Dragon Analysis Tool

User's Guide

Version 1.0e

University of Houston

Computer Science Department
High Performance Tools Group
Dr. Barbara Chapman

Last Modified: 9 / 14 / 2 0 0 4

Preface

This User's Manual describes the Dragon Analysis Tool, a system that provides source code analysis capabilities to the user. It provides a range of information about the structure of a source program in graphical format. Future versions of this system will provide additional functionality to support the development and improvement of OpenMP and MPI programs. Dragon has a graphical user interface using MOTIF/LESSTIF and X11 together with some PD widget sets to display graphs and navigate within these graphs.

Audience Description

This manual is a user's guide only. Users are expected to have a basic knowledge of the structure of programs and some experience in C or Fortran programming. Dragon is available on Red Hat 7.2/7.3. It is assumed that users are familiar with the basic commands on these systems.

Organization

This User's Guide is structured into the following chapters and appendices:

- *Chapter 1*: **Introduction** gives an overview of Dragon Analysis Tool and describes its major goals

- *Chapter 2:* **Basic Usage of Dragon**, describes naming conventions, how to manipulate the graphs.

- *Chapter 3:* **The Features of Dragon**, describes the features of the local and global information windows of Dragon and how to navigate within the graphs.

- *Chapter 4:* **Ongoing and Future Work**, a short preview of what is going on around Dragon and plans for the near future.

- *Appendix A:* **Installation**, describes the hardware and software requirements for installing Dragon on Linux platforms.

- *Appendix B:* **A Case Study**, provides an example of parallelizing an existing serial program with OpenMP directives using Dragon.

Chapter 1. Introduction

In this chapter we give an overview of the overall features of Dragon and describe the structure of the system.

1.1 Major Goals of Dragon

Whenever application software is migrated to a new platform, optimized for an existing one, or extended in its functionality, the source code must be carefully analyzed in order to understand many details of the current implementation. Dragon is a prototype software tool to support an application developer or code owner who wishes to understand more about his or her C or Fortran application. It provides a range of information about the structure of a source program in a graphical browseable form, at the level of detail desired. The current input languages for Dragon are FORTRAN 77/90, C, OpenMP.

OpenMP is the de facto standard for shared memory programming that can be used to program SMPs and distributed shared memory systems. However, significant user effort is required to parallelize serial codes with OpenMP directives or further analyze and optimize the performance of OpenMP programs. Dragon, with its comprehensive intra and interprocedural analysis information, can be an indispensable assistant for writing, analyzing and optimizing OpenMP applications.

Dragon is an interactive system. It provides a powerful graphical user interface including tools for manipulating the graphical displays and for navigating within them. A hierarchically structured menu system based on MOTIF/LESSTIF and X11 helps to find the different facilities of Dragon.

Dragon is an on-going research project developed at the University of Houston, with funding from DOE and Dell Computer Corp. Its functionality is based on a branch (Open64.UH) of the Open64 compiler infrastructure that was made available to the community by Silicon Graphics Inc. and is now maintained by Intel Corp. Dragon is also intended to be a resource for the community.

Chapter 2. Basic Usage of Dragon

In this chapter the user will find

- Some hints on preparing programs for Dragon
- Invoking Dragon and first time use
- Naming/Graphical conventions
- How to manipulate graphs

2.1 Preparing Your Program for Dragon

The first step of preparing a program to be analyzed by Dragon is to compile it with our branch of the Open64 compiler (Open64.UH) using a special set of flags. The user is responsible for modifying the application's makefiles, in order to reflect the compiler and flags setting changes. Since Open64.UH is based on SGI's compiler suites, the following are the main drivers/compilers:

- open90 (Fortan90/77 compiler)
- opencc (C compiler)
- openCC (C++ compiler) [Currently not supported by Dragon]

In order to prepare your program, you **must add** the following flags: *-ipa -O2 -dragon* for the compile (-c) and link commands as follows.
Compile commands:

> *openf90 -c -ipa -O2 -dragon myfile.f*
> *opencc -c -ipa -O2 -dragon myfile2.f*

Link commands:

> *openf90 -ipa -O2 –dragon myfile.o myfile2.o –o myprogram*

Note: add -mp flag into compile and link commands if it is an OpenMP program.

If you want to view the data dependence information you need to clean previous compilation results and recompile the application, with the following flags:

> *openf90 -c -O3 -dragon myfile.f*
> *opencc -c -O3 -dragon myfile2.f*

Link commands:

> *openf90 –O3 –dragon myfile.o myfile2.o –o myprogram*

Note: do not add –mp flag in previous compile and link commands even if it is an OpenMP application.

In this example, Open64.UH will generate a series of *.cfg, *.dep, *.rgn* files for each compilation unit, a project file *.d* (where * denotes the name of the target file) and the empty target file.

In addition to static callgraph and control flowgraph, Dragon can also display the feedback information from previous executions in dynamic callgraph and dynamic control flowgraph on Itanium machines. To achieve this:

Step 1. compile and link the program using opencc/openf90 with option

-fb_create myfeedback -fb_type=1 -fb_phase=0

This will generate executable files with profiling activated.

Step 2. Run the program. A feedback file named "myfeedback" storing the profiling data is generated.

Step 1 and step 2 can be repeated as many times as you want to merge profiling information from several executions.

Step 3. compile and link the program again using

Opencc/openf90 -fb_opt myfeedback -dragon -ipa -O2

This will generate the dynamic callgraph and dynamic control flowgraph information stored in .d and .cfg file

Step 4. invoke Dragon to load the *.d file and navigate the generated dynamic callgraph/control flowgraph.

Currently, dynamic information is available only on IA-64 platform.

