AN INTERACTIVE PARALLEL PROGRAM SLICER (PPS) FOR C PROGRAMS ON THE iPSC/2 SYSTEM

By

TING-HUAN HSIAO

Bachelor of Science in Pharmacy

Taipei Medical College

Taipei, Taiwan,

Republic of China

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1992

Thesis
1992
H8725i

AN INTERACTIVE PARALLEL PROGRAM SLICER (PPS) FOR C PROGRAMS ON THE iPSC/2 SYSTEM

Thesis Approved:

[illegible]
Thesis Adviser

[illegible]

Blayne E. Mayfield

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Dr. Mansur Samadzadeh for his guidance and encouragement throughout my thesis. Profound thanks are due to Drs. G. E. Hedrick and Blayne Mayfield for their advice and willingness to serve on my committee. Their support and suggestions were very helpful throughout the study.

Thanks go to Dr. K. Teague for providing access to the iPSC/2 System, Mr. I. Ahmad for providing the test parallel programs, and Mr. W. D. Miller for serving on the examination committee. They made this study go so smoothly.

Many thanks are due to my parents, my brother, and my sister for their moral support. Finally, I would like to thank my friend Li-Chen Ko for her suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Computer programs are products. Every product has its limited useful lifetime. Compared with the cost of producing a new product, the cost of prolonging the life span of an existing product is much lower. Maintaining an existing program is the usual way to prolong the life span of a program. The maintenance of a program includes debugging, testing, modifications, and adding features. Understanding what a program does is the basic requirement for the maintenance of a program. A tool which can make existing programs to be understood more easily is a useful tool for program maintenance.

According to the data flow and control flow graph of a program and a given slicing criterion, program slicing decomposes a large existing program into relatively smaller programs [Weiser 82]. Those small programs are called slices [Weiser 82 and 84]. The notation of a slicing criterion, which was first introduced by Weiser [Weiser 81], is a tuple <i, V>, where i is a specified line number of the code in a program and V is a subset of variables in the program. A slice retains the original properties of the specified variables.

The approach of program slicing makes an originally large program more amenable to debugging, testing, and understanding. Because program slicing decomposes an existing program and focuses on the desired variables of the program, it is more powerful for maintaining existing programs than designing new programs. When maintaining existing large programs, programmers can apply this technique to focus only on those statements which may influence the slicing criterion. This slicing criterion is not sufficient for parallel programs written with explicit message passing for distributed-memory parallel processors. Every such parallel program comprises at least two programs: one host program and one node program.

This thesis introduces an extended slicing criterion for applying slicing to distributed-memory parallel programs. The extended slicing criterion extends Weiser's slicing criterion to be <f, i, V>, where f specifies the end of back-tracing in program slicing. The definitions of i and V are the same as Weiser's. The extended slicing criterion solves the insufficiency of Weiser's slicing criterion for distributed-memory parallel programs. The first advantage of the extended slicing criterion is that it can be applied to distributed-memory parallel programs as well as sequential programs. The second advantage is that the definition of f can be extended to be a set of files instead of one file. For a distributed-memory parallel program, it is not necessary to be restricted to one host program and one node program to form a parallel program. For a sequential program, it is not necessary to put all the procedures and functions in a single program.

This thesis applies the extended slicing criterion and the static slicing technique to develop an interactive parallel program slicer (PPS) for the distributed-memory parallel programs which are written in C language on the iPSC/2 System. This thesis is organized as follows: Chapter II, literature review, comprises Flynn's and Duncan's classifications of computer architectures and program slicing. Chapter III consists of the definition, applications, and advantages of the extended slicing criterion. Chapter IV discusses the development of the parallel program slicer (PPS). Finally, Chapter V contains the summary and future work of this thesis.

CHAPTER II

LITERATURE REVIEW

2.1 Parallel Computer Architectures

2.1.1 Flynn's Taxonomy of Computer Architectures

According to Flynn's taxonomy of computer architectures [Flynn 66], computer architectures can be divided into four different types: SISD, SIMD, MISD, and MIMD.

SISD, single instruction and single data stream, is a serial computer architecture. SISD is a uniprocessor environment [Hennessy 90].

SIMD, single instruction and multiple data streams, is an environment that deals with one instruction and several different data streams at the same time. All synchronously executing processors may deal with the data streams but they are working on the same instruction [Hennessy 90]. The SIMD architecture is divided into two different types: processor array architectures and associative memory processor architectures [Duncan 90].

MISD, multiple instructions and single data stream, is an environment that deals with several instructions and one data stream at the same time [Hennessy 90]. This is not a useful architecture category.

MIMD, multiple instructions and multiple data streams, is an environment that deals with several instructions and several data streams at the same time.

Flynn's classification, which was developed in 1966, is not enough for more recent computer architectures [Duncan 90 and Hennessy 90]. Many modern computer architectures do not belong to a single type in Flynn's taxonomy. Computer designers not only combine some types of Flynn's taxonomy in a new computer architecture, but they also develop new computer architectures that are beyond that taxonomy.

2.1.2 Duncan's Taxonomy of Parallel Computer Architectures

Many modern computer architectures cannot be assigned to any of the classes in Flynn's taxonomy. In 1990, Duncan developed a new classification of parallel computer architectures [Duncan 90]. Duncan's taxonomy of parallel computer architectures includes some of the traditional types of Flynn's taxonomy and many modern computer architectures. Duncan's taxonomy includes three types: synchronous, MIMD, and MIMD paradigm. The first type is synchronous which includes vector, SIMD, and systolic architectures. The SIMD includes processor array and associative memory architectures. The second type is MIMD which includes distributed memory and shared memory architectures. The third type is MIMD paradigm which comprises MIMD/SIMD, data flow, reduction, and wavefront architectures. This taxonomy solves the insufficiency of Flynn's taxonomy.

Different characteristics of parallel programs depend largely on the targeted parallel computer architectures. In this thesis, the word "MIMD" refers to Duncan's taxonomy of parallel computer architectures.

2.1.3 Classifications of MIMD Architectures

There are a number of different parallel computer architectures. All of these architectures support the executions of parallel programs by providing more than one processor [iPSC/2 90]. All parallel processors "cooperate to solve problem through concurrent execution" [Osterhaug 89]. According to Duncan's taxonomy, MIMD architectures are divided into two different types: shared memory and distributed memory [Duncan 90].

Shared memory architectures have two types of memories. The first type of memory is a shared memory [Osterhaug 89] which is global [Hennessy 90]. Every process can access the shared data in the global memory which may consist of several memory modules [Duncan 90]. The second type of memory is a private memory [Osterhaug 89] which is a local memory [Duncan 90 and Hennessy 90]. Although there is only one shared main memory, every processor has its own cache as a local memory. This kind of memory architecture is also called a tightly-coupled memory architecture [Osterhaug 89].

Because every processor shares the same main memory, the accesses of the real memory storage depend on the different virtual memory mapping schemes available on different architectures. There are many different interconnection schemes in shared memory architectures, such as bus interconnection, crossbar interconnection, and multistage interconnection [Duncan 90].

Distributed memory architectures have no global memory that can be shared by every processor. Every processor has its own local memory, i.e., a cache memory [Fox 88]. The interconnections between processors of a distributed memory system are strictly based on passing messages [Duncan 90]. The interconnection topologies include rings, meshes, trees, hypercubes, and reconfigurable architectures [Duncan 90].

2.2 The iPSC/2 System

2.2.1 Introduction

By applying hypercube topology, the iPSC/2 System achieves a MIMD parallel computer architecture that has a distributed memory [Brandis 86]. The interconnection between processors is done by passing messages [iPSC/2 90 and 91]. Users can ask for the desired number of processors and the desired size of memory for their programs, but the number of processors and the size of memory must be a power of two [Fox 88, iPSC/2 90 and 91]. A group of allocated nodes is called a cube [iPSC/2 91]. All the allocated nodes are a group which is used to solve an unique problem.

2.2.2 Characteristics of the iPSC/2 System

The iPSC/2 System applies hypercube topology and achieves a MIMD distributed-memory parallel computer architecture [Brandis 86]. The iPSC/2 System consists of a host and one or more cabinets [iPSC/2 90]. A host is a workstation or a local system resource manager [iPSC/2 90]. The host provides the only interconnection between the user and the allocated nodes [Brandis 86 and iPSC/2 91]. A cabinet includes storage devices and nodes [iPSC/2 90]. There are two different storage devices: disk drives and tape drives. There are four different nodes: compute nodes, I/O nodes, integrated nodes, and service nodes [iPSC/2 91]. A node consists of a processor and a local memory. All nodes are connected. The iPSC/2 System supports both remote hosts and local hosts [iPSC/2 91]. The iPSC/2 System supports a maximum number of 128 nodes [Brandis 86].

There are two different operating systems on the iPSC/2 System. The operating system of the host is UNIX and the operating system of the node is NX/2. The iPSC/2 System allows different users to allocate different cubes at the same time.

In the iPSC/2 System, every allocated node has a binary identification number and every two connected nodes can only have one different bit between the two binary numbers [Fox

88 and iPSC/2 90]. If the user asks for 2^n nodes from the system, every node is "connected to" n nodes and the identification numbers of those nodes range from 0 to $2^n - 1$ [Fox 88 and iPSC/2 90]. The intermediate nodes for connecting any two nodes consist of the number of different bits of the binary identification numbers of the two nodes minus 1. The maximum number of intermediate nodes for connecting any two nodes is n - 1.

2.2.3 Programming on the iPSC/2 System

A parallel program for the iPSC/2 System consists of host and node programs. The host and node programs are separate files. The host and node programs are compiled and linked on the host. The compiled host program is then loaded and executed on the host. According to the request of the user program, the compiled node program will be loaded and executed on some or all of the allocated nodes. The iPSC/2 System supports C and Fortran programming languages. The libraries for the host program and the node program are different. For compiling a host program, the "host" must be specified as a parameter. For compiling a node program, the "node" must be specified as a parameter. For example, if the host program is named "host.c" and the node program is named "node.c", the host program can be compiled as "cc -o host host.c -host"; the node program can be compiled as "cc -o node node.c -node". After the compilations of the host and node programs, the user can type "host" to execute the parallel program.

2.3 The iPSC System Simulator

By providing the Simulator libraries and Simulator program, the iPSC System Simulator produces an environment which accepts the commands and system calls of the iPSC/2 System [iPSC 91]. The Simulator program runs on Sequent S81 and it is only for designing a prototype of a parallel program [iPSC 91]. The iPSC System Simulator is not the same as the iPSC/2 System. There are a number of differences between the iPSC/2 System and the iPSC System Simulator. In fact, not all parallel programs of the iPSC/2 System and the iPSC System Simulator are compatible.

The major difference between the iPSC/2 System and the iPSC System Simulator is that the iPSC/2 System provides parallel executions of parallel programs but the iPSC System Simulator provides only sequential executions [iPSC 91]. This causes the speed of execution to be relatively slower on the iPSC System Simulator [iPSC 91]. Another difference is that the iPSC/2 System has different libraries for the host and node programs but the iPSC System Simulator has only one library. The iPSC/2 System Simulator cannot check for improper system calls issued by host and node programs [iPSC 91]. Also, the number of nodes that can be allocated by a user on the iPSC System Simulator is limited by the operating system.

2.4 Program Slicing

2.4.1 Introduction

Given a slicing criterion, program slicing produces a slice by deleting all of the statements that do not affect the values of the desired variables of the slicing criterion [Nanja 90a, 90b, and Weiser 81]. A slice is relatively smaller than the original program [Weiser 81]. When maintaining existing large programs, programmers can apply this technique to focus only on those statements which may influence the slicing criterion [Gallagher 90, 91, and Weiser 81]. Program slicing can be divided into two different types: dynamic slicing and static slicing.

2.4.2 Static Slicing

Static slicing produces a program segment that consists of those statements that may possibly interfere with the desired slicing criterion or may possibly be executed if the program is sliced according to the desired criterion [Venkatesh 91 and Weiser 84]. Static slicing produces a slice that captures all possible executions of the original program.

A slicing criterion, which was first introduced by Weiser, is an ordered pair <i, V>, where i is a specified line number of code in a program and V is a subset of variables in the program [Weiser 81]. Those smaller programs are called slices [Weiser 82]. Every slice is a subset of the original program and it keeps the original behavior of the desired variable set V before the specified i'th statement of the original program. The program in Figure 1 calculates the summation of ten numbers which range from 0 to 9. The slice in Figure 2 is a static slice of the program in Figure 1. The slicing criterion is <8, {total}>.

```
        include "stdio.h"
        int i;
        int j;
        int total;
        main()
        {
1          i = 0;
2          j = 0;
3          total = 0;
4          while ( i<10 )
           {
5            total = total + 1;
6            i = i + 1;
7            j = j + 1;
           }
8          printf("total = %d\n",total);
        }
```

Figure 1. A program for calculating the summation of ten numbers

2.4.3 Dynamic Slicing

Dynamic slicing is developed from static slicing [Agrawal 90 and Venkatesh 91]. Dynamic slicing only concentrates on a particular execution of a program to be sliced. It does not capture all the possible executions [Venkatesh 91]. Dynamic slicing can be divided into two different models: Korel and Laski's dynamic slicing and Agrawal and Horgan's dynamic slicing.

```
      main()
      {
1       i = 0;
3       total = 0;
4       while ( i<10 )
        {
5         total = total + 1;
6         i = i + 1;
        }
8       printf("total = %d\n",total);
      }
```

Figure 2. A static slice of the program in Figure 1

Korel and Laski's dynamic slicing produces an executable program segment in which all of the desired variables keep the same behavior as the original program in a particular execution of the program [Korel 88 and Venkatesh 91]. The slices retain the same behavior of the original program [Agrawal 90 and Venkatesh 91]. Korel and Laski's dynamic slicing follows a special execution which is based on a certain criterion, then it discards the unexecuted and unnecessary statements. The rest of the program is an executable slice. In Korel and Laski's dynamic slicing, the statements that affect the control flow of the original program are kept. Figure 3 shows a Korel and Laski's dynamic slice of the program in Figure 1.

```
       main()
       {
1        i = 0;
3        total = 0;
4        while ( i<10 )
         {
5          total = total + 1;
6          i = i + 1;
          }
8        printf("total = %d\n",total);
       }
```

Figure 3. A Korel and Laski's dynamic slice of the program in Figure 1

Agrawal and Horgan's dynamic slicing produces a program segment consisting of the statements that affect the desired variables in a particular execution of the program [Agrawal 90 and Venkatesh 91]. Statements are discarded if they do not affect the desired variables, no matter whether or not they are involved in the control flow of the original program. Thus, Agrawal and Horgan's dynamic slices may be inexecutable.

Compared with Korel and Laski's dynamic slicing, Agrawal and Horgan's dynamic slicing produces relatively smaller dynamic slices. The slice in Figure 4 is an Agrawal and Horgan's dynamic slice of the program in figure 1 and it includes an infinite loop on line 4.

2.4.4 Interprocedural Slicing

Weiser [Weiser 84] introduced an algorithm for interprocedural slicing in order to apply program slicing to a program that consisted of more than one procedure. If a program consists of more than one procedure, the procedure calls should exist in the program. Procedure calls may involve some parameters. Because every variable in a program has its own scope, this situation will increase the difficulty of slicing.

```
        main()
        {
  1       i = 0;
  3       total = 0;
  4       while ( i<10 )
          {
  5         total = total + 1;
          }
  8       printf("total = %d\n",total);
        }
```

Figure 4. An Agrawal and Horgan's dynamic slice of the program in Figure 1

According to Weiser's approach [Weiser 84], interprocedural slicing needs two steps. In the first step, a procedure P is sliced by a given slicing criterion. In the second step, the original slicing criterion is translated to the corresponding slicing criteria for each procedure that either calls procedure P or that is called by procedure P. The two steps alternate with each other until no further slicing criteria are produced.

The program in Figure 5 checks which element in the integer array "data" is an odd number and which element is an even number, and outputs the square of each number. By slicing this example program, a detailed demonstration of interprocedural slicing is presented below.

```
     #include <stdio.h>
     int i;
     int action;
     int data[10];
     int square;
     int dummy;
     main()
     {
 1     i = 0;
 2     action = -1;
 3     square = -1;
 4     dummy = 100;
 5     for (i=0;i<10;i++)
 6       data[i] = i + 1;
 7     for (i=0;i<10;i++)
       {
 8       action = check_odd_or_even(data[i]);
 9       if (action == 1)
10         printf("data[%d], %d, is an odd number\n",
                                            i,data[i]);
11       else if (action == 0)
12         printf("data[%d], %d, is an even number\n",
                                            i,data[i]);
13       action = -1;
14       square = get_square(data[i]);
15       printf("square of %d is %d\n",data[i],square);
       }
     }

16 get_square(number)
17 int number;
     {
18     number = number * number;
19     return (number);
     }

20 check_odd_or_even(number)
21 int number;
     {
22     number = number % 2;
23     return (number);
     }
```

Figure 5. Program example.c

In this example, the slicing criterion is <15, {square}> and the scope of the slicing criterion is procedure main(). Following Weiser's algorithm, the first step produces the slice in Figure 6. In the second step, the original slicing criterion <15, {square}> is translated to new slicing criterion <19, {number}> and the scope of this slicing criterion is procedure get_square(). After Step 2, the slice in Figure 7 is obtained. After applying Steps 1 and 2, no more new slicing criteria is produced. Finally, the program in Figure 8 is a slice of the program in Figure 5 and the slicing criterion is <15, {square}>.

```
    main()
    {
 1    i = 0;
 3    square = -1;
 5    for (i=0;i<10;i++)
 6      data[i] = i + 1;
 7    for (i=0;i<10;i++)
      {
14      square = get_square(data[i]);
      }
    }
```

Figure 6. The first slice of example.c in Figure 5

```
16 get_square(number)
17 int number;
   {
18   number = number * number;
19   return (number);
   }
```

Figure 7. The second slice of example.c in Figure 5

```
       main()
       {
 1       i = 0;
 3       square = -1;
 5       for (i=0;i<10;i++)
 6         data[i] = i + 1;
 7       for (i=0;i<10;i++)
         {
14         square = get_square(data[i]);
         }
       }

16 get_square(number)
17 int number;
       {
18       number = number * number;
19       return (number);
       }
```

Figure 8. The final slice of example.c in Figure 5

2.4.5 <u>Slicing of Separately Compiled Programs</u>

Applying program slicing to separately compiled programs is Weiser's other approach [Weiser 84]. If a program consists of several separately compiled procedures, the external procedure calls exist within the execution of the program.

Applying program slicing to separately compiled programs causes one problem. The problem is that if some actual code of those separately compiled procedures is unavailable, then a worst case must be taken into consideration [Weiser 84]. The worst-case assumption is to assume that all the global variables will be referenced and changed within the actual code of those unavailable procedures.

According to Weiser's algorithm, the process of applying program slicing to a separately compiled program requires three steps. In the first step, a procedure P is sliced by an original slicing criterion and produces a slice S. In the second step, a function is created to translate the original slicing criterion to a set of corresponding slicing criteria for all of the procedures that are involved in the slice S. If the set of slicing criteria is empty or the corresponding slicing criterion of no procedure can be translated, it means at least the actual code of one procedure is unavailable which is the worst case because all of the unavailable code must be included in the resulting slice. In the third step, all procedures present in the set that were obtained as a result of the second step are sliced by their corresponding slicing criteria to produce a set of slices. Each slice of a procedure is sliced within the scope of that procedure. For all procedures whose translated corresponding slicing criteria are unavailable in the set, their slices are obtained as a worst case assumption. Steps 2 and 3 alternate with each other until no further slicing criterion is produced.

The programs in Figures 9 and 10 are separated from the program in Figure 5. The program in Figure 9 is main.c and the program in Figure 10 is utility.c. After successful compilations of the two programs, the object files are linked together to form an executable sequential program. The result of execution of the executable program is the same as the result of execution of the program in Figure 5.

```
      #include <stdio.h>
      int i;
      int action;
      int data[10];
      int square;
      int dummy;
      main()
      {
 1      i = 0;
 2      action = -1;
 3      square = -1;
 4      dummy = 100;
 5      for (i=0;i<10;i++)
 6        data[i] = i + 1;
 7      for (i=0;i<10;i++)
        {
 8        action = check_odd_or_even(data[i]);
 9        if (action == 1)
10          printf("data[%d], %d, is an odd number\n",
                                             i,data[i]);
11        else if (action == 0)
12          printf("data[%d], %d, is an even number\n",
                                             i,data[i]);
13        action = -1;
14        square = get_square(data[i]);
15        printf("square of %d is %d\n",data[i],square);
        }
      }
```

Figure 9. Program main.c

```
1   get_square(number)
2   int number;
    {
3     number = number * number;
4     return (number);
    }

5   check_odd_or_even(number)
6   int number;
    {
7     number = number % 2;
8     return (number);
    }
```

Figure 10. Program utility.c

The slices in Figures 11 and 12 are both sliced from the program in Figure 9 by the same slicing criterion <15, {dummy}>. Differing availability of the actual code of the program in Figure 10 causes the difference between the two slices.

```
     main()
     {
 1     i = 0;
 2     action = -1;
 3     square = -1;
 4     dummy = 100;
 5     for (i=0;i<10;i++)
 6       data[i] = i + 1;
 7     for (i=0;i<10;i++)
       {
 8       action = check_odd_or_even(data[i]);
14       square = get_square(data[i]);
       }
     }
```

Figure 11. A slice of the worst case of the program in Figure 9

```
     main()
     {
 4     dummy = 100;
     }
```

Figure 12. A slice of the program in Figure 9

The worst case of slicing the program in Figure 9 by a slicing criterion <15, {dummy}> is shown in Figure 11. This worst case is caused when the actual code of the program in

Figure 10 is unavailable. If the actual code of the program in Figure 10 is available, the slice of the program in Figure 9 will be reduced to the slice given in Figure 12.

2.4.6 Limitations of Weiser's Slicing Criterion

Weiser's definition of a slicing criterion is not sufficient for distributed-memory parallel programs. Every such parallel program comprises at least two different programs, a host program and a node program. These programs are in fact two separate files. Each one of them contains a procedure main(). This causes two problems. Firstly, the domain of the line number i becomes ambiguous in a distributed- memory parallel environment. Secondly, if users want to slice distributed-memory parallel programs, a slicer cannot automatically recognize that the two programs (a host program and a node program) are related. Thirdly, a slicer has no knowledge about where to find the actual code of a separately compiled program, this may cause every slice to be made based on a worst-case assumption.

The program in Figure 13 is a node program for the iPSC/2 System. This program calculates the summation of 10 numbers and outputs the result on the screen. Figure 14 is a static slice of the program in Figure 13. The slicing criterion is <8, {subtotal}>.

