A Parallel Algorithm for Static Slicing of Concurrent Programs

D. Goswami, R. Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur, INDIA
email: $< diganta, rajib >$@cse.iitkgp.ernet.in

Abstract

Slicing of concurrent programs is a compute-intensive task. To speed up the slicing process, we develop a parallel algorithm. For this purpose we use *Concurrent Control Flow Graph (CCFG)* as the intermediate representation. We use a network of communicating processes to develop our parallel algorithm. We have implemented our parallel algorithm and the experimental results appear promising.

Key Words: Program slicing, Static slicing, Process network, Program representation, Parallel algorithm, Concurrent program.

1 Introduction

Program slicing is a technique for extracting only those statements from a program which may affect the value of a chosen set of variables at some point of interest in the program. Excellent surveys on the applications of program slicing and existing slicing methods are available in [1, 2]. Slicing is a useful tool and was first introduced by Weiser [3]. Slicing is especially useful for debugging and analyzing the behavior of distributed and concurrent programs. These programs are large and run on several processors often at distributed location. A programmer is often at a loss to determine the cause for a failure as freezing a computation (i.e. global snapshot) is very difficult because the concurrent processes proceed independently with intermittent synchronization and message passing. The accepted way to slice such a program is by first converting the program into an intermediate graph representation and then processing this intermediate representation. The computational complexity of these slicing algorithms is polynomial in the size of the intermediate graphs. For a program with a large number of coarse-grained processes, the intermediate graph may even contain millions of nodes. Also, it is necessary to compute slices very fast because often these slices are required for run-time use, e.g. debugging. In this paper, we attempt to parallelize our previously published results on slicing of concurrent programs [4], to compute slices faster.

Parallelization is a natural choice to speed up compute intensive programs. Harman *et al* have presented a parallel algorithm for static slicing of sequential programs using networks of communicating processes acting concurrently [5]. Each process in this network is defined by a simple function on streams of messages. In this paper, we extend the parallel algorithm proposed by Harman *et al* for slicing concurrent programs.

The rest of this paper is organized as follows. Section 2 briefly reviews a parallel algorithm for static slicing of sequential programs for providing the basic background of the work. We review our intermediate graph representation for concurrent programs in Section 3. In Section 4, we discuss Weiser Slice of concurrent programs. Section 5 describes our proposed parallel algorithm for static slicing. Section 6 gives the correctness proof for our parallel algorithm. Experimental result and comparison with related work are presented in Section 7. We conclude this paper in Section 8.

2 Parallel Algorithms for Static Slicing

A *parallel slicing algorithm* to compute slices for sequential programs was presented by Harman *et al* [5]. In their method, a *process network* is constructed from the program to be sliced. A *process network* is a network of concurrent processes and is defined as a graph whose nodes represent processes and whose edges represent communication channels among processes.

The CFG of the program is first constructed which is used to construct the process network. The *Reverse CFG (RCFG)* is constructed by reversing the direction of every edge in the CFG. Every node in the $RCFG$ represents a process and the edges correspond to communication channels. Edges entering a node i represent inputs to process i and edges leaving node i represent outputs from process i.

Since every node in a $RCCFG$ represents a process in the process network, we will refer to a CFG node i as process i in the context of process network i.e., the identifier i will be used to represent a node and a process in different context.

The following definition of *controlled nodes* would be useful in the next section.

Definition 1 (Controlled Nodes) *The execution of a predicate node 'controls' the* *execution of other nodes in the CFG by determining whether or not control will definitely pass to these nodes or not. For each predicate p, the set of nodes that depend on the truth value of predicate p are termed as the controlled nodes of p.*

For example, for a *while* loop the controlled nodes of the predicate node are simply those appearing in the body of the loop.

2.1 Process Behavior

Each process repeatedly sends and receives messages that are sets consisting of variables and node identifiers. The behavior of each process i depends precisely on the following information, also denoted as *process state*, derived directly from the *CFG* of the program to be sliced:

∘ i : The node identifier,
∘ $ref(i)$: The set of variables referenced in node i,
∘ $def(i)$: The set of variables defined in node i,
∘ $C(i)$: The set of nodes controlled by i, where i is a predicate.

Assuming a side-effect free program, we have $def(i) = \phi$, where i is a predicate node. If i is not a predicate node, then obviously $C(i) = \phi$.

The behavior of each process depends on its *state* and the input message. Let the input to a process i be the set I which is a set of variables and node identifiers. The behavior of each process in the process network is defined in functional notation as follows.

$$
\begin{aligned}
F_i(I) = {} & \text{if } I \bigcap (def(i) \bigcup C(i)) \neq \phi \\
& \text{then } (I - def(i)) \bigcup ref(i) \bigcup \{i\} \\
& \text{else I}
\end{aligned}
$$

This is interpreted as follows:

- If the set I contains any of the variables defined by i or contains any of the node identifiers controlled by i (in case i is a predicate node), then the process i outputs a message on all its output channels consisting of:

  1. all the variables in I that it does not define,
  2. all variables that it references,
  3. its node identifier, i.

- Otherwise (i.e., if I has no elements in common with the defined variables or controlled nodes of i), the process i merely outputs I on all outgoing channels.

The process i then waits for the next input message and upon receiving a message repeats this action.

The output by process i as described above will be referred as *basic output* in later sections.

2.2 Constructing A Slice

To compute a slice for the slicing criterion $< n, V >$, where V is the set of variables and n is the CFG node representing a statement, network communication is initiated by outputting the message V from process n in the process network. Messages will then be generated and passed around the network until it eventually 'stabilizes', i.e. when no new message arrives from any node.

Once stabilized, the slice is computed by observing the output messages by each node: Node i should be included in the slice if and only if process i has output its node identifier i. That is, the algorithm computes the slice of a program by including the set of nodes whose identifiers are input to the *entry* node of the CFG, because the entry node is reachable via every node in the $RCFG$ and thus messages output by all nodes will eventually reach the entry process.

2.3 Correctness

The parallel slicing algorithm has been shown to be correct and finitely terminating by Harman *et al.* We reproduce some of their important results in this regard from [5].

