.
Message: 1023110, 88 lines  
Posted: 5:34pm EST, Mon Nov 25/85, imported: ....  
Subject: Challenge from Alan Karp  
From GOLUB@SU-SCORE.ARPA  
I have just returned from the Second SIAM Conference on  
Parallel Processing for Scientific Computing in Norfolk,  
Virginia. There I heard about 1,000 processor systems, 4,000  
processor systems, and even a proposed 1,000,000 processor  
system. Since I wonder if such systems are the best way to  
do general purpose, scientific computing, I am making the  
following offer.  
I will pay $100 to the first person to demonstrate a  
speed-up of at least 200 on a general purpose, MIMD  
computer used for scientific computing. This offer  
will be withdrawn at 11:59 PM on 31 December 1995.  
Some definitions are in order.  
.....
1.1 Standard notions of speedup

Speedup $S_P$ using $P$ processors:

$$S_P = \frac{T_1}{T_P}$$

Efficiency $E_P$ using $P$ processors:

$$E_P = \frac{S_P}{P} = \frac{T_1}{PT_P}$$

Amdahl’s Law says that if $f$ is the fraction of a code that is sequential then

$$S_P \leq \frac{1}{f + (1 - f)/P} \leq \frac{1}{f}$$
1.2 Data-dependent speedup

Suppose that $N$ measures the size of data in the problem.

Scaled speedup $S_{P,N}$ using $P$ processors:

$$S_{P,N} = \frac{T_{1,N}}{T_{P,N}}$$

Efficiency $E_{P,N}$ using $P$ processors:

$$E_{P,N} = \frac{S_{P,N}}{P} = \frac{T_{1,N}}{P T_{P,N}}$$

Amdahl’s Law broken when $f_N \to 0$ as $N \to \infty$.

A problem is scalable if $E_{P(N),N} \geq \epsilon > 0$ and $P(N) \to \infty$ as $N \to \infty$. 
Figure 4: Architectural space for parallel “supercomputers.”
Figure 5: Divide and conquer algorithm for reduction operations.

\[
mee = \text{myProc} \\
\text{id} = n \text{Proc} \\
\text{dee} = \log(\text{id})/\log(2) \\
\text{do } k=1,\text{dee} \\
\quad \text{gee} = \text{id} \times \text{int}(\text{mee}/\text{id}) \\
\quad \text{id} = \text{id}/2 \\
\quad \text{t}(\text{gee+id}) = \text{t}(\text{gee}) \\
\text{endo}
\]

Broadcast on a ring of \( P = 2^d \) processors using \( \log_2 P \) steps. All communication in this algorithm appears in a single line of code with two @ symbols.
2 Implementation strategy for fusions

The @ symbol denotes *guarded memory* [3].

To compile an expression such as

\[ t@p = t@q \]

use the “golden rule” (better to give than receive) [14]:

1. Processor q first sends data to processor p.
2. Then processor p receives data from processor q.

This order insures completion.

The processor identifiers can be functions of myProc.

\[ x = y @ \min(\text{myProc}+1, \text{nProc}-1) \]

There can be multiple values of q where q=f(myProc), so the sending processor may need to send data to multiple processors.
3 Failure mode

Fusions can fail if the sending processor is not aware of the fusion.
This can occur if branches split processors into separate groups.
Communication between separate groups is undefined.
A run-time system can track group splits and merges to allow determination of group membership in communications.
If no group membership failure is detected, computation is deterministic.
Currently implemented in PC [11].
4 Overlapping communications

Ernesto Gomez developed the SOS system in part to support implementation of fusions in branched parallel codes [9].

Each communication statement is replaced by a finite state machine that can perform the communication operation in different ways depending on the state of the receiver.

For example, in a shared memory system it may be possible to write directly to memory, bypassing a buffer system typically required in message passing.

This can reduce the overhead of communications substantially.
5 Determining merges of implicit process sets in SPMD parallel computation

SPMD (single program, multiple data) parallel computation has multiple copies of the same program running concurrently, each with its own data.

This style of programming has facilitated the successful deployment of parallel computers, from the ubiquitous small clusters of workstations now commonly used as a computational server for a single work-group to the worlds most powerful machines which comprise tens of thousands of processors.

The main advantage of SPMD programming is that you only have to write one program that can be interpreted independently by different, but cooperating, processes.
5.1 SPMD review

Different copies of SPMD codes have the same variable names, but potentially different values, for various reasons.

For example, SPMD codes always have a way to identify the process(or) number executing a particular copy of the code.

For simplicity, we identify that by the variable $\text{me}$ here.

Thus values in different processes could become distinct based on computations involving the variable $\text{me}$. 
5.2 Data exchange review

Different styles of data exchange are supported in different SPMD paradigms, but we will use the send-receive metaphor of message passing to describe all data exchange between processes.

To avoid writing “process(or)” repeatedly, we will use the word “process” to mean either as appropriate.

SPMD codes can be running on different processors, or on different processes in a single processor, or some combination of both.

There are several challenges with SPMD; we focus here on dead-lock detection. We give a simple example to illustrate one type of difficulty frequently encountered.
6 Problem motivation

Many important algorithms have a fundamentally irregular structure that does not map simply onto the SPMD model [10]. Even a standard algorithm for summing numbers in a ring of processors indicates what needs to be considered. Suppose that $x$ is the variable at each process to be summed, and we want $s$ to contain the sum at the end, at process number zero. For simplicity, suppose that the number $P$ of processes is a power of two: $P = 2^d$. Initialize $k=1$ and $s=x$. Then loop over $i = 1, \ldots, d = \log_2 P$ doing the following pseudo-code (the logical expression $k \mid me$ in the conditional in the first line is true if the integer $k$ divides the integer $me$ evenly):

\[
\begin{align*}
\text{if} \ (k \mid me) & \\
& \text{if} \ (me/k \text{ is odd}) \ \text{send} \ s \ \text{to process no.} \ (me-k) \\
& \text{if} \ (me/k \text{ is even}) \ \text{receive data from process no.} \ (me+k) \\
& \quad \text{and add it to} \ s \\
\text{else} \ & \text{do nothing}
\end{align*}
\]

$k=2*k$
6.1 Problem motivation, continued

The main features of this code are that there are conditionals that divide the processes into subsets which execute different code, the conditionals are nested and in loops, and communication occurs intertwined with all of this.

This type of programming is necessary in many cases to achieve the efficiency inherent in algorithms like this divide-and-conquer implementation of a standard reduction operation.

More complex examples of SPMD codes are common with these features. The SPMD model is often used to port legacy codes, for example [5].

The main difficulty that can arise in programs like the above is that communication can potentially involve processes which have executed different branches of the conditionals.
6.2 Problem motivation, continued

Of course, the simple code above can be checked for correctness and determinism, but we are interested in being able to handle the general case by automated techniques.

We focus on detection of a particular form of communication failure in which some process is waiting for another process to send it something, but the other process will never execute the required send because it is on a different conditional branch.

This kind of communication failure can be a form of “live” lock since it is an interaction between a stopped process and one that may not be.

We propose a system that uses compile-time analysis to adapt a run-time system tailored to the control structure of the code which allows detection of this type of communication failure and would potentially allow a graceful termination of a parallel execution.
6.3 Live-lock detection

There are two key aspects to detecting this type of live-lock.