2.2 Invoking Dragon

To run **Dragon** you have go to the directory where Dragon is installed and type at the command prompt "*./dragon*". Dragon will automatically display its user interface and enable the menus to load a new project (*.d* file)

If you want to load a new Fortran program, select the Program menu and then "Load Source Files"; please refer to figure 2.1. Then a file selector box will appear where you can select the file(s) of the program that you wish to load. When you have specified your program files, the preprocessor will read the input program, process all the include files and analyze your program. At this point, Dragon is ready to answer your queries via the user interface.
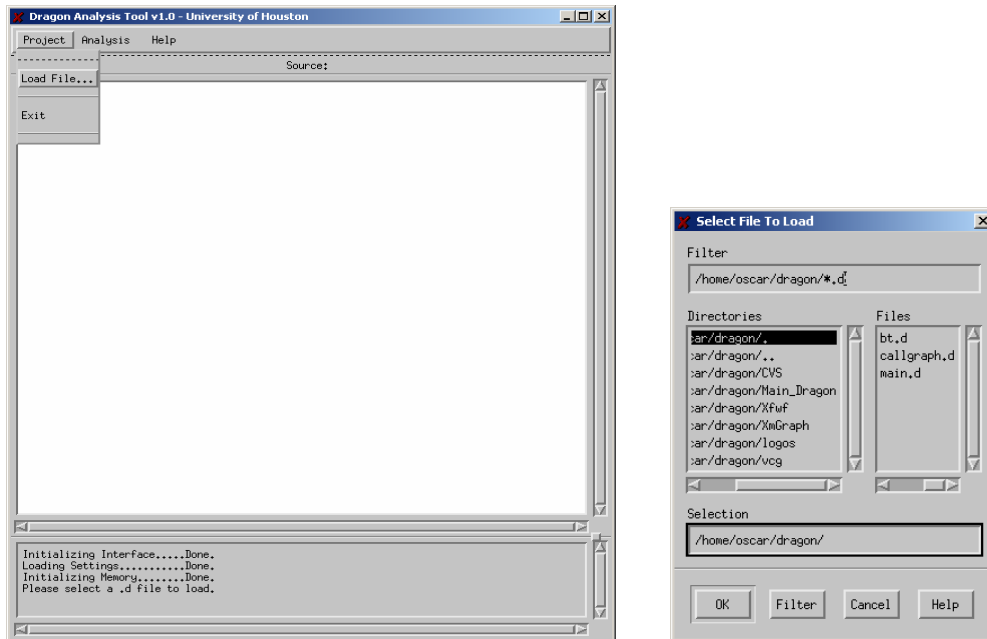
Figure 2.1. Project Menu / File box selector

2.2.1 Filename Conventions

Dragon uses a file selector box to select the files to process. Dragon accepts as input project files with extension *.d (*filename.d*). You can only load one *.d at a time.

2.2.2 Other Naming Conventions

We often refer to procedures of a program by the Fortran term "program unit" which includes the main program and each subprogram, whether subroutine or function.

2.3 Manipulating the Graphs

Basic information about graphs:

- Nodes, arcs, and regions of a graph may be selected.
- Information is often related to a node and its selection enables the display of information.
- A selected node is displayed with a rectangle around the node, just like a selected button.
- Selected arcs are displayed with different color and/or different width.

- Items (arcs or nodes) are deselected by clicking on a graph area without nodes or arcs.

You will need a mouse with at least 2 buttons to manipulate the graphs that are used in Dragon. Use mouse button 1 or left mouse button to:

- Select a node or an arc
- Deselect a node or an arc
- Click a node or an arc to get further information

Use button 2 or middle mouse button (in case of three mouse buttons) to move nodes of a graph by simply clicking on the node, holding down the middle mouse button and moving the mouse. An outlined rectangle of the node and the connected outlined arcs follow the mouse pointer that is shown as a cross.

Chapter 3. The Features of Dragon

3.1 Overview

Dragon windows contain menus for displaying the program and its parts. The menu items provide information that may be local, i.e. related to one procedure, or global, i.e. related to two or more program units.

- The structure of the complete program is provided in the form of a graph (the callgraph), with the optional dynamic information via profiling.
- The individual program procedures are represented both as source text and in the form of a flowgraph. The flowgraph shows only the control flow of a procedure and compresses straight-line code into a single node. The frequency of each edge can be obtained via profiling.
- Some of the Callgraph nodes are colored for identification: the main procedure is in orange, terminal/leaf nodes in green (procedures that don't invoke other procedures) and dead functions in red (procedures that are guaranteed to never be invoked).
- The coloring of the Control Flowgraph, shows branch nodes in green, entry/exit nodes in yellow, do/for loops in blue, and OpenMP nodes in orange.

3.2 Code Browser Window

This window is usually the starting point for obtaining information related to an individual procedure.

Figure 3.2: Local Analysis Window

At the top, it shows the filename of the procedure selected. All queries will be assumed to be requests for information about the procedure that is currently displayed in the source text pane. A procedure is loaded and displayed in the Code browser window by either clicking its name from the procedure list menu or by clicking a corresponding node in the Callgraph window. We will explain this later in the document.

The local analysis window contains a palette with items to select if you want to view the callgraph or flowgraph.
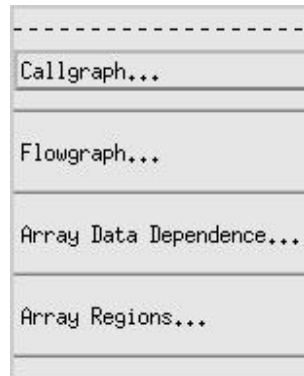
Figure 3.3: Analysis Menu

