Parallel Static Single Assignment Form and Constant Propagation
for Explicitly Parallel Programs *

Jaejin Lee and David A. Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
{jlee,padua}@csrd.uiuc.edu

Abstract

Static Single Assignment (SSA) form has shown its usefulness for powerful code optimization techniques, such as constant propagation, of sequential programs. We introduce a new Parallel Static Single Assignment (PSSA) form and the transformation algorithm for the explicitly parallel programs with interleaving semantics and post-wait synchronization. The parallel construct considered in this paper is cobegin/coend. A new concept, π-assignment, which summarizes the information of interleaving statements among threads, is introduced. Parallel Control Flow Graph, which contains the information of conflicting statements in addition to control flow and synchronization information, is used as an intermediate representation for the PSSA transformation. A parallel extension of the Sparse Conditional Constant propagation algorithm based on the PSSA form makes it possible to apply the constant propagation optimization to explicitly parallel programs.

1. Introduction

Due to the impact of rapidly changing integrated circuit logic technology, a single chip will provide more than enough room for a microprocessor. It will be natural to have multiple processors on a single chip in near future. As a result, parallel programming is becoming more popular and important. It has already been popular in numerical and scientific computations. Despite of the importance of parallel programming, there are very little studies done on the optimization of explicitly parallel programs.

It is not possible to apply classical optimization techniques directly to parallel programs because of interleavings of statements and shared variables[8]. As shown in Figure 1, constant propagation optimization on a parallel program results in an incorrect program in the presence of busy-waiting. The value of variable a should be 3 at the point just after the assignment a = b but it is 4 in the example because we did not consider the interaction between threads. Thus it is common for programmers to turn off the optimization switch when they compile parallel programs.

To overcome these limitations, it is important to develop optimization algorithms and the corresponding intermediate representations for parallel programs. For the last few years, static single assignment (SSA) form has been proposed to represent data flow properties of sequential programs and being used in optimizing compilers[2, 4]. The SSA form has shown its usefulness for powerful code optimization techniques such as constant propagation [14], common subexpression elimination[10], partial redundancy elimination, code motion, and induction variable analysis. Recently, a parallel static single assignment form was proposed by Srinivasan et al.[13, 12]. However, it is restricted to the subset of parallel constructs in the PCF Parallel Fortran with copy-in/copy-out semantics in which the result of a parallel execution does not depend on the particular choice of the interleaving of statements in

---

*Supported in part by Army contract #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the government.
the explicitly parallel program. In addition, they have not considered synchronization mechanism in explicitly parallel programs.

In this paper, we introduce a new parallel static single assignment form for the language with interleaving semantics and synchronization mechanism. To convert an explicitly parallel program into its parallel static single assignment (PSSA) form, we introduce the parallel control flow graph, a parallel counterpart of the control flow graph in sequential program. After showing the algorithm for converting explicitly parallel programs into PSSA form, we describe an algorithm for constant propagation, which is a parallel extension of the sparse conditional constant propagation algorithm by Wegman and Zadeck[14] for the parallel static single assignment form. To the best of our knowledge, no studies have been done on the constant propagation for explicitly parallel programs.

After a brief overview of the sequential Static Single Assignment form in Section 2, we discuss the language model in Section 3. Section 4 describes Parallel Control Flow graphs. We develop Parallel Static Single Assignment form in Section 5. Section 6 describes the parallel sparse conditional constant propagation. In section 7, we discuss some related work. Section 8 is conclusions and future work for this paper.

2. Static Single Assignment Form

In this section we describe the traditional Static Single Assignment (SSA) representation[4]. In SSA form, a program has the property that only one definition (assignment) of each variable in the program can reach its uses. The SSA form contains \( \phi \)-functions that distinguish values of variables coming from different incoming control flow edges in the control flow graph[1] of the program. A \( \phi \)-assignment has the form \( V' = \phi(V_1, \ldots, V_n) \) where \( V', V_1, \ldots, V_n \) are variables and \( n \) is the number of incoming control flow edges for the node where the \( \phi \)-assignment is placed. We place \( \phi \)-assignments for a variable at its join nodes[4]. Multiple definitions of the variable reach at the join node through its distinct incoming control flow edges. Figure 2 shows an example SSA transformation.

3. Language Model

In this section, we describe the language used in this paper. We are using Fortran like language with cobegin/coend as the only parallel construct and the event synchronization mechanism Set/Wait. However, there is no goto or exit statement in this language, i.e. we are using a structured language. Figure 3 shows an example code which uses the language.

The parallel construct used is cobegin/coend. The construct consists of blocks of code and the block can contain another cobegin/coend construct. However, any looping mechanism such as do/endo, while/endwhile, and repeat/until may not contain cobegin/coend. Each block is separated from the next by //. Threads are generated using the cobegin statement and threads are synchronized at the coend statement. The syntax is:
4. Parallel Control Flow Graph

In this section we introduce parallel control flow graphs. The Parallel Control Flow Graph (PCFG) is an intermediate representation of explicitly parallel programs that has some similarities to the Parallel Program Graphs by Sarkar and Simons [11] and the Parallel Control Flow Graphs and Parallel Precedence Graphs by Wolfe and Srinivasan [15]. However, it is different in that it contains conflict edges in addition to synchronization edges and control flow edges. Also, PCFG does not have any concept such as super node of Parallel Program Graphs [11]. First we define the concept conflicts as follows:

**Definition 1** Two memory references in different threads conflict if they reference the same memory location and at least one is a write. Two statement conflict if they mutually contain conflicting memory references.

The notion of a basic block for explicitly parallel programs is different from the one in sequential setting. The definition of parallel basic blocks are as follows:

**Definition 2** A Parallel Basic Block has the following properties.

1. A sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end (this is the definition of a basic block [1] in sequential setting).
2. Only the first statement can be a Wait or contain a use of a conflicting variable.
3. Only the last statement can be a Set or contain a definition of a conflicting variable.
4. cobegin or coend is the only statement in a parallel basic block if the block contains cobegin or coend.

