Barrier Inference

Alexander Aiken* and David Gay*
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
{aiken,dgay}@cs.berkeley.edu

Abstract

Many parallel programs are written in SPMD style, i.e. by running the same sequential program on all processes. SPMD programs include synchronization, but it is easy to write incorrect synchronization patterns. We propose a system that verifies a program's synchronization pattern. We also propose language features to make the synchronization pattern more explicit and easily checked. We have implemented a prototype of our system for Split-C and successfully verified the synchronization structure of realistic programs.

1 Introduction

Explicitly parallel programming—where the programmer specifies the parallelism in a computation—is arguably the most widely used parallel programming paradigm. In exchange for a programming model that gives direct control over performance, programmers must manage the coordination of parallel processes, a task that is facilitated or hindered by the programming language. Despite years of practical experience, there is little research exploring language and compiler support for writing explicitly parallel programs. We propose a static semantics for global synchronization that guarantees an explicitly parallel program has no global synchronization errors. Our proposal is based on a formalization of widespread programming practices. We have proven the soundness of our method and implemented a prototype system. Experimental evidence gathered from testing our system on realistic benchmarks supports our hypothesis that the global synchronization structure of realistic programs can be formalized and automatically verified.

Our system was developed in the context of a distributed memory, shared address space programming language (Split-C, an SPMD language developed at Berkeley [13]), but we found it equally applicable to checking the synchronization structure of shared memory, shared address space parallel programs; our method can show the synchronization correctness of the SPLASH-2 [25] benchmarks. Whether a similar result holds for pure message passing programs requires further research.[1]

*This material is based in part upon work supported by NSF Young Investigator Award No. CCR-9457812, DARPA contract F30602-95-C-0136 and a Microsoft graduate fellowship.

1.1 Global Synchronization

A simple and popular parallel programming model is SPMD (for Single Program, Multiple Data). SPMD programs are explicitly-parallel programs written in sequential languages extended with communication and synchronization primitives. A typical SPMD skeleton is

```
work1();
barrier;
work2();
barrier;
work3();
```

where a `barrier` causes a process to block until all other processes have also reached a `barrier`. In SPMD execution, all processes execute a copy of the program independently. In this example, the `barrier`s serve to guarantee that all processes finish `work1()` before proceeding to `work2()`. The only synchronization is at the `barrier`s—processes execute `workn()` asynchronously.

While conceptually simple, the combination of asynchronous execution and explicit global synchronization introduces subtle issues of program structure and correctness. Figure 1 gives examples illustrating correct and incorrect synchronization patterns. In these examples, `different()` returns a different value in every process (causing different branch decisions in different processes) and `workn()` is a function with no synchronization. In all the examples `barrier`s are executed conditionally; we have observed that almost all SPMD programs have conditional synchronization.

There are two basic forms of incorrect synchronization. In Figure 1a, processes execute different numbers of `barrier`s,

[1] Such programs may not rely on global synchronization to the same degree as shared address space programs, but standard message passing libraries such as MPI [20] do include global synchronization primitives.

```
if (different()) barrier;
work1(); barrier;
work2();
```

(a) processes left behind

```
while (different()) barrier;
work1(); barrier;
work2(); barrier;
work3();
```

(b) processes "trapped" in a loop

```
if (different()) barrier else broadcast;
```

(c) conflicting barrier/broadcast

```
a <- different();
if (a) barrier;        (*)
x <- x + 1;
if (not a) barrier;    (*)
```

(d) correct but not structurally correct

```
if (x) barrier else work();
```

(e) correct if processes agree on x's value

```
i <- 0;
while i < 10
  (if (i = 1) barrier;
   i <- i + 1);
barrier;
```

(f) correct loop

```
if (different())
  (barrier; barrier)
else
  (work1(); barrier; work2(); barrier)
```

(g) if with matching barriers

```
i <- 0
if (different())
  (while (i < 10) (barrier; i <- i + 1))
else
  (j <- i + 10;
   while (j < 20) (work1(); barrier; j <- j + 1))
```

(h) structurally correct but not verifiable

Figure 1: Examples of correct and incorrect synchronization.

causing the program to "hang" when some processes terminate while others wait at a `barrier`. The same problem occurs in loops containing `barriers` if processes execute differing numbers of iterations (Figure 1b). The second problem is illustrated by Figure 1c, where some processes execute `barrier` while others execute `broadcast`. In SPMD languages, simultaneously executing different synchronization operations causes a runtime error (or, in some implementations, undefined behavior).

Even correct SPMD synchronization can be subtle. Figure 1e is correct, provided the value of variable `x` (which is *replicated*, i.e. each process has a variable `x` local to the process) is the same in all processes. This pattern—conditional synchronization where the program's design guarantees processes make the same branch decisions—is ubiquitous in SPMD programs. Figure 1f gives a more complex example illustrating the same point. However, processes in correct programs need not always make the same branch decisions, as Figures 1d, g, and h show.

1.2 Synchronization Verification

Figure 1e shows that an important component of understanding synchronization behavior is knowing which replicated variables must have the same value in all processes: We call such variables *single-valued*. Replicated variables with different values in different processes are *multi-valued*. Informally, a variable x is single-valued if for every assignment $x = e$, either e is a constant, a broadcast, or a function of other single-valued variables. One can show by induction on the length of program executions that single-valued variables take on the same sequence of values in all processes. A formal definition of single-valued requires more development (see Section 3.1).

In practice, SPMD programmers use synchronization in a highly structured way. All SPMD programs we have seen observe the following notion of synchronization correctness, which relies on knowing single-valued variables.

Definition 1.1 (Structural Correctness) An expression is structurally correct if all subexpressions e satisfy the following: Let V be the set of single-valued variables on entry to e and V' the set of single-valued variables on exit from e. If processes begin execution of e in environments having the same value for each variable in V and all processes terminate (i.e., no process loops), then all processes execute the same sequence of synchronization operations and end execution in environments having the same value for each variable in V'.

It is easy to check that Figure 1f, g, and h are structurally correct and that Figure 1e is structurally correct assuming `x` is single-valued. Figure 1d is an example without synchronization errors that is not structurally correct (because of the expressions marked $(*)$).

1.3 Barrier Inference

We have developed a static semantics that verifies that a program has structurally correct synchronization. Because `barriers` are the most common form of SPMD synchronization, we call this process *barrier inference*. Statically checking synchronization behavior guarantees that programs never fail by "hanging" at barriers or executing conflicting synchronization operations. SPMD programmers do make such mistakes,[2] and our techniques eliminate this class of bugs. Equally important, our method makes explicit the heretofore implicit assumptions about single-valued variables in SPMD programs. In our experience, this extra information is extremely useful for understanding SPMD programs written by others. Barrier inference also gives the compiler a more precise understanding of the portions of the program that execute in parallel, which makes SPMD optimizations, e.g. [14], more precise.