3.2.1 Flowgraph

Selecting the menu button flowgraph from the Analysis menu (figure 3.3) will trigger a window with the local flowgraph of the current procedure.
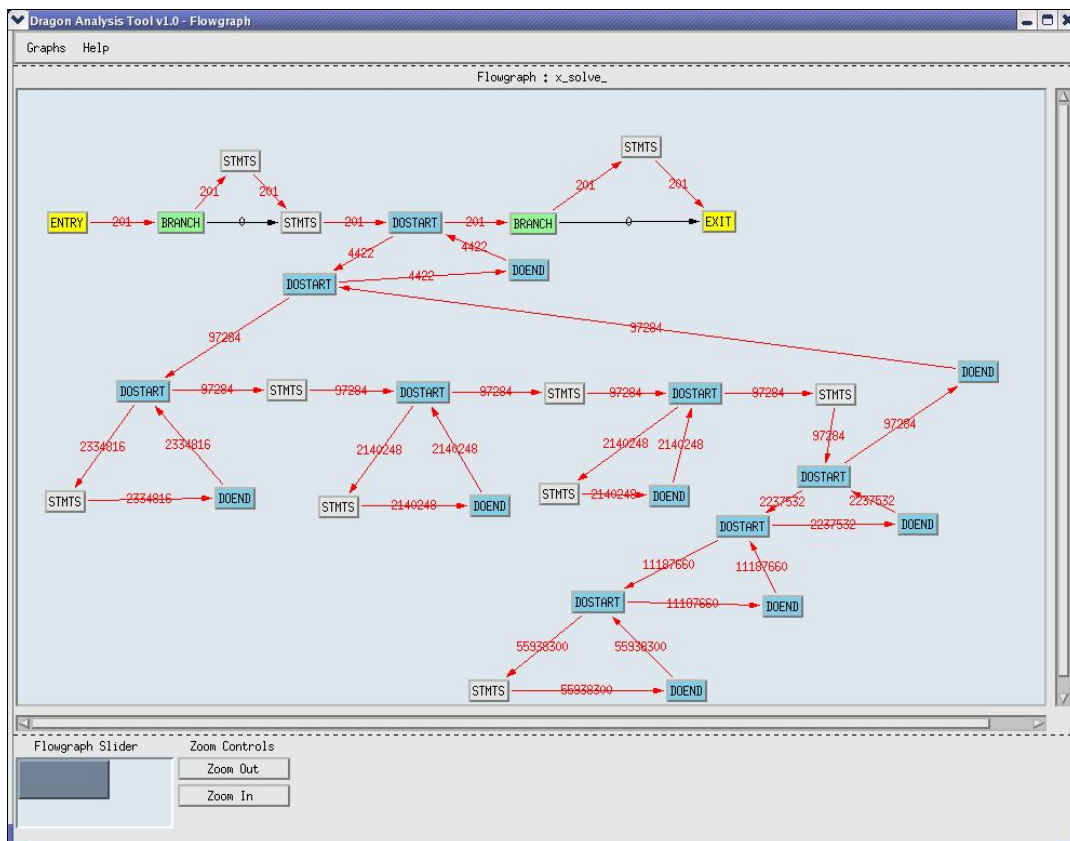

Figure 3.4: Flowgraph

This flowgraph consists of a basic block tree with the following possible elements:

- DOSTART nodes represent DO statements (DO loop headers). There are 2 arcs connecting the DO node with the body of the DO loop and the next statement in the flowgraph.
- ENDO nodes represent END DO or CONTINUE statements
- LOG-IF nodes represent logical IF statements. If the IF statement has a True and a False branch in the flowgraph, 2 arcs connect it to these branches
- ENTRY nodes represent entry to a procedure
- EXIT nodes represent the exit to a procedure
- STARTREGION nodes represent the start of a parallel OpenMP region.
- ENDREGION represents the end of a parallel region in an OpenMP code.

All statements that are not described above are represented by:

- STMTS nodes, which represent a set of statements with single entry and single exit, and no explicit or implicit control flow.
- STMTS/EXIT nodes which represent several statements in a basic block with the exit of the procedure.
- IO node with I/O statements including OPEN, CLOSE, READ or WRITE a file.

For dynamic control flowgraph via profiling, the edge between two nodes may be in red (activated) or black (non-activated). The number on the edge indicates how many times this control flow path has been activated during previous executions.

On top of the Flowgraph window, the name of the currently loaded procedure is displayed. When clicking on a node of the flowgraph, the corresponding text in the program is highlighted in the code browser window. One node in the flowgraph can represent several consecutive statements in a program.

3.2.2 Array Data Dependence Information

Selecting the menu button Array Data Dependence from the Analysis menu will trigger a window with the local data dependence information of the current procedure as shown in figure 3.6.
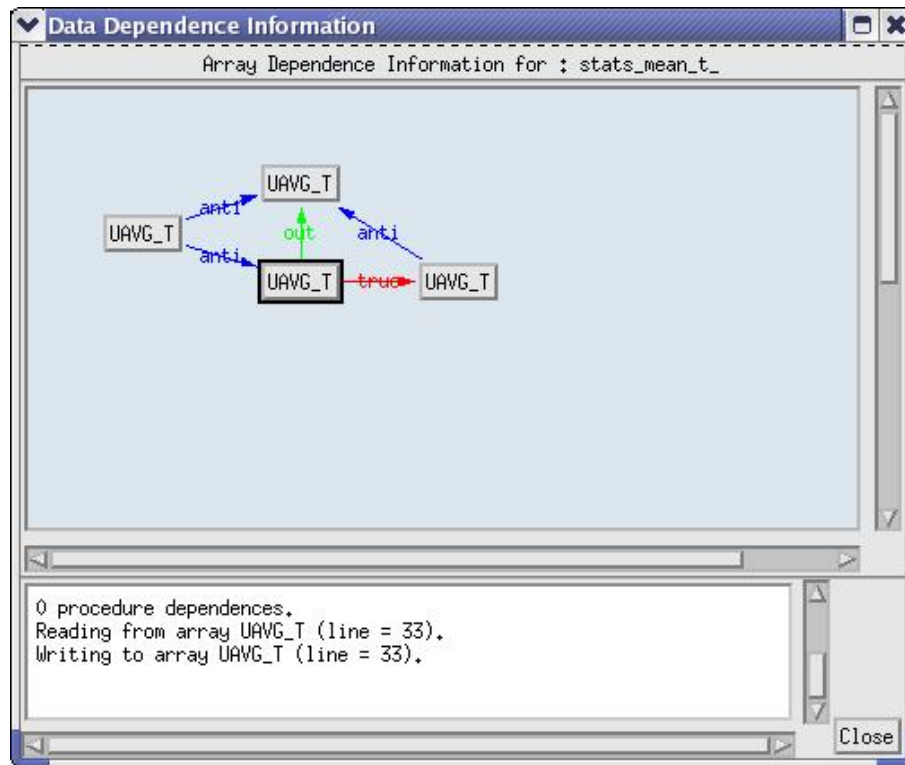
Figure 3.6: Global Analysis Window

Each node in the graph represents an array access or function name. An array access can be a read, or write to an element of an array. When clicking on the nodes, Dragon will map the array access to the source code. The arcs in the graph represent dependences.