Theorem 1 (Termination) *The process network used in the parallel slicing algorithm can be defined in terms of recursion equation over finite sets of variables and node identifiers and such recursion equation systems give rise to terminating computations [5].*

Theorem 2 (Correctness) *The parallel slicing algorithm is correct in the sense that every statement included in a Weiser slice is also included using the parallel slicing algorithm [5].*

3 Intermediate Representation of Concurrent Programs

In this section, we propose a representation of concurrent programs for the purpose of using it for parallel slicing. This representation is a generalization of CFG for sequential programs. We name this representation *concurrent control flow graph (CCFG).*

Informally, $CCFG$s are forests of *control flow graphs (CFGs)*, one for each process in the program, with nodes and edges added to represent interprocess communications.

3.1 Overview of Parallel Programming Constructs Used

In our subsequent discussions, we will use primitive constructs for process creation, interprocess communication and synchronization which are similar to those available in the Unix environment [6]. The main motivation behind our choice of Unix-like primitives is that the syntax and semantics of these primitive constructs are intuitive, well-understood, easily extensible to other parallel programming models and also can be easily tested.

The language constructs that we consider for message passing are *msgsend* and *msgrecv*. The syntax and semantics of these two constructs are as follows:

- *msgsend(msgqueue, msg):* When a msgsend statement is executed, the message *msg* is stored in the message queue *msgqueue*. The msgsend statement is non-blocking, i.e. the sending process continues its execution after depositing the message in the message queue.

- *msgrecv(msgqueue, msg):* When a msgrecv statement is executed, the variable *msg* is assigned the value of the corresponding message from the message queue *msgqueue*. The msgrecv statement is blocking, i.e. if the *msgqueue* is found to be empty, the receiving process waits for the corresponding sending process for depositing the message.

We have considered nonblocking send and blocking receive semantics of interprocess communication because these have traditionally been used for concurrent programming applications. In this model, no assumptions are made regarding the order in which messages arrive in a message queue from the msgsend statements belonging to different processes except that messages sent by one process to a message queue are stored in the same order in which they were sent by the process i.e., the message queue preserves the order of messages sent from any single process. A process executing a *msgrecv(msgqueue, msg)* statement removes the first available message from the msgqueue.

A fork() call creates a new process called *child* which is an exact copy of the parent. It returns a nonzero value (process ID of the child process) to the parent process and zero to the child process [6]. Both the child and the parent have separate copies of all variables. However, shared data segments acquired by using the shmget() and shmat() function calls are shared by the concerned processes. Parent and child processes execute concurrently. A wait() call can be used by the parent process to wait for the termination of the child process. In this case, the parent process would not proceed until the child terminates.

Semaphores are synchronization primitives which can be used to control access to shared variables. In the Unix environment, semaphores are realized through the semget() call. The value of a semaphore can be set by semctl() call. The increment and decrement operations on semaphores are carried out by the semop() call [6]. However, for simplicity of notation, we shall use P(sem) and V(sem) as the semaphore decrement and increment operations respectively.

This section introduces a method to graphically represent concurrent programs in Unix process environment. This representation is later used to compute static slices. Our method can handle only static creation of processes. We construct the graph representation of a concurrent program through three hierarchical levels: *process graph*, *concurrency graph*, and *CCFG*. Construction of the first two levels is described next only providing necessary definitions without going into details of the construction of various intermediate graphs to save space. The interested reader can find these in [4].

3.2 Process Graph

In [4], we have proposed a hierarchical graphical representation for concurrent programs. This is done through three levels and the first level of these is denoted as process graph. A *process graph* captures the basic process structure of a concurrent program.

Definition 2 (Process Graph) *A process graph, $G_p = (N_p, E_p, f_p)$, is a directed graph where N_p is a set of nodes referred as process nodes. $E_p \subseteq N_p \times N_p$ is a set of edges. f_p is a function assigning statements to process nodes. The edges of a process graph can be of two types: fork edges and join edges.*

Informally, a *process node* consists of a sequence of statements of a concurrent program which would be executed by a process.

Definition 3 (Fork Edge) *A fork edge from a node P1 to a node P2 in a process graph exists if the last statement in the statement sequence represented by P1 is a fork call and P2 represents the statement sequence to be executed by the parent process immediately after this fork call or by the child process created by this fork call.*

Definition 4 (Join Edge) *A join edge from a node P2 to a node P1 in a process graph exists if all the following hold:*

1. *the statement sequence represented by P1 contains a wait call.*

2. *the statement sequence represented by P2 does not contain any fork call.*

3. *P0 is the first predecessor node of both P1 and P2 which represents a statement sequence where the last statement is a fork call.*

3.3 Concurrency Graph

A process graph captures only the basic process structure of a program. This has to be extended to capture other Unix programming mechanisms such as interprocess communication and synchronization. We achieve this by constructing a *concurrency graph*. A concurrency graph is a refinement of a process graph where the process nodes of the process graph containing message passing statements are split up into three different kinds of nodes, namely *send node, receive node,* and *statement node* [4]. Interprocess communication due to message passings are represented by *communication edge*s.

Definition 5 (Communication Edge) *A communication edge from a send node S to a receive node R in a concurrency graph exists if both the msgsend statement in the send node S and the msgrecv statement in the receive node R uses the same message queue.*

Definition 6 (Concurrency Graph) *A concurrency graph $G_c = (N_c, E_c, f_c)$ is a directed graph where N_c is a set of nodes referred as components. $E_c \subseteq N_c \times N_c$ is a set of edges and f_c is a function assigning statements to components. Edges of a concurrency graph can be of following types: fork edges, communication edges, and control edges.*

Each node of the concurrency graph is called a *component*. The maximal set of components which are capable of concurrent execution is called a *concurrent set of components* or just a *concurrent component*. A concurrency graph captures the dependencies among different components arising due to message passing communications among them. However, processes may also interact through other forms of communication such as using shared variables. Access to shared variables may either be unsynchronized or synchronized using semaphores. Determination of such shared dependences (represented by *shared dependence edge*s) cannot be done from a simple analysis of the source code.

Definition 7 (Shared Dependence Edge) *A shared dependence edge from a statement s_1 to another statement s_2 exists if the following hold:*

1. $x \in def(s_1)$,

2. $x \in ref(s_2)$,

3. *s_1 and s_2 can execute concurrently.*

To determine shared dependence across two components one must know whether these two components are concurrent or not. We presented an algorithm based on graph reachability for determining the set of components concurrent to a given component [4].