If slice is executable, the static slice in Figure 14 is ambiguous. A node program is only part of a parallel program. It should be activated by a host program. Without a host program, a node program is undefined.

```
     #include "stdio.h"
     long subtotal;
     int i;
     main()
     {
1      subtotal = 0;
2      if (mynode() == 0)
3        printf("root node\n");
4      if (mynode() > 0)
       {
5        for (i=0; i<10; i++)
6          subtotal = subtotal + 10 * mynode() + i;
7        printf("%d: %d\n", mynode(), subtotal);
       }
     }
```

Figure 13. A node program for the iPSC/2 System

```
     main()
     {
1      subtotal = 0;
4      if (mynode() > 0)
       {
5        for (i=0; i<10; i++)
6          subtotal = subtotal + 10 * mynode() + i;
7        printf("%d: %d\n", mynode(), subtotal);
       }
     }
```

Figure 14. A static slice of the node program in Figure 13

CHAPTER III

EXTENDED SLICING CRITERION

3.1 Introduction

Different parallel computer architectures can handle different application software. An extended slicing criterion is introduced for applying slicing to distributed-memory parallel programs. The extended slicing criterion is based on Weiser's slicing criterion [Weiser 81] and addresses the insufficiency of Weiser's criterion for applying the operation of slicing to distributed-memory parallel programs. The rest of this Chapter discusses the definition, applications, and advantages of the extended slicing criterion.

3.2 Definition

An extended slicing criterion consists of three elements: <f, i, V>, where f specifies the end of back-tracing in parallel program slicing, i is a specified line number, and V is a subset of the variables used in the program. The definition of extended slicing criterion is based on Weiser's definition of a slicing criterion. Compared with Weiser's definition, an extended slicing criterion includes one more element, f, that is used to specify where the back-tracing ends in a distributed-memory parallel program (in the host program or in the node program). The definitions of i and V are the same as Weiser's. The slices that are produced by applying the extended slicing criterion are not necessarily executable.

3.3 Applications

3.3.1 Applied to Distributed-memory Parallel Programs

There are four possible cases to consider for slicing a distributed-memory parallel program involving host and node programs. These four cases are tabulated in Table I.

TABLE I

FOUR POSSIBLE CASES TO CONSIDER FOR SLICING A DISTRIBUTED-MEMORY PARALLEL PROGRAM INVOLVING HOST AND NODE PROGRAMS

Case	Program in the Slicing Environment	Extended Slicing Criterion	Domain of i and V
1	Host Program	<host, i, V>	Host Program
2	Host Program	<node, i, V>	Host Program
3	Node Program	<node, i, V>	Node Program
4	Node Program	<host, i, V>	Node Program

Cases 1 and 3 slice only part of a parallel program and produce slices that may not be executable. Cases 2 and 4 produce slices that form an executable parallel program.

3.3.2 Applied to Sequential Programs

In Table I, Cases 1 and 3 are special cases for applying the extended slicing criterion to sequential programs. Because there is only one program, the extended slicing criteria of Cases 1 and 3 will be the same. The slices resulting from Cases 1 and 3 are executable. For example, if the name of the program in Figure 1 is "sum.c" and the extended slicing criterion is <sum.c, 8, {total}>, then the static slice will be the same as the slice in Figure 2, which slices the program in Figure 1 by Weiser's original slicing criterion <8, {total}>.

3.4 Advantages

One advantage of the extended slicing criterion is that it can be applied to both distributed-memory parallel programs and sequential programs. The possible cases for slicing distributed-memory parallel programs were discussed in the previous section. For sequential programs, Cases 1 and 3 of Table I can be imagined as special cases for slicing sequential programs.

Another advantage of the extended slicing criterion is that the component f can refer to a set of files. For instance, for a distributed-memory parallel program, it will not be necessary for the slicing criterion to be restricted to one host program and one node program to form a parallel program. Thus for a sequential program it will not be necessary to put all of the procedures and functions into a single program.

By applying the extended slicing criterion, a slicer can automatically check every possible program file to find needed functions or procedures. If a needed function or procedure cannot be found in any of the program files that are specified by the user, a worst-case assumption will be made for the given extended slicing criterion.

For example, the programs in Figures 15 and 16 are obtained from the program in Figure 10. If the program in Figure 9 is loaded in a slicing environment and the extended slicing criterion is <{utility1.c, utility2.c}, 15, {dummy}>, then the slice will be the same as the slice in Figure 12. If the extended slicing criterion is <{main.c}, 15, {dummy}>, then the slice will be the same as the slice in Figure 11.

```
get_square(number)
int number;
{
  number = number * number;
  return (number);
}
```

Figure 15. Program utility1.c

```
check_odd_or_even(number)
int number;
{
  number = number % 2;
  return (number);
}
```

Figure 16. Program utility2.c

CHAPTER IV

PARALLEL PROGRAM SLICER (PPS)

4.1 Introduction

By applying the extended slicing criterion and static slicing technique, a slicing environment called parallel program slicer (PPS) was developed for distributed-memory parallel programs. PPS is written in C on the iPSC/2 System. Because every parallel computer architecture has its unique system calls for performing parallelism, PPS is especially developed for the iPSC/2 System. The rest of this chapter discusses how to apply the extended slicing criterion to distributed-memory parallel programs on the iPSC/2 System and the design approach of PPS.

4.2 Parallel Program Slicing on the iPSC/2 System

Message passing is a major characteristic of the iPSC/2 System [iPSC/2 90]. Without message passing between the host program and the node program, either the host program or the node program can be treated as an individual program. This special case is not a parallel program and will not be discussed in this thesis. This thesis focuses on programs with "csend" and "crecv" commands of the iPSC/2 System.

The programs in Figure 17 and Figure 18 are simple examples that are used for demonstrating how PPS slices a distributed-memory parallel program. The program in Figure 17 is a host program for calculating the summations and averages of two sets of numbers. This host program gets sixteen nodes from the iPSC/2 System and activates eleven nodes to calculate the summations and averages of two sets of numbers. The elements of the first set are from 1 to 100. The elements of the second set range from 101 to 200.

The program in Figure 18 is a node program of the host program in Figure 17. This node program is loaded into sixteen nodes of the iPSC/2 System but only eleven nodes will be activated. Except for the root node, every activated node calculates the summations of ten numbers in each set. The root node calculates the averages of the two sets of numbers and sends the results to its host program.

The static slice in Figure 19 keeps only those statements which may influence the slicing criterion. The system calls for cube control are deleted from the host program in Figure 17. This slice is not a complete host program because there is no system call to get cubes from the iPSC/2 System and no node program has been loaded. Because no node program exists, statement 18 is undefined. PPS needs special rules to handle the typical system calls of the iPSC/2 System for applying slicing to the parallel programs of the iPSC/2 System.

```
#include "stdio.h"
long total1;
long total2;
long average1;
long average2;
int data1[100];
int data2[100];
int work_node;
int i;

main()
{
  total1 = 0;
  total2 = 0;
  average1 = 0;
  average2 = 0;
  work_node = 10;
  for (i=0;i<100;i++)
  {
    data1[i] = i+1;
    data2[i] = i+101;
  }
  getcube("hsiao","16m1",NULL,0,0);
  setpid(100);
  load("node",-1,0);
  csend(1, &work_node, sizeof(int), -1, 0);
  csend(1, data1, 100*sizeof(int), -1, 0);
  csend(1, data2, 100*sizeof(int), -1, 0);
  crecv(1, &total1, sizeof(int));
  crecv(1, &average1, sizeof(int));
  crecv(1, &total2, sizeof(int));
  crecv(1, &average2, sizeof(int));
  printf("((1 + 100) * 100)/2 = %d\n",total1);
  printf("average = %d\n",average1);
  printf("((101 + 200) * 100)/2 = %d\n",total2);
  printf("average = %d\n",average2);
  killcube(-1,-1);
  relcube("hsiao");
}
```

Figure 17. A host program for calculating the summations and the averages of two sets of numbers

```
#include "stdio.h"
long subtotal1;
long subtotal2;
long average1;
long average2;
long dummy;
int data1[100];
int data2[100];
int work_node;
int i;

main()
{
  subtotal1 = 0;
  subtotal2 = 0;
  average1 = 0;
  average2 = 0;
  dummy = 0;
  crecv(1, &work_node, sizeof(int));
  crecv(1, data1, 100*sizeof(int));
  crecv(1, data2, 100*sizeof(int));
  if ((mynode() != 0) && (mynode() <= work_node))
  {
    for (i=0;i<10;i++)
    {
     subtotal1=subtotal1+data1[10*(mynode()-1)+i];
     subtotal2=subtotal2+data2[10*(mynode()-1)+i];
    }
  }
  gisum(&subtotal1,1,&dummy);
  gisum(&subtotal2,1,&dummy);
  if (mynode() == 0)
  {
    average1 = subtotal1 / work_node;
    average2 = subtotal2 / work_node;
    csend(1,&subtotal1,sizeof(int),myhost(),100);
    csend(1,&average1,sizeof(int),myhost(),100);
    csend(1,&subtotal2,sizeof(int),myhost(),100);
    csend(1,&average2,sizeof(int),myhost(),100);
  }
}
```

Figure 18. A node program for calculating the summations and the averages of two sets of numbers

```
     main()
     {
 4     average2 = 0;
18     crecv(1, &average2, sizeof(int));
22     printf("average = %d\n",average2);
     }
```

Figure 19. A static slice of the host program in Figure 17

4.3 Design Approach

4.3.1 <u>Definitions</u>

The following definitions are useful for applying slicing to distributed-memory parallel programs. Some of these definitions are introduced by Weiser [Weiser 84] and some are developed for PPS.

DEF(n) is the set of variables whose values are changed at statement n [Weiser 84].

REF(n) is the set of variables whose values are used at statement n [Weiser 84].

SYS() is the set of statements that include any cube control system call of the iPSC/2 System.

def(v) is the set of statements in which the value of variable v is changed.

ref(v) is the set of statements in which the value of variable v is used.

CS(TP, FP) is a set of ordered pairs, where TP is the line number of a "csend" command in a domain program and FP is the line number of a corresponding "crecv" command in the paired program of the domain program.

CR(TP, FP) is a set of ordered pairs, where TP is the line number of a "crecv" command in a domain program and FP is the line number of a corresponding "csend" command in the paired program of the domain program.

BT(n, v) is an ordered pair, where n specifies a desired line number and v is a variable name. It is a special case of Weiser's slicing criterion <i, V>, when the set V consists of only one member. The set BT(n, v) is a collection of statements that may influence the value of variable v.

4.3.2 Basic Rules for Applying Parallel Program Slicer (PPS) to the iPSC/2 System

Every parallel computer architecture has its unique system calls for performing parallelism. PPS focuses on the C-based parallel programs written for the iPSC/2 System. There are some basic rules about applying slicing to the parallel programs of the iPSC/2 System.

Rule 1: PPS uses the extended slicing criterion which is introduced in Chapter III.

Rule 2: PPS applies the definitions that were introduced in the previous section.

Rule 3: PPS modifies Weiser's algorithms for interprocedural slicing and slicing of separately-compiled program, and applies an analogous slicing approach to parallel programs.

Rule 4: PPS keeps nine of the twenty two of the system calls for cube control in the C version used on the iPSC/2 System, even if they do not influence the extended slicing criterion. Those system calls include attachcube(), getcube(), killcube(), killproc(), load(), relcube(), setpid(), waitall(), and waitone(). For the rest of the system calls for cube control, if they appear in REF(n) but are not involved in BT(n, v), they will not be added to the slice. The reason for keeping some of the system calls for cube control is discussed later in this section. The twenty two system calls for cube control are listed in Appendix C.

By applying Rule 4, the slice in Figure 20 can be obtained, which is a static slice of the host program in Figure 17. The extended slicing criterion is <host, 24, {average2}>, the program loaded into the PPS environment is the host program in Figure 17.

In this example, the system calls for cube control are not deleted. Compared with the slice of Figure 19, it is reasonable to show line 18 in the slice of Figure 20. First, the node program has been loaded into some nodes because lines 9 and 11 are kept. Second, when the "crecv" command is called by a host program, the corresponding node program must issue the "csend" command. Line 10 is kept for the "csend" command of the node program because an assigned host id is a necessary parameter for issuing the "csend" command. These are the reasons for Rule 4 to keep the system calls for cube control.

```
        main()
        {
 4        average2 = 0;
 9        getcube("hsiao","16m1",NULL,0,0);
10        setpid(100);
11        load("node",-1,0);
18        crecv(1, &average2, sizeof(int));
22        printf("average = %d\n",average2);
23        killcube(-1,-1);
24        relcube("hsiao");
          }
```

Figure 20. A static slice of the host program in Figure 17

4.3.3 Example

In this section, a detailed example of applying PPS to C-based distributed-memory parallel programs is given. The example parallel program consists of the host program in Figure 17 and the node program in Figure 18. In this example, the host program in Figure 17 is loaded into the PPS environment.

According to the previous Section, PPS applies Rule 1 to accept an extended slicing criterion <f, i, V>. In this example, the extended slicing criterion is <node.c, 22, {average2}>.

By applying Rule 2, Tables II and III will result which consist of the sets DEF(n), REF(n), and SYS() of the host and node programs, respectively. Tables IV and V consist of the sets def[average2], ref[average2], CS(TP,FP), CR(TP,FP), and BT[22,average2] of the host and node programs, respectively. After applying Rule 2, the four steps of Rule 3 are implemented as follows.

TABLE II

THE DEF[], REF[], AND SYS[] OF THE HOST PROGRAM IN FIGURE 17

Line No.	DEF[]	REF[]	SYS[]
1	total1		
2	total2		
3	average1		
4	average2		
5	work_node		
6	i		
7	data1	i	
8	data2	i	
9			getcube()
10			setpid()
11			load()
12		work_node	
13		data1	
14		data2	
15	total1		
16	average1		
17	total2		
18	average2		
19		total1	
20		average1	
21		total2	
22		average2	
23			killcube()
24			relcube()

TABLE III

THE DEF[], REF[], AND SYS[] OF THE NODE PROGRAM IN FIGURE 18

Line No.	DEF[]	REF[]	SYS[]
1	subtotal1		
2	subtotal2		
3	average1		
4	average2		
5	dummy		
6	work_node		
7	data1		
8	data2		
9		work_node	
10	i	i	
11	subtotal1	subtotal1 data1	
12	subtotal2	subtotal2 data2	
13	subtotal1 dummy		
14	subtotal2 dummy		
16	average1	subtotal1 work_node	
17	average2	subtotal2 work_node	
18		subtotal1	
19		average1	
20		subtotal2	
21		average2	

TABLE IV

THE def[average2], ref[average2], CS(TP, FP), CR(TP, FP), AND BT[22, average2] OF THE HOST PROGRAM IN FIGURE 17

def[v]	ref[v]	CS(TP,FP)	CR(TP,FP)	BT[22,v]
4	22	(12, 6)	(15,18)	18
18		(13, 7)	(16,19)	4
		(14, 8)	(17,20)	
			(18,21)	

TABLE V

THE def[average2], ref[average2], CS(TP, FP), CR(TP, FP), AND BT[21, average2] OF THE NODE PROGRAM IN FIGURE 18

def[v]	ref[v]	CS(TP,FP)	CR(TP,FP)	BT[21,v]
4	21	(18,15)	(6,12)	21
17		(19,16)	(7,13)	17
		(20,17)	(8,14)	15
		(21,18)		14
				12
				10
				9
				8
				6
				5
				4
				2

Step 1: We can find that statements 4 and 18 are included in the set BT(22, average2) in Table IV. The slice of the host program initially includes statement 4 and 8. In Table IV, we can see that line 18 exists in sets def[average2] and CR(18, 21). This situation requires an additional slicing on the node program and the extended slicing criterion is <host, 21, {average2}).

Step 2: We can get BT(21, average2) from Table V. The set BT(21, average2) is the initial slice of the node program. In Table V, we can see that statement 6 exists in sets BT(21, average2) and CR(6, 12). Statement 8 exists in sets BT(21, average2) and CR(8, 14). These cases need to slice the host program. The extended slicing criteria are <node.c, 12, {work_node}> and <node.c, 14, {data2}>.

Step 3: The host program is sliced according to the extended slicing criterion <node.c, 12, {work_node}>. There is no additional slicing required in this Step.

Step 4: The host program is sliced according to the extended slicing criterion <node.c, 14, {data2}>. Because there is no additional slicing required in this step, this slicing is completed except to for checking to see which system calls should be kept.

After applying Rule 3, Table VI results which consists of the slices of the host and node programs according to the four Steps of this example.

According to Rule 4, the system calls of getcube(), killcube(), load(), relcube(), and setpid() will be added to the slice of the host program. Figure 21 is the final slice of the host program. Figure 22 is the final slice of the node program. Slices of Figures 21 and 22 form an executable parallel program.

TABLE VI

THE CORRESPONDING SLICES OF THE HOST AND NODE PROGRAM AFTER EACH STEP

Step 1		Step 2		Step 3		Step 4	
host	node	host	node	host	node	host	node
4		4	2	4	2	4	2
18		18	4	5	4	5	4
			5	12	5	6	5
			6	18	6	8	6
			8		8	12	8
			9		9	14	9
			10		10	18	10
			12		12		12
			14		14		14
			15		15		15
			17		17		17
			21		21		21

```
      main()
     {
  4    average2 = 0;
  5    work_node = 10;
  6    for (i=0;i<100;i++)
       {
  8      data2[i] = i+101;
       }
  9    getcube("hsiao","16m1",NULL,0,0);
 10    setpid(100);
 11    load("node",-1,0);
 12    csend(1, &work_node, sizeof(int), -1, 0);
 14    csend(1, data2, 100*sizeof(int), -1, 0);
 18    crecv(1, &average2, sizeof(int));
 22    printf("average = %d\n",average2);
 23    killcube(-1,-1);
 24    relcube("hsiao");
   }
```

Figure 21. A static slice of the host program in Figure 17

```
      main()
      {
  2     subtotal2 = 0;
  4     average2 = 0;
  5     dummy = 0;
  6     crecv(1, &work_node, sizeof(int));
  8     crecv(1, data2, 100*sizeof(int));
  9     if ((mynode() != 0) && (mynode() <= work_node))
        {
 10       for (i=0;i<10;i++)
          {
 12        subtotal2=subtotal2+data2[10*(mynode()-1)+i];
          }
        }
 14     gisum(&subtotal2, 1, &dummy);
 15     if (mynode() == 0)
        {
 17       average2 = subtotal2 / work_node;
 21       csend(1,&average2,sizeof(int),myhost(),100);
        }
      }
```

Figure 22. A static slice of the node program in Figure 18

4.4 Advantages and Limitations of PPS

Firstly, PPS produces slices by interactively communicating with the user. Secondly, PPS can slice separately compiled parallel programs written for the iPSC/2 System. Thirdly, PPS provides load, view, compile, slice, and run functions that serve as a rudimentary environment for slicing.

PPS is developed for novice programmers. PPS handles C-based parallel programs on the iPSC/2 System and it concentrates on the "csend" and "crecv" commands. In its current implementation, PPS cannot handle the following: function calls and switch control statements.

For applying the technique of static program slicing and interprocedural slicing to a program, a slicer must traverse every character of the program in order to create sufficient information about the caller and the called procedures. This will increase the complexity of the slicer. Because PPS concentrates on the "csend" and "crecv" commands of the iPSC/2 System, it can indeed carry out the techniques of interprocedural slicing and slicing of separately-compiled programs. The "csend" and "crecv" commands of the iPSC/2 System are a pair and the relationship between a "csend" command and a "crecv" command is the same as a function call with its parameters (call by value or call by address). Because PPS focuses on "csend" and "crecv" commands, the function calls within a program are not implemented at this time.

Since switch control statements can declare their own variables, switch control statements must be treated as procedures or functions. For a C program, the actual code of a called procedure or function are separate from the caller procedure but the switch control statements are within the procedures or functions. This will increase the complexity of a slicer. Since PPS can handle "while", "for", "if", "else if" and nested loops, the switch control statement is not implemented at this time.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

In Chapter I, a brief overview of classifications of computer architectures and program slicing were introduced. The advantages of applying program slicing to programs and the difficulties of applying program slicing to parallel programs were outlined.

In Chapter II, Flynn's and Duncan's classifications of computer architectures, MIMD parallel computer architectures, the iPSC/2 System, the iPSC System Simulator, and program slicing were presented. The shared-memory and distributed-memory computer architectures were discussed. The differences between the iPSC/2 System and the iPSC System Simulator were outlined. Both static and dynamic slicing techniques were discussed. In addition, Korel and Laski's dynamic slicing and Agrawal and Horgan's dynamic slicing, interprocedural slicing, and slicing of separately-compiled program were briefly introduced. The slicing examples were present for each slicing type. The limitations of Weiser's slicing criterion were also discussed.

In Chapter III, a definition of the extended slicing criterion was introduced for applying program slicing to distributed-memory parallel programs. Four possible cases to consider for slicing a distributed-memory parallel program involving host and node programs were discussed. The applications and advantages of extended slicing criterion were presented.

In Chapter IV, the details of the static slicing design of the Parallel Program Slicer (PPS) were discussed. Because every parallel computer architecture has its unique system calls for performing parallelism, PPS is especially developed for the iPSC/2 System. Some basic rules for applying PPS to the iPSC/2 System were introduced. A detailed example was also presented. The advantages and limitations of PPS were discussed.

5.2 Future Work

PPS can be enhanced by extending its capabilities to handle function calls and switch control statements. It can be made more powerful. The element f of the extended slicing criterion (see Section 3.4) can be extended to be a set of files. This idea was discussed in Chapter III. If PPS applies dynamic slicing techniques instead of static slicing techniques, the slices will be relatively smaller than the slices that are produced by the current implementation of PPS.

REFERENCES

[Agrawal 90]
H. Agrawal and B. Horgan, "Dynamic Program Slicing," Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June 1990, pp. 246-256.

[Brandis 86]
R. C. Brandis, "IPPM Interactive Parallel Program Monitor," Technical Report No. CS/E 88-010, Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR, August 1986.

[Duncan 90]
R. Duncan, "A Survey of Parallel Computer Architectures," IEEE Computer, vol. 23, no. 2, February 1990, pp. 5-16.

[Fox 88]
G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, Solving Problems on Concurrent Processors, vol. 1, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Gallagher 90]
K. B. Gallagher, "Using Program Slicing in Software Maintenance," Ph.D. Dissertation, University of Maryland, Baltimore County, MD, 1990.

[Gallagher 91]
K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," IEEE Transactions on Software Engineering, vol. 17, no. 8, August 1991, pp. 751-761.

[Hennessy 90]
J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, Inc., Polo Alto, CA, 1990.

[iPSC/2 90]
iPSC/2 and iPSC/860 Programmer's Reference Manual, June 1990.

[iPSC/2 91]
iPSC/2 and iPSC/860 User's Guide, April 1991.

[iPSC 91]
iPSC System Simulator Manual, December 1991.

[Korel 88]
B. Korel and J. Laski, "Dynamic Program Slicing," Information Processing Letters, vol. 29, no. 3, October 1988, pp. 155-163.

[Nanja 90a]
S. Nanja, "An Interactive Debugging Tool for C Based on Program Slicing and Dicing," Master Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, May 1990.

[Nanja 90b]
S. Nanja and M. Samadzadeh, "A Slicing/Dicing-Based Debugger for C", 8th Annual Pacific Northeast Software Quality Conference, Porland, OR, October 1990, pp. 204-212.

[Osterhaug 89]
A. Osterhaug, Guide to Parallel Programming on Sequent Computer Systems, Prentice Hall, Englewood Cliffs, NJ, 1989.

[Venkatesh 91]
G. A. Venkatesh, "The Semantic Approach to Program Slicing," Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 1991, pp. 107-119.

[Weiser 81]
M. Weiser, "Program Slicing," Proceedings of the Fifth International Conference on Software Engineering, March 1981, pp. 439-449.

