# Code Parallelization with CAPO — A User Manual

Haoqiang Jin, Michael Frumkin and Jerry Yan

*NASA Advanced Supercomputing (NAS) Division*
*M/S T27A-2 • NASA Ames Research Center*
*Moffett Field • CA 94035-1000*
*capo@nas.nasa.gov*
*http://www.nas.nasa.gov/Tools/CAPO/*

NAS Technical Report NAS-01-008, August 2001

# CONTENTS

# USING CAPO

## Contents

# 1. General Information

## 1.1. What is CAPO

CAPO (CAPTools-based Automatic Parallelizer using OpenMP) automates the insertion of compiler directives to facilitate parallel processing on shared memory parallel (SMP) machines. While CAPO is currently integrated seamlessly into CAPTools [3] (developed at the University of Greenwich), CAPO is independently developed at NASA Ames Research Center as one of the components for the Legacy Code Modernization (LCM) project. Utilizing the data dependence information produced by CAPTools, CAPO produces either OpenMP or SGI multiprocessing directives for sequential FORTRAN programs with nominal user interaction. Due to the broad support of the OpenMP standard [12], the generated OpenMP codes can potentially run on a wide range of SMP machines. Generation of a mixed message-passing (e.g. MPI [11]) and OpenMP code is possible because of the implementation of CAPO within CAPTools.

The success of CAPO relies on accurate interprocedual data dependence information which is provided by CAPTools. CAPO generates compiler directives in three stages:

1) identification of parallel loops in the outer-most level,
2) construction and optimization of parallel regions around parallel loops, and
3) insertion of directives with a proper list of private, reduction, and shared variables.

Attempts have also been made to identify potential pipeline parallelism (implemented with point-to-point synchronization). Although the user is still expected to inspect the generated code before actual execution, the task has been simplified tremendously by the automation process and the built-in graphical user interface, known as the Directives Browser. The Directives Browser provides tools for the user to interact with the parallelization process. It presents information in such a way that the user can easily isolate problematic code sections from the rest of the code and find a solution quickly.

## 1.2. Distribution and Contact Information

CAPO is currently implemented within CAPTools and distributed directly from NASA Ames Research Center. It is released in a similar way as the standard CAPTools distribution. The distributed executable of CAPO includes all the functionality of CAPTools for generating message-passing programs as well as the capability of producing OpenMP codes. So the user needs only to maintain one copy that is distributed with CAPO to access the functionality of both CAPTools and CAPO.

To obtain a copy of CAPO, the user should send a request to *capo@nas.nasa.gov*. A license is needed to run CAPTools/CAPO. A test license may be obtained from the CAPTools web site (see below) or by sending email to *captools@gre.ac.uk*. For NASA users, please contact *capo@nas.nasa.gov* directly.

For any feedback and bug reporting on CAPO, please send email to:

CAPO Development Team at *capo@nas.nasa.gov*.

For any feedback and user support on CAPTools, please contact:

*captools-support@gre.ac.uk* or check the web site at *http://captools.gre.ac.uk/*.

For more information on the LCM project, check:

*http://www.nas.nasa.gov/Groups/Tools/Projects/LCM*.

## 1.3. Installation and Execution

Once the user has obtained a copy of CAPO in a compressed tar file, extract files by

```
% gunzip -c capo-sgi-1.1.tar.gz | tar xvf -
```

The CAPO distribution is maintained in a similar directory structure as the CAPTools distribution does. For example the executable of CAPO is in

```
captool/bin/{machine}/capo
```

where `{machine}` is `sgi` for SGI machine running IRIX, `sun` for SUN workstation running Solaris, and `linux_x86` for Intel machine running Linux.

The user should follow the same installation procedure to CAPTools to set up CAPO. For the installation and use of CAPTools, please refer to the CAPTools User Manual [9] and the web site at *http://captools.gre.ac.uk/*. In summary, the user needs to set up the following environment variables:

| | |
|---|---|
| `CAPHOME` | – home directory for the CAPTools / CAPO installation |
| `OPENWINHOME` | – home directory for the XVIEW library |
| `CAPLIBHOME` | – home directory for CAPLib (not necessary for OpenMP codes). |

and add "`$CAPHOME/bin/{machine}`" to the searching path, e.g. in `csh`:

```
setenv CAPHOME /usr/local/captool
setenv OPENWINHOME $CAPHOME/openwin
set path = ($CAPHOME/bin/sgi $path)
```

CAPO is then ready for use.

## 1.4. How to Use This Manual

The manual is organized into three parts around the use of CAPO:

1) **Using CAPO** – discusses the fundamentals of using CAPO to parallelize codes,
2) **Tutorials** – gives hands-on experiences, and
3) **Appendix** – lists detailed references of parameters and the graphical user interface.

For major changes in different versions of CAPO, see the ***`WhatsNew`*** file included in the CAPO distribution.

Convention generally followed in this manual:

| | |
|---|---|
| *Italic* | address (including email), URL, remarks, emphasis |
| `Courier` | code list, syntax description, program outputs |
| **Bold** | window name, menu name, list name |
| ***Bold italic*** | summary head, menu item |
| $\boxed{\text{Box}}$ | button, setting selection |

Throughout this document, the references to *CAPO* describe the OpenMP generation and the relevant components and the references to *CAPTools* describe all other features, but sometimes these two terms are used interchangeably for shared components.

# 2. Computer-Aided Parallelization Process

The shared memory and distributed memory programming paradigms are two of the most popular models used to transform existing serial codes to a parallel form. For a distributed memory parallelization it is necessary to consider the whole program when using an SPMD paradigm. Data placement is an essential consideration to efficiently use the available distributed memory, while the placement of explicit communication calls requires careful consideration. Nowadays, scalability and high performance mostly involve hand-written parallel programs using message-passing libraries (e.g. MPI [11]). However, this process is very difficult.

The parallelization on a shared memory system is relatively easier because of the globally addressable space. The data placement appears to be less crucial than for a distributed memory parallelization. Historically, the lack of a programming standard for using directives and the rather limited performance due to scalability have affected the acceptance of the shared memory programming model approach. In recent years significant progress has been made in hardware and software technologies, as a result the performance of parallel programs with compiler directives has also made improvements. The introduction of an industrial standard for shared-memory programming with directives, OpenMP [12], has addressed the issue of portability.

In general the parallelization process in any case is error-prone, time-consuming and requires a detailed level of expertise. Programming with directives may not necessarily produce a result that enhances performance. In the worst case, the inserted directives can create erroneous results when used incorrectly. While vendors may have provided tools to perform error-checking and profiling, automation in directive insertion is very limited and often fails on large programs, primarily due to the lack of a thorough enough data dependence analysis. Presence of these deficiencies motivated the development of the parallelization tool, CAPO. The tool automatically inserts OpenMP directives in Fortran programs and applies a degree of optimization with nominal user interaction. CAPO is aimed at taking advantage of the detailed interprocedural data dependence analysis provided by Computer-Aided Parallelization Tools (CAPTools) [3], developed by the University of Greenwich, to reduce potential errors made by users and, with nominal help from user, achieve performance close to that obtained when directives are inserted by hand. Our approach is different from other tools and compilers in two respects: 1) emphasizing the quality of dependence analysis and relaxing much of the time constraint on the analysis; 2) performing directive insertion and preserving the original code structure for maintainability. Translation of OpenMP codes to executables is left to dedicated OpenMP compilers.

In this section, we outline the OpenMP programming model, give an overview of CAPTools, and then its extension, CAPO, for generating OpenMP programs. To better understand and use the tools, we also describe the basic concept of data dependence here.

## 2.1. The OpenMP Programming Model

OpenMP [12] was designed to facilitate portable implementation of shared memory parallel programs. It includes a set of compiler directives and callable runtime library routines that extend Fortran, C and C++ to support shared memory parallelism. It can provide an incremental path for parallelizing sequential software, as well as targeting the scalability and performance for any complete rewrites or new construction of applications.

OpenMP follows the *fork-and-join* execution model. A fork-and-join program initializes as a single lightweight process, called the *master thread*. The master thread executes sequentially until the first parallel construct (`OMP PARALLEL`) is encountered. At that point, the master thread creates a team of threads, including itself as a member of the team, to concurrently execute the statements in the parallel construct. When a work-sharing construct such as a parallel do (`OMP DO`) is encountered, the workload is distributed among the members of the team. An implied synchronization occurs at the end of the `DO` loop unless a "`NOWAIT`" is specified. Data sharing of variables is specified at the start of parallel or work-sharing constructs using the `SHARED` and `PRIVATE` clauses. In addition, reduction operations (such as summation) can be specified by the `REDUCTION` clause. Upon completion of the parallel construct, the threads in the team synchronize and only the master thread continues execution. The fork-and-join process can be repeated many times in the course of program execution. However, it should be appreciated that for every fork (`PARALLEL` region) there is an associated setup cost that is machine dependent.