There are structurally correct programs our system cannot verify, such as Figure 1h. Intuitively, the problem with this example is that although both branches execute the same number of barriers, our system only infers that the branches each execute some unknown number of barriers and cannot tell that these numbers are equal. In contrast, our system verifies Figure 1g by inferring that both branches execute two barriers. While we have seen examples similar to Figure 1g, we have seen no programs with the structure of Figure 1h.

We present our barrier inference algorithm, which statically verifies the correctness of an SPMD program's synchronization behavior (Section 3), along with a proof of soundness (Section 3.1). We propose language features that make the synchronization structure of SPMD programs explicit (Section 4.1). We have implemented a prototype system to validate the algorithm and to empirically study the proposed language features. We tested the prototype on a substantial number of Split-C programs (Section 5). Experience with our implementation is positive; the system successfully checks the benchmarks with a few minor modifications to the programs, including one to correct a bug detected by our system. We also examined the Splash-2 benchmarks [25] by hand and found that all but one can be checked with our system (Section 5.2). These experiments were for medium-size programs; we believe that static verification of synchronization is especially important for larger systems because these are not amenable to manual verification, and also for higher-order languages (e.g. parallel object-oriented languages) where control-flow is less explicit.

2 The Language

We present our system using $\mathcal{L}$, a small procedural language extended with three parallel operations: `barrier`, `broadcast` (which is like `barrier` except a distinguished value is sent to all processes), and `communicate` (which allows asynchronous communication). As our interest is in synchronization operations such as `barrier` and `broadcast`, we leave the semantics of `communicate` unspecified. The grammar for $\mathcal{L}$ is:

$$
\begin{array}{lcl}
\texttt{Expr} & ::= & \texttt{i} \\
& | & \texttt{id} \\
& | & \textbf{barrier} \\
& | & \textbf{broadcast} \\
& | & \textbf{communicate} \\
& | & \texttt{id(Expr}, \ldots, \texttt{Expr)} \\
& | & \texttt{id} \leftarrow \texttt{Expr} \\
& | & \textbf{if}\ \texttt{Expr Expr}\ \textbf{else}\ \texttt{Expr} \\
& | & \texttt{Expr; Expr} \\
& | & \textbf{let}\ \texttt{id}\ \textbf{in}\ \texttt{Expr} \\
& | & \textbf{letrec}\ \texttt{id(id}, \ldots, \texttt{id)} = \texttt{Expr}\ \textbf{in}\ \texttt{Expr}
\end{array}
$$

[2] It is difficult to provide direct evidence for this claim, but we have committed such programming mistakes ourselves and found them in existing, presumably debugged, programs.

Values in $\mathcal{L}$ are integers and all variables are replicated. A `let` introduces a new variable and a `letrec` introduces a potentially recursive function definition; the other expressions are also standard. There are some predefined functions, such as $+$, which are mathematical functions, i.e. their result depends solely on their arguments. In examples we write `while` $\texttt{e}_1$ $\texttt{e}_2$ as shorthand for

$$\texttt{letrec f()} = \texttt{if}\ \texttt{e}_1\ (\texttt{e}_2\texttt{; f())}\ \texttt{else 0 in f()}$$

This spare language is sufficient to illustrate the novel aspects of our techniques. In Section 4.2 we discuss extensions to the C- and FORTRAN-based languages used in practice. Figure 2 gives a simple, CPS-inspired state-transition semantics for $\mathcal{L}$. The computation of one process is a sequence of steps:

$$\texttt{State} \rightsquigarrow \texttt{State}$$

where a state $\texttt{FunEnv} \times \texttt{Env} \times \texttt{Cont} \times \texttt{Expr}$ consists of an expression e to be evaluated, environments for the variables and function names in scope at e, and the computation to perform after evaluating e (a continuation). Termination is indicated by returning a special state without a continuation or a function environment. Readers familiar with CPS semantics will note that this CPS semantics is non-standard, because a continuation is a function returning only the next state in the computation, rather than the final answer of the entire computation. This modification exposes intermediate states of the computation, which is needed to define the semantics of `barrier` and `broadcast`.

The semantics of $\mathcal{L}$ model synchronization structure, but not the details of the communication primitives. The synchronization primitives, `barrier` and `broadcast`, are the only operations requiring global interaction. For `barrier`, once all processes reach a `barrier` each process proceeds with its continuation. The rule for `broadcast` is identical. The values returned by the communication operations are predicted by an $oracle()$ function. The only place where the communicated value is important is in `broadcast`: it returns the same value in all processes, but the actual value is not important for synchronization verification. The `barrier` operation does not communicate any values, so its result is always 0 (an arbitrary choice).

For simplicity, we assume variables and functions are given unique names (i.e., no names hide names in outer scopes). This property can be enforced by renaming variables.

The semantics of Figure 2 uses a few operations: $FF(f)$ is the set of function names in scope at f's definition; $FV(f)$ is the set of identifiers (other than f's formal parameters) in scope at f's definition. The set $dom(E)$ is the domain of E.

$$
\begin{array}{ll}
F & \texttt{FunEnv} = \texttt{FunctionName} \to \texttt{FunctionDefinition} \\
E & \texttt{Env} = \texttt{Var} \to \mathcal{N} \\
C & \texttt{Cont} = \texttt{Env} \times \mathcal{N} \to \texttt{State} \\
 & \texttt{State} = \texttt{FunEnv} \times \texttt{Env} \times \texttt{Cont} \times \texttt{Expression} + \texttt{Env} \times \mathcal{N}
\end{array}
$$