[Weiser 82]
M. Weiser, "Programmers Use Slices When Debugging," Communications of the ACM, vol. 25, no. 7, July 1982, pp. 446-452.

[Weiser 84]
M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. SE-10, no. 4, July 1984, pp. 352-356.

APPENDIXES

APPENDIX A

USER'S MANUAL FOR PPS

PPS is a slicer for C-based parallel programs on the iPSC/2 System. PPS applies the static slicing techniques to programs. It is designed for helping novice C parallel programmers of the iPSC/2 System.

Enter the PPS Environment

By typing "PPS", users can enter the PPS environment and the greeting screen in Figure 23 will be shown on the screen. After the prompt has been changed to be "PPS>", all the legal commands of PPS will be accepted. The following commands are legal commands of PPS.

```
      *********************************************
      *                                           *
      *                   PPS                     *
      *          A Static Program Slicing         *
      *                  Based                    *
      *         Parallel Program Slicer           *
      *********************************************
{compile | Compile | c | C}: compile
{help | Help | h | H}: help
{load | Load | l | L}: load
{run | Run | r | R}: run
{slice | Slice | s | S}: slice
{view | View | v | V}: view
{quit | Quit | q | Q}: quit
PPS>
```

Figure 23. PPS greeting screen.

l, L, load, or **LOAD:** This command loads a program into the PPS environment. After users enter this command, PPS will ask the user to enter the file name containing the program.

Example:
```
PPS>l
PPS>Enter File Name of Target Program: Input_Prog_Name
```

c, C, compile, or COMPILE: This command invokes the iPSC/2 System C compiler to compile the target program.

Example:
```
PPS>c
PPS>Compiling ...
```

v, V, view, or **VIEW:** This command displays the target or the focus program on the screen in PPS format.

r, R, run, or **RUN:** This command calls the "compile" twice for compiling the target and the focus programs. After compilations, the parallel program will be executed.

Example:
```
PPS>r
PPS>Compiling ...
    Compiling ...
```

s, S, slice, or **SLICE:** This command asks the user to enter a series of requests as the extended slicing criteria.

Example:
```
PPS>s
PPS>Enter The Focus File: Input_Prog_Name
PPS>Enter The Desired Line Number: Input_Line_No
PPS>Enter The Desired Variables: Input_Var1 [Input_Var2...]
```

The "Input_Line_No" must be less than or equal to the total number of lines of the target program. The "Input_Var1" and "Input_Var2" are variables in the target program.

h, H, help, or **HELP:** This command shows the legal commands of PPS and the syntax of each command.

q, Q, quit, or **QUIT:** This command will result in exit from PPS environment.

APPENDIX B

GLOSSARY AND TRADEMARK INFORMATION

ASIM: Simulator of the iPSC System on the Sequent S-81.

Domain Program: A program contains certain "csend" or "crecv" command that its corresponding "crecv" or "csend" command needs to be found.

Dynamic Slicing: Dynamic slicing focuses on one particular execution. It does not capture all the possible executions of a program.

Extended Slicing Criterion: A slicing criterion which extends Weiser's slicing criterion from two to three elements <f, i, V>, where f specifies the end of back-tracing in determining a program slice. The definitions of i and V are the same as Weiser's. The extended slicing criterion is used for the slicing of distributed memory parallel programs.

Focus Program: A focus program is the domain of corresponding "csend" or "crecv" command of target program.

Host Program: A program runs on the host of the iPSC/2 System and responsible for communicating with the allocated nodes in the iPSC/2 System.

MIMD: Multiple instructions and multiple data streams, MIMD, is an environment which deals with several instructions and several data streams at the same time.

Node Program: A program runs on an allocated node on the iPSC/2 System. A node program must be activated by its corresponding host program.

Parallel Program Slicer (PPS): According to a given extended slicing criterion, PPS is a program slicer for slicing C based parallel program on the iPSC/2 System.

Program Slicing: According to the data flow and control flow graph of a program and a given slicing criterion, program slicing decomposes a large program into relatively smaller programs.

Slice : A slice is an executable subset of a given program. A slice is reduced from a given program by applying a slicing criterion.

Slicing Criterion: A slicing criterion was first introduced by Weiser. It is an ordered pair <i, V>, where i is a specified line number in a program and V is a subset of the variables in the program.

Static Slicing: Static slicing produces a slice that captures all possible executions of the original program.

Target Program: A target program is the domain of the line number and the set of variables of an extended slicing criterion.

TRADEMARK INFORMATION

UNIX is a trademark of AT&T Corporation.
iPSC, Hypercube, and NX2 are trademarks of Intel Corporation.
Sequent is a trademark of Sequent Computer System, Inc.

APPENDIX C

SYSTEM CALLS FOR CUBE CONTROL ON THE iPSC/2 SYSTEM

There are twenty two system calls for cube control on the iPSC/2 System. The system calls listed in this appendix are for C version [iPSC/2 90].

1. attachcube()
2. cubeinfo()
3. flick()
4. getcube()
5. handler()
6. killcube()
7. killproc()
8. killsyslog()
9. load()
10. myhost()
11. mynode()
12. mypid()
13. newserver()
14. nodedim()
15. numnodes()
16. plogoff()
17. plogon()
18. relcube()
19. setpid()
20. setsyslog()
21. waitall()
22. waitone()

APPENDIX D

A PROGRAM AND SOME OF ITS SLICES

The following program segments are demonstrations of applying static slicing to a program. The slicing criteria in these examples are defined according to Weiser's definition. Based on different given slicing criteria, different slices are generated.

Original program:

```
     include "stdio.h"
     int i;
     int j;
     int total;
     main()
  1  {
  2     i = 0;
  3     j = 0;
  4     total = 0;
  5     while ( i<10 )
  6     {
  7       total = total + i;
  8       i = i + 1;
  9       j = j + 1;
 10     }
 11     printf("total = %d\n",total);
 12  }
```

Slice on <12, {j}).

```
     main()
  1  {
  2     i = 0;
  3     j = 0;
  5     while ( i<10 )
  6     {
  8       i = i + 1;
  9       j = j + 1;
 10     }
 12  }
```

Slice on <12, {i}>

```
        main()
      1 {
      2     i = 0;
      5     while ( i<10 )
      6     {
      8       i = i + 1;
     10     }
     12 }
```

Slice on <4,{i,j}>.

```
        main()
      1 {
      2     i = 0;
      3     j = 0;
     12 }
```

Slice on <5,{total}>.

```
        main()
      1 {
      4     total = 0;
     12 }
```

Slice on <12, {total}>.

```
        main()
      1 {
      2     i = 0;
      4     total = 0;
      5     while ( i<10 )
      6     {
      7       total = total + i;
      8       i = i + 1;
     10     }
     11     printf("total = %d\n",total);
     12 }
```