The following dependences are shown:

- True dependences (red arcs)
- Anti dependences   (blue arcs)
- Output dependences (green arcs)
- Procedure dependences. (orange arcs)

3.3 Global Analysis Window
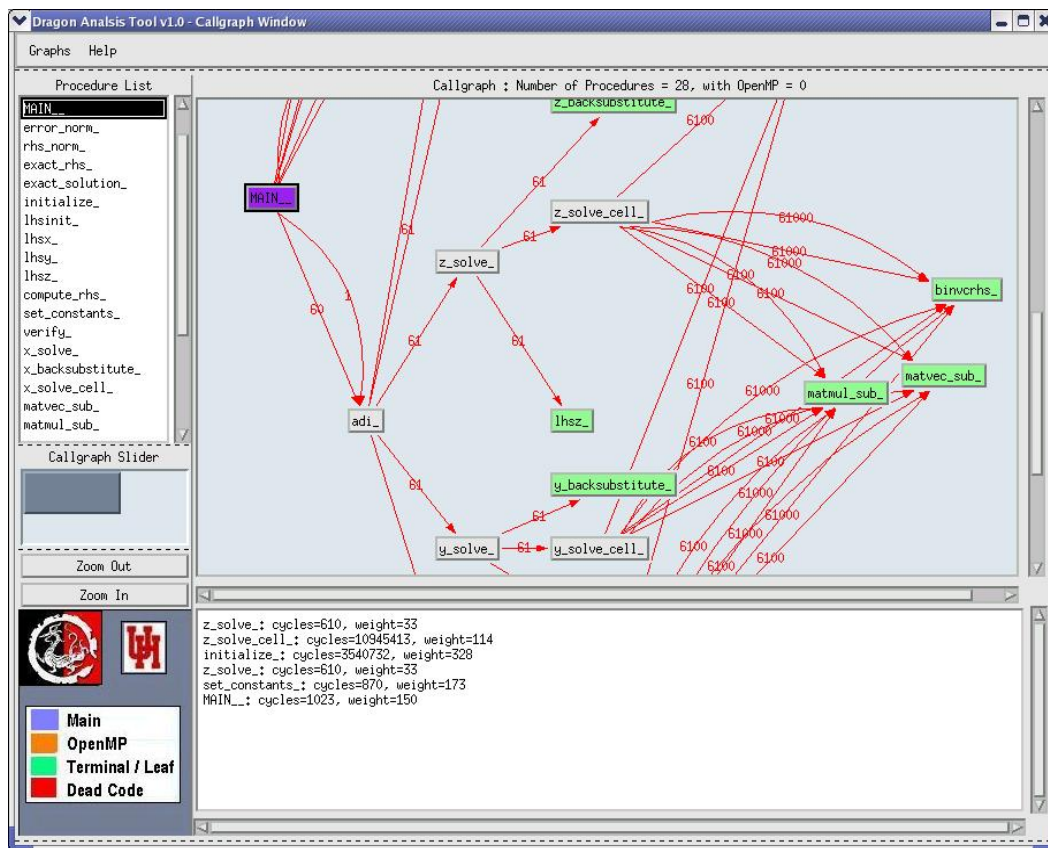
Figure 3.6: Global Analysis Window

3.3.1 Callgraph

The callgraph gives the structure of the program, where there is a node for each procedure in the program and a directed edge links a pair of nodes if and only if the procedure corresponding to the source node may invoke the sink node's procedure at run time. If you click on a node, the source text of the corresponding procedure will be displayed in the browser window. The total number or procedures in the program is shown at the top of the callgraph.

To the left of the callgraph, a list of all the procedures is displayed. Selecting a procedure from this list is equivalent to selecting its node in the graph. The coloring of the callgraph highlights the main procedure in orange, terminal/leaf nodes in green (those procedures that don't invoke other procedures) and dead functions in red (those procedures that are never invoked).

For dynamic callgraph, the number on an edge indicates how many times the caller has called the callee during the previous executions. The bottom window shows the CPU cycles consumed for each procedure.

3.3.2 Array Regions Analysis

Array Regions analysis graph shows the array references in the whole program. Currently, we mainly show the array regions that are either global arrays or formal parameters.

