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 a 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 optimiztion 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 a 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.

```
flag = 0
cobegin
  b = 4
  while flag = 0 do
  endwhile
  a = b
  print a
//
  b = 3
  flag = 1
coend

          (A)
```

```
flag = 0
cobegin
  b = 4
  while flag = 0 do
  endwhile
  a = 4
  print 4
//
  b = 3
  flag = 1
coend

          (B)
```

Figure 1. Incorrect constant propagation in the presence of busy-waiting. (A):Before constant propagation. (B):After constant propagation.

To overcome these inabilities, 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.

```
a = 4
b = 5
if P then
   a = a + b
else
   a = a - b
endif
c = a + 1
```

(A)

```
a0 = 4
b0 = 5
if P then
   a1 = a0 + b0
else
   a2 = a0 - b0
endif
a3=ϕ(a1,a2)
c1 = a3 + 1
```

(B)

Figure 2. An example of SSA transformation. (A): Original code. (B): SSA form.

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 a 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 algoritm 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 ϕ-functions that distinguish values of variables coming from different incoming control flow edges in the control flow graph[1] of the program. A ϕ-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 ϕ-assignment is placed. We place ϕ-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.

```
a = 0
b = 4
c = 3
d = 10
e = 0
f = 3
g = 0
k = 0
if (c <= 2) then
  e = a + d
  g = e + d
else
  cobegin
    if (c > 3) then
      f = a + 7
      Set(S)
    else
      c = 4 * c
      Set(S)
      a = b + c
    endif
    a = a - 3
  //
    a = f + a
    d = 3 * d
    Wait(S)
    do k = 1, 10
      a = a + k
    enddo
    e = a + d
  coend
  g = a + 8
endif
print g
```

Figure 3. An example code

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/enddo`, `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:

```
cobegin
    stmt-list
//
...
//
    stmt-list
coend
```

Where *stmt-list* is a block which consists of statements. The blocks share the same address space and execute concurrently with each other. When the block does not contain another `cobegin`/`coend`, it is called a thread.

An *event* is an integer variable with two states, *posted* and *cleared*. Only three operations, `Set`, `Wait`, and `Clear` are allowed on an event variable `S`. The `Set(S)` statement sets the event variable `S` to *posted*, if not already in *posted* state, the `Wait(S)` suspends the calling thread until the event variable `S` is set to *posted*. The `Clear(S)` resets the state of an event variable `S` to *cleared*, enabling it to be reused. However, we will not consider the case, in which `Clear` statements are used, in this paper. In addtion, any `Set` or `Wait` may not appear inside a loop.

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* $G = (N, E, Ntype, Etype)$ *such that,*

- N *is the set of nodes in* G.
- $E = E_{ct} \cup E_{sy} \cup E_{cf}$ *where,*
  - $E_{ct} = \{(m,n) \mid m,n \in N \wedge Etype(m,n) \in \{T,F,U\}\}$ *is the set of control flow edges.*
  - $E_{sy} = \{(m,n) \mid m,n \in N \wedge Etype(m,n) \in \{S\}\}$ *is the set of synchronization edges which show the order enforced by synchornization operations.*
  - $E_{cf} = \{(m,n) \mid m,n \in N \wedge Etype(m,n) \in \{DU,DD,UD\}\}$ *is the set of conflict edges.*
- $Ntype$ *is a function which tells the class of nodes, such that* $Ntype : N \mapsto T$ *where,* $T = \{Entry,\ Exit,\ Cobegin,\ Coend,\ Condition,\ Header,\ Compute,\ ThreadEntry,\ ThreadExit\}$.
- $Etype$ *is a function such that,* $Etype : N \times N \mapsto \{T,F,U,S,DU,DD,UD\}$.

The *Entry* and *Exit* nodes are special nodes which have no predecessors and no successors in PCGF respectively. Several threads are created at *Cobegin* node and the number of threads are the outgoing edges of *Cobegin* node. Threads are merged at *Coend* node. *ThreadEntry* and *ThreadExit* nodes are special nodes that mark the beginning and the end of a thread in between *Cobegin* and *Coend* nodes.