First of all, it is essential to identify corresponding send-receive pairs.

We do this by a legislative approach, that is, we require all communications to be what we call fusions in which both sender and receiver are identified (at least a run time).

Secondly, it is essential to know which sets of processes are executing which branches of code.

For a general solution, we have to consider nested conditionals, loops and subroutines, and any potential interactions among them.

Identifying where process sets can split into separate sets executing different code is relatively easy: this only occurs at conditionals.

But the key is to identify (conservatively) where these separate sets can merge again.
6.4 Live-lock detection support

We identify a novel data structure, the *merge tree* which supports identification where these separate sets can merge again.

The use of a single text being followed by all processors in SPMD allows certain types of static program analysis.

Here we show how an SPMD code can be analyzed statically and code can be added so that communicating sets of processors can be identified correctly at runtime.

We describe situations in which this can lead to deterministic parallel computation.
7 Implicit process sets

Subsets of processes taking different paths can be formed when conditional predicates are different at different sets of processes.

These implicit process sets (IPS) are formed dynamically in a parallel computation and present challenges to parallel programming systems which are not met by present technology.

For example, the BSP model [17] does not allow for the formation implicit process sets. Similarly, MPI [12] provides support for explicit process sets, but not implicit ones.

Our work can be seen as an extension of the BSP model to allow for the formation of implicit process subsets which function for part of the computation like independent BSP computations.
7.1 Splits and merges

Determining points in a control flow graph (CFG) where IPS can form (which we call **splits**) is quite easy.

It just requires the identification of appropriate conditionals.

However, IPS which split may later **merge**, and the identification of such points in a CFG is the major objective of this work.

We describe a novel data structure, the **merge tree**, which facilitates the process of merging IPS into larger sets, eventually to the full set of processes initiating the computation.
7.2 Terminal processes

We denote the set of processes under consideration by $\Gamma$. We consider only a static set of processes, without dynamic process creation, for simplicity.

We are interested in computations in which all processes in $\Gamma$ terminate at a well-defined end state.

Thus we are able to apply our techniques recursively to subroutines.

If a subset of $\Gamma$ enters a subroutine, they will also all exit it, under conditions that we describe here.

Thus it is sufficient to explain what we must do to monitor a code without subroutines to be sure that it will terminate together in a single end state.
7.3 MIPS

We here propose a framework for the support of process sets that subdivide and recombine dynamically at runtime, which we call MIPS (merging, or modulating, implicit process sets).

We describe a method that will allow us to identify statically (at compile time) where such process sets can change during execution.

This allows us to insert code at these points to track these sets at runtime. Our framework will allow us to specify sufficient conditions on communications to guarantee that, if no individual process fails or diverges, each process will reach an end state.

We will also be able to set forth conditions to guarantee that the computation is deterministic.
7.4 Communication requirements

The most obvious requirement for communication not to deadlock is to have all processes involved in the communication that are needed.

Said conversely, if a process is expecting to receive information from another process that is not participating in any way, deadlock will occur.

We require that we have a conservative estimate of the set of processes that are attempting to communicate at each occurrence of communication.

Thus we see that tracking the formation and merging of implicit process sets is central to our approach.

Rather than work with a specific programming language, we work instead at the level of a CFG.
Figure 6: This CFG can result from the following control structures: Node S: an IF-THEN-ELSE statement, with the left portion of the graph being executed if the condition is true, and the right side of the graph being the ELSE case. Loop L9: a FOR or WHILE loop, with exit from the header node. Loop L1: a DO-WHILE loop in which the exit node is at the tail of the back edge instead of at the header. Node 1: a simple IF statement, which either permits the execution of the code in nodes 2, 3, 4, 5, 6 and 7 or skips directly to node 8. Node 2: a SWITCH statement (in C/C++) or a computed GOTO (in Fortran).
8 MIPS and the CFG

A CFG (Control Flow Graph) [2] is the representation of the basic blocks in a program and the relation between them.

Sequential computation can be represented simply as a path
$$\cdots \rightarrow \sigma_i \rightarrow \sigma_{i+1} \rightarrow \cdots$$ through the CFG. Parallel computation can be represented in a Cartesian product of CFG’s, but a simple path no longer suffices as a description.

Instead, we can think of a parallel computation as involving transitions of the form
$$\sigma_G \Rightarrow \sigma'_G,$$
where \(\sigma\) and \(\sigma'\) denote nodes in the CFG and \(G\) and \(G'\) denote (possibly different) sets of processors which are all executing the corresponding basic blocks in the indicated order.

Here we present one way of determining points in a CFG where we need to insert code to track changing process set membership.

The objective is to have as few such points as possible, to minimize work not needed for correctness.
8.1 CFG details

We define a control flow graph formally as a directed graph \( \{A, V, s, e\} \) with vertices \( V \) which represent basic blocks, arcs \( A \) which represent transfers of execution from one block to another, a start node \( s \in V \) and an end node \( e \in V \) (see [2]).

If a CFG has multiple end nodes, it may always be transformed to have a single end node by jumps from all the previously existing end blocks to the new end node.

For purposes of our analysis we will modify the standard definition of basic block by identifying communication statements as block leaders.

A consequence of this is that there can be at most one communication statement in each basic block.
8.2 Subroutine details

We note that our treatment of the CFG also applies to subroutines.

A subroutine has a CFG that looks like one for a full program.

It has a single entry point and possibly multiple exit points, which we assume can be coalesced into a single exit.

A subroutine in one CFG just appears as an ordinary node.

An exception to this is Fortran subroutines that use the *entry* statement.

This allows defining different entry points, call parameters and return types within a single subroutine.

The implications of multiple entry points in a subroutine are the same as for a program; i.e., the CFG for the subroutine will in general be different, have different connectivity and data dependences.

Therefore a subroutine with multiple entries will be treated as a different subroutine for each entry statement.
8.3 MIPS and the CFG

We show an example of a program CFG in Figure 6.

The MIPS system augments the CFG to allow the monitoring of the splits and merges in SPMD parallel computation, as follows.

At the start node we initialize a data structure identifying the total set $\Gamma$ of all processes, as well as code to keep track at each process of subsequent information about process sets (described below).

In a successful execution all processes reach the end node.

At the end node we may need to do some system-specific clean up to exit gracefully and terminate the parallel computation.

(End nodes might also be merge nodes, and these would get additional treatment as indicated subsequently.)
8.4 MIPS and communication

Any nodes with communication must be identified.

At these, we add code to verify that a communication condition (that all processes in the communication set are present) holds.

In addition, code to handle irregular termination when the communication condition does not hold.

We will simply assume that termination is signaled to all processes in this case.

Code generation for such a case can be more complicated, allowing generation and output of information to help in debugging, but this is beyond the scope of the present work.
8.5 Split and merge nodes

We will use the notion of a split node more generally, so we define this for any directed graph.

**Definition 8.1** A *split node* is any node in a directed graph of out-degree $N$ greater than one.

An SPMD computation can split into multiple paths in the CFG at a split node. Merge points in the CFG are nodes where different processes in parallel computation which have previously split into different paths come together again. Corresponding to each split node, we wish to find a *merge node* at which the processes that split previously can be guaranteed to re-join.

We now describe these points more precisely.
Merge nodes are associated with split nodes, and there can be many for a single split.