APPENDIX E

CODE FOR PPS

```
/***********************************************************
 *  FILE NAME: pps.h
 *  PURPOSE: This file contains the definitions and global
 *           declarations for Parallel Program Slicer
 *           (PPS), pps.c.
 **********************************************************/
#include <stdio.h>
#define NO 0
#define YES 1
#define TARGET 1     /* to identify a target program */
#define SCOPE 2      /* to identify a focus program */
#define HOST 1       /* to identify a host program */
#define NODE 2       /* to identify a node program */
#define def 1        /* to identify a def set */
#define ref 2        /* to identify a ref set */
#define DEF 3        /* to identify a DEF set */
#define REF 4        /* to identify a REF set */
#define SYS 5        /* to check for a iPSC system call */
#define CS 6         /* to check for a "csend" command */
#define CR 7         /* to check for a "crecv" command */
#define P_A_COM 1    /* a complete comment within a code */
#define A_COM 2      /* a complete comment without code */
#define S_COM 3      /* an incomplete comment line */
#define I_COM 4      /* an incomplete comment line */
#define E_COM 5      /* an incomplete comment line */
#define MAX_LINE 1000
#define MAX_COLUMN 180/* the maximum no. of characters in a
                       * line */
#define Screen_line 19/* the max no. of lines of a screen */
#define size 120

/**********************************************************
 * A structure definition for holding information about a
 * code segment.
 **********************************************************/
typedef struct code_str {
  int mark;            /* if this code belongs to a slice */
  int outside_main_mark; /* if this code is outside the
                          * "main" */
  int def_mark;        /* if this code is a definition */
  int comment_mark;    /* if this code is a comment line */
  int blank_line_mark;/* if this code is a blank line */
  int sys_call_mark;   /* if this code is a system call */
  int o_num;         /* the original line number */
  int n_num;         /* the new line number after packing */
  int s_num;         /* the line number of the slice */
  char code[121];    /* content of a code */
  struct code_str *pre;  /* next code structure */
  struct code_str *next; /* previous code structure */
  struct REF_DEF_str *REF_set; /* all variables that have
                                * been referenced in this
                                * code */
  struct REF_DEF_str *DEF_set; /* all variables that have
                                * been defined in this code*/
};

/***********************************************************
 * A structure definition for holding information about a
 * variable.
 ***********************************************************/
typedef struct var_str {
  char type[10];             /* variable type */
  char var_id[80];           /* variable name */
  struct var_str *l_ptr;     /* pointer of left variable */
  struct var_str *r_ptr;     /* pointer of right variable */
  struct ref_def_sys *ref_set;/* where this variable has
                               * been referenced */
  struct ref_def_sys *def_set;/* where this variable has
                               * been defined */
};

  /**********************************************************
   *  A structure definition for holding information about a
   *  program.
   **********************************************************/
typedef struct data_str {
  int type;      /* type of a program : HOST or NODE */
  int total;     /* total line number of original program */
  int n_total;   /* total line number after packing */
  int s_total;   /* total line number of slice */
  int sys_need;  /* if system call are need */
  char file[20];             /* file name of this program */
  char pid[MAX_COLUMN];      /* the process id */
  struct var_str *var_root;  /* the root of the variables */
  struct code_str *load_cp;  /* the "load" command */
  struct code_str *first;    /* first code of the program */
  struct code_str *head;     /* the code of "main" */
  struct code_str *tail;     /* last code of the program */
  struct cs_cr_str *cr;/* for holding the "crecv" command */
  struct cs_cr_str *cs;/* for holding the "csend" command */
  struct control_scope_str *c_scope_set;
                            /* the information of the
                             * control statements within the
                             * program */
  int evar_line;            /* the converted line number */
  struct esc_var_str *evar_root;
                            /* the root of variables of the
```

```
                             * extended slicing criterion */
  struct esc_var_str *evar_history;
                            /* the root of variables that
                             * have been sliced */
  struct ref_def_sys *sys;  /* for holding system calls */
};

/**********************************************************
 * A structure definition for holding information about an
 * extended slicing criterion.
 **********************************************************/
typedef struct esc_str {
  int line_no;              /* line number of an extended
                             * slicing criterion */
  struct esc_var_str *var_root;
};                          /* the root of variable of an
                             * extended slicing criterion */

/**********************************************************
 * A structure definition for holding information about a
 * variable of a extended slicing criteria.
 **********************************************************/
typedef struct esc_var_str {
  int line_no;
  struct var_str *var;      /* pointer of a variable */
  struct esc_var_str *next;/* pointer of the next variable*/
};

/**********************************************************
 * A structure definition for holding information about
 * "ref,"  "def," and "sys" sets.
 **********************************************************/
typedef struct ref_def_sys {
  int type;                 /* check if this is a ref, def,
                             * or sys set */
  struct code_str *cp;      /* pointer of a code */
  struct ref_def_sys *next; /* pointer of the next code */
};                          /* that belongs to this set */

/**********************************************************
 * A structure definition for holding information about
 * "REF" and "REF" sets.
 **********************************************************/
typedef struct REF_DEF_str {
  int type;                 /* check if this is a REF or DEF
                             * set */
  struct var_str *vp;       /* pointer of a variable */
  struct REF_DEF_str *next;/* pointer of the next variable*/
};                          /* that belongs to this set */

  /**********************************************************
   * A structure definition for holding information about
   * "csend" or "crecv" commands.
   *********************************************************/
```

```
typedef struct cs_cr_str {
  int mark;             /* for checking if this command has
                         * been referenced */
  int level;            /* belong to which control block */
  int cs_cr_type;       /* a "csend" or "crecv" command */
  char type[MAX_COLUMN];/* the selection marker of this
                         * "csend" or "crecv" command */
  char to[MAX_COLUMN];  /* if this is a "csend", record
                         * where to send  */
  struct code_str *tp;  /* pointer to the code contains this
                         * "csend" or "crecv" command */
  struct code_str *fp;
  struct group_cs_cr_str *tp_group;
                        /* those commands that must be sliced
                         * as a group */
  struct group_cs_cr_str *fp_group;
                        /* those commands that correspond to
                         * this command */
  struct control_scope_str *in_css;
                        /* this command belongs to which
                         * branch of control statement */
  struct cs_cr_str *next;
                        /* next "csend" or "crecv" command */
};

/***********************************************************
 * A structure definition for holding information about a
 * control statement.
 ***********************************************************/
struct control_scope_str {
  int type;              /* show which type of control
                          * statement */
  int level;             /* how many nested control
                          * statement outside this control
                          * statement */
  struct code_str *start;/* the code contains this control
                          * statement */
  struct code_str *left; /* the line contains "{" */
  struct code_str *end;  /* the line contains "}" */
  struct branch_control_scope_str *branch;
                         /* pointer to a branch if this
                          * control statement has branch */
  struct control_scope_str *pre;  /* the previous control
                                   * statement */
  struct control_scope_str *next;
};                       /* the next control statement */

/***********************************************************
 * A structure definition for holding information about a
 * branch control statement.
 ***********************************************************/
struct branch_control_scope_str {
  struct control_scope_str *css;
                            /* pointer to the information of
```

```
                             * this control statement */
  struct branch_control_scope_str *pre;
                            /* pointer to the previous
                             * related branch */
  struct branch_control_scope_str *next;
                            /* pointer to the next related
};                          /* branch */

/*********************************************************
 * A structure definition for holding information about
 * related "csend" and "crecv" commands.
 *********************************************************/
typedef struct group_cs_cr_str {
  struct cs_cr_str *t_fp; /* pointer to a cs_cr_str */
  struct group_cs_cr_str *pre;
                        /* pointer to the previous cs_cr_str
                         * that must be sliced as a group */
  struct group_cs_cr_str *next;
                        /* pointer to the next cs_cr_str */
};                      /* that must be sliced as a group */

struct data_str *initiate_data_str();
                  /* allocate a piece memory for data_str
                   * and initialize the members */
struct code_str *initiate_code_str();
                  /* allocate a piece memory for code_str
                   * and initialize the members */
struct var_str *initiate_var_str();
                  /* allocate a piece memory for var_str
                   * and initialize the members */
struct esc_str *initiate_esc_str();
                  /* allocate a piece memory for esc_var
                   * and initialize the members */
struct esc_var_str *initiate_esc_var_str();
                  /* allocate a piece memory for esc_var_
                   * str and initialize the members */
struct ref_def_sys *initiate_ref_def_sys_str();
                  /* allocate a piece memory for ref_def_
                   * sys and initialize the members */
struct cs_cr_str *initiate_cs_cr();
                  /* allocate a piece memory for cs_cr_
                   * str and initialize the members */
struct group_cs_cr_str *initiate_group_cs_cr();
                  /* allocate a piece memory for group_cs_
                   * cr_str and initialize the members */
struct control_scope_str *initiate_control_scope_str();
                  /* allocate a piece memory for control_
                   * scope_str and initialize the members */
struct REF_DEF_str *initiate_REF_DEF_str();
                  /* allocate a piece memory for REF_DEF_str
                   * and initialize the members */
struct branch_control_scope_str
            *initiate_branch_control_scope_str();
              /* allocate a piece memory for branch_control_
```

```
                 * scope_str and initialize the members */
struct code_str *pop_control_s();
                       /* return a pointer to code_str */
struct data_str *TP;   /* pointer to the target program */
struct data_str *SP;   /* pointer to the focus program */
struct esc_str *ESC;   /* pointer to the extended slicing
                        * criterion */
struct code_str *C_s_stack[MAX_COLUMN];
                       /* a pointer stack for checking the
                        * scopes of control statements */
int C_id_stack[MAX_COLUMN];
                       /* an integer stack for holding the
                        * type of control statements */
int TOP_id;       /* for accessing the stack of C_id_stack */
int TOP_line;     /* for accessing the stack of C_s_stack */
int LEVEL;        /* for checking the nested control
                   * statements */
int DEBUG;
char PRE_TOKEN[MAX_COLUMN];
FILE *fpps;

/************************************************************
 * FUNCTION NAME: main()
 * PURPOSE: Performs Parallel Program Slicer (PPS).
 * ALGORITHM: Interactively processes a user input query by
 *            calling appropriate functions.
 ************************************************************/
#include </usr/u/prof/samad/hsiao/demo/pps.h>
main()
{
  int STATUS = YES;
  char user_input[80];
  char temp[80];
  FILE *fo;

  DEBUG = YES;
  TOP_id = -1;
  TOP_line= -1;
  show_pps_version();
  show_long_manu();

            /* infinity loop until received quit command */
  if (DEBUG)
    fpps = fopen("ppsdata","w");
  while (STATUS)
  {
    printf("PPS> ");
    gets(user_input);
    get_parameter(temp,user_input);
                             /* user invokes pps commands */
    if (check_pps_command(temp) == YES) {
      if ((temp[0] == 'q') || (temp[0] == 'Q'))
        STATUS = NO;
      else if ((temp[0] == 'h') || (temp[0] == 'H'))
```

```
          show_long_manu();
        else if ((temp[0] == 'l') || (temp[0] == 'L'))
          load_function(user_input);
        else if ((temp[0] == 'r') || (temp[0] == 'R'))
          run_function();
        else if ((temp[0] == 's') || (temp[0] == 'S'))
          slice_function(user_input);
        else if ((temp[0] == 'v') || (temp[0] == 'V'))
          view_function(user_input);
        else if ((temp[0] == 'c') || (temp[0] == 'C'))
          compile_function(user_input);
      }
      else {               /* user ask for system routines */
       if (temp[0] == '!') {
         temp[0] = ' ';
         strcat(temp,user_input);
         system(temp);
       }
                     /* user invokes pps in pps environment */
       else if ((strcmp("pps",temp) == 0) ||
                                   (strcmp("PPS",temp) == 0)) {
         printf("Already in Parallel Program Slicing");
         printf(" Environment!\n");
       }
       else {                            /* illegal user input */
         printf("pps : Not a Valid Command : %s\n",temp);
         printf("Type 'h' or 'H' for Help.\n");
       }
      }
    }
    return_memory(TP);
    return_memory(SP);
    if (ESC)
      free(ESC);
    check_erase_temp();
    if (DEBUG)
      fclose(fpps);
}

/************************************************************
 * FUNCTION NAME: slice_function()
 * PURPOSE: Performs slicing function.
 * ALGORITHM: Checks if extended slicing criterion exists.
 *            If extended slicing criterion is not exists,
 *            asks user to input extended slicing criterion
 *            (ESC). If extended slicing criterion exists,
 *            PPS will slice the parallel program.
 ***********************************************************/
slice_function(user_input)
char *user_input;
{
  if (!TP){        /* no target program has been loaded into
                    * PPS environment */
    printf("Load first!\n");
    return;
   }
                /* target program has already been loaded */
   printf("Target Program : %s\n",(*TP).file);
   if (!ESC)       /* no given extended slicing criterion */
     get_ESC(user_input);   /* ask user to input ESC */
   if (ESC)  {     /* extended slicing criterion exists */
     printf("Scope Program  : %s\n",(*SP).file);
     printf("Line No : %d\n",(*ESC).line_no);
     make_slice();
   }
}

/*******************************************************
 * FUNCTION NAME: make_slice()
 * PURPOSE: According to a given extended slicing criterion,
 *          this function makes corresponding slices.
 * ALGORITHM: Checks internal "csend" and "crecv" pairs
 *            within the node program. Checks cross linked
 *            "csend" and "crecv" pairs between the host and
 *            node program.  According to a given extended
 *            slicing criterion, this function slices the
 *            target program and then checks if any extra
 *            slicing criterion has been generated. PPS will
 *            alternatively slice the target and focus
 *            programs until both of them have no slicing
 *            criterion left. If any "csend" or "crecv"
 *            exists, the system calls for cube control will
 *            be added to the slice .
 ******************************************************/
make_slice()
{
  struct code_str *cp;

  printf("ESC target = %s\n",(*TP).file);
  printf("ESC focus = %s\n",(*SP).file);
  if (strlen((*TP).pid) > 0)     /* copy host pid to node */
    strcpy((*SP).pid,(*TP).pid);
  else                           /* copy host pid to node */
    strcpy((*TP).pid,(*SP).pid);
  cs_cr_group_sorting(TP,(*TP).cs);/* sort csend commands */
  cs_cr_group_sorting(TP,(*TP).cr);/* sort crecv commands */
  cs_cr_group_sorting(SP,(*SP).cs);/* sort csend commands */
  cs_cr_group_sorting(SP,(*SP).cr);/* sort crecv commands */

  if ((*TP).type == NODE)        /* if TP is node program */
    internal_cs_cr_link(TP);
  else if ((*SP).type == NODE)   /* if SP is node program */
    internal_cs_cr_link(SP);
                        /* link corresponding "csend" and
                         * "crecv" commands between the host
                         * and node program */
  cross_link_cscr(TP,SP,(*TP).cs,(*SP).cr);
  cross_link_cscr(SP,TP,(*SP).cs,(*TP).cr);
```

```
            /* keep slicing host and node programs
             * until there is no more slicing criterion */
  while (((*TP).evar_root) || ((*SP).evar_root)) {
    if ((*TP).evar_root) {    /* TP has slicing criterion */
      mark_def(TP);
      mark_slice(TP,SP);
    }
    if ((*SP).evar_root) {    /* SP has slicing criterion */
      mark_def(SP);
      mark_slice(SP,TP);
    }
  }
        /* checks if there exists any grammatical error */
  add_code_for_executable(TP);
  add_code_for_executable(SP);
        /* when csend and crecv command exists in slices */
  if ((*TP).sys_need)        /* need to mark system calls */
    mark_SYS(SP,TP);
  else if ((*SP).sys_need)
    mark_SYS(TP,SP);
                    /* assigns line number of the slices */
  set_slice_number(TP);
  set_slice_number(SP);
  view_slice(TP);    /* output the slice of TP to screen */
  view_slice(SP);    /* output the slice of SP to screen */
  create_temp(TP);
  create_temp(SP);
  printf("The slice of target program %s is saved as %s\n",
                      (*TP).file,"hostslice.c");
  printf("The slice of focus program %s is saved as %s\n",
                      (*SP).file,"nodeslice.c");
  if (DEBUG)
    output_related_data();
}

/***********************************************************
 * FUNCTION NAME: set_slice_number()
 * PURPOSE: Assigns the line number of each code in slice.
 ***********************************************************/
set_slice_number(dp)
struct data_str *dp;
{
  int i = 1;
  struct code_str *cp;

  cp = (*dp).head;
  while (cp) {     /* traverses every code of the program */
    if ((*cp).mark) {        /* the code belongs to slice */
      (*cp).s_num = i;
      i++;
    }
    cp = (*cp).next;
  }
  (*dp).s_total = i - 1;
```

```
}

/*************************************************************
 * FUNCTION NAME: output_slice()
 * PURPOSE: Outputs the slice of the program.
 * ALGORITHM: Traverses every code of the program and
 *            outputs the code that is marked as a slice.
 ************************************************************/
output_slice(dp)
struct data_str *dp;
{
  struct code_str *cp;
  struct esc_var_str *evar;

  fprint_line();   /* print a line of "=" on output file */
  fprintf(fpps,"A Slice of %s\n\n",(*dp).file);
  cp = (*dp).head;
  while(cp) {                       /* traverses every code */
    if ((*cp).mark > 0) /* the code belongs to the slice */
      fprintf(fpps,"%3d %3d  %s",(*cp).o_num,
                                 (*cp).n_num,(*cp).code);
    cp = (*cp).next;
  }
}

/*************************************************************
 * FUNCTION NAME: mark_sys()
 * PURPOSE: Marks the codes of system calls for cube control.
 * ALGORITHM: Traverses the structure of (*dp).sys and marks
 *            every one in the linked list.
 ************************************************************/
mark_SYS(dp,fp)
struct data_str *dp,*fp;
{
  struct ref_def_sys *rdsp;

  mark_def(dp);
  rdsp = (*dp).sys;
              /* traverses from the header of the list */
  if (rdsp)
    (*dp).evar_line = (*dp).n_total;
  while (rdsp) {          /* check every code in the list */
    (* ((*rdsp).cp) ).mark = YES;        /* mark the code */
    get_necessary_var(dp,(*rdsp).cp);
    rdsp = (*rdsp).next;
  }
  mark_slice(dp,fp);
}

/*************************************************************
 * FUNCTION NAME: mark_def()
 * PURPOSE: Marks the codes of definitions.
 * ALGORITHM: Traverses every code of the program and if it
 *            is a declaration of definition then marks it.
 *********************************************************/
mark_def(dp)
struct data_str *dp;
{
  struct code_str *cp;

  cp = (*dp).head;       /* check from the beginning of the
                          * program */
  while ((cp) && (cp != (*dp).first)) {
                         /* only checks those codes that are
                          * outside the "main" */
    if ((*cp).def_mark)  /* this code is a definition */
      (*cp).mark = YES;
    cp = (*cp).next;
  }
}

/*********************************************************
 * FUNCTION NAME: add_code_for_executable()
 * PURPOSE: Makes a control statement to fit grammatical
 *          requirements.
 * ALGORITHM: If a control statement has been marked but the
 *            "exit" or "break" in the scope did not be
 *            marked, then PPS will marks it.  If there is
 *            no any line in a slice, marks the codes of
 *            "main", "{", and "}".
 *********************************************************/
add_code_for_executable(dp)
struct data_str *dp;
{
  struct control_scope_str *scp;
  struct code_str *start_cp,*end_cp;
  char token[MAX_COLUMN],code[MAX_COLUMN];

  scp = (*dp).c_scope_set;
                             /* traverse the scope of every
                              * control statement */
  while (scp)   {
    start_cp = (*scp).start;
    end_cp = (*scp).end;
                             /* the slice is empty */
    if (((*scp).type == 1) && ((*start_cp).mark != YES))  {
      (*start_cp).mark = YES;
      (*end_cp).mark = YES;
      if ((*scp).left)
        (*((*scp).left)).mark = YES;
      return;
    }
                            /* this control statement is in
                             * the slice */
    if ((*start_cp).mark)   {
                       /* only one  statement in the scope
                        * of the control statement */
      if (((*end_cp).n_num - (*start_cp).n_num) == 1)    {
          if (!(*end_cp).mark)  {    /* the scope is empty */
            strcpy(code,(*end_cp).code);
            get_parameter(token,code);
                          /* check if "exit" or "break" exists
                           * in the control statement */
            if ((strcmp(token,"exit") != 0)  &&
                   (strcmp(token,"break") != 0))
               strcpy((*end_cp).code,";\n");
            (*end_cp).mark = YES;
          }
        }
      }
      scp = (*scp).next;
    }
  }

  /************************************************************
   * FUNCTION NAME: mark_slice()
   * PURPOSE: Marks the codes that fit the slicing criterion.
   * ALGORITHM: Checks every variable in the linked list of
   *            (*dp).evar_root.  Marks every code that has
   *            ever defined the variable, if the line number
   *            of the code is less than the number of
   *            (*dp).evar_line. If a code has been marked in
   *            this function and it is a "crecv" command, it
   *            needs to cross check its "csend" command in dp
   *            or fp.
   ************************************************************/
  mark_slice(dp,fp)
  struct data_str *dp,*fp;
  {
    struct var_str *vp;
    struct code_str *cp,*crcp,*cscp;
    struct cs_cr_str *cscrp;
    struct group_cs_cr_str *gcrp,*gcsp;
    struct esc_var_str *evp;
    struct ref_def_sys *rdsp;

    evp = (*dp).evar_root;
    while (evp)              /* check every variable in the */
    {                        /* extended slicing criterion  */
      rdsp = (*((*evp).var)).def_set;
      while (rdsp)   {       /* check every line that has ever
                              * defined the variable */
        cp = (*rdsp).cp;
        if ((*cp).n_num <= (*dp).evar_line){
                             /* before desired line number */
          (*cp).mark = YES;
          if ((*dp).type == NODE)
            (*dp).sys_need = YES;
          cscrp = (*dp).cr;
          while (cscrp)  {   /* check every "csend" command */
            crcp = (*cscrp).tp;
            gcrp = (*cscrp).fp_group;
          while (gcrp){ /* check related "crecv" command */
            cscp = (*((*gcrp).t_fp)).tp;
            if (crcp == cp){ /* corresponding to "cscrp" */
              (*cscp).mark = YES;
                              /* cross link between host and
                               * node program */
              if ((*((*gcrp).t_fp)).cs_cr_type != NODE)
                extra_slicing(fp,cscp);
              else            /* internal link within a node
                               * program */
                extra_slicing(dp,cscp);
            }
            gcrp = (*gcrp).next;   /* check next "crecv" */
          }
          cscrp = (*cscrp).next;
          check_necessary_scope(dp,cp); /* check if it is */
        }                      /* within a control statement */
      }
      rdsp = (*rdsp).next;              /* check next line */
    }
    evp = (*evp).next;      /* check the next variable */
    (*dp).evar_root = evp;  /* release the used variable */
  }
  (*dp).evar_line = -1;
}

/************************************************************
 * FUNCTION NAME: extra_slicing()
 * PURPOSE: Checks if any extra variable is needed for
 *          slicing a given slicing criterion.
 * ALGORITHM: Every variable that has been referenced in the
 *            code but has never been linked to (*dp).evar_
 *            root, will be added to the (*dp).evar_var as a
 *            new variable for slicing. If the line number
 *            of cscp is greater than the (*dp).evar_line,
 *            resets (*dp).evar_line to be the line number.
 ************************************************************/
extra_slicing(dp,cscp)
struct data_str *dp;
struct code_str *cscp;
{
  int n_line;
  struct var_str *vp;
  struct REF_DEF_str *Rp;

  n_line = check_line((*dp).c_scope_set,(*cscp).n_num);
  if (n_line > (*dp).evar_line)
    (*dp).evar_line = n_line;
  check_necessary_scope(dp,cscp);/* check variables in
                                  * control statements */
  Rp = (*cscp).REF_set;  /* for every variable that has
                          * been referenced in this line */
  while (Rp)  {
    vp = (*Rp).vp;   /* check from the root of variables */
    if (check_evar_history(dp,vp))  {
      link_esc_var(&((*dp).evar_history),vp,n_line);
      link_esc_var(&((*dp).evar_root),vp,n_line);
    }
    Rp = (*Rp).next;
  }
}

/************************************************************