Condition node is the same as the branch node in the Control Flow Graph[1] in sequential setting but if

Table 1. The labeling rule for conflict edges.

	def&use	def	use
def&use	DU	DD	DU
def	DU	DD	DU
use	UD	UD	

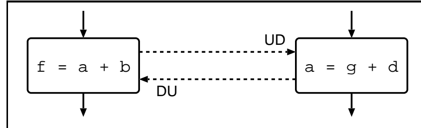


Figure 4. An example of labeling conflict edges.

it is a loop header[1] 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 *Etype* 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/enddo` 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:

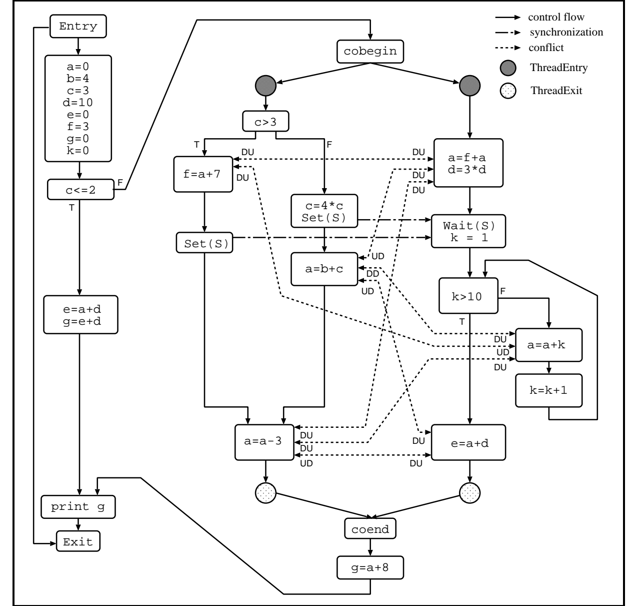
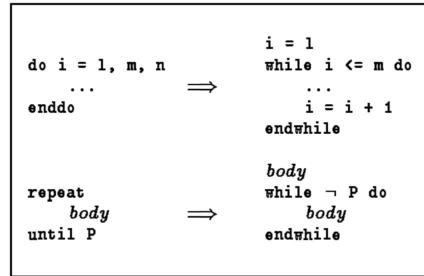


Figure 5. The Parallel Control Flow Graph of the example code in Figure 3.



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 ϕ-function in the static single assignment (SSA) form [4] in sequential setting, parallel static single assignment (PSSA) form has π-functions in addition to ϕ-functions.

The meaning of ϕ-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 ϕ-function to cover the case of *Coend* node at which threads are merged.

Definition 5 *A ϕ-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 π-function summarizes the interleaving information of conflicting variables at the place where the π-function is placed. Similar to ϕ-functions, the π-function distinguishes values of variables coming from an incoming control flow edge (the number of incoming control flow edges into the node where π-functions are placed is always 1 by the construction of PCFG) and distinct conflict edges labeled DU.

Definition 6
A π-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 a explicitly parallel program by using π-functions. The value of all the conflicting variables are well defined by the π-function at the point where the π-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 ϕ-functions (ϕ-assignments). [Section 5.2.]
3. Place π-functions (π-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 Emrath, 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/endwhile` 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 **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 $Loop(n)$ be a function that returns the set of nodes in the loop whose header is n. For a given PCFG $G = (N,\ E_{ct} \cup E_{sy} \cup E_{cf},\ Ntype,\ Etype)$, We construct a subgraph $G' = (N',\ E',\ Ntype,\ Etype)$ of G where:

$$\begin{aligned} N' &= N - \{n \mid m, n \in N \wedge n \in Loop(m) \\ &\qquad \wedge\ Ntype(m) = Header \wedge m \neq n\} \\ E' &= (E_{ct} \cup E_{sy}) \\ &\qquad - \{(m, n) \mid m, n \in N \wedge (m \notin N' \vee n \notin N')\} \end{aligned}$$

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 $Prec(n)$ for a given node n in a data flow analysis framework where,

$$Prec(n) = \{m \mid m \in N',\ m \text{ is guaranteed to precede } n \text{ in execution}\}$$

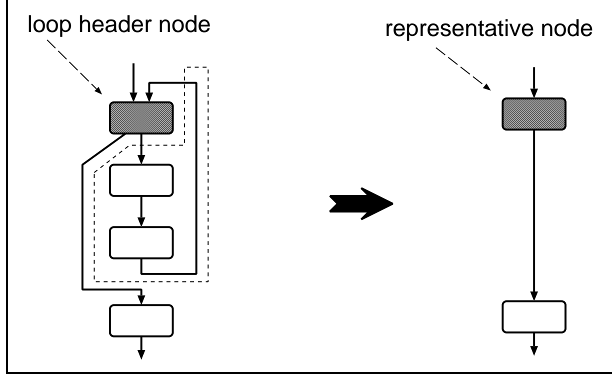


Figure 6. To get the partial ordering of the nodes in a loop body, we calculate the precedence of the loop header node and then give this ordering to each node in the loop body. To apply the data flow analysis framework, we remove the nodes and edges enclosed by the dashed line.

