Basic Compiler Algorithms for Parallel Programs *

Jaejin Lee and David A. Padua
Department of Computer Science
University of Illinois
Urbana, IL 61801
{j-lee44,padua}@cs.uiuc.edu

Samuel P. Midkiff
IBM T. J. Watson Research Center
P.O.Box 218
Yorktown Heights, NY 10598
smidkiff@us.ibm.com

Abstract

Traditional compiler techniques developed for sequential programs do not guarantee the correctness (sequential consistency) of compiler transformations when applied to parallel programs. This is because traditional compilers for sequential programs do not account for the updates to a shared variable by different threads. We present a concurrent static single assignment (CSSA) form for parallel programs containing `cobegin/coend` and `parallel do` constructs and `post/wait` synchronization primitives. Based on the CSSA form, we present copy propagation and dead code elimination techniques. Also, a global value numbering technique that detects equivalent variables in parallel programs is presented. By using global value numbering and the CSSA form, we extend classical common subexpression elimination, redundant load/store elimination, and loop invariant detection to parallel programs without violating sequential consistency. These optimization techniques are the most commonly used techniques for sequential programs. By extending these techniques to parallel programs, we can guarantee the correctness of the optimized program and maintain single processor performance in a multiprocessor environment.

*This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract ASC-96-12099; and a Partnership Award from IBM. Jaejin Lee's work is partially supported by an IBM cooperative fellowship. This work is not necessarily representative of the positions or policies of the Army, Government or IBM Corp.

1 Introduction

Under the shared memory parallel programming model, all threads in a job can access a global address space. Communication between threads is via reads and writes of shared variables rather than explicit communication operations. Processors may access a shared variable concurrently without any fixed ordering of accesses, which leads to *data races* [5, 9] and non-deterministic behavior.

Data races and synchronization make it impossible to apply classical compiler optimization and analysis techniques directly to parallel programs because the classical methods do not account for updates to shared variables in threads other than the one being analyzed. Classical optimizations may change the meaning of programs when they are applied to shared memory parallel programs [23].

```
      Program vseq
      integer a, b, c, d, e
      volatile a, b
      a = 0
      b = 0
      c = 0
      d = 0
      e = 0
C$DOACROSS SHARE(a,b,c,d,e), LASTLOCAL(i)
      do i = 1, 2
        if (i.eq.1) then
          a = 1
          c = b
        else
          d = a
          b = 1
          e = a
        end if
      end do
      print *,'a,b,c,d,e = ',a,b,c,d,e
      stop
      end

  a,b,c,d,e = 1 1 0 0 0
```

(A)

```
$33:lw    $12, 20($sp)
    lw    $13, -4($12)
    sw    $13, -16($12)
    li    $14, 1
    sw    $14, -8($12)
    sw    $13, -20($12)
$34:
```

(B)

```
$33:R13 ← a
    d   ← R13
    R14 ← 1
    b   ← R14
    e   ← R13
$34:
```

(C)

Figure 1. (A) A sample code. (B) An assembly code segment generated by the compiler for the boxed statements in (A) without any optimization switch on. (C) An intermediate code segment before register allocation that corresponds to the code in (B).

Consider the program in Figure 1(A) for an example of incorrectly applied classical optimization techniques. The

MIPSpro Fortran 77 Compiler was used for compilation on an SGI challenge machine with 4 R4400 processors. During compilation, no optimization was enabled. The output 'a,b,c,d,e = 1 1 0 0 0' of the program on 2 processors in Figure 1(A) is not what the programmer would expect from an intuitive interpretation of the original program. By "the output the programmer expects," we mean the output attained by an execution that follows sequential consistency, which is the most intuitive and natural memory consistency model for programmers. The intuitive interpretation would lead us to assume that if `e` is equal to 0, then `c` should be 1. The reason for this assumption is that if `e` is equal to 0, then `e=a` should have executed before `a=1`, which means `b=1` should have executed before `c=b`. We define sequential consistency [18] as follows.

Definition 1.1 (Sequential Consistency)
A result of an execution of a parallel program P is sequentially consistent if it is the same as the result of an execution where all operations were executed in some sequential order, and the operations of each individual thread occur in this sequence in the order specified by P.

Sequential consistency is what most programmers assume when they program shared memory multiprocessors, even if they do not know exactly what it is. An article supporting this argument at the hardware level is found in [12], where it is argued that future systems ought to implement sequential consistency as their hardware memory consistency model because the performance boost of relaxed memory consistency models does not compensate for the burden it places on system software programmers. We use sequential consistency as the correctness criterion for the execution of parallel shared memory programs.

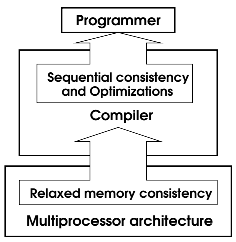


Figure 2. The compiler optimizations discussed in this paper together with fence generation algorithms present a sequentially consistent view to the programmer.

In this paper, we present analysis and optimization techniques which guarantee sequential consistency. We assume either that sequential consistency is enforced by the target machine or, if the target machine has a relaxed memory consistency model, that the back-end compiler inserts fences to enforce sequential consistency. One possible approach to enforce sequential consistency is to insert a fence instruction for each *delay* edge found by Shasha and Snir's delay set analysis [28]. A better method is discussed in [19] and is beyond the scope of this paper.

Our techniques make it possible to correctly handle the situations like the one in the example of Figure 1. Note that the problem arises from redundant load elimination (i.e., the reuse of symbolic register *R13* in the intermediate code in Figure 1(C) violates sequential consistency). The origin of the problem is that current commercial compilers were not implemented with sequential consistency or optimizations on shared variables in mind.

Under the assumption of sequentially consistent execution of the original program, the defining characteristic of a correct compiler transformation in this paper is *subset correctness*.

Definition 1.2 (Subset Correctness)
A transformation is correct if the set of possible observable behaviors of a transformed program is a subset of possible observable behaviors of the original program.

An *observable behavior* of a program is a functional behavior of the program (i.e., the program is treated as a black box and the programmer is interested in only its input and output). If a transformed program is sequentially consistent to the original program, it preserves subset correctness. But, not *vice versa*.

The remainder of the paper is organized as follows. In the next section, we describe Shasha and Snir's delay set analysis. In Section 3, we present an extension of the static single assignment (SSA) form that makes it possible to handle nested `parallel do` and `cobegin/coend` constructs and the `post/wait` synchronization. We describe algorithms for copy propagation and dead code elimination in Section 4 and global value numbering in Section 5. We present applications of the global value numbering technique, such as common subexpression elimination, redundant load/store elimination, and hoistable access detection (loop invariant detection) in Section 6. Section 7 presents related work, and Section 8 concludes our discussion.