Beyond the inclusion of parallel constructs to distribute work to multiple threads, OpenMP introduces a powerful concept of *orphan directives* that greatly simplifies the task of implementing coarse grain parallel algorithms. Orphan directives are directives outside the lexical extent of a parallel region. This allows the user to specify control or synchronization from anywhere inside the parallel region, not just from the lexically contained region.

## 2.2. CAPTools

The Computer-Aided Parallelization Tools (CAPTools) [3] is a software toolkit that was designed to automate the generation of message-passing parallel code. CAPTools accepts FORTRAN-77 serial code as input, performs extensive dependence analysis, and uses domain decomposition to exploit parallelism. The toolkit employs sophisticated algorithms to calculate execution control masks and attempts to minimize communication cost. The generated parallel code contains calls to a portable interface to message passing standards, such as MPI and PVM, through a low-overhead library (CAPLib).

There are two important strengths that make CAPTools stand out. Firstly, an extensive set of enhancements to the conventional dependence analysis techniques [8] has allowed CAPTools to obtain much more accurate dependence information, and thus, produce more efficient parallel code. Secondly, the toolkit contains a set of browsers [9] that allow the user to inspect and assist in the parallelization at different stages.

## 2.3. Generating OpenMP Directives

The goal of developing computer-aided tools to help parallelize applications is to let the tools do as much as possible and to try and minimize the amount of tedious and error-prone work performed by the user. The key to automatic detection of parallelism in a program, and thus parallelization, is to obtain accurate data dependences in the program. Generating OpenMP directives is simplified somehow because we are now working in a globally addressed space without being explicitly concerned about data distribution. However, we still have to realize that there are always cases in which certain conditions could prevent tools from detecting possible parallelization, thus, an interactive user environment is also important.

The design of CAPO had kept the above tactics in mind. CAPO uses the data dependence analysis engine in CAPTools, exploits loop level parallelism in a program and inserts OpenMP directives automatically. The schematic structure of CAPO is illustrated in Figure 1. CAPO takes a serial code as input and first performs the data dependence analysis. User knowledge on certain input parameters in the source code may be entered to assist this analysis for more accurate results. The process of generating OpenMP directives is summarized in the following three stages.

1) *Identify parallel loops and parallel regions*. The loop-level analysis is carried out to classify loops as parallel (including reduction), serial or potential pipeline based on the data dependence information. Parallel loops to be distributed with work-sharing directives for parallel execution are identified by

traversing the call graph of the program from the top downwards. Only outer-most parallel loops are considered, partly due to the very limited support of multi-level parallelization in available OpenMP compilers. Parallel regions are then formed around the distributed parallel loops. An attempt is also made to identify and create parallel software pipelines.

2) *Optimize loops and regions.* This stage is mainly for reducing the overhead caused by the fork-and-join and synchronization. A parallel region is first expanded as far as possible and may include calls to subroutines that contain additional (*orphaned*) parallel loops. Regions are then merged together if there is no violation of data usage in doing so. Region expansion is currently limited to within a subroutine. Synchronization between loops in a parallel region is optimized by trying to prove if the loops can be executed asynchronously.

3) *Transform codes and insert directives.* Variables in common blocks are analyzed for their usage in all parallel regions in order to identify threadprivate common blocks. If a private variable is used in a non-threadprivate common block, the variable is treated with a special code transformation. A routine needs to be duplicated if its usage conflicts at different calling points. By traversing the call graph, OpenMP directives are then added for identified parallel regions and parallel loops with variables properly listed. The variable usage analysis is performed at several points to identify how variables are used (e.g. private, shared, reduction, etc.) in a loop or region. Such analysis is required for the identification of loop types, the construction of parallel regions, the treatment of private variables in common blocks, and the insertion of directives.

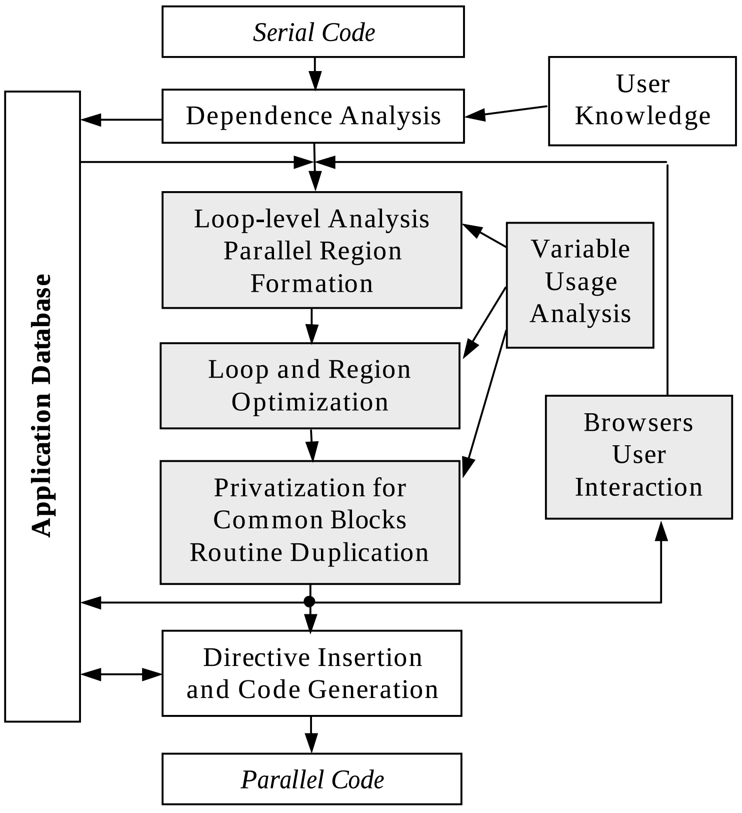

Serial Code
Dependence Analysis
User Knowledge
Application Database
Loop-level Analysis Parallel Region Formation
Variable Usage Analysis
Loop and Region Optimization
Browsers User Interaction
Privatization for Common Blocks Routine Duplication
Directive Insertion and Code Generation
Parallel Code

*Figure 1: Schematic flow chart of the CAPO architecture.*

Intermediate results can be stored into or retrieved from a database. User assistance to the parallelization process is possible through browsers implemented in CAPO (the Directives Browser) and in CAPTools. The Directives Browser is designed to provide more interactive information from the parallelization process, such as reasons why loops are parallel or serial, distributed or not distributed. The user can concentrate on areas where potential improvements could be made, for example, by removing false data dependences. This is a typical part of the iterative process of parallelization.

## 2.4. Data Dependence

To be able to use many of the facilities provided by CAPTools/CAPO it is important to understand what a dependence is in terms of the source code for a program. The dependence analysis performed by CAPTools builds a dependence graph where the nodes in the graph represent executable statements and the arcs of the graph represent relationships between statements. For the purposes of code generation and parallelism, the arcs (dependences) show the required execution order of the connected nodes (statements) to achieve semantically valid code. Dependence is the fundamental building block for the tool-based parallelization. Any generated code, parallel or serial, that does not violate any of the dependences is valid.

There are four basic types of dependences in a program source. For two statements, S1 preceding S2, in a program, the execution order of S1 and S2 cannot be changed if any one of the following conditions exists:

**1)** *True dependence* – data is written in S1 and read in S2.

For example,

```
S1: A(I) = ....
S2: .... = A(I) ....
```

present true dependence between source S1 and sink S2 caused by A(I) being assigned in S1 and used in S2. Obviously the sink of the dependence cannot execute until the source has assigned the required value.

**2)** *Anti dependence* – data is read in S1 and written in S2.

For example,

```
S1: .... = A(I)
S2: A(I) = ....
```

present anti dependence between source S1 and sink S2 caused by A(I) being reassigned in S2. S2 therefore cannot execute until after S1 has used the value that S2 will overwrite.

**3)** *Output dependence* – data is written in S1 and written again in S2.

For example,

```
S1: A(I) = ....
S2: A(I) = ....
```

present output dependence between source S1 and sink S2 caused by A(I) being reassigned in S2. The order of assignment to the memory location must be maintained and the value in that location after the execution of both statements must be that provided by the sink statement.

**4)** *Control dependence* – S2 is executed only if the condition in S1is satisfied.

For example,

```
S1: IF (A(I).EQ.5) THEN
S2:   B(J) = ....
```

present control dependence between source S1 and sink S2. Obviously the execution of S2 depends on the Boolean expression in S1 being calculated. The controlled statements (S2) cannot execute until the controlling statement (S1) has executed.

True and control dependences are actual algorithmic dependences caused by information flow. Anti and output dependences (jointly referred to as pseudo dependences) are caused by re-use of memory locations and are not inherent to the code. Pseudo dependences may be removed by introducing intermediate working variable(s).

Dependences can further be marked to indicate whether they exist between iterations of a particular loop, or if they exist independent of the surrounding loops. Loop-related dependences directly affect the parallelization of a loop.

**5)** *Loop-carried dependence* – A dependence between two statements (or two instances of the same statement) between iterations of the loop surrounding both statements. *Dependence level* is the nesting level (or depth) of the carried loop in the current loop nest.