We implement $Prec(n)$, $Prec_{cf}(n)$, and $Prec_{sy}(n)$ for a given node n with bit vectors. The set $Prec(m)$ for node m such that $m \in N - N'$ is given by the following equation. Assume that a function $Header(m)$ returns the node, which is the header node of the outermost loop enclosing m.

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

The data flow equations for the problem are as follows.

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

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

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

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

5.2. Placing ϕ-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 ϕ-functions in the given PCFG. We consider only control flow edges and ignore conflict edges and synchronization edges in the PCFG. We place ϕ-functions in the dummy node of the node in which ϕ-functions should be placed. After placing all the ϕ-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 ϕ-assignments. One is the algorithm for placing ϕ-assignments and renaming variables by Cytron et al.[4]. The other is the SSA transformation for structured programs by Brandis and Mössenböck [2]. Our algorithm for placing ϕ-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 priory*, when we meet one of those branch nodes, we do depth first traversal of its successor nodes to generate ϕ-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 Mössenböck [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 ϕ-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 ϕ-assignments is almost the same as for the *Condition* block. However, if we treat a *Cobegin* block as a *Condition* block, lots of superfluous ϕ-assignments are generated. In order to prevent this, we check each argument in ϕ-function after filling in all the vacant arguments in it. If the ϕ-assignment satisfies the following condition, we discard the ϕ-assignment, since it is superfluous.

Theorem 1 *When the join node is a Coend node, a ϕ-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.*

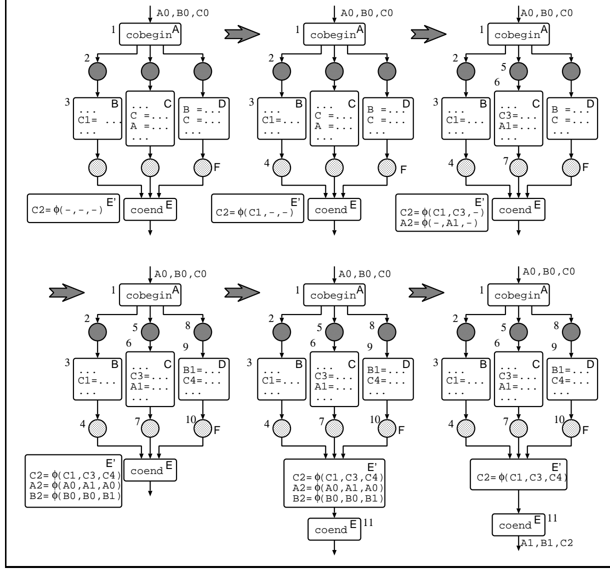


Figure 7. Placing ϕ-assignments in the join node of *Cobegin* node.

Proof:

1. It is obvious if all the parameters are the same.
2. If only one parameter p is different from all the others and there are more than two parameters, only p is defined within cobegin/coend. Since, the original variable that corresponds to p is a shared variable among threads, the definition p will be used after control merges at the *Coend* node. Therefore the ϕ-assignment is superfluous and can be removed.
3. If it has exactly two different parameters p and q, and one of its parameters q is defined before the corresponding *Cobegin* node, then only p is defined within cobegin/coend. By the similar argument to the case of more than two parameters, the ϕ-assignment is superfluous and can be removed. ∎

Figure 7 shows the transformation of the *Cobegin* node. *Cobegin* node is generated by cobegin statement in the source program and we know its join node coend at parse time. Thus, when we do parsing, we create a pointer from the *Cobegin* node to its join node. Whenever we need its join node in the transformation process, we follow this pointer.

1. The incoming values of variables into node A are A0, B0, and C0. We call them the current values of A, B, and C respectively. We do nothing at node A.
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 *Cobegin* 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.