**Definition 8.2** Consider a CFG \( \{V, A, s, e\} \) with an \( N \)-way split node \( l \in V \). A node \( m \in V \) is a **merge node** corresponding to \( l \) and a set \( D \subset A \) of \( 1 < d \leq N \) departing edges,

\[
D = \{l \to b_i : 1 \leq i \leq d\},
\]

if all paths \( l \to b_i \to^* e \) are of the form

\[
l \to b_i \to^* m \to^* e
\]

for \( i = 1, \ldots, d \). A merge node \( m \) for the set of all edges departing from \( l \) is called a **complete merge node** for \( l \).

The end node \( e \) is always a (complete) merge node, so (complete) merge nodes always exist for any split node and any set of departing edges.

The end node may be the only merge node, but in general there will be earlier merge nodes.
8.6 Uniform predicates

Finally, we note that not all conditionals need to be treated as splits.

If the predicate is \textit{uniform} across all processes (e.g., a constant identified at compile time), the processes will all go on the same branch in the CFG. Any predicate that we cannot assert to be uniform we designate as \textit{non-uniform}.

In applying the results here to codes with uniform predicates, the resulting CFG would need to have the uniform splits marked in some way.
8.7 Merges and the $\text{dom}$ relation

We can define a relation $\text{cm}$ by $m \text{cm} l$ if and only if $m$ is a complete merge node for the split node $l$.

To be more precise, if $F = \{V, A, s, e\}$ is the CFG used in defining the relation, we will denote it by $\text{cm}_F$.

We now show that this relation is closely related to the well known ‘dominator’ relation ($\text{dom}$) [1].

That is, for any two nodes $a$ and $b$ in $V$, we say $a \text{dom} b$ if every path from $s \rightarrow^* b$ passes through $a$: $s \rightarrow^* a \rightarrow^* b$.

Again, we will write this as $\text{dom}_F$ to make the particular CFG being used clear.
We summarize the key points that we will use about the dominator relation.

First, $\text{DOM}$ is asymmetric: you cannot have both $a \text{ DOM} b$ and $b \text{ DOM} a$. Second,

$$a \text{ DOM} c \text{ and } b \text{ DOM} c \implies \text{ either } a \text{ DOM} b \text{ or } b \text{ DOM} a.$$ (8.1)

Finally, $\text{DOM}$ is transitive: $a \text{ DOM} b$ and $b \text{ DOM} c \implies a \text{ DOM} c$.

Let us derive some graphs from a given graph $F = \{V, A, s, e\}$.

Firstly, we define $F' = \{V, A', e, s\}$ where $A'$ denotes the set of edges in $A$ but with the direction reversed.

Secondly, we let $F_l = \{V_l, A_l, l, e\}$ denote the subgraph of $F$ with $l$ as the start node.

More precisely, $V_l$ is the set of all nodes $n$ for which there are edges in $A$ of the form $l \rightarrow^* n \rightarrow^* e$, and $A_l$ is the set of all such edges.
Theorem 8.3  Let $F = \{V, A, s, e\}$ be a CFG, and let $l$ be any split node. Then the following are equivalent for any $m \in V$:

\[ m \text{ CM}_F l \]
\[ m \text{ DOM}_{F'} l \]
\[ m \text{ DOM}_{F_l} e \] (8.2)

Proof. The equivalence relating $\text{CM}_F$ and $\text{DOM}_{F_l}$ follows easily from the definitions, and the equivalence of $\text{CM}_F$ and $\text{DOM}_{F'}$ follows simply by reversing arrows. QED

Corollary 8.4  Let $F = \{V, A, s, e\}$ be a CFG, let $l$ be any split node, and let $m$ be a complete merge node for $l$. If $n \in V$ is any node satisfying $n \text{ DOM}_{F'} m$, then $n$ is also a complete merge node for $l$.

Proof. By Theorem 8.3, $m \text{ DOM}_{F'} l$. Since $\text{DOM}$ is transitive, we have $n \text{ DOM}_{F'} l$, so the result follows by another application of Theorem 8.3. QED
The identity of other first merge nodes is also encoded in the \texttt{dom} tree. Let us begin by proving a simple result that is analogous to Corollary 8.4.

**Lemma 8.5** Let $F = \{V, A, s, e\}$ be a CFG, let $l$ be any split node, and let $m$ be a merge node for $l$ and a set $D$ of departing edges. If $n \in V$ is any node satisfying $n \text{dom}_{F'} m$, then $n$ is also a merge node for $l$ and $D$.

**Proof.** Since $m$ is a merge node for $l$ and $D = \{l \rightarrow b_i : 1 \leq i \leq d\}$, we know that all paths $l \rightarrow b_i \rightarrow^* e$ must go through $m$: $l \rightarrow b_i \rightarrow^* m \rightarrow^* e$.

But $n \text{dom}_{F'} m$ implies that all paths $m \rightarrow^* e$ must go through $n$.

Thus we conclude that all paths $l \rightarrow b_i \rightarrow^* e$ must go through $n$: $l \rightarrow b_i \rightarrow^* m \rightarrow^* n \rightarrow^* e$. Therefore $n$ is a merge node for $l$ and $D$. \textbf{QED}
8.8 First merge nodes

There is a natural ordering among merge nodes.

Suppose both $m$ and $n$ are merge nodes for $l$ and $D$, and let $l \rightarrow b_i$ be a departing edge in $D$.

Then either

$$l \rightarrow b_i \rightarrow^* n \rightarrow^* m \quad \text{or} \quad l \rightarrow b_i \rightarrow^* m \rightarrow^* n$$

(8.3)

since all paths to the end must pass through them both.

Thus merge nodes for a given split $l$ and set of departing edges $D$ are ordered by the relation $\rightarrow$.

We are interested in finding the first merge node for a given split and set of departing edges, since this is the earliest point that process sets which split can be merged again.

The \textsc{dom} relation makes it easy to make the notion of ordering precise.
Definition 8.6  Consider a CFG \( \{V, A, s, e\} \) with an \( N \)-way split node \( l \in V \) and a set \( D \subset A \) of \( 1 < d \leq N \) departing edges, \( D = \{l \rightarrow b_i : 1 \leq i \leq d\} \). A merge node \( m \) is a first merge node for \( l \) and \( D \) if for all merge nodes \( n \) for \( l \) and \( D \) \((n \neq m)\) we have \( n \text{dom}_{F'} m \). A first merge node \( m \) for the set of all edges departing from \( l \) is called a first complete merge node for \( l \).

We will subsequently show that first merge nodes exist and are unique for a given set \( D \), but let us just make some comments first as orientation.

If \( N = 2 \), there is only one first merge node.

A first merge node \( m \) depends on both the split node \( l \) and the set of departing edges \( D \); it will typically be different for different sets of departing edges when \( N \geq 3 \).

However, the same node \( m \) could be the first merge node for more than one split node.
Since merge nodes are naturally ordered by $\rightarrow$, it makes sense to look for first merge nodes by going backwards, say, from the end node. However, we might have loops which would make it difficult to identify ‘first’ merge nodes. We show that this does not occur.

The $\text{dom}_F$ relation for any directed graph $F = \{V, A, s\}$ induces a tree $T_{F,s}$ [1], where the subscript $s$ denotes the root of the tree.