$$
\begin{array}{rcl}
\langle F, E, C, i\rangle & \leadsto & C\ (E, i) \\[1em]
\langle F, E, C, x\rangle & \leadsto & C\ (E, E(x)) \\[1em]
\langle F, E, C, communicate\rangle & \leadsto & C\ (E, oracle()) \\[1em]
\langle F, E, C_0, f(Expr_1, \ldots, Expr_n)\rangle & \leadsto & \langle F, E, C_1, Expr_1\rangle \text{ where} \\
 & & F(f) = \ f(x_1, \ldots, x_n) = Expr \\
 & & C_1 = \lambda E_2, v_1.\langle F, E_2, C_2, Expr_2\rangle \\
 & & \ldots \\
 & & C_{n-1} = \lambda E_n, v_{n-1}.\langle F, E_n, C_n, Expr_n\rangle \\
 & & C_n = \lambda E_{n+1}, v_n.\langle F|\ FF(f), E_0, C', Expr\rangle \\
 & & E_0 = (E_{n+1}|FV(f))[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n] \\
 & & C' = \lambda E', v.C_0\ ((E_{n+1}//FV(f) + E'//\{x_1, \ldots, x_n\}), v) \\[1em]
\langle F, E, C_0, p(Expr_1, \ldots, Expr_n)\rangle & \leadsto & \langle F, E, C_1, Expr_1\rangle \text{ where} \\
 & & p \text{ is a primitive} \\
 & & C_1 = \lambda E_2, v_1.\langle F, E_2, C_2, Expr_2\rangle \\
 & & \ldots \\
 & & C_{n-1} = \lambda E_n, v_{n-1}.\langle F, E_n, C_n, Expr_n\rangle \\
 & & C_n = \lambda E_{n+1}, v_n.C_0(E_{n+1}, p\,(v_1, \ldots, v_n)) \\[1em]
\langle F, E, C, x \leftarrow Expr\rangle & \leadsto & \langle F, E,\ \lambda E', v.C\ (E'[x \leftarrow v], v),\ Expr\rangle \\[1em]
\langle F, E, C, if\ Expr_1\ Expr_2\ else\ Expr_3\rangle & \leadsto & \langle F, E, C_0, Expr_1\rangle \text{ where} \\
 & & C_0 = \lambda E', v.\langle F, E', C,\ \text{if } v = 0 \text{ then } Expr_2 \text{ else } Expr_3\rangle \\[1em]
\langle F, E, C, Expr_1; Expr_2\rangle & \leadsto & \langle F, E,\ \lambda E', v.\langle F, E', C, Expr_2\rangle,\ Expr_1\rangle \\[1em]
\langle F, E, C, let\ x\ in\ Expr\rangle & \leadsto & \langle F, E[x \leftarrow 0],\ \lambda E', v.C\ (E'//\{x\}, v),\ Expr\rangle \\[1em]
\langle F, E, C, letrec\ f(x_1, \ldots, x_n) = Expr_1\ in\ Expr_2\rangle & \leadsto & \langle F[f \leftarrow f(x_1, \ldots, x_n) = Expr_1], E, C, Expr_2\rangle \\
 & & FF(f) = dom(F), \quad FV(f) = dom(E) \\[1em]
[\langle F_1, E_1,\ C_1,\ barrier\rangle, \ldots, \langle F_n, E_n,\ C_n,\ barrier\rangle] & \leadsto & [C_1\ (E_1, 0), \ldots, C_n\ (E_n, 0)] \\[1em]
[\langle F_1, E_1,\ C_1,\ broadcast\rangle, \ldots, \langle F_n, E_n,\ C_n,\ broadcast\rangle] & \leadsto & [C_1\ (E_1, v), \ldots, C_n\ (E_n, v)] \text{ where } v = oracle()
\end{array}
$$

Figure 2: Semantics for $\mathcal{L}$.

The environment $E|V$ is E with the domain restricted to variables V. The environment $E//V$ is E with variables V removed; i.e., $E|(dom(E) - V)$. The environment $E_1 + E_2$ is the combination of two environments E_1 and E_2 with disjoint domains.

The result of a (terminating) sequence of states is an environment recording the final state and an integer result. The computation of n processes executing in parallel is a sequence of steps:

$$\mathtt{State}^{\mathtt{n}} \rightsquigarrow \mathtt{State}^{\mathtt{n}}$$

The transitions for vectors of states include the synchronization rules for `barrier` and `broadcast`, plus a general rule for interleaving the transitions of individual processes:

$$[\mathtt{S}_1, \ldots, \mathtt{S}_{\mathtt{i}-1}, \mathtt{S}_{\mathtt{i}}, \mathtt{S}_{\mathtt{i}+1}, \ldots \mathtt{S}_{\mathtt{n}}] \rightsquigarrow [\mathtt{S}_1, \ldots, \mathtt{S}_{\mathtt{i}-1}, \mathtt{S}'_{\mathtt{i}}, \mathtt{S}_{\mathtt{i}+1}, \ldots \mathtt{S}_{\mathtt{n}}]$$

whenever $\mathtt{S}_{\mathtt{i}} \rightsquigarrow \mathtt{S}'_{\mathtt{i}}$.

Let I be the initial continuation $\lambda E, v.(E, v)$. The evaluation of expression e on n processors is

$$\begin{array}{c}[\langle\emptyset, \{\mathtt{pid} = 1\},\ \ \mathtt{I},\ \ \mathtt{e}\rangle, \ldots, \langle\emptyset, \{\mathtt{pid} = \mathtt{n}\},\ \ \mathtt{I},\ \ \mathtt{e}\rangle] \\ \overset{*}{\rightsquigarrow} [(E_1, i_1), \ldots, (E_n, i_n)]\end{array}$$

The initial environment of each process contains a *process id* in the variable `pid`. This value distinguishes one process from another.

If all processes halt with a final environment and integer value then that run is successful. A run is unsuccessful if (1) processes execute a different number of `barriers` (Figures 1a and 1b), (2) some processes reach a `barrier` at the same time others reach a `broadcast` (Figure 1c), or (3) one or more processes loop. Our methods are capable of statically checking realistic programs for (1) and (2).

3 Barrier Inference

Barrier Inference is an example of an *effect system* [7]. An effect associated with each expression models two aspects of SPMD computation. The first aspect is the sequence of `barriers` and `broadcasts` executed in evaluating an expression e. The rules associate an abstract synchronization sequence with e:

$$\mathcal{S} = \{\bot, f\} \cup \{b, r\}^*$$

A sequence value $s \in \{b, r\}^*$ means every process executes exactly the sequence s of `barriers` (b) and `broadcasts` (r). A sequence value f (for *fixed*), means every process executes the same unknown sequence of `barriers` and `broadcasts`. The sequence value $\bot$ means no process executes the expression. An element of $\mathcal{S}$ can be assigned to every structurally correct expression. There is an ordering on synchronization sequences:

$$\bot \preceq s \preceq f \quad \text{for any } s \in \{b, r\}^*$$

The second aspect of an expression's effect tracks single-valued variables. An abstract environment $\mathtt{AEnv} :\ \mathtt{Vars} \rightarrow \{+, -\}$ is a mapping from program variables to $+$ (indicating a variable is single-valued) or $-$ (indicating a variable may be multi-valued). There is an ordering $+ \preceq -$.

Analogous to an abstract environment there is an *abstract function environment*, which is a mapping

$$\mathtt{FEnv} :\ \ \mathtt{FunctionNames} \rightarrow \{+, -\}^{\mathtt{n}} \times \mathtt{AEnv} \times \{+, -\} \times \mathtt{AEnv} \times \mathcal{S}$$

from function names to function signatures.

Definition 3.1 A function f satisfies a signature written

$$(\mathtt{a}_1, \ldots, \mathtt{a}_{\mathtt{n}}), \mathtt{A} \rightarrow \mathtt{a}, \mathtt{A}', \mathtt{s}$$

if the following hold: f has n arguments and its free variables are those in $dom(A) = dom(A')$; processes beginning execution of f in states agreeing on values of the single-valued function arguments in $(\mathtt{a}_1, \ldots, \mathtt{a}_{\mathtt{n}})$ and single-valued variables in $\mathtt{A}$ either diverge or (1) agree on the result if $\mathtt{a} = +$, (2) agree on the value of every single-valued variable in $\mathtt{A}'$, and (3) have executed the same sequence of synchronization operations $\mathtt{s}$.