The following is the definition of conflicts between two parallel basic blocks.

**Definition 3** Two parallel basic blocks conflict if they contain statements that conflict each other.

The Parallel Control Flow Graph is defined as follows:

**Definition 4** A Parallel Control Flow Graph (PCFG) is a directed graph \( G = (N, E, N_{type}, E_{type}) \) such that,

- \( N \) is the set of nodes in \( G \).
- \( E = E_{ct} \cup E_{sy} \cup E_{cf} \) where,
  
  \[
  E_{ct} = \{(m, n) \mid m, n \in N \land E_{type}(m, n) \in \{T, F, U\}\}
  \]
  is the set of control flow edges.

  \[
  E_{sy} = \{(m, n) \mid m, n \in N \land E_{type}(m, n) \in \{S\}\}
  \]
  is the set of synchronization edges which show the order enforced by synchronization operations.

  \[
  E_{cf} = \{(m, n) \mid m, n \in N \land E_{type}(m, n) \in \{DU, DD, UD\}\}
  \]
  is the set of conflict edges.

- \( N_{type} \) is a function which tells the class of nodes, such that \( N_{type} : N \rightarrow T \) where, \( T = \{\text{Entry}, \text{Exit}, \text{Cobegin}, \text{Coend}, \text{Condition}, \text{Header}, \text{Compute}, \text{Thread Entry}, \text{Thread Exit}\} \).

- \( E_{type} \) is a function such that,
  
  \[
  E_{type} : N \times N \rightarrow \{T, F, U, S, DU, DD, UD\}.
  \]

The \text{Entry} and \text{Exit} nodes are special nodes which have no predecessors and no successors in PCFG respectively. Several threads are created at \text{Cobegin} node and the number of threads are the outgoing edges of \text{Cobegin} node. Threads are merged at \text{Coend} node. \text{Thread Entry} and \text{Thread Exit} nodes are special nodes that mark the beginning and the end of a thread in between \text{Cobegin} and \text{Coend} nodes.

\text{Condition} node is the same as the branch node in the Control Flow Graph[1] in sequential setting but if
it is a loop header node, it is Header node. Both Condition and Header nodes contain a condition for branching. Compute nodes are all the remaining nodes. Usually they contain a sequence of assignment statements.

If a control flow edges represents a true branch, it is labeled with $T$, if it represents false branch, its label is $F$. Otherwise its label is $U$ meaning unconditional. There always exists a control flow edge from Entry node to Exit node. The direction of a synchronization edge is from the node which contains Set to the node which has Wait for the same event variable. The label of a synchronization edge is $S$. If there are two conflicting statements, there are two edges between the nodes which contain the statements. The directions of the two edges are opposite. The function $E_{type}$ for conflict edges is defined by Table 1. The leftmost column is the type of conflict at the tail of the edges and the uppermost row is the type of conflict at the head of the edges. For example, if the tail of the conflict edge contains the definition of the conflicting variable, its status is def. If it contains a use, the status is use. If it contains both of them (the definition and use do not have to be for the same variable), it is def&use. In the case of head, the rule is similar to the case of tail. An example of labeling conflict edges is shown in Figure 4.

When we convert a do/endo loop or a repeat/until loop into a PCFG, we convert them into a while/endwhile loop and then convert the while/endwhile loop into the PCFG:

\[ f = a + b \]

\[ a = q + d \]

Figure 4. An example of labeling conflict edges.

Table 1. The labeling rule for conflict edges.

<table>
<thead>
<tr>
<th>def&amp;use</th>
<th>def</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>def&amp;use</td>
<td>DU</td>
<td>DD</td>
</tr>
<tr>
<td>def</td>
<td>DU</td>
<td>DD</td>
</tr>
<tr>
<td>use</td>
<td>UD</td>
<td>UD</td>
</tr>
</tbody>
</table>

If the loop is while/endwhile like loop, it is easy to separate the loop header from the nodes in the loop body because the exit from the loop is from the header. Figure 5 shows the PCFG for the explicitly parallel program in Figure 3.

5. Parallel Static Single Assignment Form

Analogous to the $\phi$-function in the static single assignment (SSA) form [4] in sequential setting, parallel static single assignment (PSSA) form has $\pi$-functions in addition to $\phi$-functions.

The meaning of $\phi$-functions in a parallel setting is the same as the one in a sequential setting, i.e. it distinguishes values of variables coming from distinct incoming control flow edges. In this paper, we extend the meaning of $\phi$-function to cover the case of $C_{end}$ node at which threads are merged.

**Definition 5** A $\phi$-function has the form
\(\phi(V_1, V_2, \ldots, V_n)\), where \(n\) is the number of incoming control flow edges of the node where it is placed. The value of \(\phi(V_1, V_2, \ldots, V_n)\) is one of the \(V'_i\)'s and the selection depends on the control flow. At Coend node, the selection depends on the interleavings of statements in different threads in execution.

The \(\pi\)-function summarizes the interleaving information of conflicting variables at the place where the \(\pi\)-function is placed. Similar to \(\phi\)-functions, the \(\pi\)-function distinguishes values of variables coming from an incoming control flow edge (the number of incoming control flow edges into the node where \(\pi\)-functions are placed is always 1 by the construction of PCFG) and distinct conflict edges labeled DU.

**Definition 6**

A \(\pi\)-function has the form \(\pi(V_1, V_2, \ldots, V_n)\), where \(n\) is the number of incoming control flow edges plus the number of incoming distinct conflict edges labeled DU. The value of the \(\pi(V_1, V_2, \ldots, V_n)\) is one of the \(V'_i\)'s and determined nondeterministically because of the concurrency among threads.