The array regions analysis graph contains five separate columns: **Procedure/Scope, Array Name, Access Mode, Dimensions** and the **Number of Array References**. The *Procedure/Scope* displays the name of the procedure if this array is a formal parameter, or displays *Global*, indicating that the array is a global variable that does not belong to any particular procedure. Furthermore, if a procedure is shown in the first column of this array region analysis graph, users can locate the name of the file where this procedure is defined by clicking on *Call Graph* as given in *section 3.2*. The second column lists all the array variable names accessed in the selected procedure/scope. The user can click on one array variable name whose access information he/she wants to view and this information is displayed in the third, fourth and fifth columns correspondingly. The third column labeled *Dimensions* indicates the number of dimensions of the selected array variable. The fourth column labeled *Access Mode* indicates one of the four modes of accesses: USE(array variable usage), DEF (assignment of values to array elements), FORMAL (using array variable as a formal parameter) and PASSED(an array variable passed as an actual parameter in a procedure call). The last column labeled *Number of Array References* gives the count of the references (a reference can be any one of USE, DEF, FORMAL) to the selected array variable.
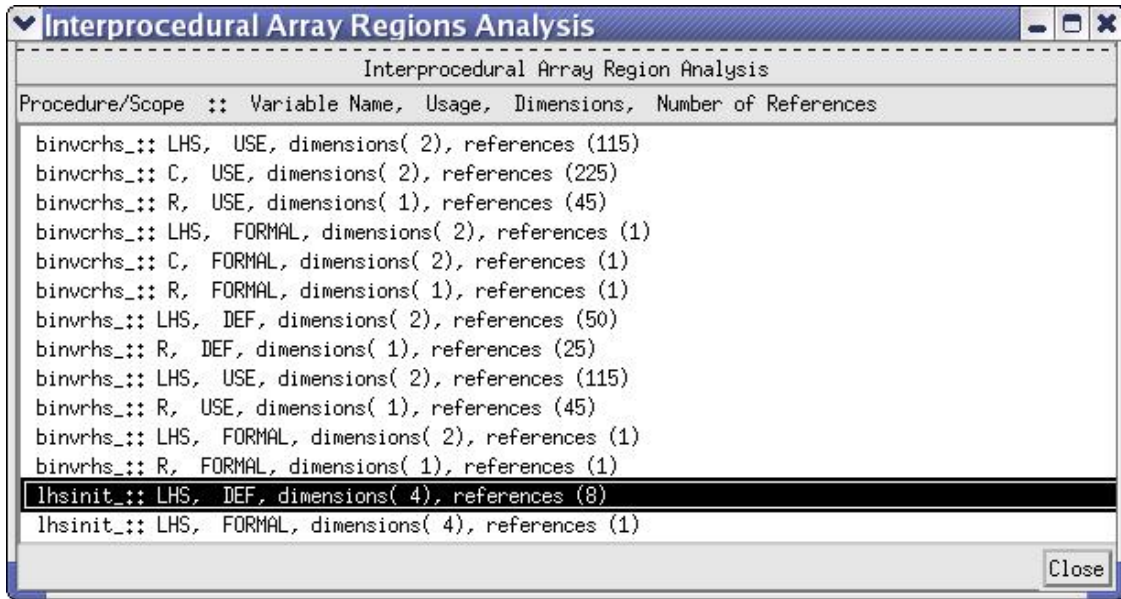
Figure 3.7: Array Region Analysis Window

Note: using up and down arrow to navigate within the region records.

We are extending the functionality to show the array regions in a triplet notation to represent a region: for example, A(1:5:1) denotes a region containing five contiguous elements from A(1) to A(5).

3.4 Graph Navigation Area

There are 2 items in the graph navigation area:
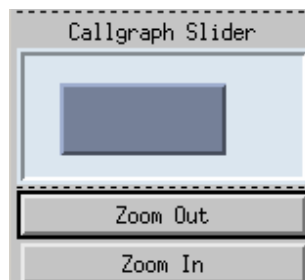
Figure 3.7: Navigation Area

- The *Slider* is a convenient way to navigate the selected graph just by dragging the slider in the slider area. Scrolling can be done in the normal ways or diagonally. The scrollbars of the selected graph are connected to the slider. The relation between slider and slider area shows approximately the relation between the visible part of the graph and the complete graph. Node that this is only true to a certain point because the slider has a minimum dimension to allow the user to drag it.

- *Zoom In / Zoom Out* enables zooming for the selected graph. The limit for *Zoom In*: only one node is shown on the visible graph area if the graph has approximately the size of the screen. The limit for zoom out is when the arcs between some of the nodes are no longer visible.

3.5 Print functionality

Dragon allows the user to print the graphs that it displays. It exports the graphs to VCG 3.0. When you select the menu item for printing a graph, a VGV window will appear. Then you should save the VGV graph in the desired format.
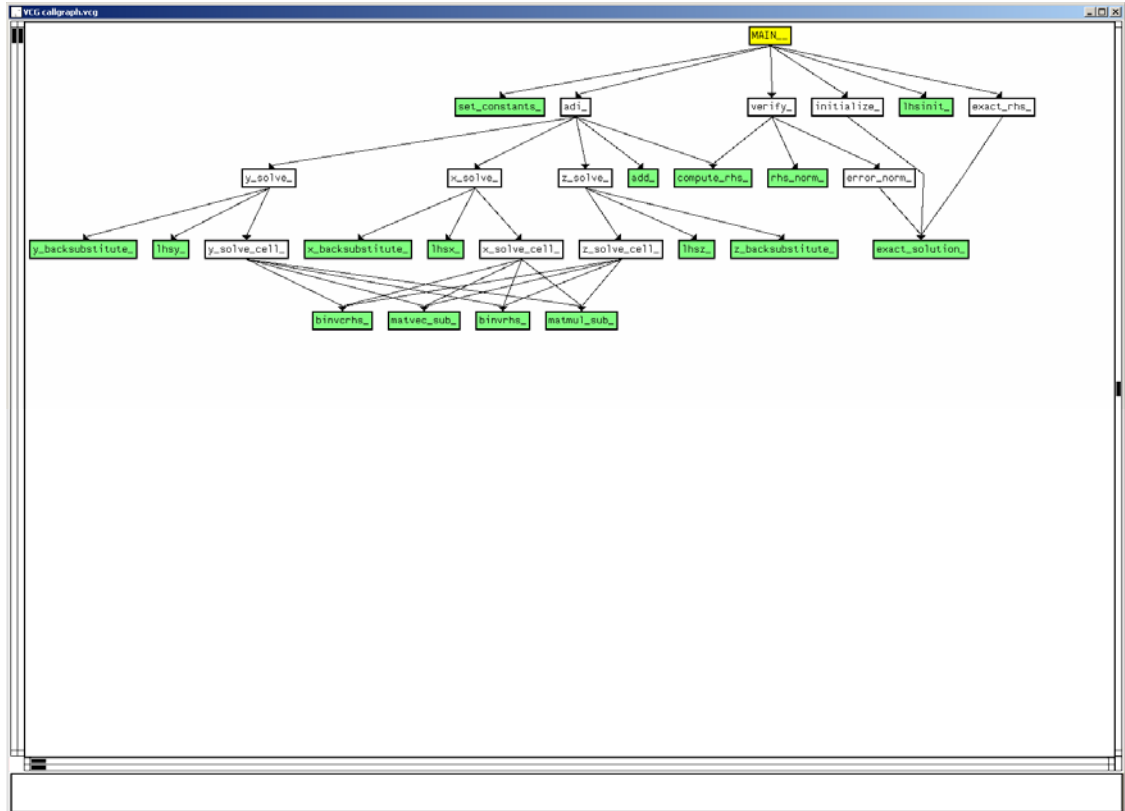
Figure 3.8: VCG window with callgraph display

Chapter 4. Ongoing and Future Work

One way to improve OpenMP performance is to minimize the data sharing among threads, either by rearranging code or by privatizing data structures where possible. Prior to this kind of non-trivial task, it is crucial to know how the arrays are being accessed by the executing threads, and determine which region of arrays are shared between multiple threads. One ongoing work of Dragon, array access patterns in OpenMP programs, is aimed to provide such important information to users and help them restructure their code to maximize data locality, reduce false sharing or even to create SPMD style OpenMP code.