For example,

```
   DO 10 J=2,NJ-1
     DO 20 I=2,NI-1
       A(I,J) = A(I,J-1)
20   CONTINUE
10 CONTINUE
```

each iteration of the J loop (except in this case the first iteration) uses a value of array A assigned in the previous iteration of the J loop. This is a loop-carried true dependence on array A by the J loop with a dependence level of 1.

**6)** *Loop independent dependence* – A dependence between two statements that exists during a single iteration of all loops that surround both statements. Loop independent dependences are marked with a level of infinity.

For example,

```
    DO 10 J=2,NJ-1
      DO 20 I=2,NI-1
S1:     A(I,J) = ....
S2:     .... = A(I,J)
 20   CONTINUE
 10 CONTINUE
```

the dependence on array A between source S1 and sink S2 is a loop independent dependence.

**7)** *Loop entry/exit dependence* – A true dependence between two statements, one being surrounded by the loop and the other being outside the dynamic extent of the loop.

Entry or exit is determined by whether the source or the sink is outside the loop. A special case is a loop-carried dependence with a dependence level less than the current loop nesting level. In this case both statements (or two instances of the same statement) may be surrounded by the loop, but dynamically one instance of the statements is outside the loop.

For example, for the I loop in the following code:

```
S1: A(2,1) = ....
    DO 10 J=2,NJ-1
      DO 20 I=2,NI-1
S2:     A(I,J) = A(I,J-1)
 20   CONTINUE
 10 CONTINUE
S3: .... = A(2,2)
```

the true dependence between source S1 and sink S2 is a loop entry dependence and the one between source S2 and sink S3 is a loop exit dependence. The true dependence between two instances of statement S2 carried by the J loop at level 1 contributes to both loop entry and loop exit dependences for the I loop.

When parallelization is considered loop entry/exit dependence determinates if a local variable needs to be *copied in* upon the entry of a loop and *copied out* upon the exit of a loop.

By analogy to loop entry/exit dependence, dependences can also be marked to indicate whether they exist upon the entry/exit of a routine. They are usually referred to as *routine input/output dependences.*

The last important concept related to dependence is encountered when the iteration space defined by iterations of the loops in a loop nest is considered for setting up pipelines (see Section 4.1 for an example of pipeline).

**8)** *Dependence vector* – A vector formed by loop-carried dependences in the loop iteration space that is defined by the loops of consideration in a loop nest.

For example,

```
       DO 10 J=2,NJ
         DO 20 I=2,NI
           A(I,J) = A(I-1,J) + A(I,J-1)
    20   CONTINUE
    10 CONTINUE
```

the loop-carried dependence of `A(I,J)` on `A(I-1,J)` forms a dependence vector of `[1,0]`, and the loop-carried dependence of `A(I,J)` on `A(I,J-1)` forms a dependence vector of `[0,1]`. The iteration space in this case is defined by the iterations of the I and J loops, as illustrated in Figure 2, and a dependence vector is shown as an arrow in the figure.

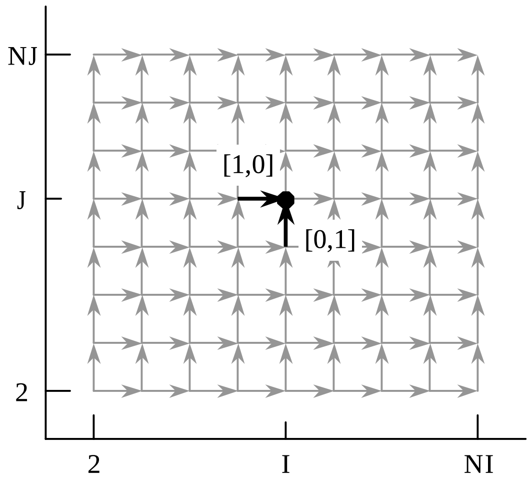


*Figure 2: Dependence vectors formed in the loop iteration space.*

# 3. Producing Parallel Code with CAPO

This section describes the usual steps a user will take to produce parallel code with CAPO. The procedure follows the outline given in Figure 1. One can refer to the Tutorials and Appendix for more information. It is also important to keep in mind that in order to get an efficient parallel code, user interaction with the tools is almost always needed. The optimization process with the CAPO Directive Browser is given in Section 4.

## 3.1. Prepare Serial FORTRAN Codes

CAPO currently works on FORTRAN 77 codes. A user can either create a single file that contains all the subroutines or provide a `.list` file that lists all the FORTRAN source files in the program. Figure 3 shows an example of an "`All.list`" file. Note that the source directory structure is preserved and the file names can be used later in the code generation.

Any unresolved symbols can be defined using dummy routines. For example, if the FORTRAN program calls C subroutines, dummy FORTRAN routines could be supplied to emulate the C functions even through these dummy routines may be deleted later on from the generated parallel code. This was a requirement of CAPTools prior to Version 2.1. The latest CAPTools provides interfaces to the dummy routines automatically.

CAPTools does not accept source codes that contain pre-processing directives. It is necessary to preprocess these files before use in CAPTools. Although the toolkit tries to preserve the original source form, these preprocessing directives will be lost.

```
lu.f
blts.f
buts.f
domain.f
erhs.f
error.f
exact.f
jacld.f
jacu.f
l2norm.f
pintgr.f
read_input.f
rhs.f
setbv.f
setcoeff.f
setiv.f
ssor.f
verify.f
../common/print_results.f
../common/timers.f
../common/wtime.f
```

*Figure 3: An example of "All.list".*

## 3.2. Make Dependence Analysis

Data dependence analysis is performed on the whole program, which is one of the key steps for the directives generation. After source files are loaded into CAPTools, user knowledge may be entered, for instance for the range of variables from the READ statements in the code. User supplied information can help obtain more accurate data dependences, and thus, more efficient parallel code. An example is illustrated in the following code:

```
      read(*,*) isize
      do 10 j=1,jm
      do 10 i=1,im
        ix = i + (j-1)*isize
        A(ix) = A(ix) + B(i,j)
   10 continue
```

The value of the parameter `isize` affects the loop parallelization. For the `j` loop, if `isize` > 0, no loop-carried data dependence exists for variable A; if `isize` = 0, there are loop-carried data dependences for variable A. The ambiguity in the `isize` value will prevent the `j` loop from being parallelized, i.e. a data dependence on variable A will be assumed. The user could supply the information "`isize` > 0" to improve the accuracy of the analysis.

Depending on the program size and the thoroughness of the analysis specified, the dependence analysis process can take minutes, hours or days to complete. Once the analysis is finished, the user should save the results to a database before proceeding further. The dependence analysis is the most CPU intensive part of the parallelization process. Table 1 lists the CPU time spent on analyzing the NPB BT benchmark on several machines. The analysis uses a single CPU. As one can see, the analysis time is roughly proportional to the clock speed of a processor.

*Table 1: CPU time spent by CAPTools on analyzing the NPB BT benchmark on several machines.*

| **Machine Type** | **OS Type** | **CPU Time** |
|---|---|---|
| Intel PIII, 500MHz<br>512MB RAM, 512KB Cache | Linux | 10.5 mins |
| Intel PII, 300MHz<br>512MB RAM, 512KB Cache | Linux | 16.4 mins |
| Sun UltraSparcII, 360 MHz<br>1GB RAM, 16KB L1, 4MB L2 | Solaris | 15.0 mins |
| Sun UltraSparcII, 300 MHz<br>2GB RAM, 16KB L1, 4MB L2 | Solaris | 17.6 mins |
| SGI R5K, 150 MHz<br>128MB RAM, 32KB L1 | IRIX | 71.4 mins |
| SGI R10K, 195MHz<br>512MB RAM, 32KB L1, 1MB L2 | IRIX | 26.4 mins |
| SGI R12K, 300MHz<br>1GB RAM, 32KB L1, 2MB L2 | IRIX | 17.8 mins |

## 3.3. Inspect Loops and Optimize Directive Generation

The parallelization strategy in CAPO is loop-based, thus an important next step is to inspect loops after the dependence analysis is performed which may involve inspecting the dependences produced by CAPTools. Quite often a dependence causing a loop to be serialized is due to insufficient knowledge of value limits for some variables, as indicated in the previous section. The user can use the dependence browser (DepGraph) to remove unnecessary dependences. However, the information in the DepGraph window could be overwhelming for a novice user.

An alternative approach for inspecting the loops is to use the Directives Browser implemented in CAPO (see Section 4 for details). The browser can be activated from the **View→Directives** menu and is designed to display information that was gathered from the directives analysis and is directly related to the directives inserted. For instance, the browser provides more interactive information on the reasons for loops to be parallel or serial and the relevant variables. The user can concentrate on loops that are indicated as serial and the possible optimization of the dependence graph if needed. It is also possible to enforce a user-defined loop type. After changes are made, the directive analysis is re-applied to take into account these changes. This is an iterative process (see Figure 1). It is always a good idea to save the incremental results to a database whenever a change is made before directives are actually inserted.