The basic idea of PSSA form is that it summarizes all the interleaving information for conflicting variables in an explicitly parallel program by using \(\pi\)-functions. The value of all the conflicting variables are well defined by the \(\pi\)-function at the point where the \(\pi\)-function is placed. Similar to SSA form in sequential setting, PSSA form has the property that all uses of a variable are reached by exactly one assignment to the variable.

Translating the PCFG of an explicitly parallel program into its PSSA form is basically a three-step process:

1. Get the partial ordering of conflicting statements. [Section 5.1.]
2. Place \(\phi\)-functions (\(\phi\)-assignments). [Section 5.2.]
3. Place \(\pi\)-functions (\(\pi\)-assignments). [Section 5.3.]

### 5.1. Partial ordering of conflicting statements

In order to get the information of the interleaving among statements in different thread, we need to get the partial ordering of the statements.

Callahan and Subhlok[3] used data flow analysis to get an event ordering for explicitly parallel programs without loops. We use a data flow analysis framework similar to theirs. Essentially this is the same as the method so called common ancestor algorithm by Emmrath, Ghosh and Padua [7, 5, 6]. This method computes an conservative approximation to the guaranteed ordering. In fact, computing the guaranteed ordering is Co-NP-hard [9].

By the way of construction of the parallel basic block, if we get the partial ordering between nodes in PCFG, it contains the partial ordering between conflicting statements in those nodes. Actually, in this paper, we are only interested in the partial orderings among nodes inside cobegin/coend since there is no loop construct which encloses cobegin/coend in our language model and we are interested in the interleaving information among statements in different threads.

The partial ordering of the nodes in the loop is not determined at compile time. We give each node in the loop the same precedence as the loop header node. There is no harm to do so, since we are only interested in the partial orderings among statements in different threads and there is no Set or Wait inside a loop by the language model.

Since all the loops in the PCFG are of the form of while/endif and there is no goto or exit statement in the language, all control flows, which go into a loop, are through the loop header (the header dominates all the node in the loop). To identify a loop header, we look for a back edge in the PCFG, then the node at the head of the back edge is the loop header [1] and the exit from the loop header is the only exit from the loop. The dominance relation is needed to identify back edges.

**Definition 7** We say node \(m\) of a PCFG dominates node \(n\), written \(m \text{ dom } n\), if every path, which consists of only control flow edges, from the Entry node of the PCFG to \(n\) goes through \(m\).

The algorithm for finding those nodes in a loop is well explained in [1]. Let \(\text{Loop}(n)\) be a function that returns the set of nodes in the loop whose header is \(n\).

For a given PCFG \(G = (N, E_u \cup E_y \cup E_c, \text{Ntype}, \text{Etype})\), We construct a subgraph \(G' = (N', E', \text{Ntype}, \text{Etype})\) of \(G\) where:

\[
N' = N - \{n \mid m, n \in N \land n \in \text{Loop}(m) \wedge \text{Ntype}(m) = \text{Header} \land m \neq n\}
\]

\[
E' = (E_u \cup E_y) - \{(m, n) \mid m, n \in N \land (m \notin N' \lor n \notin N')\}
\]

Intuitively, we merge all nodes in a loop into a representative node like in Figure 6. The representative node is the loop header. Notice that only the header node for the outermost loop belongs to \(N'\) when loops are nested. We calculate a set \(\text{Prec}(n)\) for a given node \(n\) in a data flow analysis framework where,

\[
\text{Prec}(n) = \{m \mid m \in N', \text{ m is guaranteed to precede n in execution}\}
\]
We implement $\text{Prec}(n)$, $\text{Prec}_{ct}(n)$, and $\text{Prec}_{sy}(n)$ for a given node $n$ with bit vectors. The set $\text{Prec}(m)$ for node $m$ such that $m \in N - N'$ is given by the following equation. Assume that a function $\text{Header}(m)$ returns the node, which is the header node of the outermost loop enclosing $m$.

$$\text{Prec}(m) = \text{Prec}(n) \quad \text{where } m \in N \land \text{Header}(m) = n$$

The data flow equations for the problem are as follows.

$$\text{Prec}(n) = \text{Prec}_{ct}(n) \cup \text{Prec}_{sy}(n)$$

$$\text{Prec}_{ct}(n) = \begin{cases} \bigcup_{(m,n) \in B_{ct}} (\text{Prec}(m) \cup \{n\}) & \text{if } \text{Ntype}(n) = \text{Coend} \\ \bigcap_{(m,n) \in B_{ct}} (\text{Prec}(m) \cup \{n\}) & \text{otherwise} \end{cases}$$

$$\text{Prec}_{sy}(n) = \bigcap_{(m,n) \in B_{sy}} (\text{Prec}(m) \cup \{n\})$$

We define a predicate $\text{Precede?}(m,n)$ as follows. $\text{Precede?}(m,n)$ is used later to get the partial ordering of two parallel basic blocks $m$ and $n$.

$$\text{Precede?}(m,n) = \begin{cases} \text{true} & \text{if } m \in \text{Prec}(n) \\ \text{false} & \text{otherwise} \end{cases}$$

By using the partial ordering between two conflicting statements, we can ignore some conflict edges when we generate $\pi$-assignments later. The algorithm for getting guaranteed execution ordering is given in Appendix A.

5.2. Placing $\phi$-assignments

In order to get PSSA form, we transform the PCFG into SSA form in sequential setting after synchronization analysis is done on the PCFG. In this section, we discuss how to place $\phi$-functions in the given PCFG. We consider only control flow edges and ignore conflict edges and synchronization edges in the PCFG. We place $\phi$-functions in the dummy node of the node in which $\phi$-functions should be placed. After placing all the $\phi$-functions in the dummy node, we insert the dummy node in front of the original node. All incoming control flow edges of the original node become incoming control flow edges of the dummy node and a new control flow edge, which goes from the dummy node to the original node, is created.