2 Delay Set Analysis

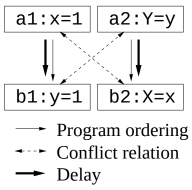
Delays are the minimal ordering between shared variable accesses needed to guarantee sequential consistency. This minimal ordering is a subset of the ordering specified by the source program. It was shown by Shasha and Snir [28] that enforcing delays is sufficient to guarantee sequential consistency. Thus, instructions not related by the minimal ordering can be exchanged without affecting sequential consistency.

Consider a program control flow graph (CFG). Let $\mathbf{P}$ be the ordering specified by the source program on the set of variable accesses. $\mathbf{P}$ is the transitive closure of the ordering represented by the control flow edges in the CFG. Let $\mathbf{C}$ be a *conflict* relation on the variable accesses. Two memory accesses *conflict* if they are in different threads that could execute concurrently, reference the same memory location, and at least one is a write. Each element of $\mathbf{C}$ is represented as a bidirectional edge in the CFG. A *critical cycle* is a cycle of $\mathbf{P} \cup \mathbf{C}$ that has no chords in $\mathbf{P}$[1]. An edge $(u,v) \in \mathbf{P}$ is

[1]For two nonadjacent nodes u and v in the cycle, a chord is a

```
X=0
Y=0
x=0
y=0
cobegin
   a1:x=1
   b1:y=1
||
   a2:Y=y
   b2:X=x
coend
```

(A)

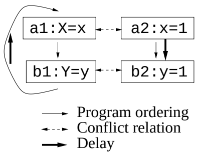


(B)

Figure 3. (A) A sample code. (B) The program edges (`a1`, `b1`) and (`a2`, `b2`) should be enforced to ensure sequentially consistent execution.

```
X=0
Y=0
x=0
y=0
cobegin
  do i=1,2
    a1:X=x
    b1:Y=y
  end do
||
  a2:x=1
  b2:y=1
coend
```

(A)



(B)

Figure 4. (A) A sample code with a loop. (B) The program edges (`b1`, `a1`) and (`a2`, `b2`) should be enforced to ensure sequentially consistent execution.

a critical pair if it occurs in a critical cycle. The critical pair relation is a minimal ordering relation that enforces sequential consistency in any execution of the program.

For example, the code shown in Figure 3(A)[2] gives a result `x`=1, `y`=1, `X`=0, and `Y`=1, which is not sequentially consistent, for the interleaving `b1 a2 b2 a1`. There is a critical cycle (`b1, a2, b2, a1, b1`) in this program and the **P** edges in the cycle are (`a1`, `b1`) and (`a2`, `b2`). Thus, the minimal ordering **D** = {(`a1`, `b1`), (`a2`, `b2`)}) must be enforced as delays to guarantee sequential consistency.

One outcome of the code shown in Figure 4(A) is `x`=1, `y`=1, `X`=0, and `Y`=1. This is not sequentially consistent and is produced by the interleaving `a1 a1 a2 b2 b1 b1`. There is a critical cycle (`a1, a2, b2, b1, a1`) in this program and the **P** edges in the cycle are (`b1`, `a1`) and (`a2`, `b2`). Thus, the minimal ordering **D** = {(`b1`, `a1`), (`a2`, `b2`)}) must be enforced as delays. In other words, `b1` in iteration i must precede `a1` in iteration $i+1$, and `a2` and `b2` cannot be reordered.

3 Intermediate Representations

We briefly discuss the concurrent control flow graph (CCFG) and the concurrent static single assignment (CSSA) form in this section. We also discuss how an early form of these representations [20, 21, 22] has been extended to include both nested `cobegin/coend` and `parallel do` constructs with `post/wait` synchronization.

3.1 Concurrent Control Flow Graph

A *basic block* in a CCFG is a sequence of instructions where any branch is at the end and there is at most one shared variable access. If there is more than one shared variable access in an assignment, we decompose the assignment into several assignments so that each contains at most one shared variable access. The shared variable access is located at the beginning of a basic block for a read access and at the end for a write access. Two basic blocks *conflict* if their variable accesses conflict. Conflict edges join conflicting basic blocks and are bidirectional. Each direction of a conflict edge has a label. The label is δ^t if it is from a definition to a use, δ^a if it is from a use to a definition, and δ^o if it is from a definition to a definition. The labels appear at the head of conflict edges. A synchronization edge labeled σ goes from `post(S)` to a `wait(S)` on the event variable `S`.

We have extended our early CCFG [20, 21, 22] to handle `parallel do` constructs. The definition of CCFG is in Appendix A [20, 21, 22]. Figure 6 shows the CCFG of the code in Figure 5. The region labeled P represents a `parallel do` construct which is represented by a `pdo` node, followed by the representation of the loop body, followed by an `endpdo` node.

```
s = 0
b = 0
f = 0
h = 0
i = 0
do while (s < 100)
  if (s > 70) then
    cobegin
      b = 3
      f = 1
      wait(g)
      t1 = s
      t2 = b
      s = t1 + t2
    ||
      do while (f==0)
      end do
      b = 4
      post(g)
      h = s
    coend
  else
    parallel do i=1,4
      if (i==1) then
        wait(m)
        t3 = b
        s = t3 + 10
      else
        b = i
        post(m)
      end if
    end parallel do
    h = s
  end if
  print h
end do
```

Figure 5. A sample code.

Each `parallel do` construct is represented using two copies of the loop body. The two copies are treated as two different threads and summarize conflict and synchronization information between all the real threads created by the `parallel do` construct so that it is treated as a `cobegin/coend` construct. Loop private variables are renamed. `t3` in the original loop body [Figure 6] is renamed to `t3'` in the copy of the loop body. The two *ThreadEntry* nodes in the `parallel do` construct contain assignments of two different constants, `_c1` and `_c2`, to the loop index variable `i` to indicate they are different threads. Also, the `endpdo` node contains an assignment for the final value of the loop index variable after execution of the `parallel do` construct. For the purposes of delay set analysis, we treat parallel loops as SPMD sections, and use the techniques[3] of Krishnamurthy and Yellick [15, 16, 17]. Therefore, one copy of the loop body contains conflict edges to itself for each loop carried dependence. Each statement in a copy of the loop body corresponds to a dual statement in the other copy. When we apply a compiler optimization to a `parallel do` construct, it must be applied to both dual statements symmetrically. If this is impossible, we do not apply the optimization.

P edge (u, v).

[2] This example is taken from [28].

[3] They used two copies of a control flow graph of an SPMD program to perform delay set analysis.

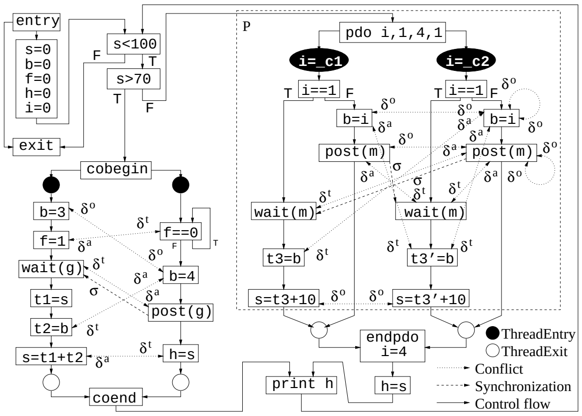


Figure 6. The concurrent control flow graph of the code in Figure 5.