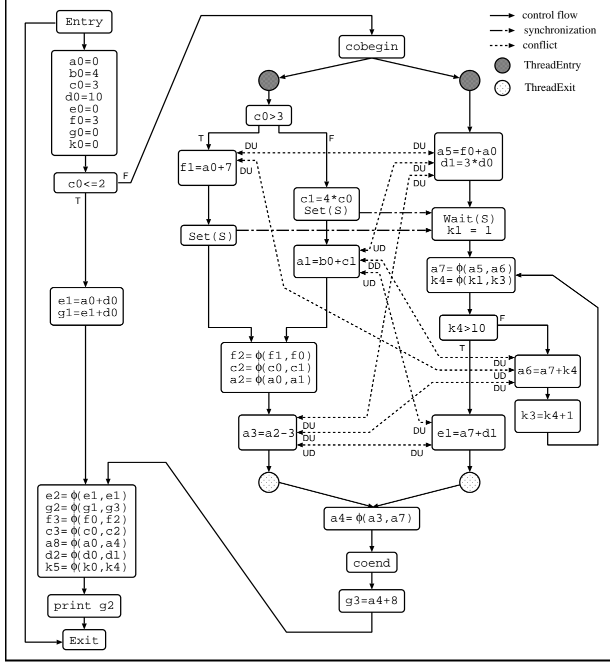


Figure 8. The result PCFG of the example code in Figure 5 after placing ϕ-assignments.

5.3. Placing π-assignments

In this section, we explain how to place π-assignments in the given PCFG after placing ϕ-functions. In sequential setting, we place the ϕ-funcion 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 satisfies 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.*

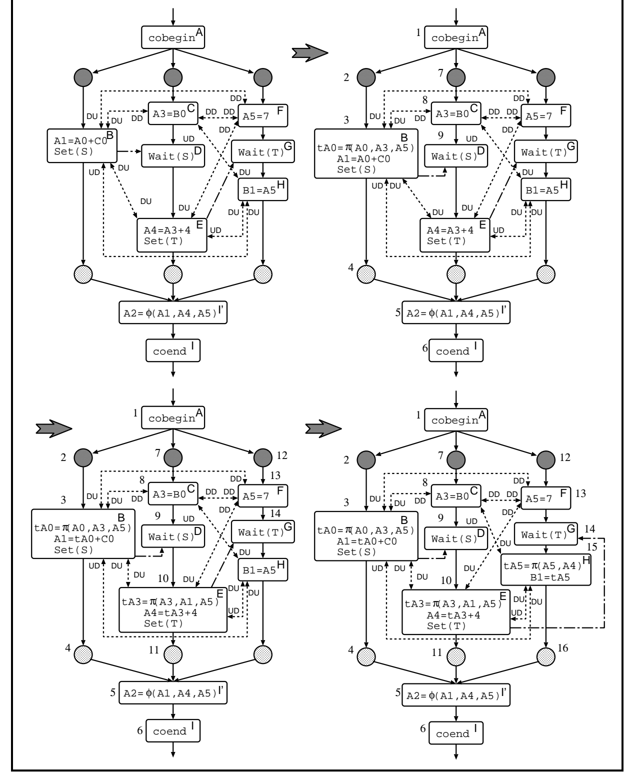


Figure 9. Placing π-assignments in a given PCFG after placing ϕ-assignments

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 π-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 nodes. We just check all the DU edges in PCFG to locate a parallel join node in order to place π-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 π-assignments.

When we place a π-assignment for a variable in the parallel join node of the variable, we generate new tem-