3.4 Concurrent Control Flow Graph

It is interesting to observe that fork edges represent 'flow of control' among processes in a concurrent programs. When a process forks, it creates a child process and executes concurrently with the child. So the fork edges in a process graph represent parallel flow and we will consider these as the *parallel flow edge*s.

We now modify the representation proposed in [4] to construct a generalized control flow graph for concurrent programs which we will refer to as *concurrent control flow graph*, ($CCFG$). The process graph is first constructed for the given program. For simplicity of notation we refer to a process node in a process graph as a *pnode*. Concurrency graph is then constructed using the method described in [4] to find out the concurrent components. It is now possible to determine the pairs of statements belonging to two different *pnodes*. The edges between such pairs of nodes constitute either a shared dependence edge or a communication edge. Construction procedure for these edges have already been explained in [4].

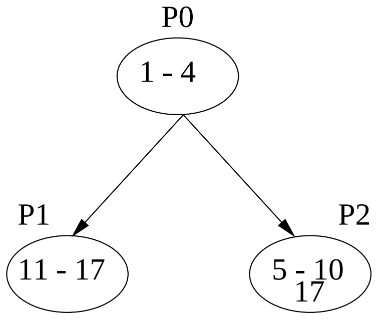
For every *pnode* of the process graph, we now construct a CFG from the statement sequence of that *pnode*. To do this, we construct special *start* nodes for all *pnodes* to mark the entry of the CFGs of these *pnode*s. For the *pnode* that does not have any predecessor in the process graph, this start node is denoted as *Entry* and denotes the point where the program starts executing. For all *pnode*s which do not have any successor in the process graph, we construct *stop* nodes to mark the termination of each of them in the CFG. Figure 1(b) shows the process graph of the example program given in Figure 1(a). The corresponding CFGs of the pnodes $P0$, $P1$, and $P2$ are shown in Figure 2.

From the construction of the individual CFGs of the pnodes from the process graph, it is obvious that the first node of every CFG is either a *start* node or an *Entry* node and the last node is either a *stop* node or a node representing a fork statement. The individual CFGs are now interconnected by adding *inter-pnode* edges to construct the $CCFG$. The various inter-pnode edges are: *parallel flow edge*s, *shared dependence edge*s, and *communication edge*s. Construction of these edges in the $CCFG$ is explained next. The $CCFG$ of the example program given in Figure 1(a) is shown in Figure 3.

Let the last statement in the statement sequence in $pnode_1$ be a fork statement

```
     main() {
      /* declaration */
1.    x = ...;
2.    y = ...;
3.    s = ....  /* shared variable */
4.    if ( fork() == 0) {
5.          y = f1(y);
6.          m = f2(y);
7.          msgsend(q1, m);
8.          if c1
9.                s = f3(y)
10.         else  s = f4 (y);
      }
    else {
11.         x = f5(y);
12.         msgrecv(q1, m);
13.         y = f6(m);
14.         x = x + s;
15.         printf("...", x);
16.         printf("...", y);
      }
17.  y = f7(x, y);
      }
```

(a)



(b)

Figure 1: (a) An Example Program and (b) Its process Graph

and *pnode*$_2$ and *pnode*$_3$ be its successors in the process graph. Then *parallel flow edge*s from the node in the corresponding CFG of *pnode*$_1$ representing the fork statement to the two start nodes of the corresponding CFGs of *pnode*$_2$ and *pnode*$_3$ are constructed to represent parallel flows. For example, two parallel flow edges exist in the $CCFG$ from node 4 to the two start nodes as shown in Figure 3.

Let $(s, s^{'})$ be a statement pair representing a shared dependence edge which is determined using the information acquired from the concurrency graph as already explained in [4]. Let $s \in pnode_i$ and $s^{'} \in pnode_j$ in the process graph. An inter-pnode edge in the $CCFG$ is then constructed from the node corresponding to the statement s in the CFG of $pnode_i$ to the node corresponding to the statement $s^{'}$ in the CFG of $pnode_j$ to represent shared dependence. By using this procedure two shared dependence edges are constructed in the $CCFG$ of the example program: one from node 9 to node 14 and the other from node 10 to node 14.

A communication edge for the statement pair $(s, s^{'})$ is similarly added to the $CCFG$ where s is a msgsend statement in one pnode and $s^{'}$ is a msgrecv statement in another pnode in the process graph such that concurrency graph reveals a potential

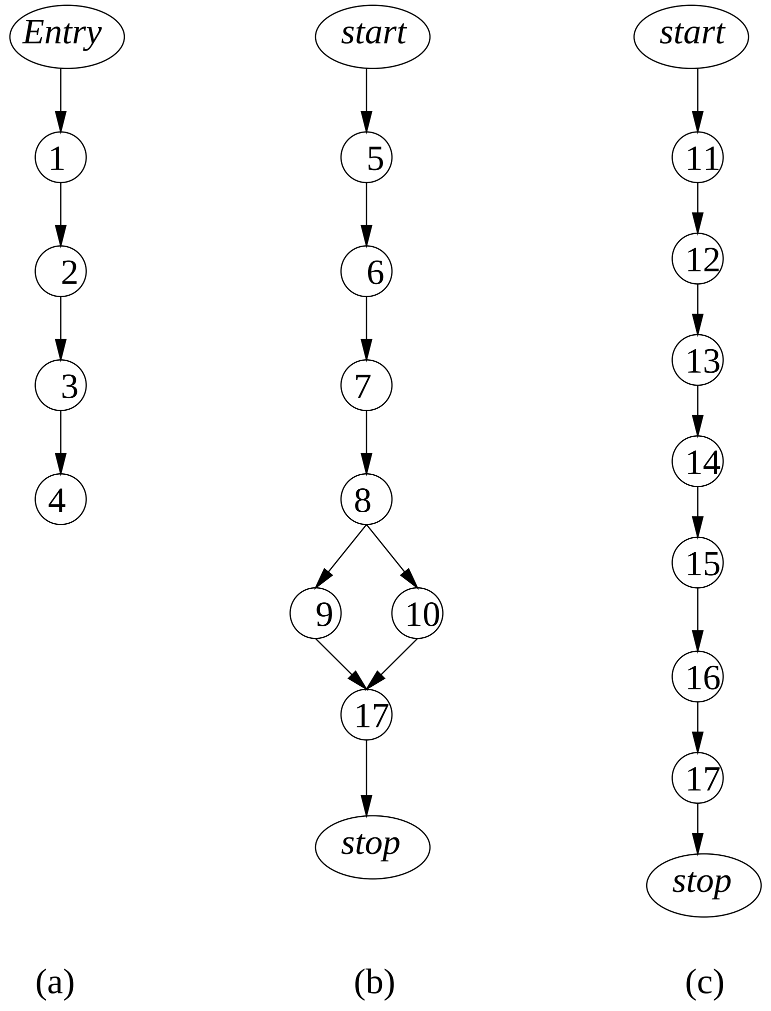


Figure 2: *CFG*s of the *pnodes* Shown in Figure 1(b)