3.2 Concurrent Static Single Assignment Form

The concurrent static single assignment (CSSA) form has these three confluence functions: ϕ (which is the same as the SSA ϕ-function [7]) and two new functions: π and ψ. These are defined as follows.

Definition 3.1 (ϕ-function)
A ϕ-function of the form $\phi(v_1, v_2, \ldots, v_n)$ is placed at a node (except for a `coend` *node or an* `endpdo` *node) where control flow edges join. n is the number of incoming control flow edges at the node. 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 path followed by the program.*

Definition 3.2 (ψ-function)
A ψ-function for a shared variable v has the form $\psi(v_1, v_2, \ldots, v_n)$, where n is the number of threads merging at a `coend` *node or an* `endpdo` *node where the ψ-function is placed. The value of $\psi(v_1, v_2, \ldots, v_n)$ is one of the v_i's and the selection depends on the interleaving of statements in the threads merging at the node.*

Definition 3.3 (π-function)
A π-function of the form $\pi(v_1, v_2, \ldots, v_n)$ for a shared variable v is placed where there is a use of the shared variable with δ^t conflict edges. n is the number of reaching definitions to the use of v through the incoming control flow edges and incoming conflict δ^t edges. The value of $\pi(v_1, v_2, \ldots, v_n)$ is one of the v_i's. The selection depends on the interleaving of statements in the threads computing v_i's.

The definition of ϕ-function in a CSSA form is the same as in the conventional SSA form. `Cobegin/coend` and `parallel do` constructs introduce new confluence points, e.g., `coend` and `endpdo` nodes. Instead of ϕ-functions, ψ-functions, which are similar to those defined by Srinivasan *et al.* [31, 32], are placed at these new confluence points and summarize the interleaving of defining assignments of shared variables in `cobegin/coend` and `parallel do` constructs. However, ϕ-assignments and ψ-assignments are insufficient to represent confluence of values because a definition of a shared variable may reach a use in a different thread. We place a π-function of a shared variable at the point where there is a use of the variable to summarize the confluence of values from different threads.

Like the SSA form for sequential programs, the CSSA form has the following properties:

- all uses of a variable are reached by exactly one (static) assignment to the variable
- for a variable, the definition dominates the uses if they are not arguments of ϕ-, ψ-, or π-functions

Translating a CCFG of a parallel program into its CSSA form is a four-step process:

1. compute the execution ordering of basic blocks guaranteed by explicit synchronization
2. place ϕ-assignments and ψ-assignments
3. rename variables
4. place π-assignments

To compute execution ordering, we start with a CCFG $G = (N, E, Ntype, Etype)$. We compute a set $Prec[n]$ of nodes $m \in N$, which is guaranteed to precede $n \in N$ during execution. An iterative algorithm can be used to compute guaranteed execution ordering. The algorithm computes control and synchronization common ancestors of two nodes [8, 22]. If there is a synchronization that depends on the loop index of a `parallel do` construct (e.g., synchronization using arrays) or that is control dependent on a node that uses the loop index of a `parallel do`, it is impossible to find statically the exact execution ordering with two copies of the loop body in the CCFG because we do not know which `post` will trigger which `wait`. We conservatively assume that there is no synchronization between two nodes that have such synchronization. However, there is the synchronization analysis technique developed by Callahan *et al.* [4] that can be applied to `parallel do` loops. It considers all the instances of statements in the iteration space and gives the ordering between them.

To insert ϕ- and ψ-assignments and rename variables, we can use the algorithm described by Cytron *et al.* [7]. At a `coend` and an `endpdo` nodes, ψ-assignments will be inserted instead of ϕ-assignments.

The ψ-function can be removed (before renaming) if the arguments are ordered. The argument with the lowest precedence kills the definitions of all other arguments and can be used instead of the ψ-function. A similar optimization is used in [31].

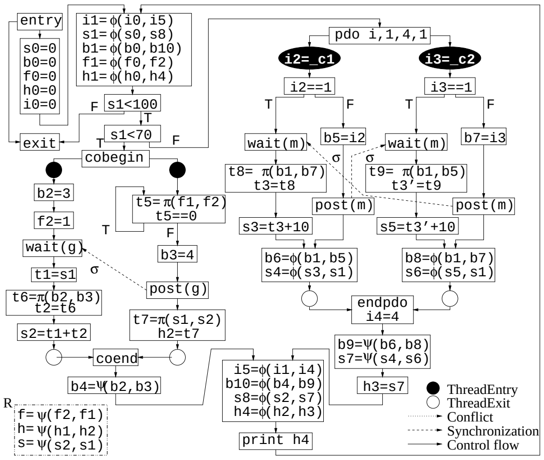


Figure 7. The CSSA form of the CCFG in Figure 6. For simplicity, conflict edges are omitted.

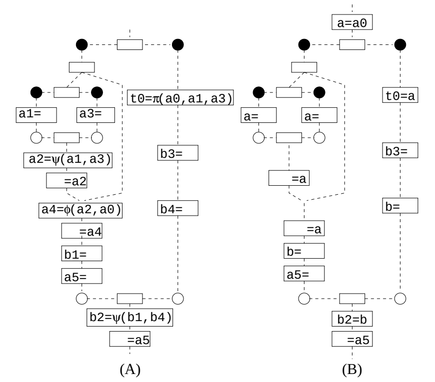


Figure 8. (A) A CSSA form. (B) After converting it to an executable intermediate form.

A reaching definition of a shared variable to its use may not be one of the arguments of the π-function for the use if it is killed by another reaching definition, or if its defining assignment executes after the use. We assign the π-function to a new temporary variable and insert this assignment immediately before the use. The use is replaced by this temporary variable. The algorithm for placing π-assignments is presented in Appendix B.

The CSSA form of the CCFG in Figure 6 is shown in Figure 7. For simplicity, conflict edges are omitted. The ψ-assignments in the lower left corner (in box R) were removed because they are redundant. Note that we do not place a ψ-assignment of the `parallel do` loop index `i` in the `endpdo` node because it is killed by the assignment to `i` in the `endpdo` node.