One should keep in mind that the CAPO/CAPTools parallelization relies on the static analysis of the serial code. The dynamic information cannot be detected and applied by the toolkit. Thus, in most cases a user-guided parallelization process is the only way to achieve a good quality parallel code.

## 3.4. Generate Parallel Code with Directives

Once the dependence analysis is completed and the loop information is inspected, directives can automatically be generated and inserted by selecting the "***Save OpenMP Directive Code***" option under the **File** menu. The type of directive(s) is controlled by the CAPO parameters (as described in Appendix 1), which are also selectable from the Setting box in the Directives Browser. One can elect to use the default setup, which is to produce OpenMP directives with a full power analysis. Steps in the generation of directives are logged to a file, by default to "`code-output.log`." Contents of the log file are described in Appendix 2.

## 3.5. Inspect the Generated Code and the Log Information

It is very important to inspect the generated parallel code together with the log information in the log file. In particular, one should look into any shared variables, private variables and I/O statements that are potentially incorrectly listed. Warnings in the last section (PASS 3) of the log file can indicate places where potential problems might exist. Of course, one can use other tools (such as ASSURE from Kuck and Associate [10]) to check for potential problems in the parallel code.

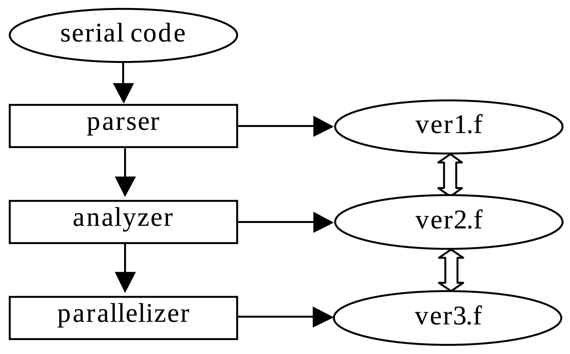


*Figure 4: Compare source codes at different stages.*

Sometimes it is useful to find out what might have been changed at different stages of code parallelization. In the framework of CAPO, one can compare codes created at three stages as shown in Figure 4: parsing, analyzing and parallelizing. The codes can be simply compared with for example the Unix '`diff`' command. Comparison of ver1.f and ver2.f will review code sections that were deemed to be redundant and were removed by the CAPTools dependence analysis process. Comparison of ver2.f and ver3.f will review the change from serial to parallel defined by the directives inserted and other code transformations.

## 3.6. Compile and Run the Parallel Code

Once the parallel code is generated use an OpenMP compiler to compile the code. Typically a compiler option is required to enable the directives. For example on the SGI Origin2000, the "`-mp`" option is needed for the SGI MIPSpro compiler to compile codes with OpenMP features.

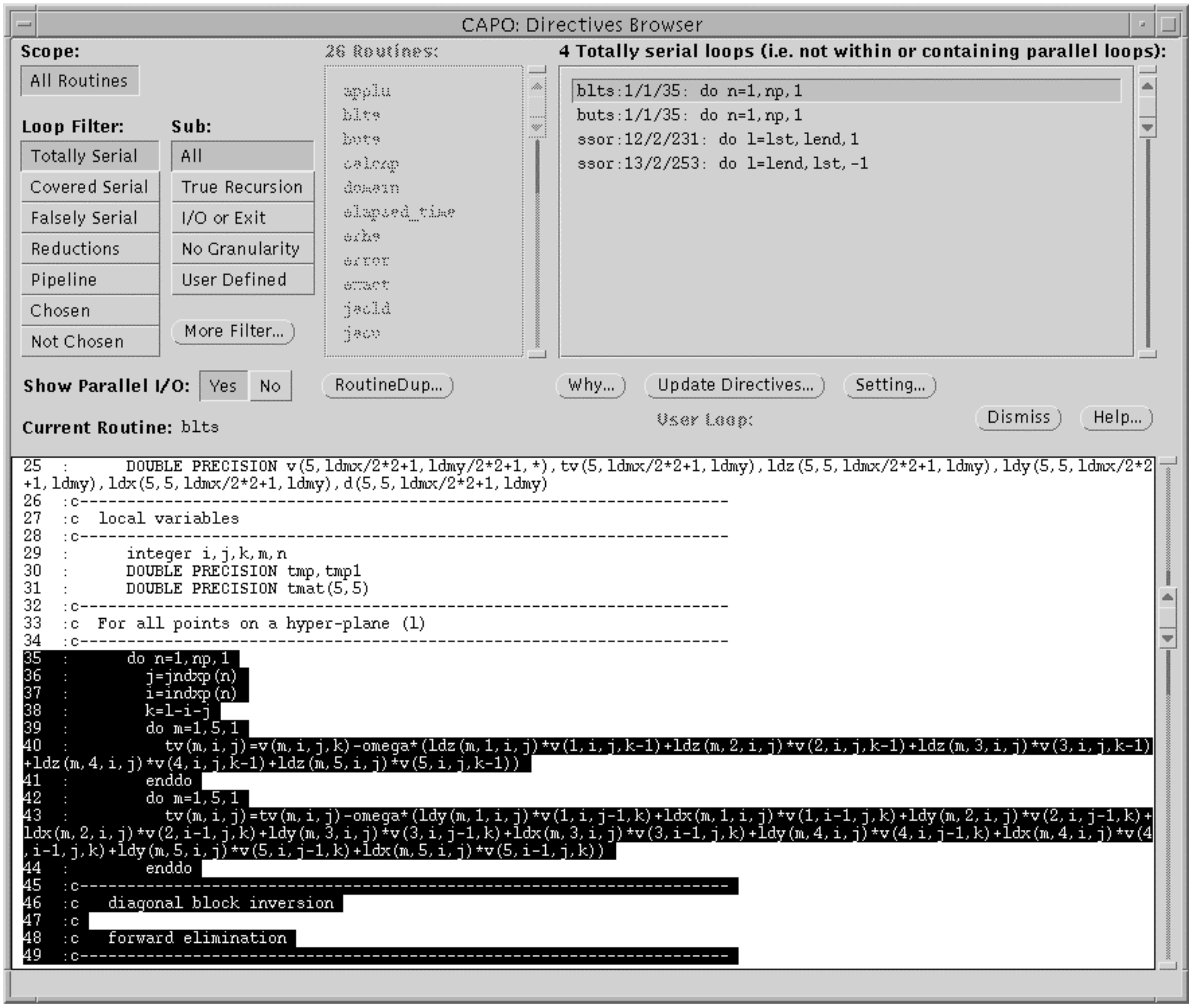
```
% f77 -o a.out -mp -O parallelcode.f
```

To run the code with 8 CPUs, do

```
% setenv OMP_NUM_THREADS 8
% ./a.out
```

# 4. Interacting with the Directives Browser

As mentioned before although the dependence analysis carried out is very detailed, it can often contain dependences that had to be assumed to exist. In these cases, user assistance can be used to improve the quality of the generated OpenMP code. This is done by classifying the different types of loops that generally exist in application codes and using the Directives Browser to inspect and interrogate all the loops in turn. The Directives Browser is activated from the ***View*** menu of CAPO after CAPO finishes the directive analysis (see Figure 5 for the main window of the browser). The browser displays loops according to their types and provides more interactive information on the reasons why loops are parallel or serial. The user can concentrate on loops that are indicated as serial (fully or covered, as given below). The user can also enforce the classification of a selected loop by re-defining the loop type or define the granularity threshold for a loop so that any loop below this level is not considered for parallelization. Another feature of the browser is to provide the access for the user to manipulate the dependence graph (in conjunction with the DepGraph Browser) and improve the quality of the parallelization.