communication edge between the two. The only communication edge that exists in the $CCFG$ of the example program is from node 7 to node 12.

We now define $CCFG$ for a concurrent graph more formally.

Definition 8 (CCFG) *A CCFG, G_F, of a concurrent program is a triple (N_F, E_F, S) where N_F is a set of nodes representing statements of the program, E_F is a set of edges of two types: inter-pnode and intra-pnode. Intra-pnode edges represent control flow edges while inter-pnode edges are of following types: parallel flow edge, shared dependence edge, and communication edge. S is a set of dummy nodes and may be of the following three types: start, stop, and Entry.*

The construction procedure of a $CCFG$ described above is summarized in the pseudocode of algorithm *Construct_CCFG* given below. We use the notations s^p and

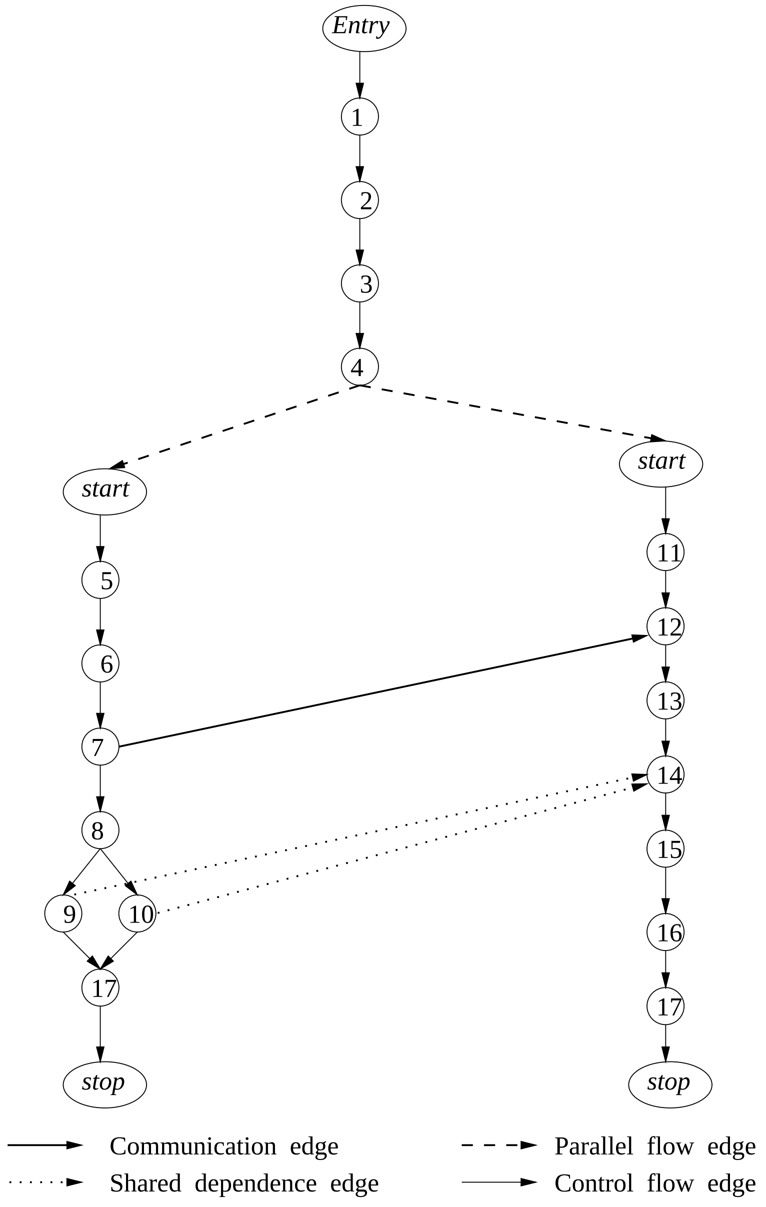


Figure 3: *CCFG* of the Example Program Shown in Figure 1(a)

s^G to indicate a statement in the statement sequence represented by a *pnode* and the corresponding CFG node respectively. We also use two sets: S_D, which represents the pair of statements from two different *pnodes* having a shared dependence edge between them. Similarly, S_C, the set of pair of statements represents communication edges.

```
Construct_CCFG
Input: Process Graph $G_p$, $S_D$, $S_C$
Output: CCFG
    for all $n_p \in N_p$ do
    begin
        Construct the $CFG$ of $n_p$
    end
    for all $n_p \in N_p$ do
    begin
        /* Examine all the statements in $n_p$ sequentially */
        for all statements in $n_p$ do
        begin
            $s^p \leftarrow$ current statement examined
            if $s^p.type$ = msgsend
            begin
                Construct communication edge $(s^G, s_k^G)$ such that $(s^p, s_k^p) \in S_C$
            end
            else if $s^p.type$=fork
            begin
                Find successors $n_i$ of $n_p$ in the process graph $G_p$
                $s_i^G \leftarrow$ corresponding start node in the $CFG$ of $n_i$
                Construct parallel flow edges $(s^G, s_i^G)$
            end
            else if $s^p.type$= shared variable read
            begin
                Construct shared dependence edges $(s_k^G, s^G)$ such that $(s_k^p, s^p) \in S_D$
            end
        end
    end
```

4 Weiser Slice of Concurrent Programs

We now develop the definition of Weiser slice of concurrent programs by extending the definition of Wesier slice for sequential programs. Based on this definition, we will compute slices of any concurrent program using our proposed technique.