After optimizations and transformations on a CSSA form, we convert it into some executable form. Basically, we restore the original shared variable name and remove π-, ψ-, and ϕ-

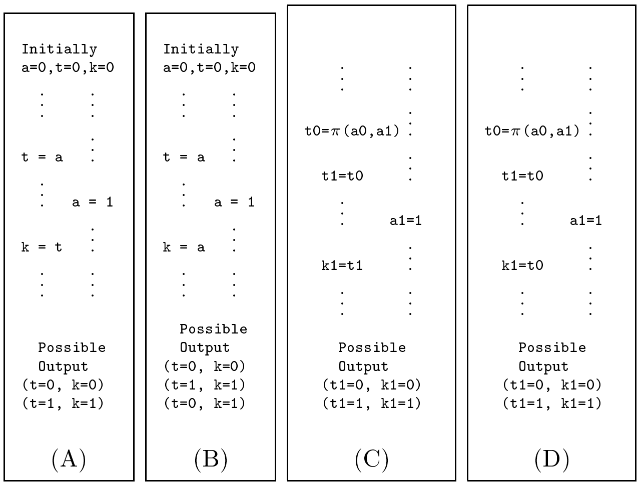


Figure 9. (A) A sample code before copy propagation. (B) The code in (A) after copy propagation. The result t=0, k=1 is not sequentially consistent. (C) The CSSA form of the code in (A). (D) The CSSA form in (C) after copy propagation. The result is sequentially consistent and does not violate subset correctness.

functions. An example is presented in Figure 8. Removing ϕ-assignments for a local variable is the same as in the sequential SSA form. Because the ψ-assignment `b2=`ψ`(b1,b4)` is in the exit node of an outermost parallel construct, we replace the ψ-function with the original shared variable `b` and replace the definitions of its arguments `b1` and `b4` with `b`. The π-function in $\mathtt{t0} = \pi(\mathtt{a0}, \mathtt{a1}, \mathtt{a3})$ is replaced with `a`, and then all definitions of its arguments `a1` and `a3` are replaced with an assignment to `a`. Because the ψ-assignment `a2=`ψ`(a1,a3)` is not in the exit node of the outermost parallel construct and the definitions of its arguments are already replaced by `a`, it is just removed. Also, the use `a2` that is not an argument of a ψ-, ϕ- or π-function is replaced by `a`. Since the definition `a2` by ψ-assignment `a2=`ψ`(a1,a3)` reaches `a4=`ϕ`(a2,a0)`, `a4=`ϕ`(a2,a0)` is removed, and the use of `a4` is replaced by `a`. We insert an assignment `a=a0` at the beginning of the outermost parallel construct because the definition of argument `a0` comes from outside the outermost parallel construct. The untouched variables `a0`, `a5`, and `b2` are treated as different variables from `a`.

4 Copy Propagation and Dead Code Elimination

For a copy assignment `t=a`, we can substitute `a` for `t` at every use u_t of `t` if [1, 3]:

1. the assignment `t=a` is the only definition of `t` reaching u_t
2. there are no assignments to `a` between `t=a` and u_t in any program execution

Classical copy propagation may violate sequential consistency in parallel programs. Consider the two threads in

Figure 9(A). After replacing `t` with `a` in `k=t`, we have the code in Figure 9(B) which can produce an incorrect result `t=0,k=1`. This copy propagation violates sequential consistency. The other thread modifies the variable `a` between `t=a` and `k=t`.

In the CSSA form [Figure 9(C)], if the propagated variable is shared, a local copy is always used for copy propagation because of π-assignments. Thus, the case in Figure 9(B) never occurs [Figure 9(D)]. Also, the case where the substituted variable is shared never occurs if we do not allow copy propagation when the substituted variable is in a ϕ-, ψ-, or π-function. Thus, the conventional copy propagation can be directly used if we do not propagate to a ϕ-, ψ-, or π-functions and the parallel program is in CSSA form [Figure 9(C) and (D)].

An instruction I is *dead* if the values it computes are not used in any executable path starting at I [25]. Some optimizations, such as copy propagation, produce dead instructions. In the sequential case, we mark *essential* instructions and follow use-def chains for the variables used in the instructions. If an instruction is in the chain, it cannot be dead coded. The *essential* instructions are those that might affect observable behaviors of the program, such as input/output, function call, and control flow instructions[4] [3, 25]. Dead code elimination for parallel programs is essentially the same as the sequential case but we must consider interactions between threads through shared variables. The CSSA form is useful for dead code elimination in parallel programs because it contains inter-thread use-def chains. The dead code elimination algorithm for parallel programs in CSSA form is the same as the sequential one except definitions from another thread are considered.

5 Global Value Numbering

The equivalence of variables in sequential programs is defined by Alpern *et al.* in [2]. This definition is used as a basis for their SSA based value partitioning algorithm. We use the same definition for equivalence of variables in parallel programs.

Definition 5.1 (Equivalence of Variables)
Two variables are said to be equivalent at a point p if they contain the same values whenever control reaches p during any possible execution of the program.

Because the notion of equivalence contains dynamic instances of variables, Alpern *et al.* defined a congruence relation of nodes in a *value graph* to detect equivalence statically [2]. There is a node in the value graph for each binary operation in the program. When the result of the operation is assigned to a variable, the variable name is used to label the node. Each edge is labeled by the position of the corresponding operand with respect to the operation. The operation label ϕ of a node that corresponds to a ϕ-function is subscripted by the CCFG node where the ϕ-function is placed. An example of the value graph of a CSSA form is in Figure 10.

[4] We cannot dead code infinite loop because it affects observable behavior of the program.

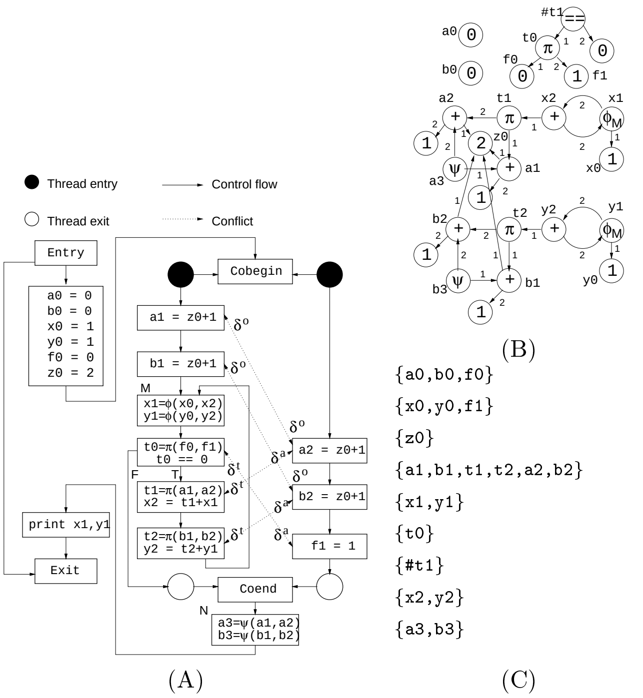


Figure 10. (A) A CSSA form. (B) The value graph. (C) After partitioning. The variables that have the same value numbers are in the same partition. `#t1` is a value number for the expression `t0==0` that does not define any variable.

Following Alpern *et al.*, we define congruence on the value graph of a parallel program as follows:

Definition 5.2 (Congruence)

- *two nodes are congruent if their operation labels are the same and are not π or ψ, and their departing edges with the same labels lead to congruent nodes.*
- *a node with the operation label π or ψ is congruent to its arguments if all arguments are congruent to each other.*

```
⋮
t0=π(a0,a1)
⋮
t1=π(b0,b1)
⋮
```

Figure 11.