For a CFG $F = \{V, A, s, e\}$, the root of the tree for $\text{dom}_{F'}$ is $e$, and we would express the tree as $T_{F',e}$; the root of the tree for $\text{dom}_{F_l}$ is $l$, and the notation for the tree is $T_{F_l,l}$.

We will also consider sub-trees of a $\text{dom}$ tree rooted at lower nodes in the tree. We are now able to characterize all first complete merge nodes via $\text{dom}$ trees.
Theorem 8.7 Let $F = \{V, A, s, e\}$ be a CFG, and let $l$ be any split node. The parent node $m^c_l$ of $l$ in the $\text{DOM}_{F'}$ tree $T_{F',e}$ is the (unique) first complete merge node corresponding to the split node $l$.

Proof. Let $m^c_l$ be the parent node of $l$ in $T_{F',e}$. Then by Theorem 8.3, $m^c_l$ is a complete merge node for $l$ ($m^c_l \text{ CM}_F l$) since $m^c_l \text{ DOM}_{F'} l$. So all we need to show is that (1) $m^c_l$ is a first complete merge node and (2) that it is unique.

If $n$ is another complete merge node for $l$ ($n \text{ CM}_F l$) then $n \text{ DOM}_{F'} l$. By (8.1), we must have either $n \text{ DOM}_{F'} m^c_l$ or $m^c_l \text{ DOM}_{F'} n$. But since $m^c_l$ is the parent of $l$ we must have $n \text{ DOM}_{F'} m^c_l$. Thus $m^c_l$ is a first complete merge node.

If $n$ is another first complete merge node, we conclude both that $n \text{ DOM}_{F'} m^c_l$ and $m^c_l \text{ DOM}_{F'} n$, which violates the property that $\text{DOM}$ is asymmetric.

Thus $m^c_l$ must be unique. QED
As might be expected, the identification of other first merge nodes is more complicated.

The following result clarifies the relationship between merge nodes and the \texttt{dom} relation.

\textbf{Lemma 8.8} \textit{Let }$F = \{V, A, s, e\}$\textit{ be a CFG, let }$l$\textit{ be any split node, and let }$D = \{l \rightarrow b_i : 1 \leq i \leq d\}$\textit{ be a set of departing edges. Then }$m$\textit{ is a merge for }$l$\textit{ and }$D$\textit{ if and only if}

$$ m \texttt{ dom}_F b_i \quad \text{for all} \quad 1 \leq i \leq d. \quad (8.4) $$

\textbf{Proof.} Follows directly from the definitions. \texttt{QED}
At a split in $F$, there are at least two successor nodes $b_i$, so Lemma 8.8 implies there must be multiple leaves in the tree $T_{F',e}$ below $m$, implying that a split in $F'$ may occur somewhere following a complete merge.

The following provides one characterization about where the split occurs.

**Lemma 8.9** Let $F = \{V, A, s, e\}$ be a CFG, and let $l$ be any split node. If $m$ is a first merge node corresponding to $l$ and a set $D$ of departing edges then $m$ is a split node in the CFG $F' = \{V, A', e, s\}$.

**Proof.** If $m$ is not a split node in $F'$, then it has a unique successor in $F'$, and hence a unique predecessor $\tilde{m}$ in $F$: $\tilde{m} \rightarrow m$.

Thus every path in $F$ to $m$ must go through $\tilde{m}$.

Thus $\tilde{m}$ must also be a merge node for $l$ and $D$, and $m \text{ dom}_{F'} \tilde{m}$.

By the asymmetry of $\text{dom}$ we cannot have $\tilde{m} \text{ dom}_{F'} m$, and thus $m$ cannot be a first merge node. **QED**
Not all split nodes in $F'$ will necessarily be first merge nodes, as Figure 8 shows. However, the $\text{dom}$ tree resolves the ambiguities.

**Lemma 8.10** Let $F = \{V, A, s, e\}$ be a CFG, and let $l$ be any split node. Suppose $m$ is a merge node corresponding to $l$ and a set $D$ of departing edges. If $n$ is the only child node of $m$ in the tree $T_{F',e}$ then $n$ is also a merge node for $l$ and $D$.

**Proof.** Since $n$ is a child node of $m$ we have $m \text{ dom}_{F'} n$. Since $n$ is the only child node of $m$, for all nodes $n' \neq n$ such that $m \text{ dom}_{F'} n'$ we must have $n \text{ dom}_{F'} n'$. In particular, we must have $n \text{ dom}_{F'} b_i$ for all edges $l \rightarrow b_i$ in $D$. The result thus follows from Lemma 8.8. **QED**

**Corollary 8.11** Let $F = \{V, A, s, e\}$ be a CFG, and let $l$ be any split node. If $m$ is a first merge node corresponding to $l$ and a set $D$ of departing edges, then $m$ is a split node in the tree $T_{F',e}$. 


Proof. Suppose that $m$ is a first merge node for $l$ and a set $D$ of departing edges. By Lemma 8.8, $m$ is not a leaf node; let $n$ be a child node of $m$ in the tree $T_{F',e}$ (so $m \text{dom}_{F'} n$) that is in a path $l \rightarrow b_i \rightarrow^* n \rightarrow^* m \rightarrow^* e$ for some edge $l \rightarrow b_i$ in $D$. If $n$ is a merge node for $l$ and $D$, we reach a contradiction, since we cannot have $n \text{dom}_{F'} m$ as required by the definition of first merge node, since $\text{dom}$ is asymmetric and we know that $m \text{dom}_{F'} n$. If $n$ is the only child node of $m$ in $T_{F',e}$, then Lemma 8.10 implies that $n$ is a merge node for $l$ and $D$, so $m$ must have more than one child node in $T_{F',e}$ and hence be a split node in this tree. QED

Now let us consider identifying all first merge nodes.

Let $m^c_l$ denote the first complete merge node for a split node $l$.

Consider the sub-tree $T_{F',m^c_l}$ of the $\text{dom}_{F'}$ tree $T_{F',e}$ rooted at $m^c_l$.

Let the tree $T_{F,l}$ denote the sub-tree of $T_{F,e}$ rooted at $l$, which consists of nodes that are reachable from the split $l$. In Figure 7, node 1 appears in the sub-tree of $T_{F',e}$ rooted at $m^c_l$ but it is not in $T_{F,l}$.
Lemma 8.12 Let \( F = \{V, A, s, e\} \) be a CFG with an \( N \)-way split node \( l \in V \), with \( N \geq 2 \). Let \( D \) be the set of \( N \) departing edges, i.e., \( D = \{l \rightarrow b_i : 1 \leq i \leq N\} \), and let \( B = \{b_i : 1 \leq i \leq N\} \) denote the corresponding nodes. Then \( B \) is the set of all leaf nodes (not equal to \( l \)) in the tree \( T_{F',m_i^c} \) that are also in \( T_{F,l} \).

Proof. Let \( m_i^c \) be the first complete merge node for \( l \). By Lemma 8.8, \( m_i^c \text{ dom}_{F'} b_i \) for all \( i \), so \( B \) is a subset of the nodes in both \( T_{F',m_i^c} \) and \( T_{F,l} \). These are leaf nodes in \( T_{F',m_i^c} \) since the only path from them in \( F' \) leads to \( l \), and Theorem 8.7 shows that \( l \) is a child node of \( m_i^c \).