There are two different classical approaches for placing $\phi$-assignments. One is the algorithm for placing $\phi$-assignments and renaming variables by Cytron et al.[4]. The other is the SSA transformation for structured programs by Brandis and Mösenböck [2]. Our algorithm for placing $\phi$-assignments is based on the latter. Their transformation is done at parse time on structured programs, but our algorithm is applied to a PCFG. We also extend their method to cobegin/coend construct.

The algorithm basically does depth first traversal on the given PCFG. Since the join nodes of the branch nodes such as Cobegin, Condition, and Header are known a priori, when we meet one of those branch nodes, we do depth first traversal of its successor nodes to generate $\phi$-functions until we meet the corresponding join node. Since the method for Condition and Header nodes is almost the same as the method in the paper by Brandis and Mösenböck [2], we skip the cases of Condition and Header in this paper. In the following section, we describe the method for Cobegin node by following an example.

Placing $\phi$-assignments in the join node of Cobegin node

In this section, we explain the method for transforming a Cobegin block into SSA form by means of an example. The placement of $\phi$-assignments is almost the same as for the Condition block. However, if we treat a Cobegin block as a Condition block, lots of superfluous $\phi$-assignments are generated. In order to prevent this, we check each argument in $\phi$-function after filling in all the vacant arguments in it. If the $\phi$-assignment satisfies the following condition, we discard the $\phi$-assignment, since it is superfluous.

**Theorem 1** When the join node is a Coend node, a $\phi$-assignment is superfluous and can be eliminated if it has more than two parameters and all or all but one of its parameters are the same, or if it has exactly two different parameters and one of its parameters is defined before the corresponding Cobegin node.
2. The incoming values of variables into node B are the same as those of node A. Since C is defined, we generate a new variable C1 and replace C with C1 in the left hand side of the assignment. There has not been allocated a dummy node for the join node E yet. So, we create a dummy node E'. We place a ϕ-assignment for the variable C in the dummy node. The new variable C2 is generated. In the beginning, the ϕ-function has the form, ϕ(-, -,-), since there are three incoming control flow edges into node E and we do not know which value of C reaches the join node E. After we make sure that the successor of node B is the join node, we know the value of C that reaches the join node E is C1. Thus we fill in the first argument of ϕ(-,-,-) with C1. We get C2=ϕ(C1, -,-).

3. The visits of node C is similar to the case of node B. After we check that the successor of C is the join node, we fill in the vacant argument of ϕ-functions, that corresponds to the thread in which C node appears. As a result, the dummy node has two incomplete ϕ-assignments, C2=ϕ(C1,C3,-), and A2=ϕ(-,A1,-). Similarly, after we visit node D, we get C2=ϕ(C1,C3,C4), A2=ϕ(-,A1,-), and B2=ϕ(-,B1). We fill in those vacant arguments with the values of variables that goes into node A after we visit node F. The final ϕ-assignments are C2=ϕ(C1,C3,C4), A2=ϕ(A0,A1,A0), and B2=ϕ(B0,B0,B1).

4. After we fill in all the vacant arguments in ϕ-functions we insert the dummy node E' just before the Coend node. The predecessors of the Coend node become the predecessors of the dummy node. We create a new edge from the dummy node to Coend node.

5. Before we visit the successor of Coend node, we discard superfluous ϕ-assignments from the dummy node E'. For example, the ϕ-assignment A2 = ϕ(A0,A1,A0) is superfluous. After updating the current value of A with A1, we discard it. Note that, we do not update the current value of A with A2. The case of variable B is similar to A. This completes the visit to Coend node.

6. Continue to visit the successor of the Coend node. The current values of variables are A1, B1, C2.

The PCFG of Figure 5 after placing all the ϕ-assignments is given in Figure 8. The algorithm for placing ϕ-assignments is given in Appendix B.
5.3. Placing $\pi$-assignments

In this section, we explain how to place $\pi$-assignments in the given PCFG after placing $\phi$-functions. In sequential setting, we place the $\phi$-function of a variable in the join node of the nodes in which the variable is defined (in fact, it is placed in the dominance frontier of the nodes in which the variable is defined [4] because of efficiency). However, there is no join node in parallel setting, which is analogous to the one in sequential setting, since the definition of a variable may reach to the use of the variable from a different thread. Therefore we need a different notion of join node which covers the notion of conflict edges (in fact, $DU$ edges). We begin this section by defining parallel join nodes.

**Definition 8** A collection of simple paths $p_i$, $2 \leq i \leq n$, is said to converge by interleaving at a node $y$ if they satisfy the following conditions.

1. They have an end node $y$ in common.
2. They do not have any node in common except $y$.
3. There exists one and only one path $p_m$ which consists of only control flow edges.
4. Every path $p_i$, $i \neq m$, consists of only one edge and the type of the edge is $DU$.

**Definition 9** Given a collection of interleavingly converging paths $p_i$ at a node $y$, $y$ is said the parallel join node of the start nodes of $p_i$'s for a variable $V$, if the start node of $p_i$ is the only node which contains an assignment to $V$ in the path $p_i$.

Note that the sequence in which a definition of a variable followed by the use of the variable never occurs in a parallel basic block by the definition of a parallel basic block, so it is sufficient that we only consider paths which have length greater than 1.

Thus, we should locate those parallel join nodes in order to place $\pi$-functions. Since a node $x$ has an incoming $DU$ edge for a variable $V$ if and only if it is a parallel join node of $V$, it is easy to locate parallel join node. We just check all the $DU$ edges in PCFG to locate a parallel join node in order to place $\pi$-assignments. However we exclude the $DU$ edges which violates the partial ordering among conflicting statement. To do so, we use the predicate $Precede$ defined in section 5.1. We go through an example to explain the method. Figure 9 shows the steps which is required to place $\pi$-assignments.