The first condition is the same as that of the sequential case. To explain the reason behind the second condition, consider Figure 11. Assume congruence between `a0` and `b0` and between `a1` and `b1`. Also, assume the value of `t0` is `a1` in execution. Because the value of a π-function can be any of of its arguments, the value of `t1` will not necessarily be that of `b1` in the execution. Thus, the first condition cannot be the condition that entails congruence between `t0` and `t1`. We assume a π-

function or a ψ-function is congruent to its arguments when all the arguments are congruent. Intuitively, a π-function has the same value as its arguments if all the arguments have the same value.

Because of the subscripts, two ϕ-functions in different CCFG nodes are never considered congruent. If the corresponding condition expression for a ϕ-function is known, the condition can be incorporated to determine congruence of ϕ-functions in different CCFG nodes [2].

The concurrent global value numbering (CGVN) algorithm is an optimistic partitioning algorithm to find congruent variables. The algorithm is an extension of a global value numbering algorithm for sequential programs that was developed by Cooper and Simpson [6, 29]. The global value numbering is essentially the same as Alpern *et al.*'s value graph partitioning [2].

The concurrent global value numbering algorithm is shown in Figure 12. In concurrent global value numbering, all variables (nodes in the value graph) are initially placed in the same partition and therefore assumed to be congruent. This assumption is refined until a fixed point is reached. The algorithm proceeds in a sequence of steps. In each step, all the nodes in the CCFG are traversed in *reverse post order* of a spanning tree. The process continues until a fixed point is reached. In **step i**, (i) a variable (node), whose operation label is not π or ψ, is in the same partition as another node if they were in the same partition in **step i-1**, and the corresponding destination of the edges leaving the nodes were pairwise in the same partition in **step i-1**, or (ii) if the operation label is π or ψ, and its arguments were all in the same partition in **step i-1**, the node is still in the same partition as its arguments.

Without loss of generality, we assume that all definitions in a CSSA form are in one of the following forms: `x=y`, `x=y op z`, `x=`$\phi(v_1,\ldots,v_m)$, `x=`$\psi(v_1,\ldots,v_m)$, and `x=`$\pi(v_1,\ldots,v_m)$, where `x` is a variable and `y` and `z` are either constants or variables. In this algorithm, a value number is a variable name. The function $LookUp(index_e, v)$ searches a hash table with a variable v and the hash index $index_e$ of its defining expression e. If the value number of the hash index is not defined (i.e., $\top$), it assigns the value number v to $index_e$ and returns v. Otherwise, it returns the value number of $index_e$. The array $ValNum$ maps a variable or a constant to its value number. In the first **for** loop, the algorithm gives the same value number to all the nodes of the value graph. The partitions of variables in Figure 10(B) after the global value numbering is in Figure 10(C).

Similar to the sequential case [2], we can show that the following theorem and corollary hold for the CSSA form by using the notion of *active* variables.

Definition 5.3 (Active variable)
A variable is active at a moment during execution if and only if it satisfies one of the following two conditions:

- *the defining assignment of the variable is not contained in an innermost loop that is currently executing, and has already been executed.*
- *the innermost loop containing the defining assignment of the variable is currently executing and the assignment has already been executed during the current iteration.*

```
A CSSA form G = (N, E, Ntype, Etype) is given.
Initialize a global hash table.
for all variable names v
  ValNum[x] ← ⊤
end for
repeat
  Changed ← false
  for each node m ∈ N in reverse post order
    for each statement S : v = exp in m in sequential order
      Idx ← IdxFormation(exp, S, m)
      Temp ← LookUp(Idx, v)
      if Temp ≠ ValNum[v] then
        ValNum[v] ← Temp
        Changed ← true
    end for
  end for
until Changed = false

Procedure IdxFormation(Exp,StmtID,Node)
  case Exp of
    x : Idx ← "ValNum[x]"
    x op y :Idx ← "ValNum[x] op ValNum[y]"
    φ(v1, ..., vp) :
        Idx ←
          "Node : φ(ValNum[v1], ..., ValNum[vp])"
    π(v1, ..., vp) :
        ValNumSet ← ∅
        for each vi ∈ {v1, ..., vp}
          ValNumSet ← ValNumSet ∪ {ValNum[vi]}
        end for
        if | ValNumSet |= 1 then Idx ← "ValNum[v1]"
        else Idx ← "StmtID : π(ValNum[v1], ..., ValNum[vp])"
    ψ(v1, ..., vp) :
        ValNumSet ← ∅
        for each vi ∈ {v1, ..., vp}
          ValNumSet ← ValNumSet ∪ {ValNum[vi]}
        end for
        if | ValNumSet |= 1 then Idx ← "Node : ValNum[v1]"
        else Idx ← "StmtID : ψ(ValNum[v1], ..., ValNum[vp])"
  end case
  return Idx
end procedure
```

Figure 12. Concurrent Global Value Numbering Algorithm. A variable in between a pair of double quotes stands for the corresponding string converted from the value of the variable.

Theorem 5.4 (Equivalence)
Two active and congruent variables have the same value in any possible execution of a program in CSSA form.

Proof: See Appendix C. □

Corollary 5.5
Two congruent variables are equivalent at a point p if their defining assignments dominate p.

Proof : If the assignment to a variable dominates a point, then the variable must be active at the point. By Theorem 5.4, they are equivalent. □

We will use the result of Corollary 5.5 for our extension of common subexpression elimination in the next section.

CGVN treats π-functions and ψ-functions conservatively in the presence of imprecisely introduced arguments due to imprecise synchronization analysis. Because two π-functions (ψ-functions) are congruent if and only if all their arguments are congruent and because the value of a π-function (ψ-function) will be the value of one of its arguments, an imprecise argument cannot affect the value of those two π-functions (ψ-functions).

6 Applications of Concurrent Global Value Numbering

In this section, we discuss some applications of the concurrent global value numbering.

6.1 Common Subexpression Elimination

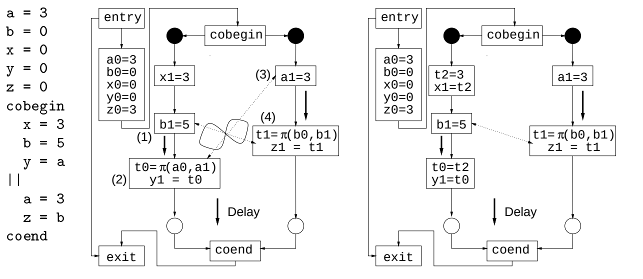


Figure 13. Since there exists a critical cycle ((1),(2),(3),(4),(1)) (i.e., the 8-like shape in (B)), we have to enforce delays on program edges ((1),(2)) and ((3),(4)) to get sequentially consistent execution. After CSE is done in (C), the CSE violates sequential consistency. The result `y1=3`, `b1=5`, `a1=3`, and `z1=0` of the code in (C) is not possible from the code in (B).