 * FUNCTION NAME: check_evar_history()
 * PURPOSE: Checks if a variable has already been referenced
 *          as a variable of slicing criterion.
 * ALGORITHM: Traverses the linked list of (*dp).evar_root
 *            and checks if vp exists.
 ***********************************************************/
check_evar_history(dp,vp)
struct data_str *dp;
struct var_str *vp;
{
  struct esc_var_str *evp;

  evp = (*dp).evar_history;
  while (evp)  {  /* traverse every variable in the list */
    if ((*evp).var == vp)     {
                  /* the variable has been used */
      if ((*evp).line_no >= (*dp).evar_line)
        return(NO);
    }
    evp = (*evp).next;
  }
  return (YES);
}

/************************************************************
 * FUNCTION NAME: check_necessary_scope()
 * PURPOSE: Checks related branch of a control statement.
 * ALGORITHM: If cp is within a branch of a control
 *            statement, all the related branch will be
 *            checked.
 ***********************************************************/
check_necessary_scope(dp,cp)
struct data_str *dp;
struct code_str *cp;
{
  struct control_scope_str *scp,*tscp;
  struct branch_control_scope_str *bcss;
  struct code_str *start_cp,*end_cp,*left_cp;

  scp = (*dp).c_scope_set;
  while (scp)  {      /* check every control scope */
    start_cp = (*scp).start;
    end_cp = (*scp).end;
    left_cp = (*scp).left;
             /* cp is within a control scope */
    if (((*cp).n_num >= (*start_cp).n_num) &&
        ((*cp).n_num <= (*end_cp).n_num))    {
      bcss = (*scp).branch;
             /* go to the head of the control statement */
      while ((*bcss).pre)
        bcss = (*bcss).pre;
      while (bcss)  {
             /* check the branch of the control statement */
        tscp = (*bcss).css;
        start_cp = (*tscp).start;
        end_cp = (*tscp).end;
        left_cp = (*tscp).left;
        (*start_cp).mark = YES;
             /* single line control statement */
        if (((*end_cp).n_num - (*start_cp).n_num) > 1)
          (*end_cp).mark = YES;
        if (left_cp)
          (*left_cp).mark = YES;
        get_necessary_var(dp,cp);
        get_necessary_var(dp,start_cp);
        bcss = (*bcss).next;
      }
    }
    scp = (*scp).next;
  }
}

/************************************************************
 * FUNCTION NAME: get_necessary_var()
 * PURPOSE: To get those variables that are involved in a
 *          control statement.
 * ALGORITHM: All the variables that have been referenced or
 *            defined within the code will be linked to
 *            (*dp).evar_root, if it has never been linked
 *            to (*dp).evar_root.
 ***********************************************************/
get_necessary_var(dp,cp)
struct data_str *dp;
struct code_str *cp;
{
  int n_line;
  struct REF_DEF_str *Rp,*Dp;
  struct var_str *vp;

  n_line = check_line((*dp).c_scope_set,(*cp).n_num);
  if (n_line > (*dp).evar_line)
    (*dp).evar_line = n_line;
  Dp = (*cp).DEF_set;
  while (Dp) {      /* check every variable in the line */
    vp = (*Dp).vp;
                    /* if the variable is not in the set of
                     * variables of slicing criterion */
    if (check_evar_history(dp,vp)) {
      link_esc_var(&((*dp).evar_history),vp,(*dp).evar_line);
      link_esc_var(&((*dp).evar_root),vp,(*dp).evar_line);
    }
    Dp = (*Dp).next;
  }
  Rp = (*cp).REF_set;
  while (Rp) {       /* check every variable in the line */
    vp = (*Rp).vp;
                     /* if the variable is not in the set of
                      * variables of slicing criterion */
    if (check_evar_history(dp,vp)) {
      link_esc_var(&((*dp).evar_history),vp,(*dp).evar_line);
      link_esc_var(&((*dp).evar_root),vp,(*dp).evar_line);
    }
    Rp = (*Rp).next;
  }
}

/************************************************************
 * FUNCTION NAME: cs_cr_group_sorting()
 * PURPOSE: Checks those related "csend" or "crecv" commands.
 * ALGORITHM: If a "csend" or "crecv" command is within a
 *            branch of a control statement, all the
 *            corresponding "csend" or "crecv" will be
 *            treated as a group.
 ***********************************************************/
cs_cr_group_sorting(dp,cscrp)
struct data_str *dp;
struct cs_cr_str *cscrp;
{
  struct control_scope_str *css;

  css = (*dp).c_scope_set;
  while (css)  {
    if ((*css).type == 6)    /* when meet a "else" branch */
                              /* check its "if" statement */
      check_in_scope(dp,cscrp,css);
    css = (*css).next;
  }
}

/***********************************************************
 * FUNCTION NAME: set_cs_cr_level()
 * PURPOSE: Checks a "csend" or "crecv" command belongs to
 *          which control statement and which nested loop.
 * ALGORITHM: Traverses every "csend" or "crecv" command of
 *            the linked list, ocscrp, and checks it belongs
 *            to which control statement.
 **********************************************************/
set_cs_cr_level(dp,ocscrp)
struct data_str *dp;
struct cs_cr_str *ocscrp;
{
  struct control_scope_str *css,*tcss;
  struct cs_cr_str *cscrp;
  struct code_str *cp,*start,*end;

  cscrp = ocscrp;
  css = (*dp).c_scope_set;
  while (cscrp)  {     /* check every "csend" command */
    tcss = css;
                       /* check if the "csend" command is
                        * in the scope of some control
                        * statement */
    while (tcss)    {  /* check every control statement */
      start = (*tcss).start;
      end = (*tcss).end;
      cp = (*cscrp).tp;
                       /* within a control scope */
      if (((*cp).n_num <= (*end).n_num) &&
          ((*cp).n_num >= (*start).n_num))  {
        (*cscrp).level = (*tcss).level;
        (*cscrp).in_css = tcss;
      }
      tcss = (*tcss).next;
    }
    cscrp = (*cscrp).next;
  }
}

/************************************************************
 * FUNCTION NAME: internal_cs_cr_link()
 * PURPOSE: Checks those "csend" and "crecv" commands that
 *          are paired within a node program.
 * ALGORITHM: Traverses every "csend" command of the node
 *            program and checks if it is paired with any
 *            "crecv" in the node program.
 ***********************************************************/
internal_cs_cr_link(dp)
struct data_str *dp;
{
  struct cs_cr_str *ocsp,*ocrp,*tcrp,*csp,*crp;
  struct group_cs_cr_str *gcstp,*gcrtp,*mcptp;

  ocsp = (*dp).cs;
  ocrp = (*dp).cr;
  while (ocsp)  {
    gcstp = (*ocsp).tp_group;
    while ((*gcstp).pre != NULL)
                          /* go to the head of list */
      gcstp = (*gcstp).pre;
    csp = (*gcstp).t_fp;
    tcrp = ocrp;
    while (tcrp) {        /* check every crecv command */
      gcrtp = (*tcrp).tp_group;
                          /* go to the head of list */
      while ((*gcrtp).pre != NULL)
        gcrtp = (*gcrtp).pre;
      crp = (*gcrtp).t_fp;
```

```
              /* an internal "csend" and "crecv" pair */
      if ((strcmp((*csp).type,(*crp).type) == 0) &&
          (strcmp((*csp).to,(*dp).pid) != 0) &&
          ((*csp).in_css != (*crp).in_css)) {
        (*tcrp).cs_cr_type = NODE;
        (*ocsp).cs_cr_type = NODE;
        link_cs_cr_fp(tcrp,ocsp,gcrtp,gcstp);
      }
      tcrp = (*tcrp).next;
    }
    ocsp = (*ocsp).next;
  }
}

/*********************************************************
 * FUNCTION NAME: link_cs_cr_fp()
 * PURPOSE: Links related "csend" and "crecv" command.
 * ALGORITHM: Checks the corresponding "csend" and "crecv"
 *            commands.
 *********************************************************/
link_cs_cr_fp(tcrp,ocsp,gcrtp,gcstp)
struct cs_cr_str *tcrp,*ocsp;
struct group_cs_cr_str *gcrtp,*gcstp;
{
  struct group_cs_cr_str *temp,*tg,*new,*target,*fp_start;
  struct cs_cr_str *csrp,*tfp;
  struct code_str *cp;

  target = (*ocsp).fp_group;
  if (!target)  {       /* fp_group is empty */
    temp = gcrtp;
    while (temp){       /* check every crecv command of a
                         * group the fp_group of crecv
                         * command is not empty */
      if ((*((*temp).t_fp)).fp_group)       {
        tg = (*((*temp).t_fp)).fp_group;
        while (tg)    {
                        /* check every csend command */
          tfp = (*tg).t_fp;
          if ((*ocsp).in_css == (*tfp).in_css)
            return;
          tg = (*tg).next;
        }
      }
      temp = (*temp).next;
    }
  }
  else  {              /* fp_group is empty */
    while (target) {   /* check every csend of a group */
      temp = gcrtp;
      while (temp)  { /* check every crecv of a group */
        tfp = (*temp).t_fp;
        while (tfp) { /* check every csend of a group */
          if ((*((*target).t_fp)).in_css == (*tfp).in_css)
```

```
            return;
          tfp = (*tfp).next;
        }
        temp = (*temp).next;
      }
      target = (*target).next;
    }
  }

  target = (*ocsp).fp_group;
  if (target)          /* go to the head of list */
    while((*target).next)
      target = (*target).next;
  temp = gcrtp;
  while (temp) {       /* check every crecv commands */
    new = initiate_group_cs_cr();
    (*new).t_fp = (*temp).t_fp;
    if (!(*ocsp).fp_group)
                       /* initialize the list */
      (*ocsp).fp_group = new;
    else {             /* link to the end of the list */
      (*target).next = new;
      (*new).pre = target;
    }
    target = new;
    temp = (*temp).next;
  }
  target = (*tcrp).fp_group;
  if (target)
                      /* go to the head of lest */
    while ((*target).next)
      target = (*target).next;
  temp = gcstp;
  while (temp) {      /* check every csend commands */
    new = initiate_group_cs_cr();
    (*new).t_fp = (*temp).t_fp;
    if (!(*tcrp).fp_group)
                      /* initialize the list */
      (*tcrp).fp_group = new;
    else {            /* link to the end of the list */
      (*target).next = new;
      (*new).pre = target;
    }
    target = new;
    temp = (*temp).next;
  }
}

/************************************************************
 * FUNCTION NAME: check_in_scope()
 * PURPOSE: To check which csend or crecv commands are a
 *          group.
 * ALGORITHM: Checks every csend and crecv command in an
 *            "else" and "else if" control statements. If
 *            every branch has a csend or crecv command,
 *            these csend or crecv commands will be put in
 *            a group.
 ***********************************************************/
check_in_scope(dp,cscrp,ocss)
struct data_str *dp;
struct cs_cr_str *cscrp;
struct control_scope_str *ocss;
{
  struct control_scope_str *tcss,*css;
  struct branch_control_scope_str *bcss,*nbcss;
  struct cs_cr_str *tcscrp,*t_fp;
  struct code_str *cp,*start,*end;
  struct group_cs_cr_str *gcscrp;

  gcscrp = NULL;
  tcscrp = cscrp;
  bcss = (*ocss).branch;
  while((*bcss).pre) /* go to the head of the branch list */
    bcss = (*bcss).pre;
  while (tcscrp)  {          /* check every csend command */
    nbcss = bcss;
    while (nbcss)     {               /* check every branch */
      tcss = (*nbcss).css;
      start = (*tcss).start;
      end = (*tcss).end;
      cp = (*tcscrp).tp;
                               /* a csend command in a branch */
      if ((*tcscrp).level == (*tcss).level)   {
        if (((*cp).n_num <= (*end).n_num) &&
             ((*cp).n_num >= (*start).n_num))    {
          if (!gcscrp) /* check first item in csend group */
            gcscrp = (*tcscrp).tp_group;
          else if (gcscrp)    {
            t_fp = (*gcscrp).t_fp;
            if (((*t_fp).in_css != tcss) &&
                 (strcmp((*t_fp).to,(*dp).pid) != 0) &&
                 (strcmp((*t_fp).type,(*tcscrp).type) == 0)) {
              (*gcscrp).next = (*tcscrp).tp_group;
              (*((*tcscrp).tp_group)).pre = gcscrp;
            }
          }
        }
      }
      nbcss = (*nbcss).next;
    }
    tcscrp = (*tcscrp).next;
  }
}

/***********************************************************
 * FUNCTION NAME: cross_link_cscr()
 * PURPOSE: Checks those "csend" and "crecv" commands that
 *          are paired between the host and node programs.
 * ALGORITHM: Traverses every "csend" command of the host
 *            program and checks if it is paired with any
 *            "crecv" in the node program. Traverses every
 *            "csend" command of the node program and checks
 *            if it is paired with any "crecv" in the host
 *            program.
 *********************************************************/
cross_link_cscr(dp1,dp2,target_csp,target_crp)
struct data_str *dp1,*dp2;
struct cs_cr_str *target_csp,*target_crp;
{
  int status = YES;
  int result,match;
  struct var_str *rvp1,*rvp2;
  struct cs_cr_str *tpcsp,*spcsp;
  struct cs_cr_str *otpcrp,*ospcrp,*temp_spcrp;
  struct group_cs_cr_str *gspcrp,*gtpcsp;

  tpcsp = target_csp;
  ospcrp = target_crp;
  while (status)  {              /* checking every "csend" */
    if ((tpcsp) && ((*tpcsp).cs_cr_type != NODE))    {
      temp_spcrp = ospcrp;
      while (temp_spcrp)      {
        if (strcmp((*tpcsp).type,(*temp_spcrp).type) != 0) {
          check_var((*tpcsp).type,(*dp1).var_root,&rvp1,
                                                  &result);
         check_var((*temp_spcrp).type,(*dp2).var_root,&rvp2,
                                                  &result);
          if (strcmp((*rvp1).type,(*rvp2).type) == 0)
             match = YES;
          else if (strcmp((*rvp2).type,"-1") == 0)
            match = YES;
          else
            match = NO;
        }
        else
          match = YES;
                          /* the csend command and the crecv
                           * command are a pair */
        if (match == YES)       {
                                /* interconnection between
                                 * host and node program */
          if ((*temp_spcrp).cs_cr_type != NODE)         {
            gspcrp = (*temp_spcrp).tp_group;
                    /* go to the head of the crecv list */
            while ((*gspcrp).pre)
              gspcrp = (*gspcrp).pre;
            gtpcsp = (*tpcsp).tp_group;
                    /* go to the head of the csend list */
            while ((*gtpcsp).pre)
              gtpcsp = (*gtpcsp).pre;
            link_cs_cr_fp(temp_spcrp,tpcsp,gspcrp,gtpcsp);
          }
        }
        temp_spcrp = (*temp_spcrp).next;
      }
    }
    tpcsp = (*tpcsp).next;
    if (tpcsp == NULL)
      status = NO;
  }
}

/************************************************************
 * FUNCTION NAME: check_f_t()
 * PURPOSE: Gets user input of the name and type of a file
 *          that will be loaded into PPS environment.
 * ALGORITHM: If the user input do not include the name and
 *            type of a file to be loaded, asks the user to
 *            input required information. After got the
 *            required information, the information will be
 *            stored in the structure of data_str.
 ***********************************************************/
check_f_t(dp,file_name,file_type)
struct data_str *dp;
char *file_name,*file_type;
{
  int status = YES;
  int counter = 0;
  char user_input[MAX_COLUMN];

  if (strlen(file_name) == 0)
  {
                       /* no file name has been specified */
    while ((status) && (counter < 3)) /* may try 3 times */
    {
      counter ++;
      if (dp == TP)         /* load target program */
        printf("Target Program : ");
      else if (dp == SP)    /* load focus program */
        printf("Scope Program : ");
      gets(user_input);
      get_parameter((*dp).file,user_input);
      if (strlen((*dp).file) >0)
                            /* got specified file name */
        status = NO;
    }
    if (status)  {         /* user did not input file name */
     printf("Load failed: no file name has been specified\n");
     return (NO);
    }
  }
  else                      /* store the file name */
    strcpy((*dp).file,file_name);
  if (strlen(file_type) == 0)  {
    get_parameter(file_type,user_input);
    if (strlen(file_type) == 0)  { /* missing file type */
```

```
      printf("Program Type (h : host | n : node) : ");
      gets(file_type);
    }
  }
  if (strcmp(file_type,"h") == 0)       /* a host program */
    (*dp).type = HOST;
  else if (strcmp(file_type,"n") == 0) /* a node program */
    (*dp).type = NODE;
  else
    return(NO);                         /* load failed */
  return(YES);
}

/************************************************************
 *  FUNCTION NAME: erase_temp()
 *  PURPOSE: Erases those temporary files that are created
 *           by compile_prog().
 *  ALGORITHM:  If the default file names exist, erases them.
 ***********************************************************/
erase_temp()
{
  FILE *fo;

                                /* check if ppshost.c exists */
  if (fo = fopen("ppshost.c","r")) {
    fclose(fo);
    system("rm ppshost.c");      /* erase temporary file */
  }
                                /* check if ppshost.c exists */
  if (fo = fopen("ppsnode.c","r")) {
    fclose(fo);
    system("rm ppsnode.c");      /* erase temporary file */
  }
                                /* check if ppshost.c exists */
  if (fo = fopen("ppshost","r")) {
    fclose(fo);
    system("rm ppshost");        /* erase temporary file */
  }
}

/************************************************************
 * FUNCTION NAME: create_temp()
 * PURPOSE: Creates temporary files for compile_prog().
 * ALGORITHM: If the type of the program is "h", the slice
 *            will be saved as "ppshost.c".  If the type of
 *            the program is "n", the slice will be saved
 *            as "ppsnode.c". For making executable slices,
 *            this function will check the load command of
 *            the host program and change the name of node
 *            program to be "ppsnode".
 ***********************************************************/
create_temp(dp)
struct data_str *dp;
{
  int i,j;
  FILE *fo,*fs;
  struct code_str *cp,*load_cp;
  struct ref_def_sys *rdsp;
  char temp_load[MAX_COLUMN],temp[MAX_COLUMN];

  if ((*dp).type == HOST) {/* create temporary host file */
    fs = fopen("hostslice.c","w");
    fo = fopen("ppshost.c","w");
  }
  else if ((*dp).type == NODE) {
                            /* create temporary node file */
    fs = fopen("nodeslice.c","w");
    fo = fopen("ppsnode.c","w");
  }
  cp = (*dp).head;
  while (cp) {        /* output slice to temporary file */
    if ((*cp).mark == YES)  {    /* the code is in slice */
      fprintf(fs,"%s",(*cp).code);
      if ((*dp).load_cp == cp) { /* replace load command */
        strcpy(temp,(*cp).code);
        i = 0;
        j = 0;
        while (temp[i] != '"')
          temp_load[j++] = temp[i++];
        temp_load[j++] = temp[i++];
        temp_load[j] = '\0';
        strcat(temp_load,"ppsnode"); /*load default file */
        while (temp[i] != '"')
          i++;
        j += 7;
        while (i <= strlen(temp))
          temp_load[j++] = temp[i++];
        fprintf(fo,"%s",temp_load);
      }
      else
        fprintf(fo,"%s",(*cp).code);
    }
    cp = (*cp).next;
  }
  fclose(fo);
  fclose(fs);
}

/************************************************************
 * FUNCTION NAME: check_outside_main()
 * PURPOSE: Finds the declarations of definitions and global
 *          variables.
 * ALGORITHM: Traverses every code before a "main" is found.
 *            Checks if the traversed code is a definition
 *            of a variable. If the code is a declaration of
 *            definition, mark the def_mark of the code.
 ***********************************************************/
check_outside_main_var(dp)
```

```
struct data_str *dp;
{
  int outside_flag = YES;
  struct code_str *cp;
  char code[MAX_COLUMN],token[MAX_COLUMN],
       value1[MAX_COLUMN],value2[MAX_COLUMN];

  cp = (*dp).head;
                    /* skip all blank and comment lines */
  while ((*cp).n_num < 0)
    cp = (*cp).next;
                    /* before the "main" has been found */
  while ((cp) && (outside_flag)) {
    strcpy(code,(*cp).code);
    get_parameter(token,code);
    if (token[0] == '#')  {
                    /* check if it is a definition */
      (*cp).def_mark = YES;
      if (strcmp(token,"#define")== 0) {
                    /* a definition has been found */
        get_parameter(token,code);
        get_parameter(value1,code);
        get_parameter(value2,code);
        while ((strcmp(value2,"/*") != 0) &&
                                 (strlen(value2) > 0)){
          strcat(value1,value2);
          get_parameter(value2,code);
        }
        link_var_drive(dp,token,value1);
      }
    }
    if (strcmp(token,"main") == 0){  /* "main" was found */
      (*dp).first = cp;
      outside_flag = NO;
    }
    (*cp).outside_main_mark = outside_flag;
    is_var(dp,cp,token,code);  /* check global variables */
    cp = (*cp).next;
                    /* skip all blank and comment lines */
    while ((*cp).n_num < 0)
      cp = (*cp).next;
  }
}

/**********************************************************
 * FUNCTION NAME: check_local_var()
 * PURPOSE: Checks the declarations of local variables.
 * ALGORITHM: Traverses every code after the code that
 *            contains the "main".
 **********************************************************/
check_local_var(dp)
struct data_str *dp;
{
  int var_flag = YES;
  struct code_str *cp;
  char code[MAX_COLUMN],token[MAX_COLUMN];

  cp = (*dp).first;
                    /* skip all blank and comment lines */
  while ((*cp).n_num < 0)
    cp = (*cp).next;
                    /* for checking some same type variables
                     * have been declared in same line */
  while ((cp) && (var_flag))  {
    strcpy(code,(*cp).code);
    get_parameter(token,code);
    if (!is_var(dp,cp,token,code))   /* no more variables */
      var_flag = NO;
    cp = (*cp).next;
                    /* skip all blank and comment lines */
    while ((*cp).n_num < 0)
      cp = (*cp).next;
  }
}

/************************************************************
 * FUNCTION NAME: get_ESC()
 * PURPOSE: Gets extended slicing criterion.
 * ALGORITHM: Checks if the user_input includes the required
 *            file name and file type. If the required
 *            information are not match, asks the user to
 *            input required information.
 ************************************************************/
get_ESC(user_input)
char *user_input;
{
  int status = NO;
  char file_type[MAX_COLUMN];
  char file_name[MAX_COLUMN];
  struct esc_var_str *escvar;

  SP = initiate_data_str();
  ESC = initiate_esc_str();
  get_parameter(file_name,user_input);
  get_parameter(file_type,user_input);
  if ((check_f_t(SP,file_name,file_type)) && (read_input(SP)))
  {
    view_function((*SP).file);
    if (get_sc())
      status = YES;
  }
  if (!status) {/* load failed and release allocated memory */
    return_memory(SP);
    free(ESC);
    SP = NULL;
    ESC = NULL;
    return (NO);
  }
  status = NO;
  if (ESC)  {     /* already got extended slicing criterion */
    if (DEBUG) {
      fprintf(fpps,"Target Program: %s\n",(*TP).file);
      fprintf(fpps,"Extended Slicing Criterion: <%s, %d, {",
                                (*SP).file,(*ESC).line_no);
    }
    escvar = (*ESC).var_root;
    while (escvar)    {        /* output desired variables */
      printf("var %s\n",(*((*escvar).var)).var_id);
      if (DEBUG) {
        if (!status) {
          status = YES;
          fprintf(fpps,"%s",(*((*escvar).var)).var_id);
        }
        else
          fprintf(fpps,", %s",(*((*escvar).var)).var_id);
      }
      escvar = (*escvar).next;
    }
    if (DEBUG)
      fprintf(fpps,"}>\n");
  }
  return (YES);
}

/************************************************************
 * FUNCTION NAME: get_sc()
 * PURPOSE: Gets the line number and variables of the
 *          extended slicing criterion.
 * ALGORITHM: Checks if the desired line number is less than
 *            the total lines of the program and if the
 *            desired variables are exists in the program.
 *            If the line number and variables are legal,
 *            the information will be stored in ESC.
 ************************************************************/
get_sc()
{
  int result = NO;
  int status = NO;
  char var_id[MAX_COLUMN];
  char input_var[MAX_COLUMN];
  struct var_str *rvp;

  printf("Line Number (%d to %d) : ",
          (*((*TP).first)).n_num,(*TP).n_total);
  scanf("%d",&((*ESC).line_no));
  getchar();
                             /* user input illegal line number */
  if (((*ESC).line_no > (*TP).n_total) ||
                ((*ESC).line_no < (*((*TP).first)).n_num)) {
   printf("%d : Unavailable line number.\n",(*ESC).line_no);
   return(NO);
  }
  (*TP).evar_line = check_line((*TP).c_scope_set,
                                               (*ESC).line_no);
  trav_var((*TP).var_root);     /* output legal variables */
  printf("Desired Variables : ");
  gets(input_var);
  get_parameter(var_id,input_var);
  while (strlen(var_id) != 0)  {     /* check user input */
    result = NO;
    check_var(var_id,(*TP).var_root,&rvp,&result);
    if (result)
      link_esc_var(&((*ESC).var_root),rvp,(*TP).evar_line);
    get_parameter(var_id,input_var);
  }
  if ((*ESC).var_root == NULL){      /* no legal variable */
    printf("input variable is not existed\n");
    printf("Slice function failed\n");
    return (NO);                     /* slice filed */
  }
  (*TP).evar_root = (*ESC).var_root;
  (*TP).evar_history = (*ESC).var_root;
  return(YES);
}

/************************************************************
 * FUNCTION NAME: check_line()
 * PURPOSE: To convert a line number to be a legal slicing
 *          line number.
 * ALGORITHM: If line_no is within a control statement, the
 *            line_no will be convert to be the line number
 *            of the end code of the scope of the control
 *            statement.
 ***********************************************************/
check_line(css,line_no)
struct control_scope_str *css;
int line_no;
{
  int convert_line;
  struct control_scope_str *temp;

  convert_line = line_no;
  temp = css;
  while (temp)  {     /* check every control statement */
    if ((*temp).type != 1)      /* do not check "main" */
                      /* if line_no is within the scope */
      if ((convert_line >= (*((*temp).start)).n_num) &&
          (convert_line <= (*((*temp).end)).n_num))
                     /* convert line_no to be max number */
        if (convert_line < (*((*temp).end)).n_num)
           convert_line = (*((*temp).end)).n_num;
    temp = (*temp).next;
  }
  return convert_line;
}
```

```
/*********************************************************
 * FUNCTION NAME: load_function()
 * PURPOSE: Opens a file and links all the code as a linked
 *          list.
 * ALGORITHM: If any program has already been loaded, this
 *            function will release the loaded program and
 *            allow the user to input a new program.
 *********************************************************/
load_function(user_input)
char *user_input;
{
  char file_type[MAX_COLUMN];
  char file_name[MAX_COLUMN];
  char response[MAX_COLUMN];

  if (TP)  {   /* target program has already been loaded */
    printf("%s : is already loaded.\n",(*TP).file);
    printf("Reload a new target program (Y/N) ?  ");
    gets(response);
               /* ask if user want to reload new program */
    if ((response[0] == 'N') || (response[0] == 'n'))
      return;
    else if ((response[0] == 'Y')||(response[0] == 'y')) {
      return_memory(TP);
      return_memory(SP);
      if (ESC)
        free(ESC);
      TP = NULL;
      SP = NULL;
      ESC = NULL;
    }
  }
  TP = initiate_data_str();
  get_parameter(file_name,user_input);
  get_parameter(file_type,user_input);
  if ((check_f_t(TP,file_name,file_type))&&(read_input(TP)))
    view_function((*TP).file);
  else  {                                /* load failed */
    return_memory(TP);
    TP = NULL;
  }
}

/*********************************************************
 * FUNCTION NAME: link_sys()
 * PURPOSE: Links the code that contains the system calls
 *          for cube control.
 * ALGORITHM: If the list is empty, initialize the list. If
 *            the list is not empty, traverses to the end
 *            of the list and link the new code.