A major limitation of Dragon is its lack of portability: it currently runs only on Intel Itanium machines. We are trying to make Dragon available on more platforms, mainly implementing it by porting the underlying Open64.UH compiler to more computer architectures.

The era of extreme-scale computing is coming. We also expect Dragon would support the analysis and optimization of future parallel programs running on thousands of CPUs and in thousands of threads.

Appendix A. Installation

SYSTEM REQUIREMENTS:

- Intel Pentium CPU
- Memory > 128M
- Disk > 250M
- Linux Kernel 2.4.2 or compatible with X11 (Red Hat 7.x)
- Minimum Resolution 1024 * 768
- Recommended Resolution 1600 * 1200
- Lesstif/Motif 2.1 (www.lesstif.org distribution)
- VCG 1.30

FILES FOR DRAGON

- README   - brief overview of the system
- dragon   - the executable file of the dragon tool
- dragon.res   - the resource file
- vcg/libPrintGraph.so   - the vcg printing lib
- logos/menu.xpm   - the logo picture
- logos/dragon.xpm   - the logo picture

INSTALLATION NOTES

1. The Dragon tool is based on the Open64.UH compiler. Please install the Open64.UH compiler first, using root privileges:

   tar -xvf open64_install.tar
   cd open64_install
   ./INSTALL_OPEN64

   If you have questions about what is inside the INSTALL_OPEN64 script, please review it, but do not attempt to change the paths for the target filenames, otherwise the compiler might not be able to install correctly.

2. Install the *vcg* tool for the purpose of printing the callgraph and control flowgraph. See *vcg* tool documentation for installation.

3. For Linux systems compliant with the above requirements, use:
   tar -xvf dragon.tar

   to extract the *dragon* executable file into your home directory. Change directory `to the dragon/ directory. Type ./dragon` to start the dragon tool.

4. Set environment variables:

Both Open64 and Dragon need some environment variables to work properly. Below are the details. You may put them into your .bash_profile and source it.

#the installation path for Open64 is defined as TOOLROOT
export TOOLROOT=$HOME/open64

#Adding the path of the libraries used by Open64 into the shared library's path
export LD_LIBRARY_PATH=$TOOLROOT/usr/lib/gcc-lib/ia64-open64-linux/0.16:$LD_LIBRARY_PATH

the installation path for Dragon, also the path of picture resource file used in Dragon
export DRAGON_PATH=$HOME/asamadragon

Finally add open64 and Dragon binaries into command search path
export PATH=$TOOLROOT/usr/bin:$HOME/asamadragon:$PATH

Appendix B. A case study

In this section, we will explain from scratch how to apply Dragon to an existing serial Fortran 77 code to gather program information, and use this information to parallelize it with OpenMP directives. (The subsections of *Edit the makefile* and *make the program* also mostly apply to the OMP version of this benchmark suite.)The example here is the NAS Parallel Benchmarks (NPB), a suite of eight codes designed to test the performance of parallel supercomputers. One of the eight codes, the serial version of BT (Block tri-diagonal solver) from NPB 3.1 is used to illustrate these steps while you can change BT to any other benchmark's name in this suite such as cg, ep, etc. This test suite is usually shipped with Dragon, so you may install it into your test directory, say */home/yourname/test/NPB3.1*. It will have three subdirectories: NPB3.1-MPI, NPB3.1-OMP and NPB3.1-SER.

Detailed information about NPB 3.1 can be obtained from the following website http://www.nas.nasa.gov/Software/NPB/. We only give the simplest user instructions here: to obtain further user help, type commands directly under any of these three subdirectories. For example, for NPB3.1-SER:

<p align="center">*[NPB3.1-SER]make*</p>

will display the user help for this benchmark

  To compile an NAS benchmark type

     make <benchmark-name> CLASS=<class>

  where <benchmark-name> is one of "bt", "cg", "ep", "ft", "is", "lu",
         "lu-hp", "mg", or "sp"
    <class> is one of "S", "W", "A", "B", "C" or "D"

To compile BT, you should then type:

<p align="center">*[NPB3.1-SER]make bt;*</p>

This will compile the benchmark bt with default problem size (W) and save the final executable file in the *bin* directory as bt.W.

<p style="text-align:center">*[NPB3.1-SER]make bt CLASS=S*</p>

This will compile bt with problem size small. The executable file's name is bt.S in this case.

<p style="text-align:center">*[NPB3.1-SER]bin/bt.S or bt.W*</p>

This will execute the compiled bt benchmark.

Editing the makefile

We need to modify the makefiles of NPB3.1-SER to gather the information needed by Dragon.