For example, the signature

$$f : (+, -), \emptyset \rightarrow +, \emptyset, \epsilon$$

says that $f(a, b) = f(a, c)$ for all b and c (provided both evaluations terminate) and f executes no synchronization operations. The inference system proves statements of the form

$$\mathtt{B},\ \ \mathtt{A} \vdash \mathtt{Expr} : \mathtt{a},\ \ \mathtt{A}',\ \ \mathtt{s}$$

which is read: Given functions matching abstract function environment `B`, if all processes begin the execution of `Expr` with the same values for variables marked single-valued in `A`, then all processes that terminate (1) agree on the values of variables marked single-valued in $\mathtt{A}'$, (2) agree on the result if $\mathtt{a} = +$, and (3) have executed the same sequence of synchronization operations `s`. Thus, any such proof shows e's structural correctness (Definition 1.1). The synchronization sequence s depends on information about single-valued variables and expressions, but not vice-versa. We find it most convenient to express both components in one set of rules.

The inference rules are in Figure 3. The remainder of this section discusses the rules, presents a soundness result, and illustrates barrier inference with examples. The [Int] rule is simple; evaluating an integer is single-valued (all processes compute the same integer), does not affect the set of single-valued variables, and executes no synchronization operations. The [Id] rule is similar; the result is single-valued only if all processes have the same value for the identifier in the environment. A `communicate` is assumed to be multi-valued, as processes may receive different values.[3] A `barrier` and a `broadcast` are always single-valued and each executes a single synchronization operation. The [Prim]

[3]When a process needs to communicate a value to all processes, `broadcast` is more efficient than n `communicate` operations, and makes explicit that the result is single-valued. Our experience with the Split-C programs of Section 5 shows that this rule is nearly universally followed.

$$\frac{}{B,\ A \vdash i : +,\ A,\ \epsilon} \quad \text{[Int]}$$

$$\frac{}{B,\ A \vdash id : A(id),\ A,\ \epsilon} \quad \text{[Id]}$$

$$\frac{}{B,\ A \vdash communicate : -,\ A,\ \epsilon} \quad \text{[Comm]}$$

$$\frac{}{B,\ A \vdash barrier : +,\ A,\ b} \quad \text{[Barrier]}$$

$$\frac{}{B,\ A \vdash broadcast : +,\ A,\ r} \quad \text{[Broadcast]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \ldots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \\ B(f) = (a'_1, \ldots, a'_n), A \rightarrow a, A', s \\ A_n | dom(A) \preceq A \\ \forall 1 \leq i \leq n.\ a_i \preceq a'_i \end{array}}{B,\ A_0 \vdash f(Expr_1, \ldots, Expr_n) : a,\ A_n // dom(A') + A',\ s_1 \oplus \ldots \oplus s_n \oplus s} \quad \text{[Fun]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ \ldots \\ B,\ A_{n-1} \vdash Expr_n : a_n,\ A_n,\ s_n \end{array}}{B,\ A_0 \vdash p(Expr_1, \ldots, Expr_n) : a_1 \sqcup \ldots \sqcup a_n,\ A_n,\ s_1 \oplus \ldots \oplus s_n} \quad \text{[Prim]}$$

$$\frac{B,\ A \vdash Expr : a,\ A',\ s}{B,\ A \vdash x \leftarrow Expr : a,\ A'[x \leftarrow a],\ s} \quad \text{[Assign]}$$

$$\frac{B,\ A[x \leftarrow +] \vdash Expr : a,\ A',\ s}{B,\ A \vdash let\ x\ in\ Expr : a,\ A' // \{x\},\ s} \quad \text{[Let]}$$

$$\frac{\begin{array}{l} dom(A) = dom(A') = dom(A_0) \\ S = (a_1, \ldots, a_n), A \rightarrow a, A', s \\ A' = A'' // \{x_1, \ldots, x_n\} \\ B[f \leftarrow S], A[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n] \vdash Expr_1 : a,\ A'',\ s \\ B[f \leftarrow S], A_0 \vdash Expr_2 : a'_2,\ A_2,\ s_2 \end{array}}{B,\ A_0 \vdash letrec\ f(x_1, \ldots, x_n) = Expr_1\ in\ Expr_2 : a'_2,\ A_2,\ s_2} \quad \text{[LetRec]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : +,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : a_2 \sqcup a_3,\ A_2 \sqcup A_3,\ s_1 \oplus (s_2 \sqcup s_3)} \quad \text{[If-Single]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : -,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \\ B,\ A_1 \vdash Expr_3 : a_3,\ A_3,\ s_3 \\ s_2 \sqcup s_3 \prec f \\ A' = A_1 \triangleleft (AV(Expr_2) \cup AV(Expr_3)) \end{array}}{B,\ A_0 \vdash if\ Expr_1\ Expr_2\ else\ Expr_3 : -,\ A',\ s_1 \oplus (s_2 \sqcup s_3)} \quad \text{[If-Multi]}$$

$$\frac{\begin{array}{l} B,\ A_0 \vdash Expr_1 : a_1,\ A_1,\ s_1 \\ B,\ A_1 \vdash Expr_2 : a_2,\ A_2,\ s_2 \end{array}}{B,\ A_0 \vdash Expr_1;\ Expr_2 : a_2,\ A_2,\ s_1 \oplus s_2} \quad \text{[Sequence]}$$

Figure 3: Inference rules.

rule says that primitive, side-effect-free functions are single-valued if all their arguments are single-valued.

In rule [Fun], actual parameters must be single-valued wherever the function signature requires single-valued arguments (the comparisons $a_i \preceq a_i'$). Similarly, the environment of the call must be single-valued in all variables the signature requires be single-valued. We define $A_1 \preceq A_2$ if $dom(A_1) = dom(A_2)$ and for all $x \in dom(A_1)$ we have $A_1(x) \preceq A_2(x)$.

The conclusion of [Fun] and several of the other rules combine synchronization sequences. The sequence $s_1 \oplus s_2$ is the best description of s_1 followed by s_2:

$$s_1 \oplus s_2 = \begin{cases} s_1 \cdot s_2 & \text{if } s_1, s_2 \in \{b, r\}^* \\ \perp & \text{if } s_1 = \perp \vee s_2 = \perp \\ s_1 \sqcup s_2 & \text{otherwise} \end{cases}$$

where $s_1 \cdot s_2$ is the concatenation of strings s_1 and s_2. The operator $\oplus$ is monotonic in both arguments.

Note the difference between the treatment of primitive and user-defined functions. The result of a primitive function is single-valued if all its arguments are single-valued, which is a kind of subtyping rule. Thus, some uses of a primitive function can be single-valued and others not. All calls to a user-defined function are either single-valued or not, depending on the function's signature in the abstract function environment. This distinction is necessary, because user-defined functions may modify single-valued state: If a function were sometimes called with arguments not matching its signature, the function might set single-valued state to a non-single value. We have not found this restriction on user-defined functions to be a problem in practice (see Section 5.1).