 *********************************************************/
link_sys(dp,cp)
struct data_str *dp;
struct code_str *cp;
```

```
{
  struct ref_def_sys *rp,*temp;

  rp = initiate_ref_def_sys_str(SYS);
  (*rp).cp = cp;
  if ((*dp).sys == NULL)           /* list is empty */
    (*dp).sys = rp;
  else  {                          /* list is not empty */
    temp = (*dp).sys;
    while ((*temp).next != NULL)   /* traverse to the end */
      temp = (*temp).next;
    (*temp).next = rp;
  }
}

/************************************************************
 * FUNCTION NAME: packing()
 * PURPOSE: Resets the line number.
 * ALGORITHM: By skipping all the blank lines and comment
 *            lines, this function will reset the line
 *            numbers of the rest codes.
 ************************************************************/
packing(dp)
struct data_str *dp;
{
  int i = 1;
  struct code_str *cp;

  cp = (*dp).head;
  while (cp)  {       /* check every code */
                      /* skip all blank and comment lines */
    if (((*cp).comment_mark<2) && !((*cp).blank_line_mark))
      (*cp).n_num = i++;
    cp = (*cp).next;
  }
  (*dp).n_total = i - 1;       /* assign new total number */
}

/************************************************************
 * FUNCTION NAME: check_comment_blank()
 * PURPOSE: Marks blank lines and comment lines.
 * ALGORITHM: Applies string processing techniques.
 ************************************************************/
check_comment_blank(dp)
struct data_str *dp;
{
  int i;
  int blank_line_flag = YES;
  int comment_start = 0;
  int comment_flag1 = NO;
  int comment_flag2 = NO;
  struct code_str *cp;

  cp = (*dp).head;
  while(cp != NULL)  {
    i = 0;
    blank_line_flag = YES;
    while (((*cp).code[i]!='\n')&&(i<=strlen((*cp).code))){
                        /* start of comment has been found */
      if (((*cp).code[i]=='/')&&((*cp).code[i+1] == '*')) {
        if (!comment_flag1) {
          comment_start = (*cp).o_num;
          comment_flag1 = YES;
        }
        i++;
      }                   /* end of comment has been found */
      if (((*cp).code[i] == '*')&&((*cp).code[i+1] == '/'))
        comment_flag2 = YES;
      if ((*cp).code[i] != ' ')
        blank_line_flag = NO;
      i++;
    }
                          /* both start and end of comment
                           * have been found */
    if ((comment_flag1) && (comment_flag2)) {
      mark_comment_codes(dp,comment_start,(*cp).o_num);
      comment_flag1 = NO;
      comment_flag2 = NO;
    }
    if (blank_line_flag)
      (*cp).blank_line_mark = YES;
    cp = (*cp).next;
  }
}

/************************************************************
 * FUNCTION NAME: mark_comment_code()
 * PURPOSE: Marks the comment lines.
 * ALGORITHM: Applies string processing techniques.
 ************************************************************/
mark_comment_codes(dp,start,end)
struct data_str *dp;
int start,end;
{
  struct code_str *cp;
  char token[MAX_COLUMN],code[MAX_COLUMN];

  cp = (*dp).head;
  while ((*cp).o_num < start)
    cp = (*cp).next;
  if (start == end)  {             /* one line comment */
    strcpy(code,(*cp).code);
    get_parameter(token,code);
    if (strcmp(token,"/*") == 0)   /* all comment */
      (*cp).comment_mark = A_COM;
    else                           /* not all comment */
      (*cp).comment_mark = P_A_COM;
  }
  else  {
    if (start < end)    {          /* more than one line */
      (*cp).comment_mark = S_COM;
      cp = (*cp).next;
                                 /* seeking for end of comment */
       while ((cp != NULL) && ((*cp).o_num < end)) {
         (*cp).comment_mark = I_COM;
         cp = (*cp).next;
       }
      (*cp).comment_mark = E_COM;
    }
  }
}

/***********************************************************
 * FUNCTION NAME: view_function()
 * PURPOSE: To show the contents of a program to screen.
 * ALGORITHM: Checks if the file exists in PPS environment.
 *            Formats the screen as a 19-line screen.
 **********************************************************/
view_function(file_name)
char *file_name;
{
  int i = 1;
  int line = Screen_line;
  int page_no;
  int status = NO;
  int pps_error = NO;
  char error_message[80];
  char choose[80],temp[MAX_COLUMN],temp_file[MAX_COLUMN];
  struct data_str *dp;

  if (!TP)  {        /* no file has been loaded into PPS */
    printf("Loaded first!\n");
    return;
  }

  if (strlen(file_name) != 0)  {
                     /* check user input file name */
    strcpy(temp,file_name);
    get_parameter(temp_file,temp);
    if (strcmp((*TP).file,temp_file) == 0)
      dp = TP;       /* default file is the target file */
    else if ((SP!=NULL)&&(strcmp((*SP).file,temp_file)==0))
      dp = SP;
    else    {        /* file did not exists in PPS */
      printf("File %s is not loaded in PPS\n",file_name);
      return;
    }
  }
  else
      dp = TP;
  page_no = (*dp).n_total % line;
  if (page_no != 0)
```

```
    page_no = ((*dp).n_total / line) + 1;
  else
    page_no = (*dp).n_total / line;
  while (!status)  {         /* user did not enter "quit" */
    show_screen_header(dp,&pps_error,1,i,page_no,
                                          error_message);
    cat(dp,((i-1)*line)+1,i*line);
    show_screen_bottom(&status,&pps_error,&i,&page_no,
                                          error_message);
  }
}

/**********************************************************
 * FUNCTION NAME: read_input()
 * PURPOSE: Reads a program from disk and links its code
 *          as a linked list.
 * ALGORITHM: Checks if the file exists in disk. Reads
 *            every code in the file. Creates necessary
 *            information sets for the file.
 **********************************************************/
read_input(dp)
struct data_str *dp;
{
  FILE *fin;
  char buffer[121];
  int i = 1;

  fin = fopen((*dp).file,"r");
  if (fin == NULL)  {                /* file not exists */
    printf("%s : File not existed\n",(*dp).file);
    return(NO);
  }
  else  {                            /* file exists */
    LEVEL = -1;
                         /* read file until end of file */
    while (fgets(buffer,size,fin) != (char *) NULL)    {
      link_code(i,buffer,dp);
      i++;
    }
    (*dp).total = i - 1;
    fclose (fin);

    check_comment_blank(dp);
                        /* mark blank and comment lines */
    packing(dp);        /* reset line number of the codes */
    check_outside_main_var(dp);
                        /* mark definitions & global var */
    check_local_var(dp);          /* mark local variables */
    create_ref_def_sys_cs_cr(dp);
                 /* create ref, def, SYS, CS, and CR sets */
    create_control_scope(dp); /* create control scope set */
    check_branch(dp);   /* check branch control statement */
    set_cs_cr_level(dp,(*dp).cs);     /* set nested level */
    set_cs_cr_level(dp,(*dp).cr);     /* set nested level */
    return(YES);
  }
}

/**********************************************************
 * FUNCTION NAME: check_branch()
 * PURPOSE: Checks the scope of related branch control
 *          statement.
 * ALGORITHM: Checks if the control statement is "else" or
 *            "else if". For every "else" and "else if",
 *            checks those closed "if" statement.
 **********************************************************/
check_branch(dp)
struct data_str *dp;
{
  struct control_scope_str *css,*pre_css,*target,*branch;
  struct branch_control_scope_str *bcss,*nbcss;

  css = (*dp).c_scope_set;
  while (css)  {         /* check every control statement */
    if ((*css).type > 5)  {  /* when "else" and "else if" */
      target = NULL;
      pre_css = (*css).pre;
      while (pre_css)      {
        if ((*pre_css).type == 5)        {
          if ((*((*pre_css).end)).n_num <
                                (*((*css).start)).n_num){
            if (target == NULL)
              target = pre_css;
            else  {
              if ( (*((*target).end)).n_num <
                                (*((*pre_css).end)).n_num)
                target = pre_css;
            }
          }
        }
        pre_css = (*pre_css).pre;
      }
      nbcss = (*css).branch;
      bcss = (*target).branch;
      while ((*bcss).next != NULL) /* link related branch */
        bcss = (*bcss).next;
      (*bcss).next = nbcss;
      (*nbcss).pre = bcss;
    }
    css = (*css).next;
  }
}

/**********************************************************
 * FUNCTION NAME: link_code()
 * PURPOSE: Links a code structure to the specified program.
 * ALGORITHM: If the code list is empty, initializes the
 *            code list. If the code list is not empty,
 *             links the code to the end of the list.
 ***********************************************************/
link_code(i,buffer,dp)
int i;
char *buffer;
struct data_str *dp;
{
  struct code_str *cp;

  if ((*dp).head == NULL) {  /* the code list is empty */
    (*dp).head = initiate_code_str();
    (*dp).tail = (*dp).head;
    (*((*dp).head)).next = (*dp).head;
    (*((*dp).head)).o_num = i;
    strcpy((*((*dp).head)).code,buffer);
  }
  else  {                  /* the code list is not empty */
    cp = initiate_code_str();
    strcpy((*cp).code,buffer);
    (*cp).pre = (*dp).tail;
    (*cp).o_num = i;
    (*((*dp).tail)).next = cp;
    (*dp).tail = cp;
  }
}

/************************************************************
 * FUNCTION NAME: cat()
 * PURPOSE: Finds the code that fits the start number and
 *          calls trav() to output code.
 ***********************************************************/
cat(dp,start,end)
struct data_str *dp;
int start;
int end;
{
  int i = 0;
  struct code_str *cp;

  if ((*dp).head == NULL)     /* the code list is empty */
    return;
  else  {                  /* the code list is not empty */
    if ((*((*dp).head)).n_num == start)
      trav((*dp).head,end);
    else    {
      cp = (*((*dp).head)).next;
      while (start > (*cp).n_num)
        cp = (*cp).next;
      trav(cp,end);
    }
    if (end > (*dp).n_total)    {
      for (i=0;i<(end-(*dp).n_total);i++)
     printf("\n");
    }
  }
}

/*************************************************************
 * FUNCTION NAME: cat_slice()
 * PURPOSE: Finds the code of the slice that fit the start
 *          number and calls trav_slice() to output code.
 ************************************************************/
cat_slice(dp,start,end)
struct data_str *dp;
int start;
int end;
{
  int i = 0;
  struct code_str *cp;

  if ((*dp).head == NULL)     /* the code list is empty */
    return;
  else  {                    /* the code list is not empty */
    if ((*((*dp).head)).s_num == start)
      trav_slice((*dp).head,end);
    else    {
      cp = (*((*dp).head)).next;
      while (start > (*cp).s_num)
        cp = (*cp).next;
      trav_slice(cp,end);
    }
    if (end > (*dp).s_total)    {
      for (i=0;i<(end-(*dp).s_total);i++)
     printf("\n");
    }
  }
}

/************************************************************
 * FUNCTION NAME: trav_slice()
 * PURPOSE: Outputs the actual code of a code structure.
 * ALGORITHM: Traverses a code list. If the line number of
 *            s_num is less than the value of "end",
 *            outputs the code.
 ************************************************************/
trav_slice(cp,end)
struct code_str *cp;
int end;
{
  if ((cp == NULL) || (end < (*cp).s_num))
    return;
  else  {
    if ((*cp).s_num > 0)
    printf("%3d %s",(*cp).n_num,(*cp).code);
    trav_slice((*cp).next,end);
  }
}
/************************************************************
```

```
 * FUNCTION NAME: trav()
 * PURPOSE: Outputs the actual code of a code structure.
 * ALGORITHM: Recursively traverses a code structure. If the
 *            line number of the code is less than the
 *            value of "end", outputs the code.
 *******************************************************/
trav(cp,end)
struct code_str *cp;
int end;
{
  if ((cp == NULL) || (end < (*cp).n_num))
    return;
  else  {
    if ((*cp).n_num > 0)
    printf("%3d %s",(*cp).n_num,(*cp).code);
    trav((*cp).next,end);
  }
}

/*******************************************************
 * FUNCTION NAME: initiate_data_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          data_str and initializes all the members of the
 *          structure.
 *******************************************************/
struct data_str *initiate_data_str()
{
  struct data_str *p;

                            /* allocate a piece of memory */
  p = (struct data_str *) malloc (sizeof (struct data_str));
  strcpy((*p).pid,"");                (*p).type = 0;
  (*p).total = 0;                     (*p).n_total = 0;
  (*p).sys_need = NO;                 (*p).evar_line = -1;
  (*p).evar_root = NULL;              (*p).evar_history = NULL;
  (*p).var_root = NULL;               (*p).s_total = 0;
  (*p).load_cp = NULL;                (*p).c_scope_set = NULL;
  (*p).first = NULL;                  (*p).head = NULL;
  (*p).tail = NULL;                   (*p).cr = NULL;
  (*p).cs = NULL;                     (*p).sys = NULL;
  return(p);
}

/*******************************************************
 * FUNCTION NAME: initiate_code_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          code_str and initializes all the members of the
 *          structure.
 *******************************************************/
struct code_str *initiate_code_str()
{
  struct code_str *p;
                            /* allocate a piece of memory */
  p = (struct code_str *) malloc (sizeof (struct code_str));
  (*p).mark = NO;                  strcpy((*p).code,"");
  (*p).def_mark = NO;              (*p).outside_main_mark = NO;
  (*p).blank_line_mark = NO;       (*p).comment_mark = NO;
  (*p).sys_call_mark = NO;         (*p).o_num = -1;
  (*p).n_num = -1;                 (*p).s_num = -1;
  (*p).pre = NULL;                 (*p).next = NULL;
  (*p).REF_set = NULL;             (*p).DEF_set = NULL;
  return(p);
}

/*********************************************************
 * FUNCTION NAME: initiate_var_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          var_str and initializes all the members of the
 *          structure.
 *********************************************************/
struct var_str *initiate_var_str()
{
  struct var_str *p;

                                   /* allocate a piece of memory */
  p = (struct var_str *) malloc (sizeof (struct var_str));
  strcpy ((*p).type,"");              strcpy ((*p).var_id,"");
  (*p).l_ptr = NULL;                  (*p).r_ptr = NULL;
  (*p).ref_set = NULL;                (*p).def_set = NULL;
  return(p);
}

/*********************************************************
 * FUNCTION NAME: initiate_esc_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          esc_str and initializes all the members of the
 *          structure.
 *********************************************************/
struct esc_str *initiate_esc_str()
{
  struct esc_str *p;

                                   /* allocate a piece of memory */
  p = (struct esc_str *) malloc (sizeof (struct esc_str));
  (*p).line_no = -1;
  (*p).var_root = NULL;
  return(p);
}

/*********************************************************
 * FUNCTION NAME: initiate_REF_DEF_str()
 * PURPOSE: Allocates a piece of memory for structure REF_
 *          DEF_str and initializes all the members of the
 *          structure.
 *********************************************************/
struct REF_DEF_str *initiate_REF_DEF_str()
{
  struct REF_DEF_str *p;
```

```
                              /* allocate a piece of memory */
  p = (struct REF_DEF_str *) malloc
                                (sizeof (struct REF_DEF_str));
  (*p).vp = NULL;
  (*p).next = NULL;
  return(p);
}

/*************************************************************
 * FUNCTION NAME: initiate_esc_var_str()
 * PURPOSE: Allocate a piece of memory for structure esc_
 *          var_str and initializes all the members of the
 *          structure.
 *************************************************************/
struct esc_var_str *initiate_esc_var_str()
{
  struct esc_var_str *p;

                              /* allocate a piece of memory */
  p = (struct esc_var_str *) malloc
                                (sizeof (struct esc_var_str));
  (*p).var = NULL;
  (*p).next = NULL;
  return(p);
}

/*************************************************************
 * FUNCTION NAME: initiate_ref_def_sys_str()
 * PURPOSE: Allocates a piece of memory for structure ref_
 *          def_sys_str and initializes all the members of
 *          the structure.
 *************************************************************/
struct ref_def_sys *initiate_ref_def_sys_str(type)
int type;
{
  struct ref_def_sys *p;

                              /* allocate a piece of memory */
  p = (struct ref_def_sys *) malloc
                                (sizeof (struct ref_def_sys));
  (*p).type = type;
  (*p).cp = NULL;
  (*p).next = NULL;
  return(p);
}

/*************************************************************
 * FUNCTION NAME: initiate_group_cs_cr()
 * PURPOSE: Allocates a piece of memory for structure group_
 *          cs_cr_str and initializes all the members of the
 *          structure.
 *************************************************************/
struct group_cs_cr_str *initiate_group_cs_cr()
{
  struct group_cs_cr_str *p;

  p=(struct group_cs_cr_str *)malloc
                          (sizeof(struct group_cs_cr_str));
  (*p).t_fp = NULL;
  (*p).pre = NULL;
  (*p).next = NULL;
  return(p);
}

/************************************************************
 * FUNCTION NAME: initiate_cs_cr()
 * PURPOSE: Allocates a piece of memory for structure cs_cr_
 *          str and initializes all the members of the
 *          structure.
 ***********************************************************/
struct cs_cr_str *initiate_cs_cr()
{
  struct cs_cr_str *p;

                          /* allocate a piece of memory */
  p = (struct cs_cr_str *) malloc
                              (sizeof (struct cs_cr_str));
  (*p).cs_cr_type = -1;                (*p).level = -1;
  (*p).tp = NULL;                      (*p).fp = NULL;
  strcpy((*p).to,"");                  (*p).tp_group = NULL;
  (*p).fp_group = NULL;                (*p).next = NULL;
  return(p);
}

/************************************************************
 * FUNCTION NAME: initiate_control_scope_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          control_scope_str and initializes all the
 *          members of the structure.
 ***********************************************************/
struct control_scope_str *initiate_control_scope_str()
{
  struct control_scope_str *p;

                          /* allocate a piece of memory */
  p = (struct control_scope_str *) malloc
                     (sizeof (struct control_scope_str));
  (*p).level = -1;                   (*p).type = -1;
  (*p).start = NULL;                 (*p).left = NULL;
  (*p).end = NULL;                   (*p).branch = NULL;
  (*p).next = NULL;                  (*p).pre = NULL;
  return(p);
}

/************************************************************
 * FUNCTION NAME: initiate_branch_control_scope_str()
 * PURPOSE: Allocates a piece of memory for structure
 *          branch_control_scope_str and initializes all the
 *          members of the structure.
 ***********************************************************/
struct branch_control_scope_str
                        *initiate_branch_control_scope_str()
{
  struct branch_control_scope_str *p;

                           /* allocate a piece of memory */
  p = (struct branch_control_scope_str *) malloc
               (sizeof (struct branch_control_scope_str));
  (*p).css = NULL;
  (*p).pre = NULL;
  (*p).next = NULL;
}

/**********************************************************
 * FUNCTION NAME: show_pps_version()
 * PURPOSE: To output the version of PPS.
 ***********************************************************/
show_pps_version()
{
  printf("     ***************************************\n");
  printf("     *                                     *\n");
  printf("     *                PPS                  *\n");
  printf("     *      A Static Program Slicing       *\n");
  printf("     *               based                 *\n");
  printf("     *       Parallel Program Slicer       *\n");
  printf("     *                                     *\n");
  printf("     *             Version 1.0             *\n");
  printf("     *                                     *\n");
  printf("     ***************************************\n");
}

/**********************************************************
 * FUNCTION NAME: show_long_manu()
 * PURPOSE: To output legal PPS commands.
 ***********************************************************/
show_long_manu()
{
  printf("    {compile | Compile | c | C} : compile\n");
  printf("    {help | Help | h | H} : help\n");
  printf("    {load | Load | l | L} : load\n");
  printf("    {run | Run | r | R} : run\n");
  printf("    {slice | Slice | s | S} : slice\n");
  printf("    {view | View | v | V} : view\n");
}

/**********************************************************
 * FUNCTION NAME: check_pps_command()
 * PURPOSE: To check if the string "str" is in the array of
 *          command_available.
 ***********************************************************/
check_pps_command(str)
char *str;
```

```
{
  int i;
  static char command_available[][10] =
               {"s","S","slice","Slice","l","L","load","Load",
                "h","H","help","Help","q","Q","quit","QUIT",
                "r","R","run","Run","v","V","view","VIEW",
                "c","C","compile","Compile"};

  for (i=0;i<28;i++)
    if (strcmp(command_available[i],str) == 0)
      return(YES);
  return(NO);
}

/************************************************************
 * FUNCTION NAME: pps_sys_command()
 * PURPOSE: To check if the string "str" is in the array of
 *          reserved_sys.
 ************************************************************/
pps_sys_command(str)
char *str;
{
  int i;
  static char reserved_sys[][11] =
            {"attachcube","getcube","killcube","killproc",
             "load","relcube","setpid","waitall","waitone"};

  for (i=0;i<9;i++)
    if (strcmp(reserved_sys[i],str) == 0)
      return(i+1);
  return(NO);
}

/***********************************************************
 * FUNCTION NAME: trav_var()
 * PURPOSE: To output all the legal variables of a program.
 * ALGORITHM: Recursively traverses a var_str and outputs
 *            every variable.
 ***********************************************************/
trav_var(vp)
struct var_str *vp;
{
  if ((*vp).l_ptr != NULL)
    trav_var((*vp).l_ptr);
  if (check_is_var((*vp).type))
    printf("type = %-7s  var id =  %s\n",
                                 (*vp).type,(*vp).var_id);
  if ((*vp).r_ptr != NULL)
    trav_var((*vp).r_ptr);
}

/************************************************************
 * FUNCTION NAME: create_control_scope()
 * PURPOSE: To create a list of control statements.
 * ALGORITHM: Checks every line of a program, except for the
 *            comment and blank lines. Calls function
 *            check_control_statement() to check what kind
 *            of control statement. If it is a "main", "do",
 *            "while", "if", "else", and "for" statement,
 *            the code pointer will be pushed into the
 *            C_s_stack and the type of control statement
 *            will be pushed into C_id_stack. When reach a
 *            "{", its code pointer will be pushed into the
 *            C_s_stack.  When reach a "}", the "{" and
 *            control statement will be popped out from the
 *            C_s_stack, and the type of control statement
 *            will be popped out from the C_id_stack.  These
 *            three code pointers form a control scope. If
 *            a control statement contains only one line of
 *            code, the pointer of "{" will be NULL.
 *******************************************************/
create_control_scope(dp)
struct data_str *dp;
{
  int type,counter,tid;
  char token[MAX_COLUMN],code[MAX_COLUMN];
  char temp_token[MAX_COLUMN],temp_code[MAX_COLUMN];
  struct code_str *cp,*tcp,*pcp,*lcp;

  cp = (*dp).first;        /* start from the line "main()" */

                           /* skip comment and blank line */
  while (((*cp).n_num < 0) && (cp))
    cp = (*cp).next;
                           /* check every code of a program
                            * except the comment and blank
                            * line */
  while (cp)  {
    strcpy (code,(*cp).code);
    get_parameter(token,code);  /* get token from code */
    if (strlen(token) > 0) {
         /* check if the token is a control statement */
      type = check_control_statement(token);
                       /* if token is a control statement
                        * the type will > 0 */
      if (type) {
         if (type == 6) {  /* an "else" statement */
         strcpy(temp_code,code);
         get_parameter(temp_token ,temp_code);
                     /* check if an "else if" statement */
         if (strcmp(temp_token,"if") == 0)    {
           get_parameter(temp_token ,temp_code);
           type = 7;
         }
        }
        LEVEL ++;           /* increase the nested level */
        push_control_id(type);
                 /* store the type of control statement */
        push_control_s(cp); /* store the pointer of code */
        link_control_scope_drive(dp,type,cp,NULL,NULL);
                              /* initialize a control scope
                               * structure */
        check_scope(dp,code,&cp,type);
                              /* check if a one code  */
      }                       /* control statement */
      else if (strcmp(token,"}") == 0) {
                              /* when reach a "}" */
        lcp = pop_control_s();
        pcp = pop_control_s();
        tid = pop_control_id();
        link_control_scope_drive(dp,tid,pcp,cp,lcp);
        LEVEL --;         /* decrease the nested level */
      }
    }
                          /* skip comment and blank lines */
    cp = (*cp).next;
    while (((*cp).n_num < 0) && (cp))
     cp = (*cp).next;
  }
}

/************************************************************
 * FUNCTION NAME: link_control_scope_drive()
 * PURPOSE: To link a control statement to the list of
 *          control statements.
 * ALGORITHM: Calls search_scope_set() to get the desired
 *            control statement. Assigns the nested level of
 *            the control scope. Assigns the type, start,
 *            end, and left pointer to the control scope.
 ***********************************************************/
link_control_scope_drive(dp,type,start,end,left)
struct data_str *dp;
int type;
struct code_str *start,*end,*left;
{
  struct control_scope_str *css;
  struct branch_control_scope_str *bcss;

  search_control_set(dp,&css,type,start);
             /* css will be the returned control scope */
  if ((*css).level == -1)/* level has not been assigned */
    (*css).level = LEVEL;
  if (end != NULL)       /* end has not been assigned */
    (*css).end = end;
  else  {       /* first call for link a control scope */
    (*css).type = type;  /* assign the type */
    (*css).start = start;/* assign the start code pointer */
   bcss = initiate_branch_control_scope_str();
                         /* get a branch structure */
    (*bcss).css = css;   /* assign itself to the branch */
    (*css).branch = bcss;/* assign the branch pointer */
  }
  if (left != NULL)      /* "{" has been found */
    (*css).left = left;  /* assign the code pointer */
}

/***********************************************************
 * FUNCTION NAME: search_control_set()
 * PURPOSE: To find a specified control statement.
 * ALGORITHM: Traverses the linked list of control scope
 *            to check if any control scope matches the type
 *            and start code pointer.
 ***********************************************************/
search_control_set(dp,css,type,start)
struct data_str *dp;
struct control_scope_str **css;
int type;
struct code_str *start;
{
  struct code_str *cp;
  struct control_scope_str *tcss,*pre_css;

  if ((*dp).c_scope_set == NULL) {
                        /* control scope list is empty */
    *css = initiate_control_scope_str();