First, edit NPB3.1/NPB3.1-SER/config/make.def, adding support to Dragon variables during make:

```
# switch to open64 compiler if "dragon" is defined during the make
ifdef dragon
DRAGON = ${dragon}
endif

ifdef DRAGON
F77 = openf90
else
F77 = f77
endif
```

Then add corresponding compile time and link time options to support different values of the dragon variable:

```
#----------------------------------------------------------------------------
# Global *compile time* flags for Fortran programs
#----------------------------------------------------------------------------
ifdef DRAGON

# generate static callgraph and control flowgraph
# if you are woking on the OMP benchmark, add –mp to FFLAGS
ifeq ($(DRAGON),static)
FFLAGS  = -dragon -ipa -O2
endif

# generate dependence information of programs
# if you are woking under NPB3.1-OMP, add –mp to FFLAGS
ifeq ($(DRAGON),dependence)
FFLAGS  = -dragon -O3
endif

#set up profiling during compiling
```

<p style="text-align:center">22</p>

```
# feedback cannot work with -mp currently, no –mp even you are
working NPB3.1-OMP
ifeq ($(DRAGON),setfeedback)
FFLAGS = -fb_create myfeedback -fb_type=1 -fb_phase=0
endif

#generate dynamic information from previous executions
# feedback cannot work with -mp currently, no –mp even you are
working NPB3.1-OMP

ifeq ($(DRAGON),getfeedback)
FFLAGS = -fb_opt ../myfeedback -dragon -ipa -O2
endif

else
 FFLAGS = -O
endif
```

Also set up link time flags, they are the same as compile flag in this case:

```
#------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#------------------------------------------------------------------------
ifdef DRAGON
FLINKFLAGS = $(FFLAGS)
else
FLINKFLAGS = -O
Endif
```

Since the NPB benchmark uses some C functions in Fortran programs, do not forget to change the compile options for C programs:

```
#------------------------------------------------------------------------
# This is the C compiler used for C programs
#------------------------------------------------------------------------

ifdef DRAGON
CC = opencc
else
CC = cc
Endif
#------------------------------------------------------------------------
# Global *compile time* flags for C programs
#------------------------------------------------------------------------
ifdef DRAGON
CFLAGS = $(FFLGAS)
else
CFLAGS  = -O
```

```
                Endif
#-------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#-------------------------------------------------------------------------
                ifdef DRAGON
                CLINKFLAGS = $(CFLAGS)
                else
                CLINKFLAGS = -O
                Endif
```

The above are the main modifications needed in the makefile. The following two things are needed to clean files generated by previous make. One is used to clean object files, the other is to clean both object files and Dragon files.

Note: if you want to make all benchmark at the same time, remember to do the following:
Under test/NPB3.1/NPB3.1-SER/config or test/NPB3.1/NPB3.1-OMP/config
1. Type the command:
cp suite.def.template suite.def
2. Then edit suite.def to get the desired problem size for each benchmark, or leave it intact to use the default S value.

One: edit NPB3.1/NPB3.1-SER/sys/make.common or NPB3.1/NPB3.1-OMP/sys/make.common

```
# invoke "make clean" whenever the Dragon variable is defined. Otherwise you must
# manually "make clean"  each time before you make benchmark with dragon variable.
# The reason is that we want Open64.UH run each time on each source file to get
program information, while Open64.UH will skip the source files if there are already
object files for them from previous make.
# The reason for several Dragon runs is that some optimization options can not be
combined together.

ifdef DRAGON
config:
   @cd ..; ${MAKE} clean
   cd ../sys; ${MAKE} all
   ../sys/setparams ${BENCHMARK} ${CLASS}
else
config:
   @cd ../sys; ${MAKE} all
   ../sys/setparams ${BENCHMARK} ${CLASS}
Endif
```

Two: clean files generated for Dragon tool as well as the object files. This is usually done when you want to get a clean source tree.

Edit NPB3.1/NPB3.1-SER/makefile or NPB3.1/NPB3.1-OMP/makefile
Adding following to the tail:
#clean files for Dragon, such as region, control flowgraph, dependence graph and
#feedback files

```
dragonclean: clean
    - rm -f bin/*.d bin/*.rgn bin/*.rgn*
    - rm -rf bin/*.ipakeep
    - rm -f */*.cfg */*.dep
    - rm -f myfeedback.*
```

Basically, the user needs to change the original makefile a little to switch from the original compiler to the Open64.UH compiler, and must set the right flags.

Make program for Dragon

It is now very easy to prepare files for Dragon. Just type the following command under directory: test/NPB3.1/NPB3.1-SER or test/NPB3.1/NPB3.1-OMP

1. To generate static callgraph and control flowgraph for the BT program.
                        [NPB3.1-SER] make bt dragon=static
2. To generate dependence information. This may take some time because the analysis involved is costly.
                    [NPB3.1-SER] make bt dragon=dependence
3. Then you can invoke Dragon to open the Dragon project file **bt.W.d** located in test/NPB3.1/NPB3.1-SER/bin. Just type:
                        [NPB3.1-SER]dragon&

Note: To make all eight benchmark at the same time, you can use "make suite dragon= .." instead of "make bt dragon=.." in previous step 1 and 2. And the step 3 becomes to invoke Dragon to open any .d file you want to see.

To generate dynamic information, you need to follow the three steps below from the directory test/NPB3.1/NPB3.1-SER or test/NPB3.1/NPB3.1-OMP

1. set profiling into the executable file
                    [NPB3.1-SER] make bt dragon=setfeedback
2. run program to generate feedback file
                                    bin/bt.W
3. recompile the program to merge dynamic information into static information
                    [NPB3.1-SER] make bt dragon=getfeedback
4. Finally, you can invoke Dragon to open the bt.W.d file with both static and dynamic information

Note: You can also use "make suite .." instead of "make bt .." in previous step 1 and 3. But remember to run each program one by one before the step 3. And the step 4 becomes to invoke Dragon to open any .d file you want to see.