4 Totally serial loops (i.e. not within or containing parallel loops):

```
blts:1/1/35:  do n=1,np,1
buts:1/1/35:  do n=1,np,1
ssor:12/2/231:  do l=lst,lend,1
ssor:13/2/253:  do l=lend,lst,-1
```

Current Routine: blts

```
         DOUBLE PRECISION v(5,ldmx/2*2+1,ldmy/2*2+1,*),tv(5,ldmx/2*2+1,ldmy),ldz(5,5,ldmx/2*2+1,ldmy),ldy(5,5,ldmx/2*2
+1,ldmy),ldx(5,5,ldmx/2*2+1,ldmy),d(5,5,ldmx/2*2+1,ldmy)
c---------------------------------------------------------------------
c  local variables
c---------------------------------------------------------------------
         integer i,j,k,m,n
         DOUBLE PRECISION tmp,tmp1
         DOUBLE PRECISION tmat(5,5)
c---------------------------------------------------------------------
c  For all points on a hyper-plane (l)
c---------------------------------------------------------------------
         do n=1,np,1
           j=jndxp(n)
           i=indxp(n)
           k=l-i-j
           do m=1,5,1
             tv(m,i,j)=v(m,i,j,k)-omega*(ldz(m,1,i,j)*v(1,i,j,k-1)+ldz(m,2,i,j)*v(2,i,j,k-1)+ldz(m,3,i,j)*v(3,i,j,k-1)
+ldz(m,4,i,j)*v(4,i,j,k-1)+ldz(m,5,i,j)*v(5,i,j,k-1))
           enddo
           do m=1,5,1
             tv(m,i,j)=tv(m,i,j)-omega*(ldy(m,1,i,j)*v(1,i,j-1,k)+ldx(m,1,i,j)*v(1,i-1,j,k)+ldy(m,2,i,j)*v(2,i,j-1,k)+
ldx(m,2,i,j)*v(2,i-1,j,k)+ldy(m,3,i,j)*v(3,i,j-1,k)+ldx(m,3,i,j)*v(3,i-1,j,k)+ldy(m,4,i,j)*v(4,i,j-1,k)+ldx(m,4,i,j)*v(4
,i-1,j,k)+ldy(m,5,i,j)*v(5,i,j-1,k)+ldx(m,5,i,j)*v(5,i-1,j,k))
           enddo
c---------------------------------------------------------------------
c   diagonal block inversion
c
c   forward elimination
c---------------------------------------------------------------------
```

*Figure 5: The Directives Browser main window.*

## 4.1. Loop Classification

The loops are identified in the browser for the following types:

i. *Totally serial loops* – These loops contain a loop-carried true data dependence that causes the serialization of the loop i.e. data assigned in an iteration of the loop is used in a later iteration. (Other possible reasons for a loop to be defined as serial include the presence of I/O or loop exiting statements within the loop body). In addition, this loop type does not contain any nested parallel loops and also is not contained within a parallel loop. The directive browser shows a list of the variables and a textual explanation of why the loop is serial. However, the data dependence(s) may have been assumed to exist and the user may be able to supplement the dependence analyzer with additional information to prove that the data dependence(s) do not exist. Alternatively, the user may wish to enforce the removal of a serializing data dependence, again using the dependence browser.

ii. *Covered serial loops* – These are also serial loops containing a loop-carried true data dependence, so they can be treated in a similar way to totally serial loops. However, this type of serial loop is either nested within a parallel loop or contains parallel loops within it. In the latter case, if the serial loop can be made parallel (see *totally serial loops*) then the parallelism can be defined at a higher level and may therefore enhance the performance of the execution in parallel.

iii. *Falsely serial loops* – These loops are not serial due to a loop-carried true dependence. Instead, they will need to execute in serial due to the existence of pseudo dependencies that represent memory re-use as this needs to be considered when working within a globally addressable memory. The directive and dependence browsers can be used together with any additional information the user may wish to offer to re-examine if the variable(s) concerned can be privatized. In the process, dependencies into or out of the loop are examined to test if the variable could be made `PRIVATE`, or to re-examine if the loop carried pseudo dependencies are needed, in an attempt to allow the loop to execute in parallel.

iv. *Reduction loops* – The analysis is used to determine if the loop body computations represent a global reduction operation such as a `MAX` or summation. These loops provide a partial update of the results by each thread followed by a global update to give the final reduction value.

v. *Pipeline loops* – This is a special class of serial loops with loop-carried true dependences. The use of directive-based software pipelines exploits parallelism in this type of loops. Figure 6 shows an example where OpenMP function calls are used to define the pipeline start-up before the J-loop and the pipeline shutdown after the loop. The example is taken from a version of the NAS LU benchmark. This is a similar strategy to that adopted for a software pipeline used in a distributed memory parallelization with message passing. For comparison a software pipeline implementation using a high level message-passing library (CAPLib) is shown in the lower panel of Figure 6.

vi. *Chosen parallel loops* – These are the parallel loops at which the `OMP DO` construct is defined. These loops may contain serial or parallel loops within their nesting but are not surrounded by other parallel loops.

vii. *Not chosen parallel loops* - Also parallel loops, but these have not been selected for application to the `OMP DO` directive. This is because these loops are surrounded by other parallel loops at a higher nesting level. In general, the OpenMP compiler suppliers do not currently support nested parallelism, therefore, even though parallelism exists at these lower levels, it is not currently exploited.

The sub filter can be used together with the loop filter to control the finer selection of loop types. Detailed explanation of these filters can be found in Appendix 3.2 and examples of using the loop filters are given in the Tutorials.

```
(a)    lloop = jend-jst
       if (lloop .gt. mthnum) lloop = mthnum
       iam = omp_get_thread_num()
       if (iam .gt. 0 .and. iam .le. lloop) then
         neigh = iam - 1
         do while (isync(neigh) .eq. 0)
!$OMP FLUSH(isync)
         end do
         isync(neigh) = 0
!$OMP FLUSH(isync)
       endif
!$OMP DO SCHEDULE(STATIC)
       do j=jst,jend,1
         do i=ist,iend,1
c forward elimination and back substitution for diag. block inversion
         enddo
       enddo
!$OMP END DO nowait
       if (iam .lt. lloop) then
         do while (isync(iam) .eq. 1)
!$OMP FLUSH(isync)
         end do
         isync(iam) = 1
!$OMP FLUSH(isync)
       endif
```

```
(b)    CALL CAP_RECEIVE(v(1,2,LOW-1,k),nx0*5-10,3,CAP_LEFT)
       do j=MAX(jst,jst+LOW-2),MIN(jend,jst+HIGH-2),1
         do i=ist,iend,1
c forward elimination and back substitution for diag. block inversion
         enddo
       enddo
       CALL CAP_SEND(v(1,2,HIGH,k),nx0*5-10,3,CAP_RIGHT)
```

*Figure 6: Implementation of a software pipeline for routine BLTS using (a) OpenMP (b) message passing.*

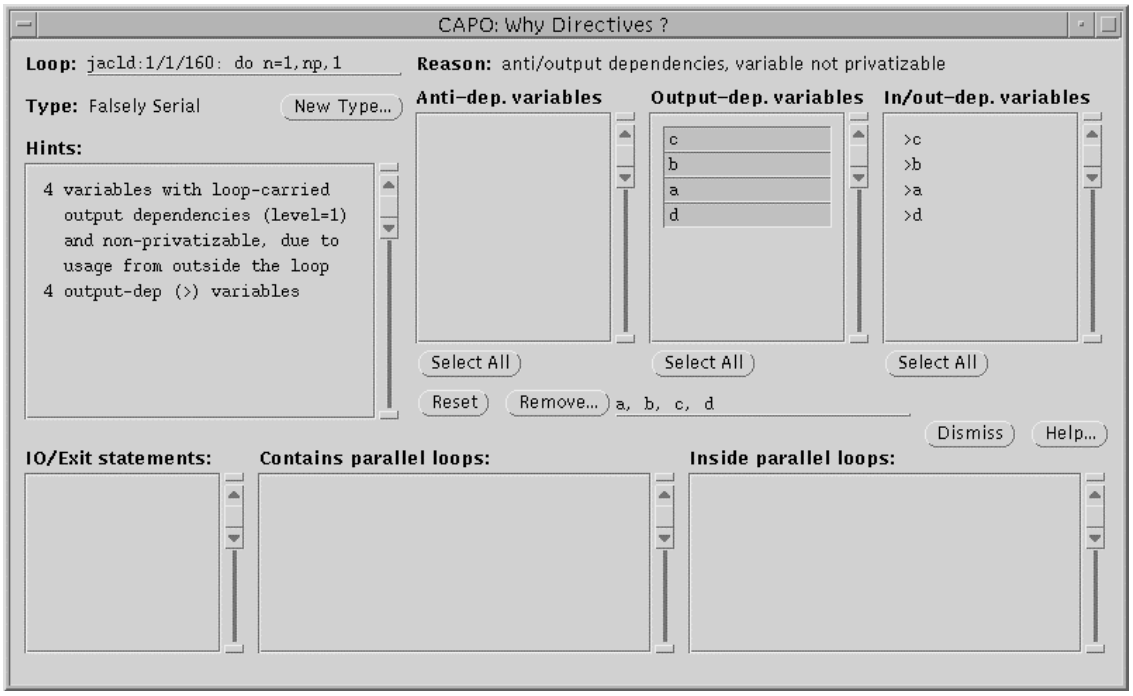
## 4.2. Browsing Different Types of Loops

The accurate dependence analysis allows the algorithm to automatically generate efficient OpenMP code in many cases. Experience has shown that this typically leaves a small proportion of cases that require user interaction. For example, the use of workspace arrays is very common in application codes. The value-based nature of the dependence analysis will often prove that no data is passed between iterations of a loop, but the memory re-use (pseudo) dependences must however be set. This correctly does not classify such loops as serial, however, the legal privatization of these arrays to allow parallel execution requires that no data is passed into or out of these arrays from or to outside the loop, i.e. no loop entry/exit dependence on these arrays (see Section 2.4).