Weiser slice for sequential programs are computed using CFGs as intermediate representations. The slicing method (due to Weiser) from a given CFG has been presented in [3]. In Section 3, we have proposed an intermediate representation of concurrent programs which is a generalization of CFG. This leads us to develop a method to compute slices of concurrent programs using Weiser's method.

Consider the slicing criterion $< P, s, V >$. For the given slicing criterion, each of the *pnodes* will contribute its part of the slice. Let the slice for the given criterion be *Slice*. $Slice_i$ denotes the section of the slice contributed by $pnode_i$. Therefore, the resulting slice can be expressed as the union of the individual parts of the slice contributed by all *pnode*s, i.e.,

$Slice = \bigcup Slice_i$

For the given criterion $< P, s, V >$ Weiser Slice *Slice* is computed using the following steps. (From now onwards, a node in the $CCFG$ corresponding to a statement s in the process graph will be annotated as s^F).

1. $Slice \leftarrow \phi$
2. If $s \in pnode_i$, then compute $Slice_i$ from the CFG of $pnode_i$ using the criterion $< s, V >$. ($Slice_i$ is the Weiser slice of the $pnode_i$).
3. $Slice \leftarrow Slice \bigcup Slice_i$
4. Generate new criteria $< pnode_j, s_m, V' >$ if all of the following hold:
   (a) $s_m \in pnode_j$ and $s_m.type = msgsend$ or *shared variable use*,
   (b) there exist shared dependence or communication edge (s_m^F, s_k^F) in the $CCFG$, and
   (c) $s_k \in Slice$
5. Repeat steps 2 to 4 using the new criteria generated until the slice *Slice* stabilizes.

We denote this slice, which is computed by extending the Weiser's technique to concurrent programs, as *Weiser Slice*.

5 A Parallel Algorithm for Static Slicing

To compute static slices of concurrent programs, we extend the parallel algorithm proposed in [5] to $CCFG$, our representation of concurrent programs. We first construct the process network for any given concurrent program. The topology of the process network is given by the *Reverse Concurrent Control Flow Graph (RCCFG)*. The *RCCFG* is constructed from the *CCFG* by reversing the direction of all edges. Every node of the *RCCFG* represents a process and every arc represents a communication channel between these processes. The *RCCFG* of the example concurrent program shown in Figure 1(a) is shown in Figure 4.

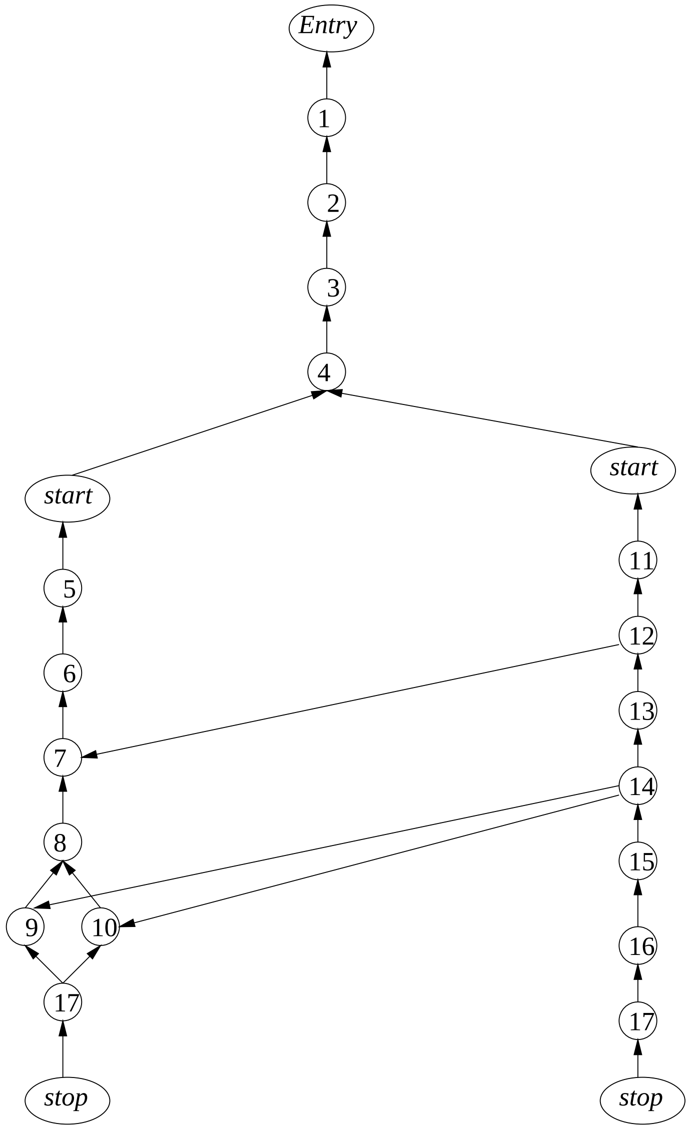


Figure 4: *RCCFG* of the Example Program Shown in Figure 1(a)

There are some additional processes and channels in the process network which do not exist in process networks for sequential programs. These processes arise in the process network of concurrent programs due to statements used for interprocess communication, process creation, etc. Behaviors of these processes and channels are different from those of ordinary ones as described in the context of sequential programs in [5]. We now outline how these processes and channels are handled to compute slices from the process network of concurrent programs.

In the standard Unix semantics each message passing statement uses a message queue *msgque* and a message *msg* [6]. The input to a process in the process network is denoted as I. We assume that the type of statement represented by a process in the process network is given by *process.type* and may be any of the following: *msgsend*, *msgrecv*, *fork*, *shared var use*, and *ordinary* (*ordinary* represents any statement other than these four). A process of type *shared var use* in a process network represents a statement which uses a shared variable. Similarly, the type of a channel representing an edge is given by *channel.type* and may be any of the following: *sd-channel* (representing a shared dependence edge), *com-channel* (representing a communication edge), and *ordinary channel* (representing any edge other than these two).

5.1 Process Behavior

Behavior of each of the processes in the process network of a concurrent program is defined as follows.