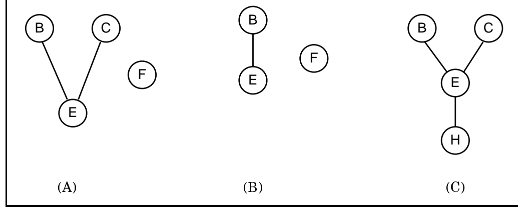


Figure 10. Partial ordering among nodes in Figure 9

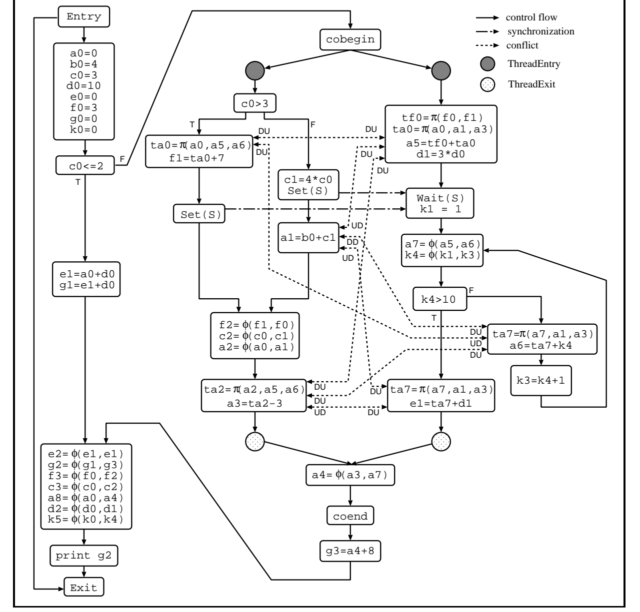


Figure 11. The result PCFG of the example code in Figure 5 after PSSA transformation.

porary variable which is annotated with the original variable name and assign the π-function to this variable. Then, replace the use of the original variable with this temporary variable. Note that there is only one use of the variable in the parallel join node by the definition of parallel basic block.

1. Node B has three incoming *DU* edges from node C, F, and E. So we place a π-assignment in this node. The use of variable A0 is replaced by the temporary tA0. We insert the π-assignment tA0=π(-,-,-,-) 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 π-function with the reaching definition through the incoming control flow edge into node A. It is A0. We get tA0=π(A0,-,-,-). 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 π-function in node B. We remove a vacant argument in the π-function. We get tA0=π(A0,-,-). 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 π-function in node B. We fill in rest of the arguments with A3, A5. The order of argument is not important except the first one. Finally, the π-assignment become tA0=π(A0,A3,A5).

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 π-assignment tA3=π(A3,-,-). Since we do not have the ordering among node E and F and node B precedes E, We fill in the π-function with A1, A5. We get tA3=π(A3,A1,A5).

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 π-assignment tA5=π(A5,-,-,-). 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 tA5=π(A5,-,-,-) and get tA5=π(A5,-). We fill in the vacant argument with the value A4 from node E and get tA5=π(A5,A4).

The final PSSA form of the code in Figure 5 is given in Figure 11. The algorithm for placing π-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 Zadeck[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

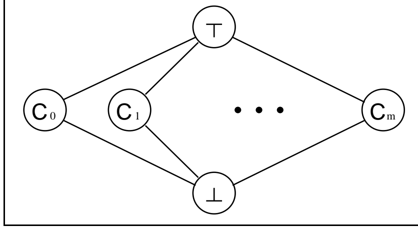


Figure 12. The lattice for constant propagation.

Table 2. The meet operation $\sqcap$ for the lattice. $(i \neq j)$

$\sqcap$	$\top$	C_i	C_j	$\bot$
$\top$	$\top$	C_i	C_j	$\bot$
C_i	C_i	C_i	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

means that nothing may yet be asserted about the variable in question. $\bot$ 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 $\bot$ 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

Table 3. The expression evaluation rules for arithmetic operations **op** such as +, −, ∗, / and logical operations such as $\wedge$, $\vee$.

op	C_j	$\bot$
C_i	C_k	$\bot$
$\bot$	$\bot$	$\bot$

$\wedge$	t	f	$\bot$
t	t	f	$\bot$
f	f	f	f
$\bot$	$\bot$	f	$\bot$

$\vee$	t	f	$\bot$
t	t	t	t
f	t	f	$\bot$
$\bot$	t	$\bot$	$\bot$

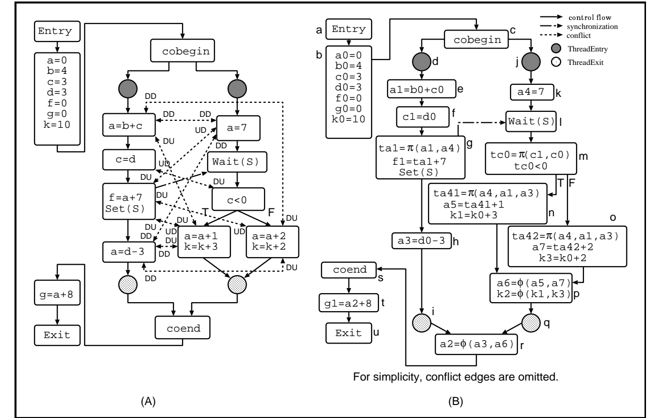


Figure 13. An example PCFG for constant propagation.

orignal PCFG, initially maps to $\bot$. 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 $Entry$ node and $PSSAE$ is empty.