For any \( x \) in the tree \( T_{F,l} \), there is a path \( l \rightarrow b_i \rightarrow^* x \rightarrow^* m_i^c \) for some \( b_i \in B \). Thus \( x \) has to appear in the tree \( T_{F',m_i^c} \) between the root \( m_i^c \) and the leaf \( b_i \). Thus the set \( B \) contains the only possible leaf nodes in the tree \( T_{F',m_i^c} \) that are also in \( T_{F,l} \), except for \( l \) itself. QED
The following provides a converse to Corollary 8.11.

**Lemma 8.13** Let $F = \{V, A, s, e\}$ be a CFG with an $N$-way split node $l \in V$, with $N \geq 2$. Suppose that $m'$ is a node in $T_{F,l}$ that is a split node in $T_{F',m_i^c}$. Let $B' = \{b_i : 1 \leq i \leq d\}$ denote the set of leaf nodes in $T_{F',m_i^c}$ that are dominated by $m'$. Let $D' = \{l \rightarrow b_i : 1 \leq i \leq d\}$. Then $m'$ is the first merge node for $l$ and $D'$.

**Proof.** Since by assumption $m'$ is in the tree $T_{F,l}$, the set $D'$ cannot be empty. Since $m'$ is a split node, we claim that $d \geq 2$.

Let $n_1$ and $n_2$ be child nodes of $m'$ in $T_{F',m_i^c}$.

By Lemma 8.12, $n_1$ and $n_2$ must dominate (at least) two different leaf nodes in $B'$. Thus $d \geq 2$. 
By Lemma 8.8, $m'$ is a merge node for $l$ and $D'$.

Now we need to show that if $m'$ is a split node, then it is a first merge node.

Again, consider any two distinct child nodes $n_1$ and $n_2$ of $m'$ in $T_{F',m_i^c}$.

The set of leaves $B_1$ and $B_2$ dominated by $n_1$ and $n_2$ cannot have any nodes in common, since $T_{F',m_i^c}$ is a tree.

If $m'$ were not a first merge node, then there would be some merge node $n$ for $l$ and $D'$ such that $m' \text{ dom}_{F'} n$.

Without loss of generality, let us assume that $n$ is on the $n_1$ branch under $m'$ in $T_{F',m_i^c}$.

Then we cannot have $n \text{ dom}_{F'} B_2$, contradicting the assumption that $n$ is a merge node for $D$, by Lemma 8.8.

Thus $m'$ must be a first merge node for $l$ and $D'$. QED
Putting together the above results leads to the main result of the paper.

**Theorem 8.14** Let $F = \{V, A, s, e\}$ be a CFG with an $N$-way split node $l \in V$, with $N \geq 2$. Let $D$ denote a subset of $d$ departing edges, $1 < d \leq N$, i.e., $D = \{l \rightarrow b_i : 1 \leq i \leq d\}$. Then $m_D$ is the first merge node for $D$ if and only if it is a node in $T_{F,l}$ that is a split node in $T_{F',m_{l}^c}$.

**Proof.** First of all, if $m_D$ is a first merge node for $D$, then $m_{l}^c \text{ dom}_{F'} m_D$ by Definition 8.6, since $m = m_{l}^c$ is also a merge node for $l$ and $D$.

Thus $m_D$ is in the sub-tree $T_{F',m_{l}^c}$, and it is also in the tree $T_{F,l}$.

Thus the ‘only if’ part of the proof follows from Corollary 8.11.

The ‘if’ part of the theorem is exactly Lemma 8.13. **QED**
Finding all first merge nodes can be done easily using the trees associated with $\text{DOM}_{F'}$ and $\text{DOM}_F$.

These trees can be computed efficiently [1], and then the identification of merge nodes is simple.

First, loop over all splits finding the corresponding first complete merge nodes. Then loop over the latter to find all interior split nodes (if any) in the corresponding subtrees.
Figure 7: **DOM** trees for the CFG of Figure 6, and identification of first complete merge nodes for split nodes. Split nodes S and 1 are indicated by dashed lines.
Figure 8: Example of a CFG $F$ with a split node (3) in $F'$ that is not a first merge.
8.9 The merge tree

We have shown how to find a merge node $m_D$ associated with a given split $l$ and set of departing edges $D$.

Lemma 8.13 shows that we can reverse this relationship and associate with each merge $m$ a set of departing edges $D_m$.

We will show that this is the unique maximal set of departing edges for which $m$ is a first merge node.

At the moment, all we know is that there is a first merge node for every split and every set of departing edges from it.

Thus it might appear that there could be a different first merge node for every subset of departing edges at a split.

We now show that there is much more order in the set of first merge nodes.

We begin with some simple rules relating merge nodes and sets of departing edges.
Lemma 8.15 Given a CFG \( \{ V, A, s, e \} \), let \( l \in V \) be an N-way split node. (i) If \( m \) is a merge node for two sets \( D \) and \( D' \) of edges departing from \( l \), then it is a merge node for \( D \cup D' \). (ii) If \( m \) is a merge node for a set \( D' \) of edges departing from \( l \), it is also a merge node for any subset of departing edges \( D \subset D' \).

Proof. Follows directly from Definition 8.2. QED

Lemma 8.16 Given a CFG \( \{ V, A, s, e \} \), let \( l \in V \) be an N-way split node. If \( m \) is a merge node for two sets \( D \subset D' \) of edges departing from \( l \), and it is a first merge node for \( D \), then it is also a first merge node for \( D' \).

Proof. Let \( n \) be any merge node for \( D' \). Since \( D \subset D' \), \( n \) is also a merge node for \( D \) by Lemma 8.15, part (ii). Since \( m \) is a first merge node for \( D \), we have \( n \mathsf{dom}_{F'} m \) by Definition 8.6. Since \( n \) was arbitrary, \( m \) is also a first merge node for \( D' \). QED
Corollary 8.17  Given a CFG \( \{V, A, s, e\} \), let \( l \in V \) be an N-way split node. If \( m \) is a first merge node for two sets \( D \) and \( D' \) of edges departing from \( l \), then it is a first merge node for \( D \cup D' \).

**Proof.** By Lemma 8.15, \( m \) is a merge node for \( D \cup D' \). By Lemma 8.16, \( m \) is a first merge node for \( D \cup D' \). QED

Corollary 8.18  Given a CFG \( \{V, A, s, e\} \), let \( l \in V \) be an N-way split node. If \( m \) is a first merge node for \( l \), there is a unique largest set \( D_m \) of departing edges for which it is a first merge node for \( D_m \). This set of departing edges is the same as in Lemma 8.13.

**Proof.** By Corollary 8.17, if \( m \) is a first merge node for any sets of departing edges, it is a first merge node for the union of the sets. So \( D_m \) can be defined by taking the union of all sets of departing edges for which \( m \) is a first merge node. This proves the first part.
The set of departing edges identified in Lemma 8.13 is based on the set of all leaf nodes in the $\text{dom}_{F'}$ tree dominated by $m$ and reachable from $l$. If the set $D_m$ defined above were larger than this, there would be an edge $l \rightarrow \hat{b}$ for which $m$ is a merge but such that $b$ is not dominated by $m$. Lemma 8.8 shows that this is not possible. QED

The following gives a simple relationship between the maximal sets of departing edges for two different first merge nodes.