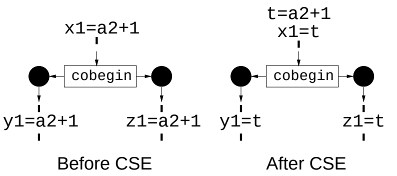


Figure 14. Because `x1=a2+1` dominates `y1=a2+1` and `z1=a2+1`, we can substitute `a2+1` with `t` in `x1=a2+1`, `y1=a2+1` and `z1=a2+1` after inserting an assignment `t=a2+1` immediately before `x1=a2+1`. CSE introduced two new shared variable accesses (i.e., `t`).

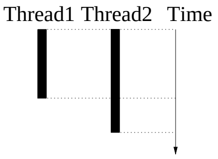


Figure 15. The execution time of a program depends on that of the critical thread Thread2.

Common subexpression elimination (CSE) is one of the most common optimization techniques for sequential programs [25]. However, it cannot be applied directly to parallel programs. Consider the code in Figure 13(A) and its CSSA form in (B). Because `a0` and `a1` are congruent, `t0` is congruent to `a0` and `a1`. Also, `x1` is congruent to `a0` and `a1`. Thus, `x1` and `t0` are congruent. Because `x1=3` dominates `t0=`π`(a0,a1)`, `x1` and `t0` have the same value at the point immediately after `t0=`π`(a0,a1)`. Thus, if we apply classical CSE to `x1` and `t0`, we obtain the program in Figure 13(C). Doing CSE in this case is equivalent to moving the statement `t0=`π`(a1,a2)` to the point immediately after `x1=3` and executing it together with `x1=3`. This makes the result of the program the same as if the order of execution were `t0=`π`(a0,a1)`, `y1=t0`, `a1=3`, `t1=`π`(b0,b1)`, `z1=t1`, `b1=5`. The result from this execution does not correspond to any result from the original program, and therefore applying classical CSE leads to incorrect results. The problem can be identified by applying delay set analysis, which would show that `t0=`π`(a0,a1)` must execute after `b1=5`, and `a1=3` must execute before `t1=`π`(b0,b1)`. That is, the delay edges in Figure 13(B) must be enforced. This order is violated by the execution sequence associated with the CSE translation.

As seen in Figure 14, applying CSE may introduce a shared variable access (`t` in Figure 14). This could be unprofitable depending on the size of the common subexpression, because shared variable accesses take more time than local variable accesses.

Thus, in order to do CSE in parallel programs, we must make use of delay set analysis and we must consider profitability. For two congruent variables v and w, we apply CSE when the following conditions hold (S_v and S_w denote the defining assignments of v and w in the CSSA form):

- (equivalence condition) S_v dominates S_w
- (sequential consistency condition) there does not exist a statement $X \neq S_v$ such that there is a delay edge (S_w, X) or (X, S_w)
- (profitability condition) application of CSE is profitable

In some cases, doing CSE does not reduce or increase the execution time of a program. As seen in Figure 15, doing CSE in *Thread 1* will not reduce or increase the execution time of the program. However, doing CSE in the critical thread *Thread 2* will be beneficial.

After CGVN is done on the CSSA form, CSE is applied to the CSSA form. The CSE algorithm is called concurrent common subexpression elimination (CCSE) [Figure 16].

6.2 Redundant Load/Store Elimination

Register allocation is an important optimization for sequential programs because the number of machine registers are limited and because register operations are much faster than memory operations. The conventional register allocation algorithm would work well on parallel programs were it not for optimizations, such as loop invariant code motion and redundant load elimination (RLE) applied in the middle of or before the register allocation phase. The sequential consistency violation example in Figure 1 comes from RLE. The redundant store elimination (RSE) removes store operations

Let $G = (N, E, Ntype, Etype)$ be the given CCFG in CSSA form. $Done$ is an array that indicates if a variable is already processed or not. $Temp$ is an array that contains a temporary variable used in CSE for each variable if there is one for the variable. $InnermostPC$ is an array that contains the innermost parallel construct for each node in the CSSA form. $First(\mathcal{W})$ returns the first element in the list $\mathcal{W}$ and remove it from $\mathcal{W}$. $Children[n]$ is the set of children of n in the dominator tree. The algorithm traverses the dominator tree in breadth first manner.

```
Do the delay set analysis on the CSSA form.
Do the concurrent global value numbering on the CSSA form.
Construct the dominator tree of the CSSA form.
let r be the root of the dominator tree.
W ← r
for each node n ∈ N in the CSSA form
  let h be the innermost parallel construct that contains n
  InnermostPC[n] ← h
end for
for each variable v in the CSSA form
  Done[v] ← false
  Temp[v] ← ⊤
end for
while W ≠ ∅
  n ← First(W)
  W ← W ∪ Children[n]
  for each variable v defined in n in sequential order
    if Done[v] = false then
      Done[v] ← true
      let P_v be the partition where v belong
      for each r ∈ P_v ∧ Done[r] = false
        let S_v be the defining statement v = e of v
        let S_r be the defining statement r = e_r of r
        if e_r is a φ- or ψ-function then
          Done[r] ← true
        else
          if InnermostPC[n] = InnermostPC[n_r] then
            let n_r be the defining node of r
            if n dominates n_r then
              if there does not exist a statement X such that
                  there is a delay (S_r, X) or (X, S_r) then
                if Temp[v] = ⊤ then
                  Generate a new local temporary variable t and
                    place an assignment t = e
                    immediately before S_v. Substitute t for e in S_v
                  Temp[v] ← t
                Substitute Temp[v] for e_r in S_r
                Done[r] ← true
      end for
    end for
end while
```

Figure 16. Concurrent Common Subexpression Elimination Algorithm.

that write the same value that was previously written to the same memory location, such that memory traffic is reduced. If we treat the variable in the load or store instruction as an expression, the CCSE can be used for both RLE and RSE of parallel programs without violating any correctness criteria. The CSSA form of the example in Figure 1 is shown in Figure 17. If we apply CGVN in this form, `t1` and `t2` are not congruent, nor are `t4` and `t5`. The reason is that `a0` and `a3` are not congruent, nor are `a0` and `a1`. Thus, the application of RLE does not change the program.