Each control flow edge has a flag $ExecutionFlag$ for the possibility of execution of the edge. If an edge is executable, its $ExecutionFlag$ is **True**, otherwise it is **False**. Each node has a flag $ExecutionFlag$. 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 π-function and the definition of PSSA edges. When we evaluate a π-function, we apply meet operation $\sqcap$ of the lattice to all of the arguments in the π-function. For example, assume the arguments `a3`, `a6`, and `a8` of a π-assignment `ta3=π(a3,a6,a8)` have the lattice value **3**, **3**, and $\top$ respectively after constant propagation. `a8` has the value $\top$ means that the definition node of `a8` is unexecutable. $ta3$ has the

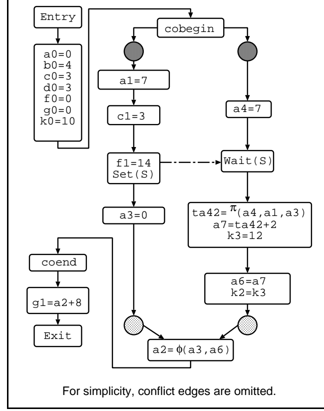


Figure 14. The PCFG in Figure 13 after constant propagation is done.

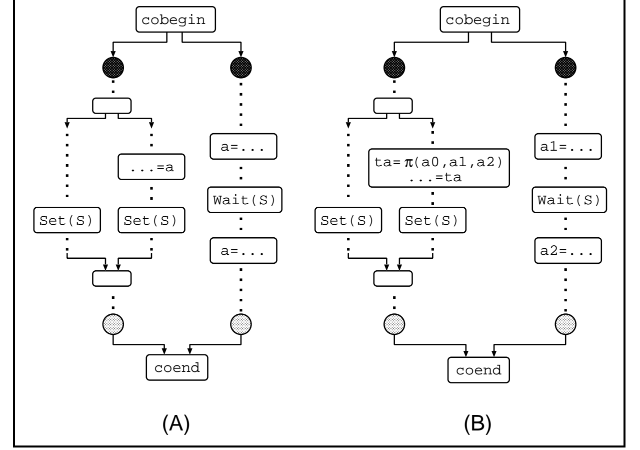


Figure 15. (A):Before PSSA transformation. (B):After PSSA transformation.

constant value 3 by applying ⊓ operation to a3, a6, and a8. Thus, this assignment is replaced by ta3=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 dipicted 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 ancesstor 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.

value of v_0	value of v_1	value of π	exact result
v_0	$\top$	v_0	v_0
C	C	C	C
C	K	$\bot$	C
v_0	$\bot$	$\bot$	v_0

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 considerd 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 explictly 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 ϕ-functions and π-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 a 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 sinchronization. It uses both ϕ- and π-assignments. The π-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

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Mark M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single assignment form for structured languages. *ACM Transactions on Programming Language and Systems*, 16(6):1684–1698, 1994.

[3] David Callahan and Jaspal Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, May 1988.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing*, pages 580–588, 1989.

[6] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, pages 69–77, January 1992.

[7] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminancy in parallel programs. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[8] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II Software*, pages 105–113, August 1990.

[9] Robert H. B. Netzer and Barton P. Miller. On the complexity of event ordering for shared memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing, Vol. II Software*, pages 93–104, August 1990.

[10] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Gloabal value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Laguages*, pages 12–27, January 1988.

[11] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *The Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[12] Harini Srinivasan and Dirk Grunwald. An efficient construction of parallel static single assignment form for structured parallel programs. Technical Report CU-CS-564-91, University of Colorado at Boulder, December 1991.

[13] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 260–272, January 1993.

[14] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[15] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In *Proceeedings of the First International Conference of the Austrian Center for Parallel Computation*, September 1991.

Appendix A. The algorithm for getting guaranteed execution ordering

Input : A parallel control flow graph
G = (N, $E_{ct} \cup E_{sy} \cup E_{cf}$, Ntype, Etype)

Output : Prec(n) for each parallel block in G.