    (*dp).c_scope_set = *css;
    return;
  }
  else  {               /* control scope list is not empty */
    tcss = (*dp).c_scope_set;
    while (tcss != NULL) {   /* when not found */
      pre_css = tcss;        /* keep the previous one */
      cp = (*tcss).start;
                      /* compare the start code pointer
                       * and the type */
      if ((start == cp)&&(type==(*tcss).type)) {
                     /* matched */
        *css = tcss;   /* assign css to be the structure */
        return;        /* when found */
      }
      else             /* try the next one */
        tcss = (*tcss).next;
    }
    if (tcss == NULL) { /* no matched structure */
      *css = initiate_control_scope_str();/* get new one */
      (*pre_css).next = *css; /* link the previous one */
      (**css).pre = pre_css;  /* link itself to the list */
    }
  }
}

/***********************************************************
 * FUNCTION NAME: check_scope()
 * PURPOSE: To find the start and end of a control
 *          statement.
 * ALGORITHM: Traverses the linked code list to find the
 *            "{" and "}" code pointer.
 ******************************************************/
check_scope(dp,code,cp,type)
struct data_str *dp;
char *code;
struct code_str **cp;
int type;
{
  int i,len,status1,status2,status3,counter,tid;
  char token[MAX_COLUMN];
  struct code_str *tcp,*pcp;

  status1 = YES;
  status2 = YES;
  status3 = YES;
  i = 0;
  counter = 0;
  len = strlen(code);        /* original length of code */
                             /* "do" and "else" have no
                              * conditional statement */
  if ((type != 1) && (type != 6))
    counter = check_counter(code);
                        /* control statement has been read */
  if (counter == 0)   {
                        /* search "{" and check if there is
                         * only one statement in the scope of
                         * the control statement */
    while ((status2) && (status1))    {
      get_parameter(token,code);
                        /* "{" has been found */
      if (strcmp(token,"{") == 0)      {
        push_control_s(*cp);    /* store the code pointer */
        status2 = NO;
      }
      else if (!status3)
        status1 = NO;
                          /* search the next line for "{" */
      else if ((strlen(token) == 0) && (status3))      {
        *cp = (**cp).next;
                          /* skip blank and comment lines */
        while (((**cp).n_num < 0) && (*cp))
          *cp = (**cp).next;
        if (!(*cp))       /* end of file */
          status2 = NO;
        else      /* copy the original code for accessing */
          strcpy(code,(**cp).code);
        status3 = NO;
      }
    }
  }
  if (status2)   {         /* one statement within scope */
    pcp = pop_control_s();
    tid = pop_control_id();
                           /* link the control scope */
    link_control_scope_drive(dp,tid,pcp,*cp,NULL);
    LEVEL--;               /* reduce the nested level */
  }
}

/*********************************************************
 * FUNCTION NAME: check_counter()
 * ALGORITHM: Checks if a control statement has been
 *            separated into more than one line.
 *********************************************************/
check_counter(code)
char *code;
{
  int i,j,k,len,status,counter;

  i = 0;
  j = 0;
  k = 0;
  status = YES;
  counter = 0;
  len = strlen(code);
                     /* check if the number of "(" equals
                      * to the number of ")" */
  do  {
    if ((code[i] == '/') && (i<len-1))    {
                     /* find the start symbol of comment */
      if (code[i+1] == '*')
        status = NO;
    }
    else     {        /* count the number of "(" and ")" */
      if (code[i] == '(')
        counter++;
      else if (code[i] == ')')      {
        k = i + 1;
        counter--;
      }
      i++;
    }
  } while ((i<len) && status);

  if (!status)                 /* terminate this string */
    code[i++] = '\0';
  if (k != 0)  {/* skip the first k characters of "code" */
    len = strlen(code);
    while(k<=len)
      code[j++] = code[k++];
  }
  return counter;
}

/*********************************************************
 * FUNCTION NAME: check_control_statement()
 * ALGORITHM: Checks if token is a control statement.
 *********************************************************/
```

```
check_control_statement(token)
char *token;
{
  int i;
  static char control_sample[][10] =
      {"main","do","while","for","if","else"};

     /* check if token is in the set of "control_sample" */
  for (i=0;i<6;i++)
    if (strcmp(token,control_sample[i]) == 0)
      return i+1;
  return NO;
}

/***********************************************************
 * FUNCTION NAME: push_control_id()
 * PURPOSE:  Puts "id" at the top of the C_id_stack.
 ***********************************************************/
push_control_id(id)
int id;
{
  C_id_stack[TOP_id+1] = id;/* put id at the top of stack */
  TOP_id++;
}

/************************************************************
 * FUNCTION NAME: push_control_s()
 * PURPOSE:  Puts "cp" at the top of the C_s_stack.


 ***********************************************************/
push_control_s(cp)
struct code_str *cp;
{
  C_s_stack[TOP_line+1] = cp;
                         /* put cp at the top of stack */
  TOP_line++;
}

/***********************************************************
 * FUNCTION NAME: pop_control_id()
 * PURPOSE: Returns the value of the top element of
 *          C_id_stack.
 * ALGORITHM: Checks if C_id_stack is an empty stack. If
 *            C_id_stack is not an empty stack, removes the
 *            top element from the stack and returns the
 *            value of the element.
 ***********************************************************/
pop_control_id()
{
  int i;

  if (TOP_id < 0)                  /* C_id_stack is empty */
    return(-1);
  TOP_id--;
  return(C_id_stack[TOP_id+1]);
}

/************************************************************
 * FUNCTION NAME: pop_control_s()
 * PURPOSE: Return the pointer of the top element of
 *          C_s_stack.
 * ALGORITHM: Checks if C_s_stack is an empty stack. If C_s_
 *            stack is not an empty stack, removes the top
 *            element from the stack and returns it.
 ***********************************************************/
struct code_str *pop_control_s()
{
  if (TOP_line < 0)             /* C_s_stack is empty */
    return(NULL);
  TOP_line--;
  return(C_s_stack[TOP_line+1]);
}

/***********************************************************
 * FUNCTION NAME: operator()
 * PURPOSE: Checks if "ch" is an operator.
 **********************************************************/
operator(ch)
char ch;
{
  int i;
  char sample[10];

  strcpy(sample,"+-*/=<>&|");
            /* check if "ch" is in the set of "sample" */
  for (i=0;i<9;i++)
    if (ch == sample[i])
      return YES;
  return NO;
}

/***********************************************************
 * FUNCTION NAME: deliminator1()
 * PURPOSE: Checks if "ch" is a " ", ";", ",", "(", or ")".
 ***********************************************************/
deliminator1(ch)
char ch;
{
  int i;
  char sample[10];

  if (ch == '\t')
    return YES;
  strcpy(sample," ;,()");
            /* check if "ch" is in the set of "sample" */
  for (i=0;i<5;i++)
    if (ch == sample[i])
```

```
      return YES;
  return NO;
}

/*********************************************************
 * FUNCTION NAME: deliminator2()
 * PURPOSE: Checks if "ch" is not a " ", ";", ",", "(",
 *          ")", or "[".
 *********************************************************/
deliminator2(ch)
char ch;
{
  int i;
  char sample[20];

             /* check if "ch" is in the set of "sample" */
  if ((ch != '\t') && (ch != '\n'))  {
    strcpy (sample," ,;()[");
    for (i=0;i<6;i++)
      if (ch == sample[i])
        return(NO);
    return(YES);
  }
  return(NO);
}

/*********************************************************
 * FUNCTION NAME: isalph()
 * PURPOSE: Checks if "ch" is a lower case or upper case
 *          alphabet.
 *********************************************************/
isalph(ch)
char ch;
{
  if (((ch<='z')&&(ch>='a'))||((ch<='Z')&&(ch>='A')))
    return YES;
  return NO;
}

/*********************************************************
 * FUNCTION NAME: create_ref_def_sys_cs_cr()
 * PURPOSE: Creates the ref, def, CS, CR, and SYS sets of
 *          the specified program dp.
 * ALGORITHM: Checks every code of the program dp. Checks
 *            if the code contains any variable, C function
 *            call, and iPSC/2 system call. If any variable,
 *            C function call, or iPSC/2 system call exists,
 *            links it to a proper list.
 *********************************************************/
create_ref_def_sys_cs_cr(dp)
struct data_str *dp;
{
  int i;
  int len;
int type;
int check_load = NO;
int result = 0;
char string1[MAX_COLUMN];
char string2[MAX_COLUMN];
struct var_str *rvp;
struct code_str *cp;
struct ref_def_sys *rp;

if (dp)
{
  cp = (*dp).first;
                       /* skip blank and comment lines */
  while ((*cp).n_num < 0)
    cp = (*cp).next;
                       /* check every code */
  while (cp)   {
    strcpy(string1,(*cp).code);
    get_parameter(string2,string1);
    if (((*cp).comment_mark == P_A_COM) &&
                            (strcmp(string2,"/*") == 0))
      strcpy(string2,"");/* no more search for this line*/
                         /* when string is a not empty */
    while (strlen(string2) > 0)  {
      rvp = NULL;
      result = 0;
      type = -1;
      if (type = c_def_function(string2)) {
        if (type == 3)         /* string2 is "fscanf" */
          check_fscanf(dp,cp,string1);
        else if (type == 5)    /* string2 is "scanf" */
          check_scanf(dp,cp,string1);
        else if (type == 7)    /* string is "fprintf" */
          check_fprintf(dp,cp,string1);
        else             /* string2 is "fclose", "fgets",
                          * "gets", or "strcpy */
          check_rest_c_def(dp,cp,string1);
      }
      else if (type = pps_def_function(string2))
                         /* string2 is "gdsum", "gisum",
                          * or "gssum" */
        check_pps_def(dp,cp,string1,string2);
      else if (random_sys_call(string2))
                      /* string2 is "srand" or "rand" */
        check_random_sys(dp,cp,string2);
      else  {
        check_var(string2,(*dp).var_root,&rvp,&result);
                         /* string2 is a variable */
        if (result == YES)
          link_ref_def_drive(dp,rvp,string1,cp,-1);
                         /* string2 is a system call
                          * for cube control */
        else if (check_load = pps_sys_command(string2)) {
          link_sys(dp,cp);
```

```
            if (check_load == 5)   /* a "load" command */
              (*dp).load_cp = cp;
                                   /* a "setpid" command */
            if (strcmp("setpid",string2) == 0)  {
              get_parameter(string2,string1);
              strcpy((*dp).pid,string2);
            }
          }
                             /* check if string2 is a "csend"
                              * or "crecv"command */
          else if (type = check_cs_cr(string2))
            link_cs_cr_drive(dp,type,cp);
        }
        get_parameter(string2,string1);
                             /* reach a comment */
        if (((*cp).comment_mark == P_A_COM) &&
                                 (strcmp(string2,"/*") == 0))
         strcpy(string2,""); /*no more checking */
      }
      cp = (*cp).next;
                            /* skip blank and comment lines */
      while ((*cp).n_num < 0)
        cp = (*cp).next;
      strcpy(PRE_TOKEN,"");
    }
  }
}

/***********************************************************
 * FUNCTION NAME: check_rest_c_def()
 * PURPOSE: To extract variables from of string1.
 ***********************************************************/
check_rest_c_def(dp,cp,string1)
struct data_str *dp;
struct code_str *cp;
char *string1;
{
  int i,result;
  char string2[MAX_COLUMN];
  struct var_str *rvp;

  get_parameter(string2,string1);
  check_var(string2,(*dp).var_root,&rvp,&result);
                             /* string2 is a variable */
  if (result == YES)
    link_ref_def_drive(dp,rvp,string1,cp,def);
}

/***********************************************************
 * FUNCTION NAME: check_pps_def()
 * PURPOSE: Checks if a code contains system calls of
 *          iPSC/2 System.
 * ALGORITHM: Checks every token of (*cp).code. If a
 *            variable is called by address, links it to def
 *            and DEF sets. If a variable is called by
 *            value, links it to ref and REF sets.
 ***********************************************************/
check_pps_def(dp,cp,string1)
struct data_str *dp;
struct code_str *cp;
char *string1;
{
  int status,i,result,len;
  char string2[MAX_COLUMN];
  struct var_str *rvp;

  get_parameter(string2,string1);
  status = YES;
  while (strlen(string2) > 0) {/* check every token */
    if (string2[0] == '&')  {  /* a var called by address */
      len = strlen(string2);
      for (i=0;i<len-1;i++)    /* skip the "&" character */
        string2[i] = string2[i+1];
      string2[i] = '\0';       /* terminate the string */
    }
    check_var(string2,(*dp).var_root,&rvp,&result);
    if (result == YES) {       /* string2 is a variable */
      if (status)     {    /* a variable called by address */
        status = NO;
        link_ref_def_drive(dp,rvp,string1,cp,def);
        link_REF_DEF(cp,rvp,DEF);
      }
      else      {          /* a variable called by value */
        link_ref_def_drive(dp,rvp,string1,cp,ref);
        link_REF_DEF(cp,rvp,REF);
      }
    }
    get_parameter(string2,string1);
  }
}

/**********************************************************
 * FUNCTION NAME: link_ref_def_drive()
 * PURPOSE: To link vp to a proper list.
 * ALGORITHM: If the type is not specified, checks the type
 *            of the vp. According to the type of vp, links
 *            vp to a proper list.
 ***********************************************************/
link_ref_def_drive(dp,vp,string,cp,type)
struct data_str *dp;
struct var_str *vp;
char *string;
struct code_str *cp;
int type;
{
  struct ref_def_sys *rdp;
  struct REF_DEF_str *RDp;
  char temp1[MAX_COLUMN],temp[MAX_COLUMN];
  strcpy(temp1,string);
  get_parameter(temp,temp1);
  if (type == -1) /* need to check if it is an assignment */
    type = check_assign_statement(temp);
  rdp = initiate_ref_def_sys_str();
  (*rdp).cp = cp;
  if (type == ref)  {              /* link to the ref set */
    link_ref_def(&((*vp).ref_set),rdp);
    link_REF_DEF(cp,vp,REF);
  }
  if (type == def)  {              /* link to the def set */
    get_parameter(temp,string);
    link_ref_def(&((*vp).def_set),rdp);
    link_REF_DEF(cp,vp,DEF);
  }
}

/*******************************************************
 * FUNCTION NAME: link_REF_DEF()
 * PURPOSE: To link vp to the proper list.
 * ALGORITHM: Initializies a REF_DEF_str for vp. According
 *            to the value of type, links the REF_DEF_str
 *            to a proper list.
 *******************************************************/
link_REF_DEF(cp,vp,type)
struct code_str *cp;
struct var_str *vp;
int type;
{
  int status = YES;
  struct REF_DEF_str *RDp;
  struct REF_DEF_str *temp;
  struct REF_DEF_str *np;
  struct var_str *tt;

  np = initiate_REF_DEF_str();
  (*np).vp = vp;
  if (type == REF)              /* link to the REF set */
    RDp = (*cp).REF_set;
  else if (type == DEF)         /* link to the DEF set */
    RDp = (*cp).DEF_set;
  if (RDp == NULL)  {          /* REF or DEF list is empty */
    if (type == REF)    {       /* initialize the REF set */
      (*cp).REF_set = np;
      RDp = (*cp).REF_set;
    }
    else if (type == DEF)    { /* initialize the DEF set */
      (*cp).DEF_set = np;
      RDp = (*cp).DEF_set;
    }
  }
  else  {
    temp = RDp;
          /* check if the pointer is already in the list */
    while (((*temp).next != NULL) && (status))    {
      if ((*temp).vp == vp)/* the pointer is in the list */
        status = NO;
      else
        temp = (*temp).next;
    }
    if ((*temp).vp == vp)  /* the pointer is in the list */
      status = NO;
    if (status)        /* the pointer is not in the list */
      (*temp).next = np; /* link the pointer to the list */
  }
}

/************************************************************
 * FUNCTION NAME: link_esc_var()
 * PURPOSE:  To link vp to the list headed by escvar.
 * ALGORITHM: Initializes a esc_var_str for vp.  If the list
 *            is empty, initializes the list. If the list is
 *            not empty and vp is not in the list, links the
 *            esc_var_str to the list.
 ***********************************************************/
link_esc_var(escvar,vp,line)
struct esc_var_str **escvar;
struct var_str *vp;
int line;
{
  int status = YES;
  struct esc_var_str *evp,*temp;

  evp = initiate_esc_var_str();
  (*evp).var = vp;
  (*evp).line_no = line;
  if (*escvar == NULL)                      /* empty list */
    *escvar = evp;
  else  {                          /* the list is not empty */
    temp = *escvar;
    while ((*temp).next != NULL)   {
        temp = (*temp).next;
    }
    (*temp).next = evp;   /* link the pointer to the list */
  }
}

/***********************************************************
 * FUNCTION NAME: link_ref_def()
 * PURPOSE: To link rdp to the list headed by refdef.
 * ALGORITHM: If the list is empty, initializes the list. If
 *            the list is not empty and the (*rdp).cp is not
 *            in the list, links rdp to the list.
 ***********************************************************/
link_ref_def(refdef,rdp)
struct ref_def_sys **refdef,*rdp;
{
  int status = YES;
  struct ref_def_sys *temp;

  if (*refdef == NULL)  /* ref, def, or sys list is empty */
    *refdef = rdp;
  else  {             /* ref, def, or sys list is not empty */
    temp = *refdef;
                     /* check if the pointer already in list */
    while (((*temp).next != NULL) && (status))    {
      if ((*temp).cp == (*rdp).cp)     /* already in list */
        status = NO;
      else
        temp = (*temp).next;
    }
    if ((*temp).cp == (*rdp).cp)       /* already in list */
      status = NO;
    if (status)         /* the pointer is not in the list */
      (*temp).next = rdp; /* link the pointer to the list */
  }
}

/***********************************************************
 * FUNCTION NAME: pps_def_function()
 * PURPOSE: To check if the string is gisum, gdsum, or
 *          gssum.
 ***********************************************************/
pps_def_function(string)
char *string;
{
  int i;
  static char pps_def_fun[][10] =
                       {"gdsum","gisum","gssum"};

     /* check if string is in the array of "pps_def_fun" */
  for (i=0;i<3;i++)
    if (strcmp(pps_def_fun[i],string) == 0)
      return(i+1);
  return(NO);
}

/***********************************************************
 * FUNCTION NAME: c_def_function()
 * PURPOSE: To check if the string is fclose, fgets,
 *          fscanf, gets, scanf, strcpy, or fprintf.
 ***********************************************************/
c_def_function(string)
char *string;
{
  int i;
  static char c_def_fun[][10] = {"fclose","fgets","fscanf",
                       "gets" ,"scanf","strcpy", "fprintf"};

        /* check if string is in the array of "c_def_fun" */
  for (i=0;i<7;i++)
    if (strcmp(c_def_fun[i],string) == 0)
```

```
      return(i+1);
  return(NO);
}

/**********************************************************
 * FUNCTION NAME: check_assign_statement()
 * PURPOSE: To check if the string is an operator of
 *          assignment.
 **********************************************************/
check_assign_statement(string)
char *string;
{
  int i;
  static char assignment[][5] = {"=","+=","-=","++","--"};

      /* check if string is in the array of "assignment" */
  for (i=0;i<5;i++)
    if (strcmp(assignment[i],string) == 0)
      return(def);
  return(ref);
}

/**********************************************************
 * FUNCTION NAME: check_cs_cr()
 * PURPOSE: To check if the string is "csend" or "crecv".
 **********************************************************/
check_cs_cr(string)
char *string;
{
  if (strcmp(string,"csend") == 0)   /* a "csend" command */
    return(CS);
  else if (strcmp(string,"crecv")==0)/* a "crecv" command */
    return(CR);
  else           /* neither a "csend" nor "crecv" command */
    return(NO);
}

/**********************************************************
 * FUNCTION NAME: link_cs_cr_drive()
 * PURPOSE: To link cp to a proper list.
 * ALGORITHM: Allocates a piece of memory for new cs_cs_str
 *            and group_cs_cr_str. Checks every token of
 *            (*cp).code. According to the type of cp,
 *            links cp to a proper list.
 **********************************************************/
link_cs_cr_drive(dp,type,cp)
struct data_str *dp;
int type;
struct code_str *cp;
{
  int i,len,result;
  struct cs_cr_str *rp,*temp;
  struct group_cs_cr_str *gp;
  struct var_str *rvp;
  struct ref_def_sys *rdp;
  char temp1[MAX_COLUMN],temp2[MAX_COLUMN],token[MAX_COLUMN];

  strcpy(temp1,(*cp).code);
  rp = initiate_cs_cr();
              /* allocate a piece of memory for cs_cr_str */
  gp = initiate_group_cs_cr();
       /* allocate a piece of memory for group_cs_cr_str */
  (*gp).t_fp = rp;
  (*rp).tp_group = gp;
  (*rp).tp = cp;
  (*rp).level = LEVEL;
  if (type == CS)                     /* a "csend" command */
    link_cs_cr(&((*dp).cs),rp);
  if (type == CR)                     /* a "crecv" command */
    link_cs_cr(&((*dp).cr),rp);
  get_parameter(temp2,temp1);
  get_parameter(temp2,temp1);               /* get data type */
  strcpy((*rp).type,temp2);
  get_parameter(temp2,temp1);       /* get passed variable */
  result = 0;
  if (temp2[0] == '&')  {                   /* skip the "&" */
    len = strlen(temp2);
    for (i=0;i<len-1;i++)
      temp2[i] = temp2[i+1];
    temp2[i] = '\0';
  }
  check_var(temp2,(*dp).var_root,&rvp,&result);
  if (result == YES);  {              /* a legal variable */
    rdp = initiate_ref_def_sys_str();
    (*rdp).cp = cp;
    if (type == CS)    {              /* a "csend" command */
      link_ref_def(&((*rvp).ref_set),rdp);
      link_REF_DEF(cp,rvp,REF);
      get_parameter(temp2,temp1);
                              /* search for the destination
                               * of "csend" command */
      while (strlen(temp2) > 0)  {
        strcpy (token,temp2);
        get_parameter(temp2,temp1);
      }
      strcpy((*rp).to,token);
    }
    if (type == CR)    {              /* a "crecv" command */
      link_ref_def(&((*rvp).def_set),rdp);
      link_REF_DEF(cp,rvp,DEF);
    }
  }
}

/************************************************************
 *  FUNCTION NAME: link_cs_cr()
 *  PURPOSE: To link a pointer of cs_cr_str to the list
 *           headed by cscr.
 *  ALGORITHM: If the list is empty, initializes the list.
 *             If the list is not empty, links the pointer
 *             to the end of the list.
 *********************************************************/
link_cs_cr(cscr,rp)
struct cs_cr_str **cscr,*rp;
{
  struct cs_cr_str *temp;

  if (*cscr == NULL)   /* the list head by cscr is empty */
    *cscr = rp;
  else  {
    temp = *cscr;
    while ((*temp).next != NULL)
             /* go to the end of the list headed by cscr */
      temp = (*temp).next;
    (*temp).next = rp;
  }
}

/*********************************************************
 *  FUNCTION NAME: check_is_var()
 *  PURPOSE:  To check if the string is a legal data type.
 *  ALGORITHM: Applies string processing techniques.
 *********************************************************/
check_is_var(token)
char *token;
{
  int i;
  char var_id[MAX_COLUMN];
  static char var_str[][10]={"char","double","float","int",
                             "long","short","FILE"};

             /* check if "var_str" is a legal data type */
  for (i=0;i<7;i++)
    if (strcmp(token,var_str[i]) == 0)
      return YES;
  return NO;
}

/*********************************************************
 * FUNCTION NAME: is_var()
 * PURPOSE: To link the declared variables to the variable
 *          list of the program dp.
 * ALGORITHM: Checks if the parameter "token" is a type of
 *            legal data type.  Checks if parameter "temp_
 *            code" contains any variable. Links the
 *            variable to the list of variables.
 *********************************************************/
is_var(dp,cp,token,temp_code)
struct data_str *dp;
struct code_str *cp;
char *token,*temp_code;
{
  int i,result,type,status,group;
  char var_id[MAX_COLUMN],n_token[MAX_COLUMN],
                                    tcode[MAX_COLUMN];
  struct var_str *rvp;
  static char var_str[][10]={"char","double","float","int",
                             "long","short","FILE"};
                     /* check if it is a legal data type */
  for (i=0;i<7;i++) {
    if(strcmp(token,var_str[i])==0){/* a legal data type */
      strcpy(tcode,temp_code);
      get_parameter(var_id,temp_code);
      do  {           /* check every string */
        link_var_drive(dp,var_id,token);
        check_var(var_id,(*dp).var_root,&rvp,&result);
        link_ref_def_drive(dp,rvp,var_id,cp,def);
        strcpy(tcode,temp_code);
        get_parameter(n_token,temp_code);
        type = check_assign_statement(n_token);
        if (strcmp(n_token,"/*") == 0)  {   /* a comment */
          i = 0;
          status = NO;
                                      /* skip the comment */
          while ((tcode[i] != '/') && (!status))    {
            if (tcode[i]==',') /* more than one variable */
              status = YES;
            else
              i++;
          }
          if (status) {        /* more than one variable */
            cp = (*cp).next;
            while ((*cp).n_num < 0)
              cp = (*cp).next;
            strcpy(temp_code,(*cp).code);
            get_parameter(var_id,temp_code);
          }
          else                        /* no more variables */
            strcpy(var_id,"");
        }
        else {    /* the parameter "token" is a variable */
          if (type == def)
                     /* declaration with initialization */
            strcpy(var_id,"");
          else
            strcpy(var_id,n_token);
        }
      } while (strlen(var_id) != 0);
       /* check every token in the parameter "temp_token" */
      return(YES);
    }
  }
  return(NO);
}

/************************************************************
```

```
 * FUNCTION NAME: check_var()
 * PURPOSE: Checks if var exists in the list of variables.
 * ALGORITHM: Traverses the list of variables to search for
 *            var.
 *******************************************************/
check_var(var,vp,cp,result)
char *var;
struct var_str *vp,**cp;
int *result;
{
  int status = -1;
  int len,i;

  if (vp)  {                      /* the var list not empty */
    if (var[0] == '*')    {       /* a pointer expression */
      len = strlen(var);
      for (i=0;i<len-1;i++)       /* skip the "*" */
        var[i] = var[i+1];
      var[i] = '\0';
    }
    status = strcmp((*vp).var_id,var);
    if (status == 0)    {              /* got the variable */
      *cp = vp;     /* return the pointer of the variable */
      *result = YES;
      return;
    }
                /* current variable id is less than "var" */
    else if (status<0)
      check_var(var,(*vp).r_ptr,&*cp,&*result);
             /* current variable id is greater than "var" */
    else if (status>0)
      check_var(var,(*vp).l_ptr,&*cp,&*result);
  }
  return;
}

/******************************************************
 * FUNCTION NAME: link_var_drive()
 * PURPOSE: To link a var_str to the list of variables.
 * ALGORITHM: Checks if the var_id starts with a '*'. If
 *            the list of variables is empty, initializes
 *            the list. If the list of variables is not
 *            empty, calls link_cs_cr() to link the var_id
 *            to a proper position of the list.
 *******************************************************/
link_var_drive(dp,var_id,var_type)
struct data_str *dp;