6.3 Hoistable Access Detection

Loop invariant code motion is another common optimization technique for sequential programs. Consider the code in Figure 18. The statement `x=a+1` in the `while` loop is a loop invariant in the classical sense [Figure 18(A)]. However, if we move the statement outside the loop [Figure 18(B)], it

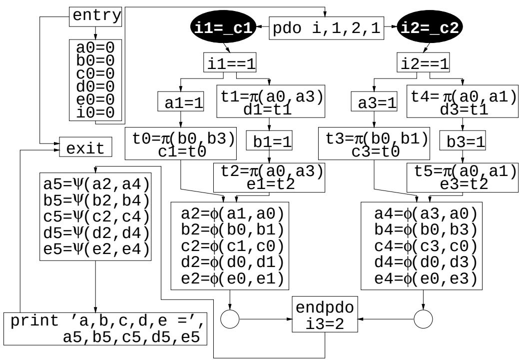


Figure 17. The CSSA form of the example in Figure 1. If we apply concurrent global value numbering in this form, `t1` and `t2` are not congruent and nor are `t4` and `t5`. Thus, we cannot apply redundant load application in this case. For simplicity, conflict edges are omitted.

```
a = 0
x = 0
y = 0
cobegin
  y = a + 1
  do while (x == 2)
    x = a + 1
  end do
||
  a = 1
coend
```

(A)

```
a = 0
x = 0
y = 0
cobegin
  y = a + 1
  x = a + 1
  do while (x == 2)
  end do
||
  a = 1
coend
```

(B)

Figure 18. (A) Before loop invariant code motion. (B) After loop invariant code motion. The `while` loop was made infinite.

might make the loop infinite. This does not happen in the original program (i.e., the loop invariant code motion violates subset correctness). For conventional compilers doing single threaded analysis, it is common to move a load instruction of a variable in a loop outside the loop if the load instruction is loop invariant. However, if we do not consider modification of the variable by another thread, moving load instructions is another source of sequential consistency violations. We define a *hoistable access* for parallel programs that is a counterpart to a loop invariant in sequential programs.

Definition 6.1 (Hoistable access)
We call any computation in a loop that can be moved outside the loop without affecting the meaning of the given program a hoistable access.

Similar to CSE, the notion of hoistable accesses also involves code motion. Thus, we must guard against sequential consistency violations. We mark a π-assignment in a loop *hoistable* if moving it outside the loop does not violate ordering enforced by delays and if all its arguments are congruent. If all the variable accesses in an instruction in a loop are hoistable, then the instruction can be moved outside the loop. The

```
Do delay set analysis on the CSSA form
Do concurrent global value numbering on the CSSA form
for each statement s in the loop
  if s is a π- or ψ-assignment and no delay edges have it
      as an endpoint and all its arguments are congruent then
    Mark s hoistable
  else if all uses of variables in s are all either constant or
        have all their reaching definitions outside the loop then
    Mark s hoistable
end for
repeat
  Changed ← false
  for each statement s in the loop that was not marked hoistable
    if s is a π- or ψ-assignment and no delay edges
        have it as an endpoint and the reaching
        definitions of all the arguments are those
        statements in the loop marked hoistable then
      Mark s hoistable
      Changed ← true
    else if all uses of variables in s are all either
          constant or have all their reaching definitions outside
          the loop or all their reaching definitions are those
          statements in the loop marked hoistable then
      Mark s hoistable
      Changed ← true
  end for
until Changed = false
```

Figure 19. Hoistable Access Detection Algorithm.

hoistable access detection algorithm based on CSSA form is in Figure 19.

7 Related Work

Despite the increasing importance of compiling parallel programs, there has been little work in the field. A static single assignment (SSA) form called *parallel static single assignment* form (PSSA), for parallel programs was proposed by Srinivasan *et al.* [30, 34, 31, 32]. PSSA was developed for the PCF Parallel Fortran [11] `parallel sections` construct with copy-in/copy-out semantics. With *copy-in/copy-out* semantics, each thread receives its own copy of the shared variables at a *fork* point and can modify only its own local copy. Thus, the result of a parallel execution does not depend on a particular interleaving of statements in the parallel program and the PSSA form cannot handle parallel programs with truly shared memory semantics.

Sarkar and Simons proposed a parallel program graph (PPG) [27] that subsumes program dependence graphs (PDGs) [10] and conventional control flow graphs (CFGs) [1]. Sarkar [26] developed reaching definitions analysis on PPGs for deterministic parallel programs.

Krishnamurthy and Yelick developed $O(n^3)$ time delay set analysis algorithm for SPMD programs. They used two copies of a control flow graph of a parallel SPMD program and inserted conflict edges in between these two copies. The graph is used for detecting critical cycles [28]. Also, they present optimization techniques such as communication optimization, message pipelining, and eliminating remote accesses through data reuse for SPMD programs [15, 16, 17].

Midkiff, Padua, and Cytron [24] extended Shasha and Snir's delay set analysis [28] to subscripted variables (i.e., arrays) by using a statement instance level flow graph.

Knoop, Steffen, and Vollmer [14, 13] presented a unidirectional bit vector data flow analysis frame work based on abstract interpretation and showed that it can be used for code motion and partial dead code elimination. Their framework handles the `cobegin/coend` construct but not synchronization mechanisms.

Lee, Midkiff, and Padua proposed the CSSA form based on CCFGs for parallel programs with `cobegin/coend` construct and the `post/wait` synchronization mechanism. They have extended Wegman and Zadeck's [33] sparse conditional constant propagation algorithm for parallel programs [20, 21, 22]. The CCFG and CSSA form has been extended to handle the `parallel do` construct in this paper.

Previous work in this field has concentrated mostly on techniques for analyzing parallel programs. There is little work on optimization techniques. We have presented several optimization techniques for parallel programs that extend common optimization techniques for sequential programs such as common subexpression elimination, copy propagation, dead code elimination, loop invariant code motion, and redundant load/store elimination. Most of the previous work either ignores or avoids using sequential consistency as a correctness criterion. Our analysis and optimization techniques guarantee sequential consistency. In addition, our work is more general than previous work in representing parallel constructs such as `cobegin/coend`, `parallel do`, and `post/wait` synchronization constructs and in dealing with non-deterministic parallel programs. The CSSA form handles parallel programs with truly shared memory semantics that cannot be handled with copy-in/copy-out semantics.

8 Conclusions

We proposed two intermediate representations, the concurrent control flow graph and the concurrent static single assignment form, for shared memory parallel programs with nested `cobegin/coend` and `parallel do` constructs and `post/wait` synchronization mechanisms. Like the conventional SSA form, the CSSA form has the following properties:

- all uses of a variable are reached by exactly one (static) assignment to the variable.
- for a variable, the definition dominates the uses if they are not arguments of ϕ-, ψ-, or π-functions

Sequential consistency and subset correctness are our correctness criteria for execution and compiler transformations. We have presented a conservative analysis technique called concurrent global value numbering that is used to detect equivalent variables in parallel shared memory programs. We have extended some classical scalar optimization techniques to guarantee sequential consistency on parallel programs in CSSA form. These optimization techniques include:

- copy propagation
- dead code elimination
- concurrent common subexpression elimination
- redundant load/store elimination
- hoistable access detection

We believe that the analysis and optimization techniques and the intermediate representations presented in this paper are the first step towards an optimizing compiler for parallel programs.

References

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

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, New York, 1998.

[4] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 21–30, March 1990.

[5] Jong-Deok Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 145–154, April 1991.

[6] Keith Cooper and Taylor Simpson. SCC-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, October 1995.