The [Assign] rule updates the environment based on the new value of the assigned variable; this reflects the fact that a variable can be single-valued at some program points and not at others. The [Let] rule introduces a new variable, which is initially single-valued as it is initialized to 0 in all processes. A new function is introduced into the function environment by the [LetRec] rule. This rule, and the [Fun] rule, express constraints on the function's signature; in [6] we outline an implementation that finds a solution to these constraints by fixed-point iteration.

The two rules for `if` are interesting. The rule [If-Single] applies when the predicate is single-valued. All processes take the same branch, but we do not know which branch. In this case a conservative upper bound over the results of both branches suffices.

The rule [If-Multi] applies when the predicate is multi-valued. The upper-bound of the synchronization sequence of the branches must be a known (not f) sequence. A subtle point is determining the single-valued variables of the final environment. Any variable modified in either branch could have different values in different processes on exit from the conditional; all such variables must be marked multi-valued in the final environment. It is easy to compute $AV(e)$, the set of variables visible at e that may be assigned in the evaluation of e (including via function calls in e). Now define $A \triangleleft \{v_1, \ldots, v_n\}$ as $A[v_1 \leftarrow -, \ldots, v_n \leftarrow -]$.

If the inference system of Figure 3 cannot assign any synchronization value to an expression, then evaluating the expression may cause processes to execute differing numbers of barriers and broadcasts—the program may get "out of synch." In this case the program is rejected. Of course, the inference system is conservative and may reject correct programs. We show in Section 5.1 that the system works well on realistic benchmarks.

3.1 Soundness

A sticky point in proving our system correct is capturing the meaning of single-valued variables. Intuitively, a variable is single-valued if all processors have the same value for the variable at the same time. However, "at the same time" is a slippery notion in a setting with asynchronous execution. Only at global synchronization points (i.e., `barriers`, `broadcasts`, and the start and end of execution) is it possible to assert anything useful about the state of all processes.

The key to this problem is to observe that the values of single-valued variables depend only on other single-valued expressions. Using this fact, it can be shown (without referring to time except within a single process) that if processes begin execution agreeing on single-valued inputs, then they terminate agreeing on the single-valued outputs.

The proof of soundness has two steps. First, we prove single-valued outputs are determined solely by single-valued inputs for a process in isolation. Second, we show that if the inference rules can derive any proof for an expression, then all processes evaluating that expression execute the same sequence of synchronization operations.

A few definitions are required. Environments E_1 and E_2 are equal with respect to an abstract environment A, written $E_1 \approx_A E_2$, if $dom(E_1) = dom(E_2) = dom(A)$ and $\forall x. A(x) = + \Rightarrow E_1(x) = E_2(x)$. A function environment F and an abstract function environment B are *compatible*, written $F : B$, if $dom(F) = dom(B)$ and for all $f \in dom(F)$:

$$\begin{gathered} F(f) = f(x_1, \ldots, x_n) = Expr \\ B(f) = (a_1, \ldots, a_n), A \to a, A', s \\ B|FF(f), A[x_1 \leftarrow a_1, \ldots, x_n \leftarrow a_n] \vdash Expr : a, A'', s \\ A' = A''//\{x_1, \ldots, x_n\} \end{gathered}$$

An execution $\mathtt{state}_1 \overset{*}{\underset{t}{\leadsto}} \mathtt{state}_2$ is an execution with *synchronization sequence* t, where t is a string with one b for each `barrier` and one r for each `broadcast` executed. The *broadcast sequence* of an evaluation $[S_1, \ldots, S_n] \overset{*}{\leadsto} [S_1', \ldots, S_n']$ is the sequence of values returned by successive calls to `broadcast` during this evaluation.

Lemma 3.2 Let e be any expression and let $B, A \vdash e : a, A', s$. Let $E_1 \approx_A E_2$, and $F : B$. If

$$\begin{array}{ccc} [\langle F, E_1, C_1, e \rangle] & \overset{*}{\underset{t_1}{\leadsto}} & [C_1(E_1', i_1)] \\ [\langle F, E_2, C_2, e \rangle] & \overset{*}{\underset{t_2}{\leadsto}} & [C_2(E_2', i_2)] \end{array}$$

and the broadcast sequences of both evaluations are identical, then the following are all true:

- $t_1 = t_2$ and $t_2 \preceq s$
- $E'_1 \approx_{A'} E'_2$
- $a = + \Rightarrow i_1 = i_2$

Theorem 3.3 Let e be any expression and let $B, A \vdash e : a, A', s$. Let $F : B$ and $E_i \approx_A E_j$ for $i, j = 1..n$. Then

$$[\langle F, E_1, I, e\rangle, \ldots, \langle F, E_n, I, e\rangle] \overset{*}{\rightsquigarrow} [(E'_1, v_1), \ldots, (E'_n, v_n)]$$

or some process diverges.

The proofs of Lemma 3.2 and Theorem 3.3 are given in [6].

The semantics of Figure 2 does not handle synchronization errors, i.e. the cases where `barriers` and `broadcasts` are mismatched or when some processes waits at a `barrier` while other processes have terminated. In those cases, the evaluation hangs. Theorem 3.3 shows that this cannot occur with barrier inference: either the program terminates, or the evaluation sequence is infinite.

3.2 Examples

We present example applications of the inference rules to Figures 1a and 1e. Other examples are included in Appendix A for the interested reader. The functions `work()` and `different()` do not contain barriers or modify visible variables.

Figure 1a fails the [If-Multi] rule - the alternatives of the `if` have different synchronization sequences.

$$\frac{\begin{array}{l}\emptyset, \emptyset \vdash \texttt{different()} : -,\ \emptyset,\ \epsilon \\ \emptyset, \emptyset \vdash \texttt{barrier} : +,\ \emptyset,\ b \\ \emptyset, \emptyset \vdash \texttt{0} : +,\ \emptyset,\ \epsilon \\ b \sqcup \epsilon = f \not\preceq f \text{ The rule fails.}\end{array}}{\emptyset, \emptyset \vdash \texttt{if different() barrier else 0} : ?}\ \text{[If-Multi]}$$

Figure 1e successfully passes the inference rules, assuming `x` is single-valued:

$$\frac{\begin{array}{l}\emptyset, \{\texttt{x} : +\} \vdash \texttt{x} : +,\ \{\texttt{x} : +\},\ \epsilon \\ \emptyset, \{\texttt{x} : +\} \vdash \texttt{barrier} : +,\ \{\texttt{x} : +\},\ b \\ \emptyset, \{\texttt{x} : +\} \vdash \texttt{work()} : -,\ \{\texttt{x} : +\},\ \epsilon\end{array}}{\begin{array}{l}\emptyset, \{\texttt{x} : +\} \vdash \texttt{ if (x) barrier else work()} : \\ \qquad -, \{\texttt{x} : +\}, \epsilon \oplus (b \sqcup \epsilon) = f\end{array}}\ \text{[If-Single]}$$

4 Realistic Languages

We now turn to the use of our techniques in realistic programming languages. Section 4.1 presents features we believe every SPMD language design should include. Section 4.2 discusses modifications needed to incorporate our techniques in programs written in C- or FORTRAN-based languages.