**Lemma 8.19** Given a CFG $\{V, A, s, e\}$, let $l \in V$ be an N-way split node. For any two first merge nodes $m \neq m'$ for $l$, if $m' \text{ dom}_{F'} m$ then $D_m \subset D_{m'}$.

**Proof.** By Lemma 8.5, we know that $m'$ is a merge node for $D_m$ (as well as $D_{m'}$). By Lemma 8.15, part (i), we conclude that $m'$ is a merge node for $D_m \cup D_{m'}$. By Lemma 8.16, $m'$ must be a first merge node for $D_m \cup D_{m'}$. By maximality, we must have $D_{m'} = D_m \cup D_{m'}$, which implies that $D_m \subset D_{m'}$. QED
**Theorem 8.20** Given a CFG \( \{V, A, s, e\} \), let \( l \in V \) be an \( N \)-way split node. For any two first merge nodes \( m \neq m' \), then either \( D_m \subset D_{m'} \) or \( D_{m'} \subset D_m \), or \( D_{m'} \cap D_m = \emptyset \).

**Proof.** If \( D_{m'} \cap D_m = \emptyset \), we are done, so let us assume that there is an edge \( l \to \hat{b} \) in \( D_{m'} \cap D_m \). Then \( m \text{ dom}_{F'} b \) and \( m' \text{ dom}_{F'} b \). By (8.1), either \( m \text{ dom}_{F'} m' \) or \( m' \text{ dom}_{F'} m \). But \( m \text{ dom}_{F'} m' \) implies \( D_{m'} \subset D_m \) and \( m' \text{ dom}_{F'} m \) implies \( D_m \subset D_{m'} \) by Lemma 8.19. QED

Consider the set \( S_l \) of all maximal sets of departing edges \( D_m \) with first merge node \( m \), as in the theorem. Then we have a well-defined mapping from the set of first merge nodes \( M_l \) to \( S_l \) given by \( m \to D_m \). Moreover, it is injective: if \( D_m = D_{m'} \) then \( m' = m \) by the uniqueness of first merge nodes. Using the theorem, the relation \( \subset \) on sets of departing edges induces a tree structure for first merge nodes, as follows.
Corollary 8.21 *First merge nodes corresponding to a single split $l$ can be arranged as a tree $T_l$ rooted in the complete merge node $m^c_l$ for $l$. We call $T_l$ the merge tree for $l$.*

**Proof.** If $D_{m_1} \subset D_m$, we add an edge $m \rightarrow m_1$ in the graph $T_l$ provided $D_{m_1}$ has no larger superset that is also a subset of $D_m$, that is, there is no $D_{m_2}$ such that $D_{m_1} \subset D_{m_2} \subset D_m$ (recall, all of the sets are distinct, none equal). Note that if $D_{m_1} \subset D_m$ and $D_{m_1} \subset D_{m'}$, then the theorem guarantees that either $D_m \subset D_{m'}$ or $D_{m'} \subset D_m$. Thus there is a unique $m$ with the required property. In particular, this means that the graph must be a tree: there can be only one arrow leading to a given node in the graph. By definition, $D_{m^c_l} \in S_l$ is the set of all departing edges from $l$. Since all $D_m \subset D_{m^c_l}$, all first-merge nodes in $M_l$ must appear in this graph. **QED**
The merge tree has the following properties that can be exploited to keep track of multiple paths taken by different processes at a split.

Suppose that $D$ is a set of edges leaving a split, corresponding to the different paths that different processes take at a split, and suppose that we want to find the appropriate merge node $m'$ for $D$, at which point we can be sure that all the processes in this set will have merged again.

There is a unique smallest $D_{m'}$ containing $D$ (by applying Theorem 8.20), and it is easy to see that $m'$ is the first merge node for $D$.

To find $m'$, suppose that $D \subset D_m$ for some merge node $m$. In particular, this always holds for $D_{m_l^c}$, and $m_l^c$ is conveniently the root of the merge tree.

Then if $D_m$ is not the maximal set for $D$, there is one and only one edge in the merge tree $m \rightarrow m_1$ such that $D \subset D_{m_1}$ (again, apply Theorem 8.20).
By walking down the merge tree, we will eventually find \( m' \). To make this an algorithm, we simply need a test for finding whether a \( D_m \) is the maximal set for \( D \).

For example, this will be the case if and only if \( D \) does not fit inside any of the sets \( D_{\hat{m}} \) for any children \( \hat{m} \) of \( m \) in the merge tree. (If, \( m \) has no children, then it must be the maximal set.)

From a practical point of view, the number of splits \( N \) would only infrequently be very large.

Thus the set of all subsets of departing edges and corresponding first merge nodes could be enumerated at compile time and a mapping from departing sets to merges could be created as a list.
8.10 Merge level

The sets $D_m$ for all $m$ corresponding to the leaves $m$ of the merge tree form a disjoint decomposition of $D$.

By walking down the tree, we see that every element of $D$ is in one of the leaf $D_m$'s.

Moreover, the sequence of merges encountered in this walk is the reverse order of merges that will be encountered upon leaving the split along the given edge.

Thus we can define a merge level for each path $\ell \rightarrow b$ corresponding to this number of merges encountered.

That is, the merge level of a given edge $\ell \rightarrow b$ is the height of the tree from the root to the leaf $D_m$ that contains it.
8.11 Merges are nested

We now show that pairs of split nodes and any of their corresponding ‘first’ merge nodes are nested. We begin with another structural relationship between $\text{dom}_{F'}$ and merge nodes.

**Lemma 8.22** Suppose $F = \{V, A, s, e\}$ is a CFG. Let $l$ be any split node, and let $m$ be a merge node for the edge $l \rightarrow b$. Suppose $x$ is any node on a path

$$l \rightarrow b \rightarrow^* x \rightarrow^* m$$  \hspace{1cm} (8.5)

that does not include $m$ anywhere else on the path other than the endpoint. Then $m \text{ dom}_{F'} x$.

**Proof.** If the conclusion were false, then there would be a path $x \rightarrow^* e$ which does not pass through $m$. Use the first part of the path (8.5) to extend this to a path $l \rightarrow b \rightarrow^* x \rightarrow^* e$ which does not pass through $m$ (by assumption, $m$ does not appear in the path $l \rightarrow b \rightarrow^* x$). This contradicts the assumption that $m$ is a merge node for the departing edge $l \rightarrow b$. Thus we must have $m \text{ dom}_{F'} x$. QED
Theorem 8.23  Suppose $F = \{V, A, s, e\}$ is a CFG, that $l_1$ is a split node and $m_1$ is any of its matching first merge nodes. Suppose $l_2$ is a split node on a path from $l_1$ to $m_1$ that includes $m_1$ only at the endpoint. Then all paths $l_2 \rightarrow^* m_2^c$, where $m_2^c$ is the complete merge node for $l_2$, are included in paths from $l_1$ to $m_1$. That is, $m_1$ is also a complete merge node for $l_2$, and the split-merge pairs are nested.