Normally the user wants to go through the following loop types and use the **WhyDirectives** window to find out the reason(s) for the setting of a particular loop type:

- `Totally Serial->True Recursion`
- `Covered Serial->True Recursion`
- `Falsely Serial->Privatization`

- `Chosen->CopyIn/Out`



```
4 variables with loop-carried
  output dependencies (level=1)
  and non-privatizable, due to
  usage from outside the loop
4 output-dep (>) variables
```

*Figure 7: The WhyDirectives window for a falsely serial loop.*

The **WhyDirectives** window (as shown in Figure 7) can be activated by clicking on the Why... button in the **Directives Browser** window once a loop is selected. The window displays information on variables that cause a loop to be so classified. The cause for a loop not to be parallel can come from several sources, for example, loop-carried TRUE/ANTI/OUTPUT dependence, non-privatizable variables (re-use of memory). If the user is sure that some of these dependences are false (mostly due to lack of input information for the dependence analysis) and can be removed, the Dep-Graph browser can be used to modify the dependence graph. A shortcut is provided in the **WhyDirectives** window where variables can be selected from the variable-list boxes and the relevant dependences can be removed by clicking the Remove button. The following relevant dependences (see Section 2.4 for an explanation of different dependences) will be removed, based on the loop type and variable list type:

| **Loop Type** | **List Type** | **Dependence Type** |
|---|---|---|
| Totally Serial | True-dep. | Loop-carried TRUE dependence |
| | Anti-dep. | Loop-carried ANTI dependence |
| | Output-dep. | Loop-carried OUTPUT dependence |
| Covered Serial | True-dep. | Loop-carried TRUE dependence |
| | Anti-dep. | Loop-carried ANTI dependence |
| | Output-dep. | Loop-carried OUTPUT dependence |
| Falsely Serial | Anti-dep. | Loop-carried ANTI dependence |
| | Output-dep. | Loop-carried OUTPUT dependence |
| | In/Out-dep. | TRUE dependence from outside of the loop |
| Chosen Parallel | CopyIn/Out | TRUE dependence from outside of the loop |

Once a change to the dependence graph (either via the Dep-Graph browser or via the WhyDirectives browser) is made, be sure to save the change to the database (***File→Save Database***) and re-perform the directive analysis (Update Directives... button).

## 4.3. Enforcing New Loop Type

A loop type as described in the previous section and defined by CAPO can be overridden by the user with the **LoopType** dialog box which is activated from the New Type button (see Figure 7). Typically this may occur when a loop is chosen for parallelization by CAPO but does not have proper granularity, or the user may want to force it to be serial and let the tool choose another loop that is nested inside this loop. Another possibility is when the user wants to enable parallelization for a loop that contains I/O statements.

Currently the following four types can be enforced by the user:

| | | |
|---|---|---|
| `Parallel` | - | from parallel loop without granularity or with I/O statements |
| `Serial` | - | from parallel loop, including reduction |
| `Reduction` | - | from serial loop with loop-carried true dependence |
| `Break` | - | from any other loop types. |

When a loop is enforced as the “`Break`” type, the loop will not be included in a parallel region. Only the conversions as indicated are possible from the dialog box. Although loop types can also be redefined from the user-defined loop file (see Appendix 1.3), use of the **LoopType** dialog box is safer. However, one should keep in mind that changing the loop type manually could potentially lead to incorrect results if the above rule is not carefully followed.

## 4.4. Routine Duplication

Routine duplication is performed after all the loop-level analyses and optimizations are done but before directives are inserted. A routine may be duplicated if it causes usage conflicts at different calling points. For example, if a routine contains parallel regions and is called both inside a parallel loop and outside another parallel loop but still inside a parallel region, the routine is duplicated so that the copy of the routine without directives is used inside the parallel loop and the second copy containing only orphaned directives without “`OMP PARALLEL`” is used inside parallel regions but outside parallel loops. Routine duplication is often used in a message-passing based parallelization to handle different data distributions in the same routine.

There are two selectable types of routine duplication (see the Settings in Appendix 3) for a routine that contains parallel regions in the dynamic extent of this routine:

- '`Loop`' as the type for routine duplication if the routine is called both inside and outside parallel loop(s).
- '`Region`' as the default type for routine duplication if the routine is called inside parallel loop(s) and inside parallel region(s) but outside parallel loop(s).

The first option removes any nesting of parallel regions. The second option allows nested parallel regions in such a form that a parallel region can be nested inside a parallel loop but not inside a non-worksharing section of a parallel region.

The **RoutineDup** browser (from ***View→Directives→RoutDup***) is used for browsing routines that will be duplicated. The browser will indicate those calls that are inside parallel loops and those that are outside parallel loops. One may inspect the calls that are outside parallel loops for possible improvements, for example, de-serializing any potential outside loop nests.

# 5. Other Features

## 5.1. CAPO Parameters and Log Information

Parameters refer to inputs that the user can supply to control the behavior of directive generation in CAPO. A list of all the parameters is given in Appendix 1. These parameters can be defined from a file, environment variables, the Setting window in the Directives Browser, or the CAPO command interface. All the parameters have default values. The Setting window from the Directives Browser is the most straightforward way to change parameters. It allows the user to select the log information type, define the directive type, set the loop granularity for parallelization, enable/disable the generation of the `THREADPRIVATE` directive, etc. For example, if the **Directive Type** is set to *No Directive*, the generated code will not contain directives and any associated transformations as indicated in the next section.

By default, the process of automatic insertion of directives is logged to the log-file "`code-output.log`." Information in this file may be examined after directives are added. There are three main sections in the log file, as outlined in Appendix 2. Depending on the log-info type, different levels of information detail may be logged. In general, the log-info type controls:

1) `min` — only minimum amount of information, such as WARNING and INFO messages,
2) `std` — information from `min`, plus summary for each routine and each region,
3) `more` — information from `std`, plus more detailed results for each loop and each region,
4) `debug` — information from `more`, plus additional debug information that are probably too much for an ordinary user.

Warning messages in the log file should be carefully examined since they may indicate potential problems in the generated parallel code.

## 5.2. Automatic Code Transformation and Optimization

CAPO performs the following code transformation and optimization automatically and logs the actions into the log file.

- Removal of the end-of-loop synchronization (using the `NOWAIT` construct) if it is proved valid. The function can be switched off from the parameter setting. Default is on.
- Loop nest interchange to improve cache performance. The array usage is analyzed against the loop nesting order for possible misalignment. Loop transformation is performed to reduce misalignment. The module is activated only when the O3 optimization is chosen. Default is O2.
- The ability to treat private variables with unknown size. A variable with unknown size is usually declared as "`(*)`" (sometimes as "`(1)`") for its last dimension in a subroutine. Use of such a variable as `PRIVATE` in a parallel region would cause ambiguity in size declaration and likely run-time error. In the current implementation, variable size is automatically detected (back tracing and usage checking) and dimension adjustment is then performed. Default is on.
- Reduction of an array is transformed into local array updates plus a global update in a critical section at the end. Default is on.
- Detection of reduction via an IF statement. The reduction is automatically transformed to local updates and a global update in a critical section at the end. This type of reduction is indicated as `IMIN` or `IMAX` in the Directives Browser. Default is on.

## 5.3. Command Interface and the Batch Mode

The command interface for CAPO is available in Version 1.1 and works closely with the CAPTools command interface. It provides a way to access the functionality of GUI components without starting the GUI. It serves as a means to record actions (to a log file) as a result of any user GUI activities so that these actions can be played back later. The commands in the command interface are usually recorded to a log file or a command file with

```
capo –logfile capo_run.cmd
```

and played back with

```
capo [-batch] capo_run.cmd.
```

The second line with the `[-batch]` option can be used to start a CAPO session in a batch mode. This is especially useful for the data dependence analysis since it is the most CPU intensive part and very little user interaction is required once the analysis is started. Refer to Appendix 4 for a list of CAPO commands and several useful CAPTools commands for the command interface.

## 5.4. Parallel I/O

Parallel I/O is not generally supported in CAPO. I/O is serialized by default, i.e., it is handled by the master thread only. If any I/O is in the dynamic extent of a loop nest, the loop will be executed sequentially. However, in some cases, one may want to exploit parallel I/O. For example in the following code:

```
DO K=1,NZ
  ...
  IF (V(K).LT.0.0) THEN
    WRITE(*,*) 'Warning: Negative value at K=', K
  ENDIF
END DO
```

The `WRITE` statement prints a warning message only when a condition is reached. If the order of the `WRITE` statement is not important, one may try to parallelize the loop.

Another commonly encountered case is that warning messages are printed inside subroutine calls while data are read/written in the current scope of a loop nest. One may want to ignore the warning messages inside subroutine calls but serialize loops containing I/O in the current scope.

The level of parallel I/O in CAPO is controlled by the parameter “CAPO_PIO”. If a value of “`incall`” is given, CAPO will ignore any I/O inside subroutine calls when parallel loops are considered. Another possible value is “`write`”, which allows any `WRITE` to stdout (`UNIT=*` or 6) inside parallel loops. This can be used for the above example. Of course, the user can always enforce a user-defined loop type. During the code generation warnings will be printed in the log file if I/O is encountered inside a parallel region. One can examine these warnings for potential problems.