4.1 SPMD Language Design

Current SPMD languages have few ways of indicating the synchronization structure of an application. Even with barrier inference, this makes SPMD programs unnecessarily difficult to read and maintain. We propose two language features that make synchronization structure more explicit: named barriers and a `single` keyword to declare single-valued variables and functions.

Some SPMD languages provide *named barriers*, with the semantics that a runtime error results if processes simultaneously execute barriers with different names. Using named barriers indicates which syntactic `barriers` may participate in a synchronization. Named barriers also make the difference between [If-Multi] and [If-Single] explicit: an [If-Multi] must use the same barrier names in both branches, while an [If-Single] may use different names. Usually named barriers are implemented using a broadcast (so the names can be compared) which is much slower than special-purpose barrier hardware (e.g., on the CM5 [17] and T3D [4]). But $\mathcal{L}$ already effectively has two barrier names: `barrier` and `broadcast`. Adding more names increases the alphabet of synchronization strings but has no impact on inference complexity. Our system thus allows named barriers to be checked at compile-time, allowing their implementation with more efficient anonymous barriers. In a language with barrier inference there are only advantages to using named barriers.

Our inference system makes clear that knowing the single-valued variables is crucial to understanding an SPMD program's synchronization structure. We believe programmers should declare single-valued variables, formal parameters, and function results. These declarations are checked by a revised inference system. We propose a keyword `single` used as a type modifier (e.g., `single int x;`). The modifications to the language are:

```
Expr  ::=  ...
        |  let Decl in Expr
        |  letrec Decl(Decl,...,Decl) = Expr in Expr

Decl  ::=  id
        |  single id
```

Declaring single-valuedness has two advantages. First, the program is clearer as the common parts of the data-flow are explicit. Second, barrier inference is simplified. Because abstract environments can be built from `single` declarations rather than computed, proofs

$$\texttt{B},\ \texttt{A} \vdash \texttt{Expr} : \texttt{a},\ \texttt{s}$$

no longer need a result environment. Function signatures

$$(\texttt{a}_1, \ldots, \texttt{a}_n) \rightarrow \texttt{a}, \texttt{s}$$

do not include environments and can be built from the declarations. Figure 4 shows the new inference rules.

$$\frac{}{B,\ A \vdash i : +,\ \epsilon} \quad \text{[Int]}$$

$$\frac{}{B,\ A \vdash id : A(id),\ \epsilon} \quad \text{[Id]}$$

$$\frac{}{B,\ A \vdash communicate : -,\ \epsilon} \quad \text{[Comm]}$$

$$\frac{}{B,\ A \vdash barrier : +,\ b} \quad \text{[Barrier]}$$

$$\frac{}{B,\ A \vdash broadcast\ x : +,\ r} \quad \text{[Broadcast]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ \ldots \\ B,\ A \vdash Expr_n : a_n,\ s_n \\ B(f) = (a'_1, \ldots, a'_n) \to a, s \\ \forall 1 \leq i \leq n.\ a_i \preceq a'_i \end{array}}{B,\ A \vdash f(Expr_1, \ldots, Expr_n) : a,\ s_1 \oplus \ldots \oplus s_n \oplus s} \quad \text{[Fun]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ \ldots \\ B,\ A \vdash Expr_n : a_n,\ s_n \end{array}}{B,\ A \vdash p\,(Expr_1, \ldots, Expr_n) : a_1 \sqcup \ldots \sqcup a_n,\ s_1 \oplus \ldots \oplus s_n} \quad \text{[Prim]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr : a,\ s \\ a \preceq A(x) \end{array}}{B,\ A \vdash x \leftarrow Expr : a,\ s} \quad \text{[Assign]}$$

$$\frac{B,\ A[x \leftarrow a] \vdash Expr : a',\ s}{B,\ A \vdash let\ a\ x\ in\ Expr : a',\ s} \quad \text{[Let]}$$

$$\frac{\begin{array}{l} S = (a_1, \ldots, a_m) \to a_0, s \\ B[f \leftarrow S], A[x_1 \leftarrow a_1, \ldots, x_m \leftarrow a_m] \vdash Expr_1 : a_0,\ s \\ B[f \leftarrow S], A \vdash Expr_2 : a'_2,\ s_2 \end{array}}{B,\ A \vdash letrec\ a_0\ \ f(a_1\ x_1, \ldots, a_m\ x_m) = Expr_1\ \ in\ \ Expr_2 : a'_2,\ s_2} \quad \text{[LetRec]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : +,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \\ B,\ A \vdash Expr_3 : a_3,\ s_3 \end{array}}{B,\ A \vdash if\ Expr_1\ \ Expr_2\ \ else\ \ Expr_3 : a_2 \sqcup a_3,\ s_1 \oplus (s_2 \sqcup s_3)} \quad \text{[If-Single]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : -,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \\ B,\ A \vdash Expr_3 : a_3,\ s_3 \\ s_2 \sqcup s_3 \prec f \\ \forall x. A(x) = + \Rightarrow x \notin (AV(Expr_2) \cup AV(Expr_3)) \end{array}}{B,\ A \vdash if\ Expr_1\ \ Expr_2\ \ else\ \ Expr_3 : -,\ s_1 \oplus (s_2 \sqcup s_3)} \quad \text{[If-Multi]}$$

$$\frac{\begin{array}{l} B,\ A \vdash Expr_1 : a_1,\ s_1 \\ B,\ A \vdash Expr_2 : a_2,\ s_2 \end{array}}{B,\ A \vdash Expr_1;\ Expr_2 : a_2,\ s_1 \oplus s_2} \quad \text{[Sequence]}$$

Figure 4: Inference rules with a `single` keyword.

4.2 Application to Existing Languages

Some features of C and FORTRAN, which are popular starting points for SPMD languages, complicate barrier inference. Unstructured control-flow, aliasing, function pointers, and unitialized data structures are problematic.

The inference of single-valued variables is very similar to the problem of *binding-time analysis* in partial evaluation [12]: Given a set of variables whose value is assumed known (or single-valued in our case), determine which expressions and variables have a value that depends solely on these variables. Algorithms for binding-time analysis for C, such as [1], handle unstructured control-flow and can be modified to compute single-valued variables and synchronization sequences.

In the presence of pointers, detecting single-valued variables can require alias analysis, a well-known hard problem [15]. We have found very conservative assumptions suffice in practice (see Section 5.1): a variable whose address is taken is multi-valued; any pointer dereference is multi-valued. Similar problems arise with function pointers, so we require that functions whose address is computed have synchronization sequence ϵ and that all visible variables they assign are multi-valued.

When a data structure is initialized with a single-valued expression at creation, it remains single-valued so long as all modifications are single-valued. Without initialization, detecting when all elements of a data structure are single-valued is much harder. We mark uninitialized data structures as multi-valued.