[7] 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.

[8] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing '89*, pages 580–588, 1989.

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

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] The Parallel Computing Forum. PCF parallel fortran extensions. *Fortran Forum*, 10(3), September 1991.

[12] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[13] Jens Knoop. Eliminating partially dead code in explicitly parallel programs. *Theoretical Computer Science*, 196(1–2):365–393, April 1998.

[14] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analysis for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[15] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[16] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–204, June 1995.

[17] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38:139–144, 1996.

[18] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[19] Jaejin Lee. *Compilation Techniques for Parallel Shared Memory Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1999. In preparation.

[20] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and concurrent sparse conditional constant propagation for explicitly parallel programs. Technical Report TR#1525, CSRD, University of Illinois at Urbana-Champaign, July 1997.

[21] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Proceedings of The 10th International Workshop on Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 114–130. Springer, August 1997.

[22] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5):563–589, 1998.

[23] Samuel. P. Midkiff and David. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing (ICPP), Vol. II Software*, pages 105–113, August 1990.

[24] Samuel P. Midkiff, David A. Padua, and Ron Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402–422, 1990.

[25] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[26] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *The Tenth Annual Workshop on Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 94–113. Springer, August 1997.

[27] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *The Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.

[28] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[29] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Department of Electrical and Computer Engineering, Rice University, April 1996.

[30] Harini Srinivasan. Optimizing explicitly parallel programs. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, July 1994.

[31] 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.

[32] 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 (POPL)*, pages 260–272, January 1993.

[33] 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.

[34] 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

Concurrent Control Flow Graph

A concurrent control flow graph (CCFG) is a directed graph $G = (N, E, Ntype, Etype)$ such that,

- N is the set of nodes in G. Each node is a basic block.
- $E = E_{cont} \cup E_{sync} \cup E_{conf}$ where:
  - $E_{cont} = \{(m,n) \mid m,n \in N \wedge\ Etype(m,n) \in \{T,F,U\}\}$ is the set of control flow edges.
  - $E_{sync} = \{(m,n) \mid m,n \in N \wedge\ Etype(m,n) \in \{\sigma\}\}$ is the set of synchronization edges which show the order enforced by explicit synchronization operations.
  - $E_{conf} = \{(m,n) \mid m,n \in N \wedge\ Etype(m,n) \in \{\delta^t, \delta^a, \delta^o\}\}$ is the set of conflict edges. Conflict edges join conflicting basic blocks and are bidirectional.
- $Ntype$ is a function which tells the class of nodes: $Ntype : N \mapsto T_N$, $T_N = \{$*Entry, Exit, Pdo, EndPdo, Cobegin, Coend, Condition, Header, Compute, ThreadEntry, ThreadExit*$\}$.
- $Etype$ is a function which tells the type of edges: $Etype : E \mapsto T_E$, $T_E = \{T,\ F,\ U,\ \sigma,\ \delta^t,\ \delta^a,\ \delta^o\}$.

Appendix B

Placing π-assignments

A CCFG $G = (N, E, Ntype, Etype)$ is given after renaming.
n_v denote the basic block where a variable v is defined.
$Precede?(m,n)$ returns **true** if $m \in Prec[n]$, otherwise **false**.
$idnum$ is the next identifying number for a new temporary variable.

$Counter \leftarrow idnum$
for each node $n \in N$
  if n contains a use v_0 of a shared variable **then**
    $\mathcal{W} \leftarrow \{v_0\}$
    $\mathcal{N} \leftarrow \{n_{v_0}\}$
    for each incoming δ^t edge (n_{v_i}, n)
      $\mathcal{W} \leftarrow \mathcal{W} \cup \{v_i\}$
      $\mathcal{N} \leftarrow \mathcal{N} \cup \{n_{v_i}\}$
    end for
    for each basic block n_{v_i} in $\mathcal{N}$
      if $Precede?(n, n_{v_i})$ **then**
        $\mathcal{W} \leftarrow \mathcal{W} - \{v_i\}$
      for each $n_{v_k} \in \mathcal{N}$
        if $Precede?(n_{v_i}, n_{v_k}) \wedge Precede?(n_{v_k}, n)$ **then**
          $\mathcal{W} \leftarrow \mathcal{W} - \{v_i\}$
      end for
    end for
    Construct $t_{Counter} = \pi(v_1, \ldots, v_m)$, where $v_j \in \mathcal{W}$.
    Insert the π-assignment at the beginning of n.
    Replace the use v_0 with $t_{Counter}$ in n.
    $Counter \leftarrow Counter + 1$
end for

Appendix C

Proof of Theorem 5.4

Assume x and y are the first two active and congruent variables but have different values. We will derive a contradiction for each possible operation labeling of the nodes r and s of x and y in the value graph. Without loss of generality, assume also the defining assignment of x has just executed.

Case 1. r's operation label is a constant. x and y should both be constants and equal. A contradiction.

Case 2. r's operation label is **op**. Because r and s are congruent, s's operation label must be **op**. Let x_1 **op** x_2 and y_1 **op** y_2 be the defining expressions of x and y. Since x and y have different values, at least one of the pairs x_1 and y_1 or x_2 and y_2 has different values. Without loss of generality, let x_1 and y_1 be such a pair. Since x and y are congruent, x_1 and y_1 must be congruent. Moreover, x_1 and y_1 must be active variables and have different values. This contradicts to the assumption that x and y are the first two active and congruent variables but have different values.

Case 3. r's operation label is ϕ_n and n is not a loop header. Because they are congruent, s's operation label must also be ϕ_n. Thus, the defining assignments for x and y occur at the same node. Let $\phi_n(x_1, \ldots, x_m)$ and $\phi_n(y_1, \ldots, y_m)$ be the defining expressions in the assignments to x and y. Then, x_i and y_i for some i give different values to x and y. In addition, x_i and y_i must be active variables. This contradicts the assumption.

Case 4. r's operation label is ϕ_n and n is a loop header. Because r and s are congruent, s's operation label must also be ϕ_n. Thus, the defining assignments for x and y occur at the same node. Moreover, the innermost loop containing the defining assignment of x and the innermost loop containing the defining assignment of y are the same, i.e., x and y are defined in the same loop. If x and y have different values before the first iteration of the loop, one pair of the congruent variables corresponding to the children of r and s, which are related to non-back edge have different values before the loop. If x and y have different values after the m_{th} iteration of the loop, then one pair of the congruent variables corresponding to the children of r and s, which are related to the back edge of the loop have different values before the end of m_{th} iteration. Either case contradicts the assumption.

Case 5. r's operation label is π. Let $\pi(x_1, \ldots, x_m)$ be the defining expression in the assignment for the node r. Then, x_i for some i gives the value to x. By the definition of congruence, x_i is congruent to both x and y. In addition, x_i is already active. This contradicts the assumption.

Case 6. r's operation label is ψ. Same as **Case 5**.

We did not consider I/O operations, but the proof can be easily extended to I/O operations. □