## 5.5. Mix of Message Passing and OpenMP

As pointed out in Section 2, CAPTools is designed to generate message-passing codes while CAPO is used to create OpenMP codes. Mixing message passing (such as MPI) and OpenMP is possible in the framework of CAPTools since CAPO is integrated into it. A commonly used *hybrid* model is to have MPI for the coarse-grained parallelization and OpenMP for the fine-grained parallelization. Such a parallelization model is very effective if an application can be divided into domains and different domains are only loosely coupled. MPI is used for inter-domain parallelism and OpenMP for intra-domain parallelism.

Tutorial 5 gives an example of producing a mixed parallel code for the NAS BT benchmark. The tutorial simply illustrates the capability of the tools to generate mixed codes. However it should be noted that using a hybrid approach for parallelization is very application dependent.

# 6. Case Studies

For completeness in this section we present case studies using CAPO to parallelize the NAS parallel benchmarks and two computational fluid dynamics (CFD) codes well known in the aerospace field: ARC3D and OVERFLOW. The parallelization process described in Section 3 was adopted. We mainly present the results and discuss issues encountered in the parallelization. Most of the results have been reported in [6].

In the case studies, we used an SGI workstation (R5K, 150MHz) and a Sun E10000 node to run CAPO. The resulting OpenMP codes were tested on an SGI Origin2000 system, which consisted of 64 CPUs and 16 GB globally addressable memory. Each CPU in the system is a R10K 195 MHz processor with 32KB primary data cache and 4MB secondary data cache. The SGI's MIPSpro Fortran 77 compiler (7.2.1) was used for compilation with the "`-O3 -mp`" flag.

## 6.1. The NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPBs) were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. The NPB suite consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. The five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. Details of the benchmark specifications can be found in [1] and [2].

In this study we used six benchmarks (LU, SP, BT, FT, MG and CG) from the sequential version of NPB2.3 [2] with additional optimization described in [5]. Parallelization of the benchmarks with CAPO is straightforward except for FT where additional user interaction was needed. User knowledge on the grid size ($\geq 6$) was entered for the data dependence analysis of BT, SP and LU. In all cases, the parallelization process for each benchmark took from tens of minutes up to one hour, most of the time being spent in the data dependence analysis. The performance of CAPO generated codes is summarized in Figure 8 together with comparison to other parallel versions of NPB: MPI from NPB2.3, hand-coded OpenMP [5], and versions generated with the commercial tool SGI-PFA [14].

CAPO was able to locate effective parallelization at the outer-most loop level for the three application benchmarks and automatically pipelined the SSOR algorithm in LU. As shown in Figure 8, the performance of CAPO-BT, SP and LU is within 10% to the hand-coded OpenMP version and much better than the results from SGI-PFA. The SGI-PFA curves represent results from the parallel version generated by SGI-PFA without any change for SP and with user optimization for BT (see [14] for details). The worse performance of SGI-PFA simply indicates the importance of accurate interprocedural dependence analysis that usually cannot be emphasized in a compiler. It should be pointed out that the sequential version used in the SGI-PFA study was not optimized, thus, the sequential performance needs to be considered in the comparison. The hand-coded MPI versions scaled better, especially for LU. We attribute the performance degradation in the directive implementation of LU to less data locality and larger synchronization overhead in the 1-D pipeline used in the OpenMP version as compared to the 2-D pipeline used in the MPI version.

The directive code generated by CAPO for MG performs 36% worse on 32 processors than the hand-coded version, primarily due to an unparallelized loop in routine `norm2u3`. The loop contains two reduction operations of different types. One of the reductions was expressed in an IF statement, which was not detected by CAPO Version 1.0 (the IF reduction will automatically be detected by Version 1.1), thus, the routine ran in serial. Although this routine takes only about 2% of the total execution time on a single node, it translates into a large portion of the parallel execution on large number of processors, for example, 40% on 32 processors. All the tested parallel versions of CG achieved similar performance.

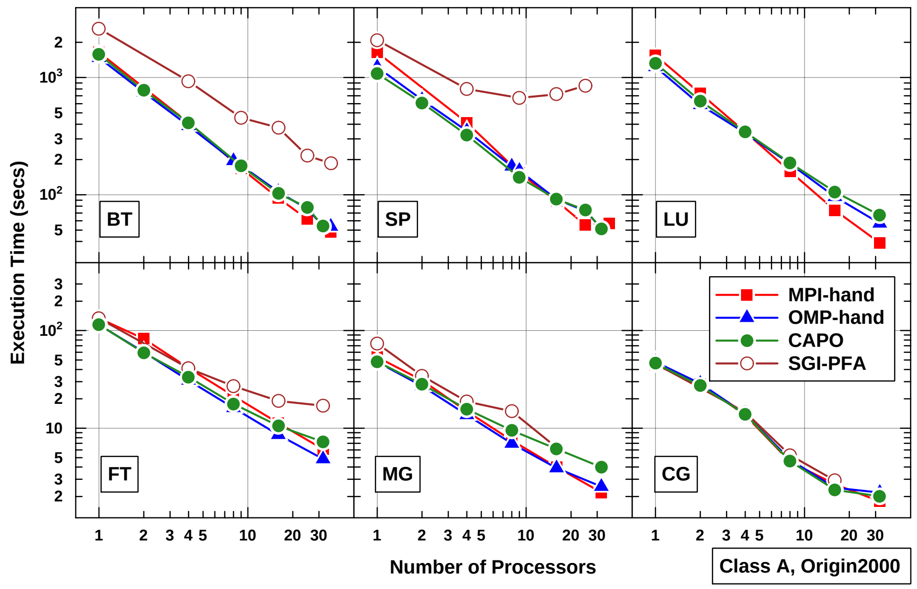


*Figure 8: Comparison of the OpenMP NPB generated by CAPO with other parallel versions: MPI from NPB2.3, OpenMP by hand, and SGI-PFA.*

The basic loop structure for the Fast Fourier Transform (FFT) in one dimension in FT is as follows.

```
DO K=1,D3
  DO J=1,D2
    DO I=1,D1
      Y(I) = X(I,J,K)
    END DO
    CALL CFFTZ(...,Y)
    DO I=1,D1
      X(I,J,K) = Y(I)
    END DO
  END DO
END DO
```

A slice of the 3-D data (X) is first copied to a 1-D work array (Y). The 1-D FFT routine CFFTZ is called to work on Y. The returned result in Y is then copied back to the 3-D array (X). Due to the complicated pattern of loop limits inside CFFTZ, CAPTools could not disprove the loop-carried true dependences on the working array Y for loop K. These dependences were deleted using the DepGraph browser to enable the analysis to identify that the K loop is a parallel loop.

The resulted parallel FT code gave a reasonable performance as indicated by the curve with filled circles in Figure 8. It does not scale as well as the hand-coded versions (both in MPI and OpenMP), mainly due to the unparallelized code section for the matrix creation which was artificially done with random number generators. Restructuring the code section was done in the hand-coded version to parallelize the matrix creation. Again, the SGI-PFA generated code performed worse of those compared.

## 6.2. ARC3D

ARC3D is a moderate-size CFD application. It solves Euler and Navier-Stokes equations in three dimensions using a single rectilinear grid. ARC3D has a structure similar to NPB-SP but contains curve-linear coordinates, turbulent models and more realistic boundary conditions. The Beam-Warming algorithm is used to approximately factorize an implicit scheme of finite difference equations, which is then solved in three directions alternatively.

For generating the OpenMP parallel version of ARC3D, we used a serial code that was already optimized for cache performance by hand [13]. The parallelization process with CAPO was straightforward and OpenMP directives were inserted without further user interaction. The parallel version was tested on the Origin2000 and the result for a 194x194x194-size problem is shown in the left panel of Figure 9. The results from a hand-parallelized version with SGI multi-tasking directives (*MT by hand*) [13] and a message-passing version generated by CAPTools (*CAP MPI*) [7] from the same serial version are also included in the figure for comparison.

As one can see from the figure, the OpenMP version generated by CAPO is essentially the same as the hand-coded version in performance. This is indicative of the accurate data dependence analysis and sufficient parallelism that was exploited in the outer-most loop level. The MPI version is about 10% worse than the directive-based versions. The MPI version uses extra buffers for communication and this could contribute to the increase of execution time.

## 6.3. OVERFLOW

OVERFLOW is widely used for airflow simulation in the aerospace community. It solves compressible Navier-Stokes equations with first-order implicit time scheme and exploits complicated turbulence model and Chimera boundary condition in multiple zones. The code has been parallelized by hand [4] with several approaches: PVM for zone-level parallelization only, MPI for both inter- and intra-zone parallelization, multi-tasking directives, and multi-level parallelization. This code offers a good test case for our tool not only because of its complexity but also its size (about 100K lines of FORTRAN 77).