When we place a $\pi$-assignment for a variable in the parallel join node of the variable, we generate new tem-
1. Node B has three incoming $DU$ edges from node C, F, and E. So we place a $\pi$-assignment in this node. The use of variable $A_0$ is replaced by the temporary $tA_0$. We insert the $\pi$-assignment $tA_0=\pi(-,-,-,-)$ in which the number of argument is the number of all incoming $DU$ edges corresponding to the variable $A$ plus 1. We fill in the first argument of the $\pi$-function with the reaching definition through the incoming control flow edge into node $A$. It is $A_0$. We get $tA_0=\pi(A_0,-,-,-)$. We go over all incoming $DU$ edges one by one. By using the predicate Precede? we get the partial ordering among the nodes which are the source of those $DU$ edges. The Hasse diagram of the partial ordering is shown in Figure 10.(A). Since node B precedes node E, we do not consider the $DU$ edge from node E for the $\pi$-function in node B. We remove a vacant argument in the $\pi$-function. We get $tA_0=\pi(A_0,-,-)$. Since there is no ordering among node B, C, and F, we have to consider those $DU$ edges from node C and F for the $\pi$-function in node B. We fill in rest of the arguments with $A_3$, $A_5$. The order of argument is not important except the first one. Finally, the $\pi$-assignment become $tA_0=\pi(A_0,A_3,A_5)$.

2. When we visit node E, we get the ordering among the nodes in Figure 10.(B). Similar to the visit on node B, we get the initial $\pi$-assignment $tA_3=\pi(A_3,-,-)$. Since we do not have the ordering among node E and F and node B precedes E, We fill in the $\pi$-function with $A_1$, $A_5$. We get $tA_3=\pi(A_3,A_1,A_5)$.

3. When we visit node H, we get the partial ordering among node B, C, E, and H in Figure 10.(C). Similar to the visit on node B, we get the initial $\pi$-assignment $tA_5=\pi(A_5,-,-,-)$. Since both node B and C precede node E and node E precedes node H, we do not count the definition from node B and C. This is because the definitions of the variable A from node B and C are killed by the definition in node E. Thus, we remove two vacant arguments in $tA_5=\pi(A_5,-,-,-)$ and get $tA_5=\pi(A_5,-)$. We fill in the vacant argument with the value $A_4$ from node E and get $tA_5=\pi(A_5,A_4)$.

The final PSSA form of the code in Figure 5 is given in Figure 11. The algorithm for placing $\pi$-assignments is given in Appendix C.

6. Constant Propagation

In this section, we describe a constant propagation algorithm for explicitly parallel programs based on PSSA representation. This algorithm is a parallel extension of the Sparse Conditional Constant (SCC) by Wegman and Zadek[14]. We call the parallel extension of the algorithm Parallel Sparse Conditional Constant (PSCC) propagation algorithm.

6.1. The lattice for constant propagation

The lattice used in the constant propagation is shown in Figure 12. During constant propagation, each variable used or defined in the program is mapped to an element of the lattice. There are three types of lattice elements. $\top$ is the highest element of the lattice and
means that nothing may yet be asserted about the variable in question. \( \perp \) is the lowest element of the lattice and means that the variable in question has been determined not to have a constant value. \( C_i \) is less than \( \top \) and greater than \( \perp \) in the lattice and means that the variable has the constant value \( C_i \). There are infinite number of \( C_i \)'s in the lattice. The meet operation \( \sqcap \) of the lattice is defined in Table 2. In addition, we need rules for evaluating expressions. These rules are summarized in Table 3. The case where an operand in an expression has the value \( \top \) never occurs in PSCC algorithm if all the variables are initialized in the original program.

### 6.2. The constant propagation algorithm

After we transform an explicitly parallel program into its PSSA form, we add def-use edges between statements. We call them PSSA (in sequential setting, they call it SSA edge[14]) edges.

**Definition 10** A PSSA edge goes from the statement where a variable is defined to a use of the variable. The edges may go between statements in different threads.

Each node \( n \) in the given PCFG has a function \( n.LatticeValue \). For each variable \( v \) which appears in node \( n \), \( n.LatticeValue \) maps \( v \) to its lattice value in node \( n \).

The algorithm proceeds by modifying the mapping \( n.LatticeValue \). The value corresponding to each variable is lowered as more information for the variable is discovered. It continues until a fixed point is reached. All the variables map to \( \top \) initially. However, the variable, which appears in read or is uninitialized in the original PCFG, initially maps to \( \perp \). Without loss of generality, we assume all the variables are initialized in the original program.

The algorithm is an work list algorithm. A work list \( CTFE \) contains control flow edges and the other work list \( PSSAE \) contains PSSA edges. Initially, the \( CTFE \) contains \textit{Entry} node and \textit{PSSAE} is empty.

Each control flow edge has a flag \textit{ExecutionFlag} for the possibility of execution of the edge. If an edge is executable, its \textit{ExecutionFlag} is \text{True}, otherwise it is \text{False}. Each node has a flag \textit{ExecutionFlag}. Its function is the same as the one of control flow edges.

The full Parallel Sparse Conditional Constant Propagation algorithm is given in Appendix D. Figure 13.(a) is the initial PCFG before PSSA transformation. Figure 13.(b) is the one after PSSA transformation. The result of the constant propagation is given in Figure 14. In this figure, all the unreachable nodes are removed.