Construct a subgraph $G' = (N', E', Ntype, Etype)$ of G such that,

```
    ;;; Loop(n) is a function that returns the set of
    ;;; nodes in the loop whose header is n.
 N' = N − {n |   m, n ∈ N ∧ n ∈ Loop(m)
                 ∧Ntype(m) = Header ∧ m ≠ n}
 E' =    (E_ct ∪ E_sy)
         −{(m, n) | m, n ∈ N ∧ (m ∉ N' ∨ n ∉ N')}
```

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

```
while Q ≠ ∅ do
    n := the first entry in Q
    Prec_old := Prec(n)
    if Ntype(n) is equal to Coend then
        Prec_cf(n) := ⋃_{(m,n)∈E_cf} Prec(m) ∪ {n}
    else
        Prec_cf(n) := ⋂_{(m,n)∈E_cf} Prec(m) ∪ {n}
    endif
    Prec_sy(n) := ⋂_{(m,n)∈E_sy} Prec(m) ∪ {n}
    Prec(n) := Prec_cf(n) ∪ Prec_sy(n)
    if Prec_old ≠ Prec(n) then
        Put control flow and sychronization
        successors of n into Q
    endif
endwhile

for each node n ∈ N − N' do
    ;;; Header(n) returns the node, which is the
    ;;; header node of the outermost loop eclosing n.
    Prec(n) := Header(n)
endfor
```

Appendix B. The algorithm for placing ϕ-assignments

Input : A parallel control flow graph
G = (N, $E_{ct} \cup E_{sy} \cup E_{cf}$, Ntype, Etype)

Output : The parallel control flow graph G' in which ϕ-functions are placed.

PlacePhiAndRename(Entry node of G, $\emptyset$, *NIL*)

```
procedure PlacePhiAndRename(n, defs, join)
    if Ntype(n) = Exit then return
    n.Visit := True
    newdefs := defs
    for each statement S in n
            in sequential order do
        if S defines a variable V then
            V_NEW := NewVar(V)
            newdefs := (newdefs - {(W, Y) | W = V
            ∧ (W, Y) ∈ newdefs}) ∪ {(V,V_NEW)}
            if join ≠ NIL then
                InsertPhi(join, V)
            endif
            Replace the definition of V in S with V_NEW
        endif
        if S is not Set or Wait statement then
            for each uses of a variable W in S do
                W_NEW := Search(W, newdefs)
                Replace the use of W with W_NEW
            endfor
        endif
    endfor
    if Ntype(n)=Header then
        X := {m | m is a successor of n
                ∧ m is not an exit of the loop
                whose header is n}
    else
        X := {m | (n,m) ∈ E_ct}
    endif
    newjoin := the join node of n
    for each m ∈ X do
        if Ntype(n)=Condition ∨ Ntype(n)=Cobegin then
            if m ≠ newjoin then
                PlacePhiAndRename(m, newdefs, newjoin)
            endif
        else if Ntype(n)=Header then
            if m ≠ n then
                PlacePhiAndRename(m, newdefs, n)
            endif
        else
            if m = join then
                FillInPhi(n, join, newdefs)
            else
                PlacePhiAndRename(m, newdefs, join)
            endif
        endif
    endfor
    if Ntype(n)=Condition ∨ Ntype(n)=Cobegin then
        CompletePhi(newjoin, defs)
        Insert the dummy node for newjoin in front of n
        ProcessDummy(the dummy node for newjoin, join)
        phidefs := GetNewDefsFromPhi(newjoin)
        newdefs := (defs - {(V, V_I) | (∃W.(V,W) ∈ phidefs)
                ∧ (V, V_I) ∈ defs}) ∪ phidefs
```

```
        PlacePhiAndRename(newjoin, newdefs, join)
    else if Ntype(n)=Header then
        CompletePhi(n, defs)
        Insert the dummy node for newjoin in front of n
        ProcessDummy(the dummy node for newjoin, join)
        phidefs := GetNewDefsFromPhi(n)
        p := the exit node of the loop whose header is n
        newdefs := (defs - {(𝒱, 𝒱_I) | (∃𝒲.(𝒱,𝒲) ∈ phidefs)
                ∧ (𝒱, 𝒱_I) ∈ defs}) ∪ phidefs
        PlacePhiAndRename(p, newdefs, join)
    endif
endprocedure

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

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

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_I := NewVar(V)
        make a new φ-function, V_I = φ_V(-,-,...,-),
        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
endprocedure

procedure Search(V, defs)
        R := W_I such that (V, W_I) ∈ defs
    return R
endprocedure

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

procedure CompletePhi(join, defs)
    for each φ_V-function in the dummy node of join do
        R := Search(V, defs)
        insert R into all the vacant position in φ_V-function
    endfor
endprocedure

procedure GetNewDefsFromPhi(join)
    defs := ∅
    for each V_I = φ_V(X, ..., W) in the dummy node of join do
        defs := defs ∪ {(V, V_I)}
    endfor
    return defs
endprocedure
```