Parallelize the program using Dragon
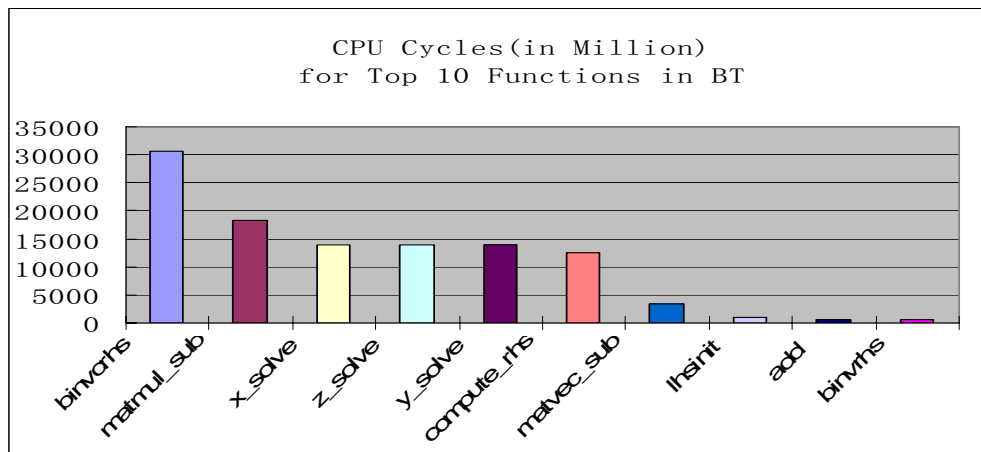
Fig. B.1. CPU cycles for top 10 functions in BT

Figure B.1 shows part of the results of profiling all functions from BT. As we can see, function *binvcrhs* consumes most of the execution cycles but it does not contain loops. Therefore, it cannot be parallelized at this level. In general, superior performance is obtained with large parallel regions, so we need to use the callgraph to determine where this procedure is invoked. It tells us that the callers of *binvcrhs* are *x_solve, y_solve* and *z_solve* where *binvcrhs* is invoked within several loops. *Matmul_sub* is the second most time-consuming routine and it contains a loop. However, this procedure is a leaf in the callgraph. It is also called by *x_solve*, *y_solve* and *z_solve*. These are the 3rd, 4th and 5th most time-consuming functions and their flowgraphs (e.g. Figure 3.4) show good opportunities for parallelization. Moreover, the caller of *x_solve, y_solve* and *z_solve* contains no loop structures; therefore nothing can be done for it. The situation is obvious now: we focus our attention on parallelization of the three subroutines: *x_solve, y_solve* and *z_solve*.

The code of *x_solve* contains many complex nested loops and is thus difficult to read. The control flowgraph can be very helpful in this case. From the graph (Fig 3.4), the programmer can easily see that *x_solve* (*y_solve* and *z_solve* are the same) has two branches with a triply-nested loop, while the innermost loop contains three statement blocks plus three more loops, the last of which is also triply-nested.

Despite the complexity of the control flow structure, the data dependences in these loops are quite simple. Fig B.2 shows the dependence graph and its corresponding source position. Associating the source position with the control flowgraph in Fig 3.4, we can

find the true dependence is actually located in the last triply-nested loop and this does not prevent us from inserting OpenMP directives to parallelize the outermost loop.

We obtain the following for the outermost loop:
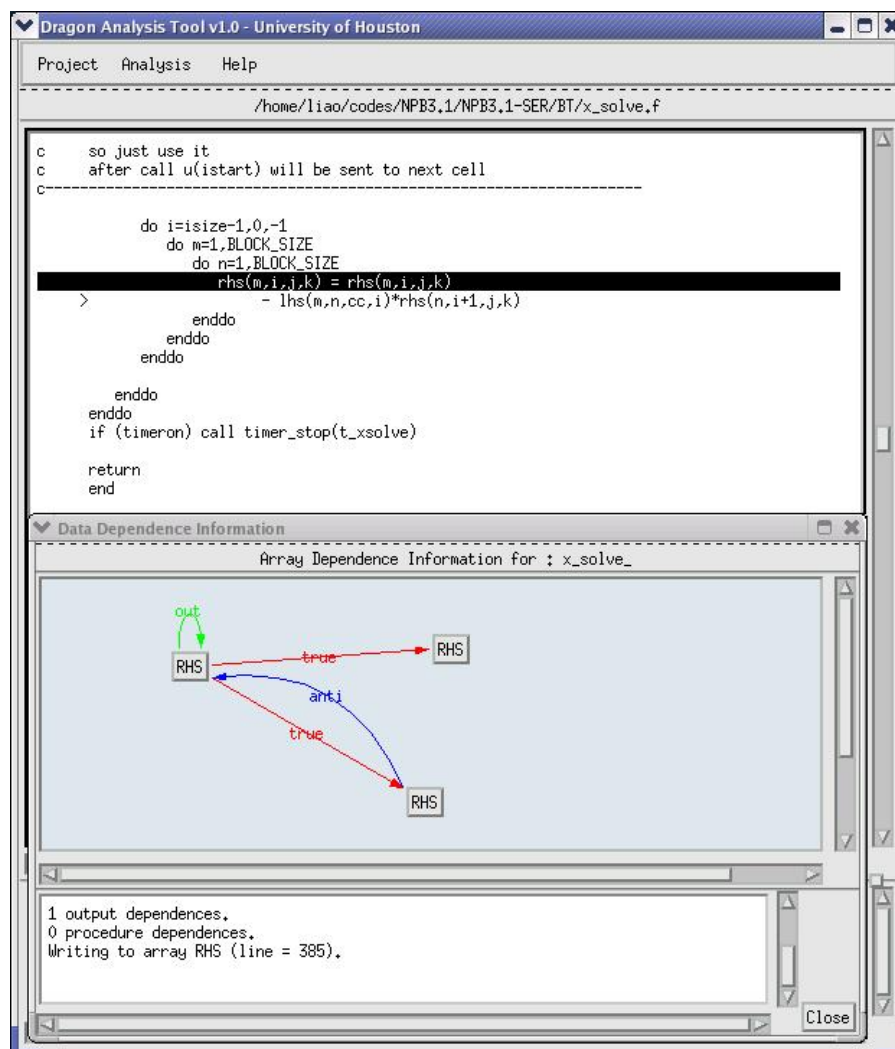
```
!$omp parallel do default(shared) shared(isize)

!$omp& private(i,j,k,m,n)

   do k = 1, grid_points(3)-2

     do j = 1, grid_points(2)-2

       do i = 0, isize   .....
```

The same procedure can be repeated for the remaining functions to quickly get an initial OpenMP version of the original serial code.

Fig B.2 Data dependence information for *x_solve*

The End---