The essential difference between PSCC and SCC[14] is the existence of \( \pi \)-function and the definition of PSSA edges. When we evaluate a \( \pi \)-function, we apply meet operation \( \sqcap \) of the lattice to all of the arguments in the \( \pi \)-function. For example, assume the arguments \( a_3, a_6, \) and \( a_8 \) of a \( \pi \)-assignment \( \tau a_3=\pi(a_3,a_6,a_8) \) have the lattice value 3, 3, and \( \top \) respectively after constant propagation. \( a_9 \) has the value \( \top \) means that the definition node of \( a_8 \) is unexecutable. \( ta_3 \) has the

![Figure 12. The lattice for constant propagation.](image)

![Table 2. The meet operation \( \sqcap \) for the lattice. \( i \neq j \)](table)

<table>
<thead>
<tr>
<th>( \sqcap )</th>
<th>( \top )</th>
<th>( C_i )</th>
<th>( C_j )</th>
<th>( \perp )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( C_i )</td>
<td>( C_j )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>( C_i )</td>
<td>( \top )</td>
<td>( C_i )</td>
<td>( C_i )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>( C_j )</td>
<td>( \top )</td>
<td>( C_j )</td>
<td>( C_j )</td>
<td>( \perp )</td>
</tr>
<tr>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
</tr>
</tbody>
</table>

![Table 3. The expression evaluation rules for arithmetic operations op such as +, -, *, / and logical operations such as \( \land \), \( \lor \).](table)

<table>
<thead>
<tr>
<th>op</th>
<th>( C_i )</th>
<th>( C_j )</th>
<th>( \land )</th>
<th>( \lor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_i )</td>
<td>( C_j )</td>
<td>( \top )</td>
<td>( \top )</td>
<td></td>
</tr>
<tr>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 13. An example PCFG for constant propagation.](image)
constant value 3 by applying ∩ operation to a3, a6, and a8. Thus, this assignment is replaced by v_{a3}=3. PSSA edges contains def-use edges across different threads in addition to SSA edges[14]. It is easy to find PSSA edges since all the uses of conflicting variables are the arguments of π-functions in a given PSSA form of PCFG.

6.3. Conservativeness of PSCC

We discuss in this section that PSCC algorithm is conservative in the sense that PSCC does not say that a variable is a constant, when it is not the case. It is well known that the SCC algorithm is conservative [1]. We show next that PSCC treats π-function conservatively in the presence of incorrectly introduced arguments. These incorrect arguments may be introduced by the inexact synchronization analysis (the common ancestor algorithm which is used to get guaranteed ordering among statements). One of these situations is depicted in Figure 15. The argument a2 in the π-assignment is incorrectly introduced due to the lack of information about execution ordering. This comes from common ancestor algorithm for synchronization analysis. The situation can be avoided by the more exhaustive synchronization analysis.

Without loss of generality, we assume the π-function has only two arguments v_0 and v_1 and v_1 is incorrectly introduced. We apply the meet operation ∩ to the arguments of the π-function for all possible cases of v_1. The result of evaluation is as follows. Here, C and K are different constants.

<table>
<thead>
<tr>
<th>value of v_0</th>
<th>value of v_1</th>
<th>value of π</th>
<th>exact result</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td>⊓</td>
<td>⊓</td>
<td>v_0</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>⊓</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>K</td>
<td>⊓</td>
<td>C</td>
</tr>
<tr>
<td>v_0</td>
<td>⊓</td>
<td>⊓</td>
<td>v_0</td>
</tr>
</tbody>
</table>

The 'exact result' in the above table means the result of evaluating the π-function without incorrectly introduced arguments. In this case, it is the same as the value of v_0. Since the value of π-function is not higher than 'exact result' in the lattice, the incorrectly introduced argument v_1 does not do any harm in constant propagation. However, the number of constants we will have found would be less than the number of constants without incorrectly introduced arguments.

7. Related Work

Currently, there are very little studies done on the application of classical optimization techniques to explicitly parallel programs. Wolfe and Srinivasan[13, 15] and Srinivasan and Grunwald[12] proposed a parallel static single assignment form for explicitly parallel programs. However, their PSSA form is restricted to the subset of parallel constructs in the PCF Parallel Fortran with copy-in/copy-out semantics in which the result of a parallel execution does not depend on the particular choice of the interleaving of statements in the program. In addition, they have not considered more general parallel constructs, such as the one with interleaving semantics and the post-wait synchronization. Thus, their method would not be much practical in real optimization of explicitly parallel programs.

They used Parallel Sections construct and Wait
clause in PCF Fortran. The parallel programs with these constructs can be easily converted into programs with cobegin/coend construct with post-wait synchronization. Then, the program can be converted into the PSSA form, which uses both $\phi$-functions and $\pi$-functions and is discussed in this paper.

8. Conclusions and Future Work

We introduced a new parallel control flow graphs (PCFG) which contains the information of conflicting statements in explicitly parallel programs. The PCFG is used as an intermediate representation for explicitly parallel programs in order to convert them into parallel static single assignment form. The new parallel static single assignment form proposed in this paper is for the explicitly parallel program with interleaving semantics and post-wait synchronization. It uses both $\phi$- and $\pi$-assignments. The $\pi$-function is a new concept and it summarizes the interleaving of statements at the point where it is placed. Using this parallel static single assignment form as an intermediate representation, we extended the classical sparse conditional constant propagation algorithm[14] to the one for explicitly parallel programs.

An extension of the work in this paper will be PSSA form for the parallel do constructs with post-wait synchronization. Also, we plan to develop the parallel counterparts of the sequential optimization techniques other than constant propagation.

References


Appendix A. The algorithm for getting guaranteed execution ordering

Input: A parallel control flow graph
\[ G = (N, E_{st} \cup E_{sy} \cup E_{sf}, \text{Ntype, Etype}) \]

Output: \( \text{Prec}(n) \) for each parallel block in \( G \).

Construct a subgraph \( G' = (N', E', \text{Ntype, Etype}) \) of \( G \) such that:

- \( \text{Loop}(n) \) is a function that returns the set of
  - nodes in the loop whose header is \( n \).