In practice we have found pointers and complex data structures rarely used in conjunction with synchronization. There are a few exceptions; in particular, in C programs command-line arguments are single-valued pointers and strings in `argv`. Many programs parse `argv` to initialize some single-valued variables. For these situations a mechanism is needed for the programmer to assert a particular expression is single-valued. In the tradition of C, we call this a *single-valued cast*. Use of this feature should of course be minimized.

5 Experiments

We implemented a prototype of our inference system for Split-C [5], an explicitly parallel extension to C. We tested our prototype on Split-C kernels and applications. The empirical question we sought to answer is: How well does barrier inference integrate with real SPMD programming? Our measure is the number of changes preexisting programs required to conform to our system. The results were promising: the checks were all successful with minor changes, except for the exception handling aspects of one application. We also hand-examined the Splash-2 benchmarks and found that all but one would be checkable with our approach.

5.1 Split-C Prototype

For our purposes, the important features of Split-C are the `barrier()` and `all_bcast()` functions, which correspond to the `barrier` and `broadcast` primitives of $\mathcal{L}$.

The prototype combines a pure inference system with the language extensions of Section 4.1: It relies on a specification of the signatures of functions and a list of single-valued global variables, but it infers the single-valued local variables. It verifies all specifications are correct.

Our implementation follows the guidelines of Section 4.2 for supporting C, except we have not implemented the analysis of data structures (which was only needed by one of the Split-C programs). The algorithm for inferring single-valued variables is similar to [1], but includes synchronization sequence analysis.

Table 5.1 presents the programs and summarizes our results. The second column counts the static occurrences of barriers in the program, while the third column reports the number of branches that control the execution of a barrier and whose condition is single-valued. The function signature and single-valued globals columns report the number of annotations necessary to check the program. The cases that required modifications to the code are summarized in the 'single-valued casts' and 'other changes' columns. Except for 'svd', all the casts are for values computed by parsing program arguments (see Section 4.2). The 'svd' algorithm uses single-valued arrays (not supported by our prototype) which accounts for 18 of the 19 casts. The last cast arises from a single-valued result returned by reference, which implies taking the address of a variable. Our system assumes that any variable whose address is taken is not single-valued.

The 'barnes' application includes exception handling (via `setjmp`), which is unchecked by our system.[4] This application also required one small, local change: It broadcasts values without using the Split-C `broadcast` primitives; we replaced this code with explicit broadcasts. One-line changes were needed in three programs, 'mm', 'wator' and 'nbody' to avoid taking the address of single-valued variables read with `scanf`. The second change in 'nbody' corrected a minor bug detected by our prototype: when unexpected arguments were supplied only some processes exited.

These results show that our system successful verifies existing Split-C applications, with few changes and annotations. All but one of the programs depend on single-valued branches, which implies that conditional synchronization is the rule and not the exception in SPMD programs, and therefore that analysis of single-valued variables is necessary. The analysis time is low enough that our system can be integrated into an existing compiler without significant cost. The times, measured on a Sun Ultra-1/167Mhz, represent the time spent in our system and do not include the time to build the standard SSA representation used by our prototype.

5.2 The SPLASH-2 Benchmarks

As a further validation of our approach, we examined the synchronization structure of the SPLASH-2 benchmarks [25],

[4] Checking use of `setjmp` and `longjmp` in C is almost impossible in any program analysis. In 'barnes', when an exception arises in one process, the whole program is terminated.

Program	Lines	Number of barriers	Single-valued branches	Function signatures	Single-valued globals	Single-valued casts	Other changes	Analysis time
cannon	501	17	1	1	-	-	-	0.14s
cg	453	18	2	3	-	-	-	0.05s
cholesky	1542	38	16	4	-	2	-	1.06s
column	651	7	3	1	-	-	-	0.04s
fft3d	1181	12	5	1	-	1	-	0.05s
mm	508	23	1	1	-	-	1	0.07s
radix	379	7	3	-	-	2	-	0.06s
sample	302	9	0	-	-	-	-	0.06s
svd	1395	1	23	13	9	19 (or 1)[a]	-	0.12s
wator	348	10	5	-	3	-	2	0.04s
nbody	546	7	6	-	2	3	2	0.09s
em3d	1080	16	1	-	-	-	-	0.11s
barnes	2804	73	17	2	6	7	2	0.32s

[a] 18 of the 19 casts are required because of the lack of support for single-valued arrays.

Kernels:
· column, sample, radix: Sorting programs.
· cannon: Matrix multiplication using Cannon's algorithm.
· cg: Conjugent-gradient based equation solver.
· cholesky: 7 implementations of Cholesky decomposition.
· fft3d: A 3-dimensional fast fourier transform.
· mm: Matrix-multiply, blocked or unblocked.
· svd: Lanczos algorithm for singular-value decomposition.

Applications:
· wator: Simulation of particle-like fish under current.
· nbody: A simple n body simulation code.
· em3d: 3-dimensional electro-magnetic simulation.
· barnes: Simulate the interaction of a system of n bodies using the Barnes-Hut hierarchical method.

Table 1: Results of checking Split-C programs.

which are written in C extended with macros for writing parallel programs. The facilities provided by the macros include named barrier synchronization. Process management is with a fork/join model, but all but one program is written in SPMD style with all processes executing the same code. The exception is 'radiosity'; as it is outside our model we cannot check it.

Our implementation is written for Split-C and therefore does not check the SPLASH-2 programs. We examined the SPLASH-2 programs by hand to see if a suitably modified system would check these programs. The results of this examination are in Table 2. The four kernels and all but one of the applications pose no particular problems for our system.

Program (*: kernel)	Lines	Number of barriers	Checkable ?
ocean	2954 4703	19 20	yes, needs single-valued arrays (both versions)
barnes	2078	6	yes
fmm	3800	13	yes
radiosity	11319	5	no, not SPMD
raytrace	10020	1	yes
water	1744 2971	9 9	yes (both versions)
volrend	3704	13	yes
cholesky*	5050	4	yes
fft*	1005	7	yes
lu*	988 763	5 5	yes (both versions)
radix*	879	7	yes

Table 2: Results of examining the SPLASH-2 benchmarks.

6 Related Work

There are two strands of related work: SIMD (Single Instruction, Multiple Data) languages and synchronization analysis.

SIMD Languages divide variables into control unit and processing unit variables. Control unit variables resemble our single-valued variables: they are variables that have only one value. Unlike single-valued variables, control unit variables are stored in only one location. Control unit variables are declared with a `CU` keyword in the Illiac IV programming language Glypnir [16]. The Connection Machine language C* [23] calls these variables *scalar*. There is no equivalent of our inference system for these languages, as the properties we are inferring are guaranteed by SIMD semantics. Our proposed `single` keyword provides similar advantages for SPMD languages.

The ELP language [21, 24], a joint SIMD/SPMD programming language where both "modes" have the same semantics, allows declaration of single-valued variables with a `mono` keyword. When in SPMD mode the compiler guarantees that the single-valued property is preserved, presumably using rules similar to ours (the paper does not give many details on the checking strategy). ELP does not include explicit barriers or language-level broadcast, so there is no equivalent to our verification of synchronization. The programming model is also very different.