Let the input to the process i in the process network received from the input channel c be the set I. Please recall that I is a set of variables and node identifiers. Behavior of process i depends on the types of process i and input channel c. Depending on the type of i we get the following four cases. The corresponding process behavior is presented with each case.

```
Case 1: i.type = msgsend
            if c.type = com-channel, then $I = I \cup \{i\}$
            Transmit $I$ on all output channels

Case 2: i.type = msgrecv
            if $msg \in I$ then {
                    $I = I \cup \{i\}$
                    If outgoing com-channel from this process $i$ is not disabled,
                    then transmit $\{msg\}$ on com-channel and then disable the
                    com-channel
                    }
            Transmit $I$ on all output channel except the com-channel

Case 3: i.type = shared var use /* Let $x$ is the shared variable */
            if $def(i) \in I$ {
                    If outgoing sd-channel is not disabled, then transmit $\{x\}$
```

```
                on sd-channel and then disable this sd-channel
                }
            Transmit *basic output* on all output channels except the sd-channel

Case 4: i.type = fork
            Transmit $I \cup \{i\}$ on all output channels

Case 5: i.type = ordinary
            Transmit *basic output* on all output channels
```

We now briefly explain the above steps.

- msgsend: If a process in a process network representing a msgsend statement receives an input I, it simply transmits I on all output channels. Before transmitting, it adds its node identifier to I if it receives I from a com-channel, otherwise it is transmitted unaltered.

- msgrecv: If a process in a process network representing a msgrecv statement receives an input I, it checks whether the input I contains the *msg* (that is used in the syntax of the msgrecv statement). If not, then it transmits I on all output channels without any change except the com-channel. Otherwise, it transmits I along with its node identifier on all output channels except the com-channel, and transmits a new message, {msg}, on the com-channel.

- Shared variable use: Whenever a process in a process network which is having sd-channel as one of its output channels receives an input I, it checks whether I contains the variable defined by the corresponding statement. The behavior of the process is same for all output channels except the sd-channel as described in Section 2.1, i.e., it outputs the *basic output*. But it outputs a new message on the sd-channel containing only the shared variable if I contains the relevant shared variable for this process, otherwise it does not output anything on the sd-channel.

- fork: A process representing a fork call always outputs the input along with its node identifier on all output channels.

The processes representing the dummy nodes, for example, *start* and *stop* simply transmit all their input unaltered through the output channels.

The messages sent along sd-channels and com-channel, if any, are transmitted only once. This is because of the fact that the message that is output on any one of these channels, is always fixed as shown in case 2 and case 3 above.

5.2 Constructing A Slice

A slicing criterion for a parallel program is a triple $< P, s, V >$, where P is a process, s is a statement in this process and V is a set of variables. To compute a slice with respect to this criterion, the process network is first initiated. Let n be the node in the $CCFG$ corresponding to the statement s in the process P. Let $n^{'}$ be the process in the process network corresponding to the $CCFG$ node n. The process network is then initiated by transmitting $\{n^{'},V\}$ on all output channels of $n^{'}$ considering the process behavior already discussed. Each process in the process network repeatedly sends and receives messages until the network stabilizes. The network stabilizes when no new messages are generated in the whole network. The set of all node identifiers that reach the *Entry* node gives the required static slice.

The steps for computation of slices are summarized below.

1. Construct the hierarchical $CCFG$ for the concurrent program.
2. Reverse the $CCFG$.
3. Compile the $RCCFG$ into a process network.
4. Initiate network communication by outputting the message $\{s, v\}$ from the process in the process network representing statement s in the process P, where $< P, s, v >$ is the slicing criterion.
5. Continue the process of message generation until no new messages are generated in the network.
6. Add to the slice all those statements whose node identifiers have reached the *entry* node of the $CCFG$.

Construction of the slice with respect to the criterion $< P1, 15, x >$ for the example program given in Figure 1(a) is explained in Figure 5(a). This figure shows the sequence of message generation and sending these messages by processes in the process network. The process network is initiated by outputting the message {15,x} by node 15 on its output channel to node 14. This is represented in the figure by (15,14) – {15,x}. On receiving the message, node 14 (which is a *shared var use* type) outputs the same message {s} on its two sd-channel to node 9 and 10 and the message {14,15,x} on the other ordinary channel according to case 3 of process behavior. Message generation and transmission by nodes continues according to the the rules of various cases as discussed above. Finally, the process network terminates when no new messages are generated in the system. To compute the slice, we take the node identifiers which reach the *Entry* node of the $RCCFG$. This gives us the slice {1,2,3,4,5,8,9,10,11,14,15}.

Computation of another slice with respect to the criterion $< P1, 16, y >$ is shown in Figure 5(b). The resulting slice is found to be {2,4,5,6,7,12,13,16}. Sequence of

```
(15,14) - {15,x}
(14,13) - {14,15,s,x}
(14, 9) - {s}
(14,10) - {s}
(13,12) - {14,15,s,x}
(12,11) - {14,15,s,x}
(11, start)- {11,14,15,s,x,y}
(start, 4) - {11,14,15,s,x,y}
(4, 3) - {4,11,14,15,s,x,y}
(3, 2) - {3,4,11,14,15,x,y}
(2, 1) - {2,3,4,11,14,15,x}
(1, Entry) - {1,2,3,4,11,14,15}
(9, 8) - {9,y}
(10,8) - {10,y}
(8, 7) - {8,9,10,y}
(7, 6) - {8,9,10,y}
(6, 5) - {8,9,10,y}
(5, start) - {5,8,9,10,y}
(start, 4) - {5,8,9,10,y}
(4, 3) - {4,5,8,9,10,y}
(3, 2) - {4,5,8,9,10,y}
(2, 1) - {2,4,5,8,9,10}
(1, Entry} - {2,4,5,8,9,10}
Terminate
```

(a)

```
(16,15) - {16,y}
(15,14) - {16,y}
(14,13) - {16,y}
(13,12) - {13,16,m}
(12,11) - {12,13,16,m}
(12, 7) - {m}
(11, start) - {12,13,16,m}
(start, 4) - {12,13,16,m}
(4, 3) - {4,12,13,16,m}
(3, 2) - {4,12,13,16,m}
(2, 1) - {4,12,13,16,m}
(1, Entry) - {4,12,13,16,m}
(7, 6) - {7,m}
(6, 5) - {6,7,y}
(5, start) - {5,6,7,y}
(start, 4) - {5,6,7,y}
(4, 3) - {4,5,6,7,y}
(3, 2) - {4,5,6,7,y}
(2, 1) - {2,4,5,6,7}
(1, Entry) - {2,4,5,6,7}
Terminate
```

(b)

Figure 5: Sequence of Message Generation and Message Passing in the Process Network during Computation of the Slice of the Example Program Shown in Figure 1(a) (a) w.r.t. the Criterion $< P1, 15, x >$ and (b) w.r.t the Criterion $< P1, 16, y >$

message generation and sending these messages by processes in the process network is shown in the figure.