**Proof.** By Lemma 8.22, $m_1 \text{dom}_{F'} l_2$. By Theorem 8.3, $m_1$ is a complete merge node for $l_2$. By Definition 8.6, $m_1 \text{dom}_{F'} m_2^c$. By Corollary 8.4, $m_1$ is also a complete merge node for $l_2$. QED

The fact that split–merge pairs are nested simplifies the code to support them; it is not necessary to use a data structure more complicated than a stack to keep track of splits and merges, and in many cases simple counters will be sufficient.
8.12 MIPS level

We have from Theorem 8.23 that split-merge pairs are nested. This allows us to define a MIPS level corresponding to the number and type of splits and merges traversed along a given path through the CFG, analogous to a parenthesis nesting level. MIPS level is more complicated than parenthesis nesting level because a single split can have more than one first merge along a particular path, and because multiple splits can have merges at the same node.

The MIPS level reflects how many merges a process will encounter before merging again with other processes. We define a MIPS level array, indexed on process number, with one entry for each process. Each process will keep its own array, and they will be different at different processes. For clarity, we will describe this data structure as a two-dimensional array, $M@p[q]$. We write it in this unusual way to emphasize that it should be thought of as a distributed one-dimensional array, where the first index indicates the process that owns the indicated copy. In particular, the algorithms that we describe for accessing $M@p$ are done only by process $p$, and not by any global operations.
Initially, the entire array $M[p]$ is set to zero. At each split, we determine the path each process will take (see below for what this requires), and thus we can compute the merge level (see section 8.10) for each process. For each process $p$, we increment $M[p][p]$ by the merge level for that path. A particular process $p$ keeps track of which processes $q$ have taken the same path that it is taking at a split, by incrementing their MIPS level in $M[p][q]$ to match its own, i.e, $M[p][q] = M[p][p]$. We describe subsequently how the other values of $M[p][q]$ are changed.

There is also a data structure to keep track of implicit process groups. In particular, each process keeps a data structure that indicates which other processes $q$ are in its own group. Again, we describe this as a distributed one-dimensional array. This could be thought of as a binary array, $G[p][q]$, for example, also set to zero initially, and with $G[p][q] = 1$ for all processes $q$ that are not in the same group as $p$. We will explain how this is to be up-dated shortly.
For processes taking different paths, using the merge tree we can determine how many merges will take place before we meet another process. More precisely, we increment $M_{\hat{p}}[q]$ by the number of edges in the merge tree from the leaf containing the path taken by $p$ to the first $D_m$ containing the paths taken by both $p$ and $q$. Finally, we note that we only increment $G_{\hat{p}}[q]$ for $q$ such that $G_{\hat{p}}[q] = 0$ at the time of the split. In addition, since these $q$ are now on separate paths, we then set $G_{\hat{p}}[q] = 1$ following the split as well.
At a merge, each process $p$ decrements its own level ($M@p[p] = M@p[p] - 1$) and the level of the processes in its group ($G@p[q] = 0$), and then it updates group membership by setting $G@p[q] = 0$ for all $q$ found to have $M@p[q] = M@p[p]$.

There may be another split that occurs before all merges are completed. However, since splits and merges are nested, the simple data structures $M$ and $G$ are sufficient to keep track of things. In particular, if a group of processes participate in a new split, their arrays $M$ will increase in the appropriate locations, and the equality test for group membership will be sufficient. Only once they get to the complete merge for that split will they decrease. It should be noted that this approach is conservative. Two processes could be executing the same block of code even though this analysis might not predict it.
**Theorem 8.24** Given a CFG \( \{V, A, s, e\} \) without loops, the data structures \( M \) and \( G \) are sufficient to determine correctly group membership. In particular, if \( G[q][p] = 0 \) for some \( p \) and \( q \), then we have \( G[q][p] = 0 \). Further, the MIPS levels will be the same for all such \( q \).

**Proof.** The conditions hold at the start of execution, since both \( M \) and \( G \) are zero then. By induction, it suffices to consider sub-sets of processes, and to show that the above conditions are valid at all merges if they are valid at all splits. This follows from the constructions. **QED**
9 Loops

Loops covered by the above analysis, but it is possible to treat them in a simpler way that may be advantageous. We now consider CFG’s which have only single-entry, single-exit (SESE) loops in the canonical form of a ‘while’ loop [6]: the entry and exit of the loop are the same node, called the head of the loop. More general loops can be transformed into this form [13].

The head $h$ of a loop is a split, where different processes could take different paths. For example, one process could enter the loop, and another could not. However, it is more useful to think of unrolling the loop to see how we should think about it (see Figure 9). Unrolling will reduce the out-degree to one. From the point of view of implicit process sets, we just need to know which processes execute which iteration of the loop. A simple counter can keep track of this. The set that enters the loop may successively be diminished in size as different processes exit the loop, but the entire set is reconstituted once all processes have exited the loop.
Figure 9: An unrolling of loop L9 from Figure 6.
What simplifies the interactions of splits and merges with loops is the observation that they do not interact in a complex way. We first show that any split that occurs outside a loop must also merge outside that loop.

**Theorem 9.1** Let $L$ be a loop in a CFG $F = \{V, A, s, e\}$ having only SESE loops. If $l \notin L$ is a split node, and $m$ is a merge node matching $l$ for some set of edges $D$, then either $m \notin L$ or $m = h$, the header node of $L$.

**Proof.** Suppose that $m \in L$. Since $l$ is not in $L$, all paths $l \rightarrow^* b_i \rightarrow^* e$ are of the form $l \rightarrow^* b_i \rightarrow^* h \rightarrow^* m \rightarrow^* e$ since $h$ is the single entry for $L$. By definition, there is a path $l \rightarrow^* b_i \rightarrow^* h \rightarrow^* e$ which by-passes completely the rest of the loop $L$. Thus the only merge node possible in $L$ is $h$. **QED**

Conversely, we show in the following theorem that any split that occurs inside a loop must also merge inside that loop.
Theorem 9.2 Suppose the CFG \( \{V, A, s, e\} \) has only SESE loops. Suppose that \( h \) is the header of loop \( L \) and \( l \) is a split node in \( L \). Then \( h \) is a merge node for \( l \) for any set of departing paths \( D \).

Proof. Since the only way to exit the loop is to pass through \( h \), \( h \) is a merge node for \( l \). \( \text{QED} \)

Multiple exit loops must either be transformed to single exit loops as described in [7] or supported through extra code that we will not consider here.

Process set membership for loops can be maintained by the array \( G \). At each loop header, a determination is made about which processes will exit the loop and which will remain. At the complete termination of the loop, the array \( G \) must be returned to its original state before entering the loop.
10  Code support for MIPS

To determine process-set membership at splits, we need to find out which branches each process will take. We call such an operation *vector combine*.

**Definition 10.1**  A *vector combine* is an all-gather (see [12], page 92) data transfer between a set $G$ of $P$ processes in which each process $p \in G$ contributes a data item in a vector $W$, at a position that corresponds to the order (index) of $p$ in $G$. The result is $P$ copies of $W$, one at each process, with data from every process.

We will not discuss in detail efficient algorithms to perform a vector combine, but for some architectures it can be done in logarithmic time by exchanging $W$ in a coordinated way in which the number of valid entries get doubled (or nearly so) at each step.
At each split, a vector combine must be performed for the set of processes arriving at the split.

This set is indicated by the data structure $G$ described in section 8.12.