- \( N' = N - \{ n | m, n \in N \land n \in \text{Loop}(m) \land \text{Ntype}(m) = \text{Header} \land m \neq n \} \)

- \( E' = (E_{st} \cup E_{sy}) - \{(m, n) | m, n \in N \land (m \notin N' \lor n \notin N') \} \)

Initialize \( \text{Prec}(n) = \emptyset \) for all \( n \in N' \).
Initialize a queue \( Q \) with the successors of the Entry node in \( N' \).

while \( Q \neq \emptyset \) do
  \( n := \) the first entry in \( Q \)
  \( \text{Prec}_{ad} := \text{Prec}(n) \)
  if \( \text{Ntype}(n) = \text{Coend} \) then
    \( \text{Prec}(n) := \bigcup (m,n) \in E_{sf} \text{Prec}(m) \cup \{ n \} \)
  else
    \( \text{Prec}(n) := \bigcap (m,n) \in E_{sy} \text{Prec}(m) \cup \{ n \} \)
  endif
endwhile

for each node \( n \in N - N' \) do
  \( \text{Header}(n) \) returns the node, which is the
  \( \text{header node of the outermost loop enclosing } n \)
  \( \text{Prec}(n) := \text{Header}(n) \)
endfor

Appendix B. The algorithm for placing \( \phi \)-assignments

Input: A parallel control flow graph
\[ G = (N, E_{st} \cup E_{sy} \cup E_{sf}, \text{Ntype, Etype}) \]

Output: The parallel control flow graph \( G' \) in which \( \phi \)-functions are placed.

Procedure \text{PlacePhiAndRename}(\text{Entry node of } G, \emptyset, \text{NIL})

if \( \text{Ntype}(n) = \text{Exit} \) then return
\( n.\text{Visit} := \text{True} \)
newdef := def
for each statement \( S \) in \( n \)
in sequential order do
if \( S \) defines a variable \( V \) then
  \( \mathcal{V}_{NEW} := \text{NewVar}(V) \)
  newdef := (newdef - \{ (\mathcal{W}, V) | \mathcal{W} = V \land (W, V) \notin \text{newdef} \} \cup \{ (V, \mathcal{V}_{NEW}) \})
if \( \text{join} \neq \text{NIL} \) then
  InsertPhi(join, \( V \))
endif
Replace the definition of \( V \) in \( S \) with \( \mathcal{V}_{NEW} \)
endif
if \( S \) is not set or wait statement then
  for each uses of a variable \( W \) in \( S \) do
    \( \mathcal{W}_{NEW} := \text{Search}((W, \text{newdef}) \cup \{ (V, \mathcal{V}_{NEW}) \}) \)
    Replace the use of \( W \) with \( \mathcal{W}_{NEW} \)
  endfor
endif
endfor

if \( \text{Ntype}(n) = \text{Header} \) then
  \( X := \{ m | m \) is a successor of \( n \land m \) is not an exit of the loop whose header is \( n \} \)
else
  \( X := \{ m | (n,m) \in E_{st} \} \)
endif
newjoin := the join node of \( n \)
for each \( m \in X \) do
if \( \text{Ntype}(n) = \text{Condition} \lor \text{Ntype}(n) = \text{Cobegin} \) then
  if \( m \neq \text{newjoin} \) then
    PlacePhiAndRename(m, newdef, newjoin)
  endif
else if \( \text{Ntype}(n) = \text{Header} \) then
  if \( m = n \) then
    PlacePhiAndRename(m, newdef, n)
  endif
else
  if \( m = \text{join} \) then
    FillInPhi(n, join, newdef)
  else
    PlacePhiAndRename(m, newdef, join)
  endif
endif
endif
endfor

if \( \text{Ntype}(n) = \text{Condition} \lor \text{Ntype}(n) = \text{Cobegin} \) then
  CompletePhi(newjoin, def)
endif
Insert the dummy node for newjoin in front of \( n \)
ProcessDummy (the dummy node for newjoin, \( \text{join} \))
phides := GetNewDefsFromPhi(newjoin)
newdef := (def - \{ (V, \mathcal{V}_{2}) | (\exists W. (V, W) \in \text{phides}) \land (V, \mathcal{V}_{2}) \in \text{def} \}) \cup \text{phides}
Appendix C. The algorithm for placing π-assignments

Input: A transformed parallel control flow graph
\[ G = (N, E_{ex} \cup E_{ay} \cup E_{cf}, N_{type}, E_{type}) \]
in which the placement of φ-assignments is done.

Output: A transformed parallel control flow graph
in PSSA form.

for each node \( n \in N \) do
    \( S := \{ v \mid (v, n) \in E \land E_{type}(v, n) = DU \} \)
    if \( S \neq \emptyset \) then
        \( D := \{ V_k \mid V \text{ is defined in the last} \)
            \( \text{statement of } v \land v \in S \)
        \( V := \text{the conflicting use in } n \)
        Insert a φ-assignment \( v = \phi(v, -\cdots, -) \)
            at the beginning of \( n \)
        \( \triangleright \) is the guaranteed execution ordering
        \( \triangleright \) among nodes in \( S \cup \{ n \} \).
        \( \triangleright \) If \( x \triangleright y \) then \( x \) is guaranteed to execute
        \( \triangleright \) before \( y \) executes.
        \( T := \{ V_k \mid v \in S \land n \triangleright v \land \exists z \geq n \land x \triangleright z \}
            \land x, n, z \text{ are all distinct} \}
        \( D := D \cup T \)
        Fill in the vacant positions in the π-assignment
        \( v = \phi(V, -\cdots, -) \) with \( W \in D \)
        Remove unfilled position in the π-assignment
        Replace the conflicting use \( V \) in \( n \) with \( v \)
            except the use in the π-assignment
        endfor
    endif
endfor

procedure PlacePhiAndRename(newjoin, newdefs, join)
else if \( N_{type}(n) = \text{Header} \) then
    CompletePhi(n, def)
    Insert the dummy node for newjoin in front of \( n \)
    \( \text{ProcessDummy} \) (the dummy node for newjoin, join)
endproc

procedure ProcessDummy(dummy, join)
for each φ-assignment for a variable \( V \) in dummy do
    InsertPhi(join, V)
endfor

procedure NewVar(V)
return a new indexed variable \( V_i \) for a variable \( V \)
endproc