6 Correctness of the Parallel Slicing Algorithm

In this section, we show that our parallel slicing algorithm has finite termination property and that it computes correct slices. We consider a slice to be correctly computed if it is identical to the Weiser slice as defined in Section 4.

Theorem 3 (Termination) *The process network used in the parallel slicing algorithm for concurrent programs gives rise to terminating computations.*

Proof: Consider that the process graph for the given program consists of a single *pnode* indicating that the program has a single process. Hence, by Theorem 1, the corresponding process network has finite termination. Theorem 1 also applies when the process graph consists of more than one *pnode* but no inter-pnode edges in the $CCFG$ except the parallel flow edges. In this case also, the process network can be expressed in terms of recursion equation as that in Theorem 1.

If there exists any inter-pnode edge in the $CCFG$, the process network will contain some *sd-channel*s and/or *com-channel*s. During execution of the process network when these channels are used, messages through these channels are transmitted only once and then disabled as described in case 2 and 3 in Section 5.1. The set of such channels in a process network is finite and hence this set will eventually be exhausted after a finite number of steps. At that point, by applying Theorem 1, we prove finite termination of the process network. $\square$

Theorem 4 (Correctness) *The parallel slicing algorithm is correct in the sense that every statement included in a Weiser slice will also be included using the parallel slicing algorithm.*

Proof: Consider the case when the process graph consists of a single *pnode* or more than one *pnode* without any inter-pnode edge except parallel flow edges in the $CCFG$. Then a slice computed with respect to any criterion will be internal to a particular process. So using Theorem 2 we find the resulting slice to be correct.

Let us consider the case where the process network consists of more than one *pnode* with inter-pnode edges. If a slice is computed using the parallel algorithm for the criterion $< pnode_i, s, V >$, then $Slice_i$ will be computed as usual by initiating network communication at the process in the process network corresponding to the $CCFG$ node s^F. $Slice_i$ will be correct by Theorem 2. Let us consider the case where $s_k \in Slice_i$ and $s_k.type$ is *shared variable use*. Hence, there is an inter-pnode edge (s^F_m, s^F_k) in the $CCFG$ such that $s_m \in pnode_j$ for some j. Now, by applying case 3 of process behavior as described in Section 5.1, a message will be transmitted on the output sd-channel (corresponding to the shared dependence edge) by the process corresponding to the $CCFG$ node s^F_k. The message is simply $\{x\}$, where x is the shared variable used in s_k. This step leads to start of computation of $Slice_j$ in $pnode_j$ by starting message generation at and transmission from the process corresponding to the $CCFG$ node s^F_m i.e. using the new criterion $< pnode_j, s_m, x >$. $Slice_j$ computed will be correct according to Theorem 2. If $s_k.type$ is msgrecv, then by applying case 2 and case 1 as described in Section 5.1 and forwarding the same argument as that in case of shared variable use type, we get correct slice $Slice_j$ in the pnode $pnode_j$. Of course, in this case we assume that there exists a com-edge (s^F_m, s^F_k) in the $CCFG$ such that $s_m \in pnode_j$ for some j. Since this process of generation of new criteria continues until the slice $Slice$ stabilizes, finally we will get the slice $Slice = \bigcup Slice_i$ which is the correct Weiser Slice (as defined in Section 4) for the given concurrent program. $\square$

7 Implementation and Experimental Results

We have implemented the parallel algorithm for computing dynamic slices of concurrent programs. The implementation environment is Digital-Unix and we have considered a subset of C language with Unix primitives for process creation and interprocess communications.

The module structure of our prototype implementation is shown in figure 6. The **Program analyzer** analyzes the program code to extract all information pertaining to process creation, join, and termination. We have used the standard Unix tools *lex* and *yacc* to carry out the syntax and semantic analysis of the program code. The information collected for every statement s during this analysis are: node id, statement type, $ref(s)$, $def(s)$, $C(s)$, input channels (numbers and types), and output channels (numbers and types).

The **Process graph constructor** uses the information collected by the program analyzer component to create the process graph. We have used data structures linked through pointers to implement the edges and nodes of the process graph.

The **Concurrency graph constructor** refers to the process graph generated by the process graph constructor as well as the information generated by the program analyzer to extract the send/receive calls made by various processes. Using this information it splits the relevant nodes of the process graph into statement nodes, send nodes and receive nodes. Communication edges are constructed between corresponding send and receive nodes. The concurrency graph constructor component also incorporates the algorithm to compute the concurrent components.

RCCFG constructor constructs the $RCCFG$ by first constructing the CFGs of individual *pnodes* and then by adding the inter-pnode edges to connect the CFGs.

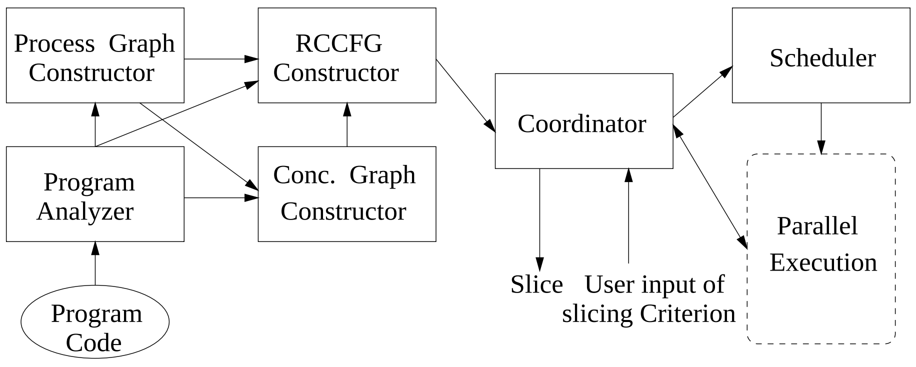


Figure 6: Architecture of the Implementation of the Parallel Algorithm

The **Coordinator** module takes the $RCCFG$ as input and creates processes for every $RCCFG$ node and then submit to the **Scheduler**. The **Scheduler** assigns these processes to available processors for parallel execution. The Coordinator also reads the user input (slicing criterion and number of processors). Based on the number of processors, the Scheduler assigns the processes to as many processors. The Coordinator is also responsible for terminating all the processes as soon as the network 'stabilizes' i.e. when no new messages are generated in the system. After termination of the parallel execution, the Coordinator collects the message received by the entry node and outputs the slice.