char *var_id,*var_type;
{
  int len,i;
  struct var_str *vp;

  if (var_id[0] == '*')  {    /* a pointer expression */
    len = strlen(var_id);
    for (i=0;i<len-1;i++)      /* skip the "*" */
      var_id[i] = var_id[i+1];
    var_id[i] = '\0';
  }
  vp = initiate_var_str(); /* allocate a piece of memory */
  strcpy ((*vp).var_id,var_id);
  strcpy ((*vp).type,var_type);
  if ((*dp).var_root==NULL) /* the variable list is empty */
    (*dp).var_root = vp;  /* initialize the variable list */
  else
    link_var((*dp).var_root,vp);      /* link to the list */
}

/*********************************************************
 * FUNCTION NAME: link_var()
 * PURPOSE: To link np to a proper position of the variable
 *          list.
 * ALGORITHM: Traverses the list of variables to find a
 *            proper position for np.
 ********************************************************/
link_var(tp,np)
struct var_str *tp,*np;
{
  if (strcmp((*np).var_id,(*tp).var_id) < 0)  {
      /* new variable id is less than current variable id */
    if ((*tp).l_ptr == NULL) {/* link to the left pointer */
      (*tp).l_ptr = np;
      return;
    }
    else                        /* go to the left pointer */
      link_var((*tp).l_ptr,np);
  }
  else if (strcmp((*np).var_id,(*tp).var_id) > 0)  {
             /* new var id is greater than current var id */
    if ((*tp).r_ptr==NULL) { /* link to the right pointer */
      (*tp).r_ptr = np;
      return;
    }
    else                       /* go to the right pointer */
      link_var((*tp).r_ptr,np);
  }
}

/*********************************************************
 * FUNCTION NAME: output_related_data()
 * PURPOSE: To output the ref, def, DEF, REF, SYS sets and
 *          legal variables and slice of the target and
 *          focus programs.
 ********************************************************/
output_related_data()
{
  output_slice(TP);
  output_slice(SP);
  fprint_line();
  fprintf(fpps,"Legal Variables in %s:\n\n",(*TP).file);
  output_legal_variables((*TP).var_root);
  fprint_line();
  fprintf(fpps,"Legal Variables in %s:\n\n",(*SP).file);
  output_legal_variables((*SP).var_root);
  output_REF_DEF(TP,REF);
  output_REF_DEF(SP,REF);
  output_REF_DEF(TP,DEF);
  output_REF_DEF(SP,DEF);
  output_SYS(TP);
  output_SYS(SP);
}

/************************************************************
 * FUNCTION NAME: output_SYS()
 * PURPOSE: Output the SYS set of a program.
 * ALGORITHM: Traverse every member of the SYS set of a
 *            program and prints the line number of the
 *            code.
 ************************************************************/
output_SYS(dp)
struct data_str *dp;
{
  int status = NO;
  struct ref_def_sys *sys;

  fprint_line();
  fprintf(fpps,"The SYS set of %s : ",(*dp).file);
  if ((*dp).sys == NULL) {       /* the set SYS is empty */
    fprintf(fpps,"{ }\n");
    return;
  }
  sys = (*dp).sys;
  while(sys) {      /* check every member of the set SYS */
    if (!status) {  /* the first member of the set SYS */
      status = YES;
      fprintf(fpps,"{%d",(*((*sys).cp)).n_num);
    }
    else
      fprintf(fpps,", %d",(*((*sys).cp)).n_num);
    sys = (*sys).next;
  }
  fprintf(fpps,"}\n\n");
}

/************************************************************
 * FUNCTION NAME: output_REF_DEF()
 * PURPOSE: To output the REF or DEF set of a program.
 * ALGORITHM: Traverses every member of the REF or DEF set
 *            of a program and prints the name of the
 *            variables.
 ************************************************************/
output_REF_DEF(dp,type)
struct data_str *dp;
```

```
int type;
{
  int status = NO;
  struct code_str *cp;
  struct REF_DEF_str *temp;

  fprint_line();        /* output a line to the output file */
  if (type == REF)                     /* output the REF set */
    fprintf(fpps,"REF sets of %s:\n\n",(*dp).file);
  if (type == DEF)                     /* output the DEF set */
    fprintf(fpps,"DEF sets of %s:\n\n",(*dp).file);
  cp = (*dp).head;
  while (cp) {       /* traverse every code of the program
                      * that specified by "dp" */
    if (type == REF)
      temp = (*cp).REF_set;
    else if (type == DEF)
      temp = (*cp).DEF_set;
    status = NO;
    while (temp){/* the set headed by "temp" is not empty */
      if (check_is_var((*((*temp).vp)).type)) {
        if (!status) { /* the first member has been found */
          status = YES;
          fprintf(fpps,"%3d  {%s",
                      (*cp).n_num,(*((*temp).vp)).var_id);
        }
        else
          fprintf(fpps,", %s",(*((*temp).vp)).var_id);
      }
      temp = (*temp).next;
    }
    if (status)  /* the set headed by "temp" is not empty */
      fprintf(fpps,"}\n");
    cp = (*cp).next;
  }
}

/************************************************************
 * FUNCTION NAME: fprint_line()
 * PURPOSE: Outputs 79 characters of "=" to the file of fpps
 ***********************************************************/
fprint_line()
{
  int i;

  fprintf(fpps,"\n");
  for (i=0;i<78;i++)
    fprintf(fpps,"=");
  fprintf(fpps,"\n");
}

/************************************************************
 * FUNCTION NAME: output_legal_variables()
 * PURPOSE: Outputs the legal variables of a program.
 * ALGORITHM: Traverses the list of variables and outputs
 *            the data types and names of the variables.
 ***********************************************************/
output_legal_variables(vp)
struct var_str *vp;
{
  if ((*vp).l_ptr != NULL)                    /* go left */
    output_legal_variables((*vp).l_ptr);
  if (check_is_var((*vp).type)) {
    fprintf(fpps,"\ntype   : %s\n",(*vp).type);
    fprintf(fpps,"var id : %s\n",(*vp).var_id);
    output_ref_def(vp,ref);         /* output the ref set */
    output_ref_def(vp,def);         /* output the def set */
  }
  if ((*vp).r_ptr != NULL)                    /* go right */
    output_legal_variables((*vp).r_ptr);
}

/************************************************************
 * FUNCTION NAME: output_ref_def()
 * PURPOSE: Outputs the ref or def set of a variable.
 * ALGORITHM: Traverses every member of the ref or def set
 *            of a variable and prints the line number of
 *            the code.
 ***********************************************************/
output_ref_def(vp,type)
struct var_str *vp;
int type;
{
  int status = NO;
  struct ref_def_sys *rdef;

  if (type == ref) {                   /* output ref set */
    fprintf(fpps,"   ref set of %s : ",(*vp).var_id);
    rdef = (*vp).ref_set;
  }
  if (type == def) {                   /* output def set */
    fprintf(fpps,"   def set of %s : ",(*vp).var_id);
    rdef = (*vp).def_set;
  }
  while(rdef){/* traverse every member of ref or def set */
    if (!status) {     /* the first member of the set headed
                        * by "rdef" has been found */
      status = YES;
      fprintf(fpps,"{%d",(*((*rdef).cp)).n_num);
    }
    else               /* not the first member of the set
                        * headed by "rdef"*/
      fprintf(fpps,", %d",(*((*rdef).cp)).n_num);
    rdef = (*rdef).next;
  }
  if (status)                       /* not an empty set */
    fprintf(fpps,"}\n");
  else                                  /* an empty set */
```

```
    fprintf(fpps,"{ }\n");
}

/************************************************************
 * FUNCTION NAME: show_screen_header()
 * PURPOSE: To show the greeting banner of PPS.
 ************************************************************/
show_screen_header(dp,pps_error,type,c_page,t_page,error)
struct data_str *dp;
int *pps_error,type,c_page,t_page;
char *error;
{
  if (*pps_error == NO)
    printf("<< Parallel Program Slicer\n");
  else    {
    printf("<< Parallel Program Slicer : %s\n",error);
    *pps_error = NO;
  }
  printf("     PAGE : %d of %d",c_page,t_page);
  if (type == 1)
    printf("            FILE NAME : %s\n",(*dp).file);
  else
    printf("             Slice of : %s\n",(*dp).file);
  printf("==========================================================");
  printf("==========================\n");
}

/*************************************************************
 * FUNCTION NAME: show_screen_bottom()
 * PURPOSE: To show the PPS screen layout.
 *************************************************************/
show_screen_bottom(status,pps_error,c_page,t_page,
                                              error_message)
int *status,*pps_error,*c_page,*t_page;
char *error_message;
{
  char choose[MAX_COLUMN];

  printf("==========================================================");
  printf("==========================\n");
  if (*c_page == 1)
  {
    if (*t_page > 1)
      printf("   N > next page    Q > quit  : ");
    else
      printf("   Q > quit  : ");
  }
  else if (*c_page == *t_page)
    printf("   P > previous page    Q > quit  : ");
  else
   printf(" P > previous page  N > next page  Q > quit : ");
  gets(choose);
  if ((choose[0] == 'q') || (choose[0] == 'Q'))
    *status = YES;
  else if ((choose[0] == 'p') || (choose[0] == 'P'))    {
    if (*c_page == 1)      {
      strcpy (error_message,"NO PREVIOUS PAGE!");
      *pps_error = YES;
    }
    else
      *c_page = *c_page - 1;
  }
  else if ((choose[0] == 'n') || (choose[0] == 'N'))    {
    if (*c_page == *t_page)      {
      strcpy (error_message,"NO NEXT PAGE!");
      *pps_error = YES;
    }
    else
      *c_page = *c_page + 1;
  }
  else    {       /* user input is not a legal command */
    strcpy (error_message,"ILLEGAL COMMAND!");
    *pps_error = YES;
  }
}

/***********************************************************
 * FUNCTION NAME: view_slice()
 * PURPOSE: To show the contents of a slice to screen.
 * ALGORITHM: Traverses every code of a program.  If a code
 *            is marked as a slice, outputs the code to
 *            screen. Formats the screen as 19-line screen.
 **********************************************************/
view_slice(dp)
struct data_str *dp;
{
  int i = 1;
  int line = Screen_line;
  int page_no;
  int status = NO;
  int pps_error = NO;
  char error_message[80];

  page_no = (*dp).s_total % line;
  if (page_no != 0)
    page_no = ((*dp).s_total / line) + 1;
  else
    page_no = (*dp).s_total / line;
  while (!status)  {       /* user did not enter "quit" */
    show_screen_header(dp,&pps_error,0,i,page_no,
                                        error_message);
    cat_slice(dp,((i-1)*line)+1,i*line);
    show_screen_bottom(&status,&pps_error,&i,&page_no,
                                        error_message);
  }
}

/**********************************************************
```

```
 * FUNCTION NAME: get_parameter()
 * PURPOSE: To extract a token from target2 and return via
 *          target1.
 * ALGORITHM: Applies string processing techniques.
 *********************************************************/
get_parameter(target1,target2)
char *target1,*target2;
{
  int i = 0;
  int j = 0;
  int len = 0;
  int keep = YES;
  int start = NO;
  int status = YES;
  char temp[MAX_COLUMN];

  len = strlen(target2);
               /* if "ch" is a " ", ";", ",", "(", or ")" */
  while (deliminator1(target2[i]))
     i++;
  if (i >= 0)
    start = YES;
  if (target2[i] == '"')  {
    i++;
    while (target2[i] != '"')
      i++;
  }
         /* "ch" is not a " ", ";", ",", "(", ")", or "[" */
  while (deliminator2(target2[i]) && (i<len) && keep)  {
    target1[j++] = target2[i++];
    if (!operator(target2[i-1]) && operator(target2[i]))
      keep = NO;
    else if (operator(target2[i-1]))    {
                                            /* check ">=" */
      if ((target2[i-1] == '>') && (target2[i] == '='))
        target1[j++] = target2[i++];
                                            /* check "<=" */
      else if ((target2[i-1] == '<')&&(target2[i] == '='))
        target1[j++] = target2[i++];
                    /* check "==", "+=", "*=", and "/=" */
      else if (target2[i-1] == '=')      {
        if (target2[i] == '=')
        target1[j++] = target2[i++];
        keep = NO;
      }
                                           /* check "&&" */
      else if (target2[i-1] == '&')      {
        if (target2[i] == '&')       {
          target1[j++] = target2[i++];
          keep = NO;
        }
        else if (isalph(target2[i]))
          target1[j++] = target2[i++];
      }
```

```
                                              /* check "||" */
      else if ((target2[i-1] == '|') && (target2[i] == '|'))
        target1[j++] = target2[i++];
                                              /* check "++" */
      else if ((target2[i-1] == '+') && (target2[i] == '+'))
        target1[j++] = target2[i++];
                                              /* check "--" */
      else if ((target2[i-1] == '-') && (target2[i] == '-'))
        target1[j++] = target2[i++];
      else if (target2[i-1] == '*')       {
        if (target2[i] == ' ')
          keep = NO;
                          /* check the start of a comment */
        else if (target2[i-1] == '/')         {
          target1[j++] = '*';
          keep = NO;
        }
        else if (strlen(PRE_TOKEN) == 0)
          keep = YES;
        else if ((strlen(PRE_TOKEN) != 0) &&
                                  (check_is_var(PRE_TOKEN)))
          keep = YES;
        else if ((strlen(PRE_TOKEN) != 0) &&
                                  (!check_is_var(PRE_TOKEN)))
          keep = NO;
      }
      else if ((target2[i-1] == '/') && (target2[i] == '*')){
        target1[j++] = '*';
        keep = NO;
      }
      else
        keep = NO;
    }
  }
  target1[j] = '\0';
  strcpy(PRE_TOKEN,target1);
  if (target1[i] == '#')        /* the code is a definition */
    strcpy(target2,"");
  if (target2[i] == '[') { /* skip the element of an array */
    i++;
    while (target2[i] != ']')
      i++;
    i++;
  }
  if (target2[i] == '\n')  {              /* end of a code */
    i++;
    strcpy(PRE_TOKEN,"");
  }
  j=0;
  while (i<=len)
    temp[j++] = target2[i++];
  strcpy(target2,temp);
}

/*********************************************************
```

```
 * FUNCTION NAME: run_function()
 * PURPOSE: Executes the sliced program.
 * ALGORITHM: Checks if host and node programs are loaded
 *            in PPS.  If both host and node programs are
 *            loaded in PPS, this function asks the iPSC/2
 *            System to compile them and execute the
 *            parallel program.
 ********************************************************/
run_function()
{
  if (!ESC)  {
    printf("Please LOAD first.\n");
    return;
  }
  if ((*TP).type == (*SP).type)  {
    printf("Need two programs.\n");
    return;
  }
  if (compile_prog(TP) && compile_prog(SP))
                        /* both TP and SP are available */
    system("ppshost");           /* execute the program */
}

/*********************************************************
 * FUNCTION NAME: compile_prog()
 * PURPOSE: Compiles the default file names of programs.
 * ALGORITHM: If the type of the program is "h", the slice
 *            will be compiled as "cc -o ppshost ppshost.c
 *            -host". If the type of the program is "n",
 *            the slice will be compiled as "cc -o ppsnode
 *            ppsnode.c -node".
 *******************************************************/
compile_prog(dp)
struct data_str *dp;
{
  FILE *f_temp;
  char target[MAX_COLUMN];

  if ((*dp).type == HOST)
    strcpy(target,"cc -o ppshost ppshost.c -host");
  else if ((*dp).type == NODE)
    strcpy(target,"cc -o ppsnode ppsnode.c -node");
  printf("Compiling ...\n");
  if (!system(target))
    return(YES);
}

/********************************************************
 * FUNCTION NAME: check_scanf()
 * PURPOSE: Extracts variables from a scanf function call.
 * ALGORITHM: Applies string processing techniques.
 ********************************************************/
check_scanf(dp,cp,string1)
struct data_str *dp;
struct code_str *cp;
char *string1;
{
  int i,result,len;
  char string2[MAX_COLUMN];
  struct var_str *rvp;

  get_parameter(string2,string1);        /* skip "scanf" */
  get_parameter(string2,string1);
  while (strlen(string2) > 0) {     /* check every token */
    if (string2[0]=='&'){/* a variable called by address */
      len = strlen(string2);
      for (i=0;i<len-1;i++)    /* skip the "&" character */
        string2[i] = string2[i+1];
      string2[i] = '\0';
    }
    check_var(string2,(*dp).var_root,&rvp,&result);
    if (result == YES)  {       /* string2 is a variable */
      link_ref_def_drive(dp,rvp,string1,cp,def);
      link_REF_DEF(cp,rvp,DEF);
    }
    get_parameter(string2,string1);
  }
}

/***********************************************************
 * FUNCTION NAME: check_fprintf()
 * PURPOSE: Extracts variables from a fprintf function call.
 * ALGORITHM: Applies string processing techniques.
 ***********************************************************/
check_fprintf(dp,cp,string1)
struct data_str *dp;
struct code_str *cp;
char *string1;
{
  int i,result,len;
  char string2[MAX_COLUMN];
  struct var_str *rvp;

  get_parameter(string2,string1);
  check_var(string2,(*dp).var_root,&rvp,&result);
  if (result == YES)              /* string2 is a variable */
    link_ref_def_drive(dp,rvp,string1,cp,def);
}

/***********************************************************
 * FUNCTION NAME: check_fscanf()
 * PURPOSE: Extracts variables from a fscanf function call.
 * ALGORITHM: Applies string processing techniques.
 ***********************************************************/
check_fscanf(dp,cp,string1)
struct data_str *dp;
struct code_str *cp;
char *string1;
```

```
{
  int i,result,len;
  char string2[MAX_COLUMN];
  struct var_str *rvp;

  get_parameter(string2,string1);
  check_var(string2,(*dp).var_root,&rvp,&result);
  if (result == YES)     /* string2 is a legal variable */
    link_ref_def_drive(dp,rvp,string1,cp,def);
  get_parameter(string2,string1);
  while (strlen(string2) > 0)  {   /* check every token */
    if (string2[0] == '&')    {
      len = strlen(string2);
      for (i=0;i<len-1;i++)
        string2[i] = string2[i+1];
      string2[i] = '\0';
    }
    check_var(string2,(*dp).var_root,&rvp,&result);
    if (result == YES)   /* string2 is a legal variable */
      link_ref_def_drive(dp,rvp,string1,cp,def);
    get_parameter(string2,string1);
  }
}

/***********************************************************
 * FUNCTION NAME: return_memory()
 * PURPOSE: Returns all of the allocated memory that is
 *          used to the hold the information of a program.
 ***********************************************************/
return_memory(dp)
struct data_str *dp;
{
  if (dp) {
    return_code_str((*dp).first);
    return_var_memory((*dp).var_root);
    return_cs_cr((*dp).cs);
    return_cs_cr((*dp).cr);
    return_control_scope_str((*dp).c_scope_set);
    return_esc_var((*dp).evar_history);
    return_ref_def_sys((*dp).sys);
    free(dp);
  }
}

/***********************************************************
 * FUNCTION NAME: return_control_scope_str()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the control statements
 *          of a program.
 ***********************************************************/
return_control_scope_str(control)
struct control_scope_str *control;
{
  struct control_scope_str *temp;
  while (control) {    /* check every control statement */
    return_branch((*control).branch);
                   /* return the allocated memory that is used to
                    * hold the information of the related branch
                    * control statements */
    temp = control;
    control = (*control).next;
    free(temp);
  }
}

/**********************************************************
 * FUNCTION NAME: return_branch()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the branch control
 *          statements of a program.
 **********************************************************/
return_branch(branch)
struct branch_control_scope_str *branch;
{
  struct branch_control_scope_str *temp,*temp_head;

  temp_head = branch;
  if (temp_head) {      /* go to the header of the related
                         * control statements */
    while ((*temp_head).pre)
      temp_head = (*temp_head).pre;
  }
  while (temp_head) {
    temp = temp_head;
    temp_head = (*temp_head).next;
    free(temp);
  }
}

/**********************************************************
 * FUNCTION NAME: return_var_memory()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the variables of a
 *          program.
 **********************************************************/
return_var_memory(var)
struct var_str *var;
{
  if (!var)             /* NULL pointer */
    return;
  if ((*var).l_ptr != NULL)   /* go left */
    return_var_memory((*var).l_ptr);
  if ((*var).r_ptr != NULL)   /* go right */
    return_var_memory((*var).r_ptr);
  return_ref_def_sys((*var).ref_set);
            /* return the allocated memory that is used to
             * hold information of the set "ref" */
  return_ref_def_sys((*var).def_set);
```

```
              /* return the allocated memory that is used to
               * hold information of the set "def" */
  free(var);
}

/***********************************************************
 * FUNCTION NAME: return_esc_var()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the variables of an
 *          extended slicing criterion.
 ***********************************************************/
return_esc_var(evar)
struct esc_var_str *evar;
{
  struct esc_var_str *temp;

  while (evar) {  /* check every variable of an extended
                   * slicing criterion */
    temp = evar;
    evar = (*evar).next;
    free(temp);
  }
}

/***********************************************************
 * FUNCTION NAME: return_ref_def_sys()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the set "ref", "def",
 *          or "sys".
 ***********************************************************/
return_ref_def_sys(set)
struct ref_def_sys *set;
{
  struct ref_def_sys *temp;

  while (set) {      /* check every member of the set "ref",
                      * "def", or "sys" */
    temp = set;
    set = (*set).next;
    free(temp);
  }
}

/***********************************************************
 * FUNCTION NAME: return_cs_cr()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the set "CS" or "CR".
 ***********************************************************/
return_cs_cr(cscr)
struct cs_cr_str *cscr;
{
  struct cs_cr_str *temp;

  while (cscr){ /* check every "csend" or "crecv" command */
```

```
    return_group_cs_cr((*cscr).fp_group);
    return_group_cs_cr((*cscr).tp_group);
    temp = cscr;
    cscr = (*cscr).next;
    free(temp);
  }
}

/*********************************************************
 * FUNCTION NAME: return_group_cs_cr()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the related "csend" or
 *          "crecv" commands.
 *********************************************************/
return_group_cs_cr(gcscr)
struct group_cs_cr_str *gcscr;
{
  struct group_cs_cr_str *temp,*temp_head;

  temp_head = gcscr;
  if (temp_head) {
              /* go to the header of the list that contains
               * the "csend" or "crecv" command */
    while ((*temp_head).pre)
      temp_head = (*temp).pre;
  }
  while (temp_head) {
    temp = temp_head;
    temp_head = (*temp_head).next;
    free(temp);
  }
}

/*********************************************************
 * FUNCTION NAME: return_code_str()
 * PURPOSE: Returns the allocated memory that is used to
 *          hold the information of the codes of a program.
 *********************************************************/
return_code_str(code)
struct code_str *code;
{
  struct code_str *temp;

  while (code) {      /* check every code of the program */
    if ((*code).REF_set)
      free((*code).REF_set);
                    /* return the allocated memory that is
                     * used for the set "REF" */
    if ((*code).DEF_set)
      free((*code).DEF_set);
                    /* return the allocated memory that is
                     * used for the set "DEF" */
    temp = code;
    code = (*code).next;
    free(temp);
  }
}

/*********************************************************
 * FUNCTION NAME: compile_function()
 * PURPOSE: Asks iPSC/2 System to compile a certain program.
 *********************************************************/
compile_function(input)
char *input;
{
  int i = 0;
  char file_name[MAX_COLUMN],file_type[MAX_COLUMN],
       token[MAX_COLUMN],default_com[MAX_COLUMN],
       response[MAX_COLUMN];

  get_parameter(file_name,input);
  if (strlen(file_name) == 0) {
    printf("Input file name: ");
    gets(file_name);
  }
  if (strlen(file_name) == 0) {
    printf("Compile filed: No file name has been
              specified\n");
    return;
  }
  get_parameter(file_type,input);
  if (strlen(file_type) == 0) {
    printf("Input file type <h: host, n: node> : ");
    gets(file_type);
  }
  if (strlen(file_type) == 0) {
    printf("Compile   filed:   No   file   type   has   been
specified\n");
    return;
  }
  strcpy(token,file_name);
                /* if the file name is "host.c", the default
                 * executable file name is "host" */
  while (i<(strlen(file_name) - 2)) {
    token[i] = file_name[i];
    i++;
  }
  token[i] = '\0';
  strcpy(default_com,"");
  strcat(default_com,"cc -o ");
  strcat(default_com,token);
  strcat(default_com," ");
  strcat(default_com,file_name);
  strcat(default_com," ");
  if (file_type[0] == 'h')
    strcat(default_com,"-host");
  else if (file_type[0] == 'n')
    strcat(default_com,"-node");
  printf("Accepts default command: %s (Y/N) ?  ",
                                              default_com);
  gets(response);
  if ((response[0] == 'Y') || (response[0] == 'y'))
                /* compile the program as default command */
    system(default_com);
  else  {       /* accept user input */
    printf("Input command : ");
    strcpy(default_com,"");
    gets(default_com);
    system(default_com);
  }
}

/***********************************************************
 * FUNCTION NAME: check_erase_temp()
 * PURPOSE: Checks if the temporary files created for
 *          procedure compile_prog() need to be erased.
 ***********************************************************/
check_erase_temp()
{
  char response[MAX_COLUMN];

  printf("Erases temporary files that generated by
                                                   PPS(Y/N)?");
  gets(response);
  if ((response[0] == 'Y') || (response[0] == 'y'))
    erase_temp();
  return;
}

/************************************************************
 * FUNCTION NAME: random_sys_call()
 * PURPOSE: To check if the string is srand or rand.
 ***********************************************************/
random_sys_call(string)
char *string;
{
  int i;
  static char random_sys[][10] = {"srand","rand"};

     /* check if string is in the array of "random_sys" */
  for (i=0;i<2;i++)
    if (strcmp(random_sys[i],string) == 0)
      return(i+1);
  return(NO);
}

/************************************************************
 * FUNCTION NAME: check_random_sys()
 * PURPOSE: Link "rand" and "srand" function calls.
 ***********************************************************/
check_random_sys(dp,cp,string)
struct data_str *dp;
struct code_str *cp;
char *string;
{
  int result = NO;
  struct var_str *rvp;
  char temp[MAX_COLUMN];

                           /* link "srand" function call */
  if (strcmp(string,"srand") == 0) {
    link_var_drive(dp,string,"func");
    strcpy(temp,string);
  }
  else {                    /* link "rand" function call */
    strcpy(temp,"s");
    strcat(temp,string);
  }
  check_var(temp,(*dp).var_root,&rvp,&result);
  link_ref_def_drive(dp,rvp,temp,cp,def);
  link_REF_DEF(cp,rvp,DEF);
}
```

VITA

Ting-Huan Hsiao

Candidate for the Degree of

Master of Science

Thesis: AN INTERACTIVE PARALLEL PROGRAM SLICER (PPS) FOR C PROGRAMS ON THE iPSC/2 SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Taipei, Taiwan, R.O.C., May 29, 1963, the son of Su-Yuan Hsiao and Su-Chen Chang.

Education: Graduated from Chien-Kuo High School, Taipei, Taiwan, R.O.C., in June 1981; received Bachelor of Science Degree in Pharmacy Sciences from Taipei Medical College, Taipei, Taiwan, R.O.C., in June 1986; Completed requirements for the Master of Science Degree at Oklahoma State University in December 1992.