Appendix C. The algorithm for placing π-assignments

```
Input :   A transformed parallel control flow graph
          G = (N, E_ct ∪ E_sy ∪ E_cf, Ntype, Etype)
          in which the placement of φ-assignments is done.
Output :  A transformed parallel control flow graph
          in PSSA form.
for each node n ∈ N do
    𝒮 := {x | (x,n) ∈ E ∧ Etype(x,n) = DU}
    if 𝒮 ≠ ∅ then
        𝒟 := {V_x | V is defined in the last
                statement of x ∧ x ∈ 𝒮
        V := the conflicting use in n
        Insert a φ-assignment tV = φ(V,−,···,−)
            at the beginning of n
        ;;; ≻ is the guaranteed execution ordering
        ;;; among nodes in 𝒮 ∪ {n}.
        ;;; If x ≻ y then x is guaranteed to execute
        ;;; before y executes.
        𝒯 := {V_x | x ∈ 𝒮 ∧ n ≻ x ∨ (∃z.z ≻ n ∧ x ≻ z)
                ∧ x,n,z are all distict}
        𝒟 := 𝒟 - 𝒯
        Fill in the vacant positions in the π-assignment
            tV = φ(V,−,···,−) with W ∈ 𝒟
        Remove unfilled position in the π-assignment
        Replace the conflicting use V in n with tV
            except the use in the π-assignment
    endif
endfor
```

Appendix D. Parallel Sparse Conditional Constant Propagation Algorithm

```
Input :   A parallel control flow graph
          G = (N, E_ct ∪ E_sy ∪ E_cf, Ntype, Etype)
          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) | Ntype(x) = Entry ∧ (x,y) ∈ E_ct }
Initialize a queue PSSAE with ∅
for each e ∈ E_ct do
    e.ExecutionFlag := False
endfor
for each n ∈ N do
    n.ExecutionFlag := False
    for each definition v in n do
        n.LatticeValue(v) := ⊤
    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 Ntype(y) = Cobegin 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 exp then
            if n.ExecutionFlag = True then
                VisitExpression(n,exp)
            endif
        endif
    endif
until (CTFE = ∅ ∧ PSSAE = ∅)
for each node n ∈ N do
    if n.ExecutionFlag = False then
        Remove n and those edges related to n from the PCFG.
    endif
endfor
for each node n ∈ N do
    for each definition in the form of v = 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,exp)
    val := Evaluate(n,exp).
    if exp is in the righthand side of
            an assignment (v = exp) then
        if val ≠ n.LatticeValue(v)
            n.LatticeValue(v) := val
            Put all PSSA edges which has v
                as a source into PSSAE.
        endif
    else if exp controls a branch then
        if val = ⊥ then
            Put all outgoing control flow edges
                of n into CTFE.
        else if val = True then
            Put the outgoing control flow edge
                e into CTFE where Etype(e) = T.
        else if val = False then
            Put the outgoing control flow edge
                e into CTFE where Etype(e) = F.
        endif
    endif
endprocedure

procedure Evaluate(n,exp)
    if exp is in the form of
            φ(v_0, v_1, ..., v_k) then
        ;;; Let (d_i, v_i) be the PSSA edge for
        ;;; the argument v_i which corresponds to
        ;;; the control flow edge e_i and let m_i
        ;;; be the node where d_i resides.
        result := ⊓_i m_i.LatticeValue(d_i)
            where e_i.ExecutionFlag = True
    else if exp is in the form of
            π(v_0, v_1, ..., v_k) then
        ;;; Let (d_i, v_i) be the PSSA edge for
        ;;; the argument v_i and let m_i
        ;;; be the node where d_i resides.
        result := ⊓_i m_i.LatticeValue(d_i)
            where m_i.ExecutionFlag = True
    else
        Apply expression evaluation rules to all of
        the operands of exp using the LatticeValue
        from the nodes where the operands are defined.
        result := the evaluation result
    endif
    return result
endprocedure
```