At the split $l$, process $p$ sets $W(p)$ equal to some indicator for the departing edge $l \rightarrow b$ which it will take.

For example, the set of all departing edges could be numbered from 1 to $N$.

The vector combine is then performed on $W$, resulting in an array which indicates the departing edge for every process.

It is possible that a single node may be a merge for several splits.

However, it is known which split is being merged because they are nested.

A simple stack can keep track of this.
In loops, process set membership is decreased as individual processes exit the loop.

Once all processes have exited, the set membership is restored to what it was before loop entry.

Thus it is necessary to keep a temporary copy of the process set membership on entry to a loop in order to restore it at exit.

Subroutines are handled in a recursive fashion.

That is to say, the process set is transferred to subroutine on entry, and a new MIPS level data structure is initialized to zero for use within the subroutine.

The set of processes executing the subroutine becomes the new total set of processes, \( \Gamma \).

Control is not returned to the calling procedure by the subroutine until all processes in the set have completed.
11 Communications and deadlock

In order to establish conditions for non-deadlocking and deterministic execution, we need to make certain restrictions on communications.

We will give examples of the restrictions in message-passing terms, although they could be extended to other approaches for information exchange.

We assume that communications are reliable.

We consider a type of communication called a fusion [16], and for simplicity, we assume that they are the only executable statement in a node in a CFG.

These are deterministic, deadlock-free communications in which the producer and consumer of information is known to all parties at run time.

To be precise, we define fusions in three steps.

First of all, a point-to-point fusion is a communication of the form \( x@m \leftarrow y@n \), where ‘←’ is assignment and the expression \( z@p \) specifies the variable \( z \) at process \( p \) (recall that in our SPMD model, each process has its own memory identified by the variable names in the ‘single program’).
A general fusion is a communication which is equivalent of a set of point-to-point fusions which occur together (atomically).

By definition, there can be no cycles in a fusion; all of the data on the right-hand side of the fusion assignment correspond to states before the communication and all of the data on the left-hand side of the fusion assignment correspond to states after the communication.

Finally, we allow in the definition of fusion anything that can be written as a general fusion followed by a deterministic computation.

This includes reductions, scans and other collective operations.

By definition, the results of a fusion communication are always the same if the same data is communicated and the states of the senders and receivers are the same.

We also assume that the producers and consumers (m and n in the point-to-point fusion) can be determined by code generated at compile time based on the text of the parallel code.
For example, a point-to-point message-passing fusion is expressed as a send-receive pair at two processes where both sender and receiver are identified at each process: it fuses send, receive and synchronization actions.

Communication in which either sender or receiver is undefined is excluded, e.g., one-sided communication represented as ‘put’ and ‘get’ statements.

Similarly, a broadcast with receivers unspecified would not constitute a fusion.

The only reason that a fusion communication can deadlock is that a required producer of data is not participating.

For a fusion $\sigma$ in the CFG we denote by $c(p, \sigma)$ the set of processes from which process $p$ expects to get information.

For any process set $G$, we can define a communication set $C(G, \sigma)$ as the union of the communication sets at each of the processes $p \in G$ that is, $C(G, \sigma) = \bigcup_{p \in G} c(p, \sigma)$. 
Definition 11.1  If \( C(G, \sigma) \subset G \), we say that the fusion \( \sigma \) is closed under communication.

We will show that as long as fusions are closed under communication then execution will be deterministic and deadlock-free.

It may be inefficient to compute \( C(G, \sigma) \) at every process (it would require a vector combine), but what we really need to know is when a fusion is not closed. In this case, it must be true that \( c(p, \sigma) \) is not contained in \( G \) for some \( p \).

Since all processes know \( G \) from the MIPS framework, each process can compute \( c(p, \sigma) \) and flag an error if it is not contained in \( G \).

Thus we have the following theorem.

Theorem 11.2  Under the above assumptions, in a MIPS-supported parallel execution individual processes can detect communication deadlock.
The question of run-time support to provide graceful termination in this case will be considered in later work.

Fusion statements are defined as single statements in the Planguages[10, 8].

A point-to-point communication is of the form \( x@m = y@n \), where ‘=’ is assignment and the expression \( z@p \) specifies the variable \( z \) at process \( p \).

MPI send-receive and other collective communication statements can also be considered fusion statements, but to our knowledge only the Planguages provide point-to-point fusions.

It would be sufficient to define a fusion to be any deterministic communication \( \sigma \) in the CFG in which we can determine the set \( c(p, \sigma) \) of processes from which process \( p \) expects to get information.

However, this one-sided definition is not really more general in the SPMD context due to the symmetry of producer and consumer (each can read the other’s code).
12 Determinism

We will now show that, given communications as described in section 11, a MIPS execution is deterministic under appropriate conditions. We assume that we have a program that uses only fusion communications, and we apply the MIPS framework to the corresponding CFG. Moreover, we assume that fusions are the only form of interaction between processes, e.g., there are no semaphores, locks, file I/O, or other devices to allow processes to interact. We assume that each sequential segment of code is deterministic and completes in finite time.

Theorem 12.1 Under the above assumptions, if all fusions are closed under communication (Definition 11.1) during execution, then the MIPS-supported parallel execution will be deterministic.
**Proof.** All processes start together and have a data structure $G$ providing the correct process set. The MIPS framework keeps track of changes in the process set that is executing each sequential segment together. The result of each communication always is the same, since they are closed under communication. Thus the start of each new sequential segment is always the same. By induction, the parallel execution must continue in a deterministic fashion. **QED**

If a sequential section diverges (never completes), then we can still assert a type of determinism, namely, that the same section will always diverge.

From a practical point of view, it may not be possible to distinguish deadlock from sequential divergence.

However, the MIPS framework could be used to help isolate the sequential divergence in an appropriate run-time system.

This will be the subject of future investigations.
We cannot assert that the parallel computation will itself reach an end state, even though each of the sub-computations do.

It is clearly possible to design a program that goes on indefinitely with fusion communications.

It is important to explain why race conditions have been ruled out by our MIPS framework and fusion communications.

A race condition involves unpredictable interactions between processors, and for us such interactions can only occur through fusions.

A typical race condition might involve the first processor arriving at a certain point broadcasting a value to all other processors.

In this case, it is not possible to predict which processor will be providing information to a given process.

We require that the producing processes be computable via code that can be generated at compile time and using variables that are explicit in the SPMD text.
Fusions are closely related to the concept of guarded memory which has been shown [4] to provide a deterministic approach to nested parallelism.

Our work here can be thought of as a way to extend [4] to the SPMD context.
13 Conclusions and Discussion

We have presented a simple strategy to track the formation and merging of implicit process groups in SPMD parallel computation. It provides a way to identify a certain type of deadlock related to communication common in irregular computations.

Our suggested approach is not the only one that could be taken using the tools discussed here. For example, just based on Theorem 8.7 alone, one could support a BSP style of computation [17]. Theorem 8.7 says how to determine complete merges following a split, and any communication occurring between them could be flagged as outside of the model. One the other hand, the approach taken here is conservative. The ‘partial merge’ at node 3 in the CFG in Figure 8 would not be identified. On the other hand, one could study all splits in the CFG $F'$ (cf. Lemma 8.9), which would include node 3 in Figure 8. We have not considered this more complex approach in detail.
References