A major aim of our implementation is to investigate the achieved speed up in computing slices. The algorithm has been tested with several programs. The length of the programs that we have used for testing vary from 30 to 100 lines. We have considered a subset of C programming language in writing these programs. The results obtained from 8 sample programs have been shown in figure 7. Program sample 1 is shortest in length and program sample 8 is the longest. For different program samples figure 7 shows the speed up achieved in two processor (solid line), three processor (dashed line), and four processor (dotted line) environment.

We have noticed encouraging results from the implementation. It is observed that speed up achieved for different programs – in two processor environment is between 1.13 and 1.56; in three processor environment is between 1.175 and 1.81; and in four processor environment is between 1.255 and 2.08. For the same number of processors used, speed up varies for different program samples. It is seen that the achieved speed up is higher for larger programs. This may be due to the fact that the number of nodes in the process network for larger programs is higher compared to smaller programs, leading to more parallelism and consequently higher utilization of the processors. It is also obvious from the result that the speed up for the same program increases with the number of processors used and varies between 1.13 to 2.08 for different program samples when the number of processors is increased from 2 to 4.

7.1 Comparison With Related Work

Speeding up slice computation has been an important objective of the researchers working in this area [1, 2, 7, 8]. However, most of the reported research in this regard has been in the form of proposing more efficient algorithm. Parallelization of slicing algorithm is scarcely reported in literature. The only published parallel algorithm we could find was proposed by Harman *et al* [5]. Their algorithm is applicable only to sequential program slicing. We have extended it to compute slices of concurrent programs. We have also implemented the parallel algorithm to investigate the achieved speed up.

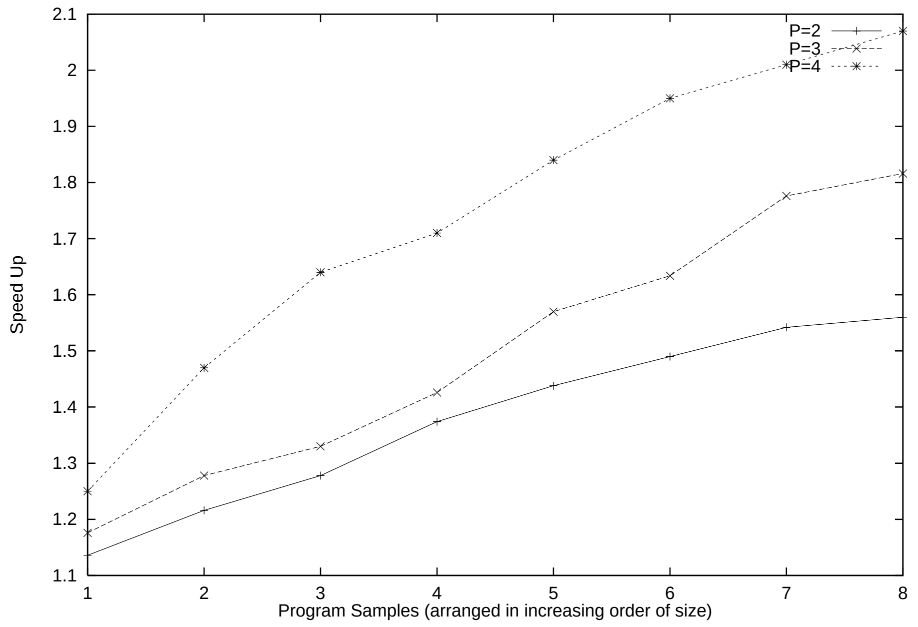


Figure 7: Speed Up Achieved for Different Program Samples by Using Our Parallel Algorithm

8 Conclusion

In this paper, we have introduced a parallel algorithm for static slicing of concurrent programs. For this purpose, we use $CCFG$ as the intermediate representation of concurrent programs. $CCFG$ represents intra-process control flow edges and inter-process data and communication edges. We have also defined Weiser Slice for concurrent programs. A parallel algorithm for slicing concurrent programs using process network has been proposed. We have shown that our parallel algorithm has finite termination property. It has also been shown to compute correct Weiser Slices. We have developed a prototype implementation of the algorithm and the performance results are encouraging.

References

[1] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, September 1995.

[2] D. Binkley and K. B. Gallagher, "Program slicing," *Advances in Computers, Ed. M. Zelkowitz, Academic Press, San Diego, CA*, vol. 43, pp. 1–50, 1996.

[3] M. Weiser, "Program slicing," *IEEE Trans. on Software Engineering*, vol. 10, no. 4, pp. 352–357, July 1984.

[4] D. Goswami, R. Mall, and P. Chatterjee, "Static slicing in unix process environment," *Software - Practice and Experience*, vol. 30, no. 1, pp. 17–36, January 2000.

[5] M. Harman, S. Danicic, and Y. Sivagurunathan, "A parallel algorithm for static program slicing," *Information Processing Letters*, vol. 56, no. 6, pp. 307–313, 1996.

[6] M. J. Bach, *The Design Of The Unix Operating System.* Prentice Hall India Ltd., New Delhi, 1986.

[7] D. Goswami and R. Mall, "An efficient method for computing dynamic program slices," *Information Processing Letters*, (to appear).

[8] H. Agrawal and J. Horgan, "Dynamic program slicing," *In Proceedings of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, SIGPLAN Notices, Analysis and Verification, White Plains, New York*, vol. 25, no. 6, pp. 246–256, June 1990.