procedure InsertPhi(join, V)
if there is no dummy node for join then
    allocate a dummy node for join
endif
if there is no φ-function for \( V \) then
    \( V_j := \text{NewVar}(V) \)
    make a new φ-function, \( V_j = \phi(v, \cdots, -) \),
    which has the same number of arguments as
    the number of incoming control flow edges into
    join and insert it into the dummy node for join
endif

procedure Search(V, def)
\( R := W_j \) such that \((V, W_j) \in \text{def} \)
return \( R \)
endproc

procedure FillInPhi(n, join, def)
index := the identification number of
    the control flow edge \((n, join)\)
for each \( \phi_V \)-function in the dummy node of \( join \) do
    R := Search(V, def)
    insert R into the index'th position in \( \phi_V \)-function
endfor
endproc

procedure CompletePhi(join, def)
for each \( \phi_V \)-function in the dummy node of \( join \) do
    R := Search(V, def)
    insert R into the vacant position in \( \phi_V \)-function
endfor
endproc

procedure GetNewDefFromPhi(join)
defs := \( \emptyset \)
for each \( V_i = \phi(V, \cdots, W) \) in the dummy node of \( join \) do
    def := defs \cup \{(V, V_i)\}
endfor
return def
endproc
Appendix D. Parallel Sparse Conditional Constant Propagation Algorithm

Input: A parallel control flow graph 
\[ G = (N, E_c, \cup E_\phi \cup E_f, N_{type}, E_{type}) \] in PSSA form
Output: A parallel control flow graph \( G' \) in PSSA form

on which constant propagation is done.

Add PSSA edges to \( G \)
Initialize a queue \( CTFE \) with

\[ \{(x, y) \mid N_{type}(x) = Entr, y \land (x, y) \in E_{ct} \} \]

Initialize a queue \( PSSAE \) with \( \emptyset \)

for each \( e \in E_{ct} \) do
  \( e_{& ExecutionFlag} := False \)
endfor

for each \( n \in N \) do
  \( n_{& ExecutionFlag} := False \)
  for each definition \( v \) in \( n \) do
    \( n_{& LatticeValue(v)} := \top \)
  endfor
endfor
repeat
  if \( CTFE \) is not empty then
    \( e := \) the first entry \((x, y)\) in \( CTFE \)
    if \( e_{& ExecutionFlag} = False \) then
      \( e_{& ExecutionFlag} := True \)
      if \( y_{& ExecutionFlag} = False \) then
        \( y_{& ExecutionFlag} := True \)
        for each expression \( exp \) in \( y \)
          VisitExpression \((y, exp)\)
        endfor
      endif
      if \( y \) has only one outgoing control flow edge \( e \) then
        Put \( e \) into \( CTFE \)
      else if \( N_{type}(y) = \) Cobiend then
        Put all the outgoing control flow edges from \( y \) into \( CTFE \)
      endif
    endif
  else if \( PSSAE \) is not empty then
    \( e := \) the first entry \((x, y)\) in \( PSSAE \)
    \( n := \) the node where \( y \) resides.
    if \( y \) is in an expression \( \text{exp} \) then
      if \( n_{& ExecutionFlag} = True \) then
        VisitExpression \((n, \text{exp})\)
      endif
    endif
  endif
until \((CTFE = \emptyset \land PSSAE = \emptyset)\)

for each node \( n \in N \) do
  if \( n_{& ExecutionFlag} = False \) then
    Remove \( n \) and those edges related to \( n \) from the PCFG.
  endif
endfor

for each node \( n \in N \) do
  for each definition in the form of \( v = \text{exp} \) do
    if \( n_{& LatticeValue(v)} \) is a constant \( C \) then
      Replace the definition with \( v = C \)
      Replace each use of \( v \) with \( C \) by following PSSA edges.
    endif
  endfor
endfor

procedure VisitExpression \((n, \text{exp})\)
  \( \text{val} := \text{Evaluate}(n, \text{exp}) \)
  if \( \text{exp} \) is in the righthand side of an assignment \( (v = \text{exp}) \) then
    if \( \text{val} \neq n_{& LatticeValue(v)} \)
      \( n_{& LatticeValue(v)} := \text{val} \)
      Put all PSSA edges which has \( v \)
      as a source into \( PSSAE \).
    endif
  else if \( \text{exp} \) controls a branch then
    if \( \text{val} = \bot \) then
      Put all outgoing control flow edges
      of \( n \) into \( CTFE \).
    else if \( \text{val} = True \) then
      Put the outgoing control flow edge \( e \)
      into \( CTFE \) where \( E_{type}(e) = T \).
    else if \( \text{val} = False \) then
      Put the outgoing control flow edge \( e \)
      into \( CTFE \) where \( E_{type}(e) = F \).
  endif
endprocedure

procedure Evaluate \((n, \text{exp})\)
  if \( \text{exp} \) is in the form of \( \phi(v_0, v_1, \ldots, v_k) \) then
    \( n_{& LatticeValue(d_i)} := \top \)
    for each \( e \) where \( \phi \) is a PSSA edge for
    \( \text{the argument} v_i \) which corresponds to
      \( \text{the control flow edge} e_i \) and let \( m_i \)
      \( \text{be the node where} d_i \) resides.
    result := \( n_{& LatticeValue(d_i)} \)
    where \( e_i_{& ExecutionFlag} = True \)
  else if \( \text{exp} \) is in the form of \( \pi(v_0, v_1, \ldots, v_k) \) then
    \( n_{& LatticeValue(d_i)} := \top \)
    for each \( e \) where \( \pi \) is a PSSA edge for
      \( \text{the argument} v_i \) and let \( m_i \)
      \( \text{be the node where} d_i \) resides.
    result := \( n_{& LatticeValue(d_i)} \)
    where \( m_i_{& ExecutionFlag} = True \)
  else
    Apply expression evaluation rules to all of
    the operands of \( \text{exp} \) using the \( \text{LatticeValue} \)
    from the nodes where the operands are defined.
  endif
return result
endprocedure

15