Analysis of the synchronization of parallel programs has been extensively studied for the purposes of deadlock and data-race detection as well as for optimization. Our survey of this work is necessarily partial, and covers only static techniques.

Jeremiassen and Eggers [11] analyse barrier synchronization for SPMD programs to improve the precision of optimization. They do not attempt to verify synchronization correctness. Their analysis relies on named barriers for precision and does not consider single-valued variables, though they do consider dependencies on multi-valued constants like `pid` [10].

A number of papers analyse 2-way synchronization, such as post/wait or Ada's accept/call mechanism, between explicitly specified tasks. As each task is specified with different code, there is no analogue of single-valued variables. Analyzing synchronization in this context is similar to analyzing the synchronization between the two branches in the [If-Multi] case, for which we only allow very simple synchronization sequences. None of the many papers on this subject present exact solutions for more general situations [2, 3, 9, 18, 19, 22, 26].

7 Conclusion

We have identified an important property of SPMD programs that current languages do not explicitly support: The portion of control and data flow governing global synchronization that is identical across all processes. This synchronization kernel structures the entire application. We have developed an inference system that both detects this structure and verifies that global synchronization is correct. An implementation of this system for Split-C successfully checks a number of programs.

The synchronization kernel is sufficiently important that it should be explicitly visible in source code. We propose language features that make SPMD programs clearer and easier to check.

We are integrating these language extensions into Titanium, a Java-based [8] successor to Split-C. This requires extending the application of the single-valued concept to more complex data structures, including references and objects, and to support language features such as exception handling.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996.

[2] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle WA, March 1990.

[3] D. Callahan and J. Subhlok. Static Analysis of Low-Level Synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison, WI USA, [1] 1989. ACM Press , New York, NY , USA. Published as SIGPLAN Notices, volume 24, number 1.

[4] Cray Research Incorporated. *The CRAY T3D Hardware Reference Manual*, 1993.

[5] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. *Introduction to Split-C*. University of California, Berkeley, 1993.

[6] D. Gay. Barrier Inference. Technical Report UCB//CSD-97-965, EECS Computer Science Division, University of California, Berkeley, July 1997.

[7] D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 REFERENCE MANUAL. Technical Report MIT-LCS//MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[9] D. P. Helmbold and C. E. McDowell. Computing Reachable States of Parallel Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):76–84, December 1991.

[10] T. Jeremiassen and S. Eggers. Computing Per-Process Summary Side-Effect Information. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 175–191, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag.

[11] T. E. Jeremiassen and S. J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Systems. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 171–180, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Co.

[12] N. D. Jones, C. K. Gomard, and P. Sestoff. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[13] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of the Supercomputing '93 Conference*, pages 262–273, Portland, OR, November 1993. IEEE Computer Society Press.

[14] A. Krishnamurthy and K. Yelick. Optimizing Parallel Programs with Explicit Synchronization. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 196–204, New York, NY, USA, June 1995. ACM Press.

[15] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[16] D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir – A Programming Language for Illiac IV. *Communications of the ACM*, 18(3):157–164, March 1975.

[17] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. The Network Architecture of the CM-5. In *Symposium on Parallel and Distributed Algorithms '92*, June 1992.

[18] S. P. Masticola and B. G. Ryder. Non-concurrency Analysis. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, New York, NY, USA, July 1993. ACM Press.

[19] C. E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, [6] 1989.

[20] Message Passing Interface Forum. Document for a standard message-passing interface. Technical Report UT-CS-93-214, University of Tennessee, Knoxville, 1993.

[21] M. A. Nichols, H. J. Siegel, and H. G. Dietz. Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):222–234, February 1993.

[22] R. N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[23] Thinking Machines Corporation. *C* Programming Guide*, 1993.

[24] L. Wang. *ELP User's Manual*. Parallel Processing Laboratory, School of Computer and Computer Engineering, Purdue University, March 1996.

[25] S. Cameron Woo, M. Ohara, E. Torrie, J. Pal Shingh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995. ACM SIGARCH and IEEE Computer Society TCCA.

[26] M. Young and R. N. Taylor. Combining Static Concurrency Analysis with Symbolic Execution. In *Proceedings Workshop on Software Testing*, pages 10–18, 1986.

A Examples

This appendix shows the results produced by our inference system on the more complex examples from Figure 1. The while loops of Figures 1b and 1e are rewritten using `letrec` so that we can directly apply the rules in Figure 3. Figure 5 shows the new code.

```
letrec w1() = if (different())
                (barrier; w1())
              else
                0
  in w1();
work1(); barrier();
work2(); barrier();
work3();
```

Example (b)

```
i <- 0;
letrec w2() = if (i < 10)
                (if (i = 1) barrier;
                 i <- i + 1;
                 w2())
              else
                0
  in w2();
barrier;
```

Example (f)

Figure 5: Loops rewritten with `letrec`.

- Figure 1b fails [If-Multi]. We end up trying to match

$$\frac{\begin{array}{l}\{\texttt{w1}:(),\emptyset \to +,\emptyset,\bot\},\emptyset \vdash \texttt{different()} : -,\ \emptyset,\ \epsilon \\ \{\texttt{w1}:(),\emptyset \to +,\emptyset,\bot\},\emptyset \vdash \texttt{(barrier; w1())} : +,\ \emptyset,\ b \\ \{\texttt{w1}:(),\emptyset \to +,\emptyset,\bot\},\emptyset \vdash 0 : +,\ \emptyset,\ \epsilon \\ b \sqcup \epsilon = f \not\prec f \text{ The rule fails.}\end{array}}{\vdash \texttt{if (different()) (barrier; w1()) else 0} :?}\ \text{[If-Multi]}$$

- Figure 1f succeeds with this signature for `w2`: $(), (\texttt{i}:+) \to +, (\texttt{i}:+), f$.

- Figure 1g successfully passes [If-Multi]

$$\frac{\begin{array}{l}\vdash \texttt{different()} : -,\ \emptyset,\ \epsilon \\ \vdash \texttt{(barrier; barrier)} : +,\ \emptyset,\ bb \\ \vdash \texttt{(work1(); barrier; work2();} \\ \quad \texttt{barrier)} : +,\ \emptyset,\ bb \\ bb \sqcup bb = bb \prec f\end{array}}{\vdash \texttt{if (different()) (...) else (...)} : -, \emptyset, bb}\ \text{[If-Multi]}$$

- Figure 1h fails because both branches have abstract synchronization sequence f

$$\frac{\begin{array}{l}\vdash \texttt{different()} : -,\ \emptyset,\ \epsilon \\ \vdash \texttt{(while ...)} : +,\ \emptyset,\ f \\ \vdash \texttt{(j = i + 10; ...)} : +,\ \emptyset,\ f \\ f \sqcup f = f \not\prec f \text{ The rule fails.}\end{array}}{\vdash \texttt{if (different()) (...) else (...)} :?}\ \text{[If-Multi]}$$