In this study, we used the sequential version (1.8f) of OVERFLOW. CAPO took 25 hours on a Sun E10K node to complete the data dependence analysis. A fair amount of effort was spent on pruning data dependences that were placed due to lack of necessary knowledge during the analysis. An example of a false dependence is illustrated in the following code segment:

```
      NTMP2 = JD*KD*31
      DO 100 L=LS,LE
        CALL GETARX(NTMP2,TMP2,ITMP2)
        CALL WORK(L,TMP2(ITMP2,1),TMP2(ITMP2,7),...)
        CALL FREARX(NTMP2,TMP2,ITMP2)
 100  CONTINUE
```

Inside the loop, the memory space for an array TMP2 is first allocated by GETARX. The working array is then used in WORK and freed afterwards. However, the data analysis has reviewed that the loop contains loop-carried true dependences caused by variable TMP2, thus, the loop can only be executed in serial. The memory allocation and de-allocation are performed dynamically and cannot be handled by CAPO. This kind of false dependence can safely be removed with the DepGraph browser. Even so, CAPO provides an easy way for the user to interact

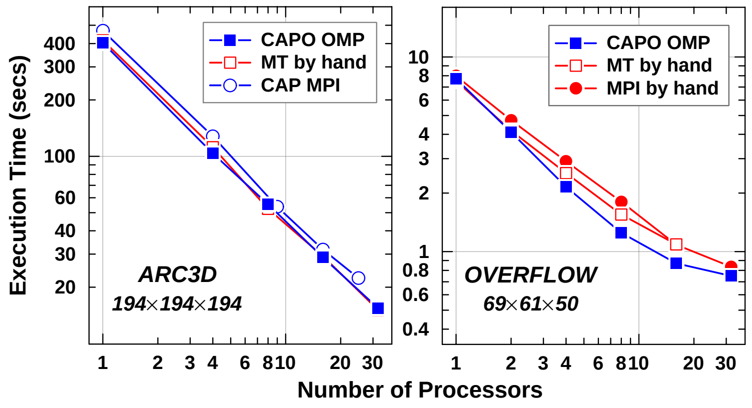


*Figure 9: Comparison of execution times of CAPO generated parallel codes with hand-coded parallel versions for two CFD applications: ARC3D on the left and OVERFLOW on the right.*

with the parallelization process. The OpenMP version was generated within a day after the analysis was completed and an additional few days were used to test the code.

The right panel of Figure 9 shows the execution time per time-iteration of the CAPO-OMP version compared with the hand-coded MPI version and hand-coded directive (MT) version. All three versions were running with a test case of size 69×61×50 (210K grid points) in single zone. Although the scaling is not quite linear (when comparing to ARC3D), especially for more than 16 processors, the CAPO version out-performed both hand-coded versions. The MPI version contains sizable extra codes [4] to handle intra-zone data distributions and communications. It is not surprising that the overhead is unavoidably large. However, the MPI version is catching up with the CAPO-OMP version on large number of processors. On the other hand, further review has indicated that the multi-tasking version used a fairly similar parallelization strategy as CAPO did, but in quite a few small routines the MT version did not place any directives for the hope that the compiler (SGI-PFA in this case) would automatically parallelize loops inside these routines. The performance number seemed to have indicated otherwise.

We also tested with a large problem of 1.5M grid points. The result was not included in the figure but CAPO's version has achieved 18-fold speedup on 32 processors of the Origin2000 (10 out of 32 for the small test case).

**Acknowledgements:** The authors wish to thank Drs. Warren Smith and Mohammad Djomehri from the NAS division at NASA Ames Research Center and Dr. Constantinos Ierotheou from the University of Greenwich for patiently reading through the manuscript and giving their valuable comments and suggestions. The work is supported by the Computational Aerospace Sciences (CAS) project under the High Performance Computing and Communications (HPCC) program.

## References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), “The NAS Parallel Benchmarks,” *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, “The NAS Parallel Benchmarks 2.0,” *NAS Technical Report RNR-95-020*, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/.

[3] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Leggett, “Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes,” *Parallel Computing*, 22 (1996) 163-195.

[4] D.C. Jespersen, “Parallelism and OVERFLOW,” *NAS Technical Report NAS-98-013*, NASA Ames Research Center, Moffett Field, CA, 1998.

[5] H. Jin, M. Frumkin and J. Yan., “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance,” *NAS Technical Report*, NAS-99-011, NASA Ames Research Center, 1999.

[6] H. Jin, M. Frumkin and J. Yan., “Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes,” in Proceedings of Third International Symposium on High Performance Computing (ISHPC2000), Tokyo, Japan, October 16-18, 2000.

[7] H. Jin, M. Hribar and J. Yan, “Parallelization of ARC3D with Computer-Aided Tools,” *NAS Technical Report*, NAS-98-005, NASA Ames Research Center, 1998.

[8] S.P. Johnson, M. Cross and M. Everett, “Exploitation of Symbolic Information In Interprocedural Dependence Analysis,” *Parallel Computing*, 22, 197-226, 1996.

[9] S.P. Johnson, P.F. Leggett, C.S. Ierotheou, E.W. Evans, and M. Cross, “Computer Aided Parallelisation Tools (CAPTools) – User Manual,” Parallel Processing Research Group, University of Greenwich, London, UK, http://captools.gre.ac.uk/.

[10] Kuck and Associates, Inc., http://www.kai.com/.

[11] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi.

[12] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.

[13] J. Taft, “Initial SGI Origin2000 Tests Show Promise for CFD Codes,” *NAS News*, July-August, page 1, 1997. (http://www.nas.nasa.gov/Pubs/NASnews/97/07/article01.html)

[14] A. Waheed and J. Yan, “Parallelization of NAS Benchmarks for Shared Memory Multiprocessors," in Proceedings of High Performance Computing and Networking (HPCN Europe '98), Amsterdam, The Netherlands, April 21-23, 1998.

# TUTORIALS

Source codes for all the tutorials described in this manual are included in the CAPO distribution and can also be obtained from site http://www.nas.nasa.gov/Tools/CAPO/. Refer to "`Examples.txt`" included in the `examples` directory for additional information.

## Contents

# Tutorial 1. A Simple Jacobi Code

This tutorial demonstrates the very basic operations you would follow to generate an OpenMP code without little user intervention. The code (jacobi.f) has an initialization loop and an iteration loop. The iteration loop computes new solutions by averaging two neighboring points and checks the maximum residual.

### *Steps of parallelization:*

1. **Perform the data dependence analysis**. In CAPO, click ***Load F77 Source*** in the **File** menu. Select jacobi.f and click Load. In the **Analyser** window, select the Full option and click Analyse. This will just take a few seconds.

2. **Save to database**. In the **File** menu, click ***Save database***. Enter a filename for the database or take the default name (jacobi_full.dbs) and click Save. It is always a good idea to save the results from different stages of the code analysis.

3. **Browse directives**. In the **View** menu, click ***Directives*** to perform the directives analysis. The Directives browser will be popped up quickly. Select the All Routines scope and browse through all loop filters. You will notice that the Jacobi code contains one *Reduction* loop (`DO 30 I=1,N`), two *Chosen* (parallel) loops (`DO 10 I=1,N` and `DO 20 I=2,N-1`), and one *Falsely Serial* loop (`DO 50 I=1,N` containing an I/O statement).

4. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (or take the default name, jacobi_omp.f) and click Save. If the directives analysis has not been performed (via Step 3), it will automatically be performed before the parallel code is generated. The log file, jacobi_omp.log, contains additional information for the parallelization process.

To compile the OpenMP code on the SGI Origin2000, do

```
% f77 –o jacobi_omp –O3 –r8 –mp jacobi_omp.f
```

To execute the parallel code with 2 threads, do

```
% setenv OMP_NUM_THREADS 2
% ./jacobi_omp
Enter the values of N and TOL ...
1000 1.0e-6
```

The output looks like

```
...
49.99968169151887
  1166848  9.9999888192314756E-07
```

You can compare the result with a single thread run or a serial version run. You will notice the program does not scale well, primarily due to little work inside each distributed loop.

# Tutorial 2. NPB LU-hp Removing False Dependences

This tutorial demonstrates the basic user interaction with CAPO: removing false dependences to improve the quality of data dependence and directives analyses. False dependences usually arise from insufficient knowledge of certain parameters (such as from `READ` statements or calculated at runtime) during CAPTools data dependence analysis. With the Directives browser, the user can inspect the results and remove these false dependences if needed.

The example is one of the benchmarks from the NAS Parallel Benchmark (NPB) suite. The benchmark, LU-hp, uses an SSOR algorithm to solve the Navier-Stokes equations in three dimensions. A hyper-plane implementation of the SSOR algorithm is used in LU-hp. The code is split into many .f files. In order to load the code to CAPO, we first create a list file “All.list” that contains names of all the .f files.

## *Steps of parallelization:*

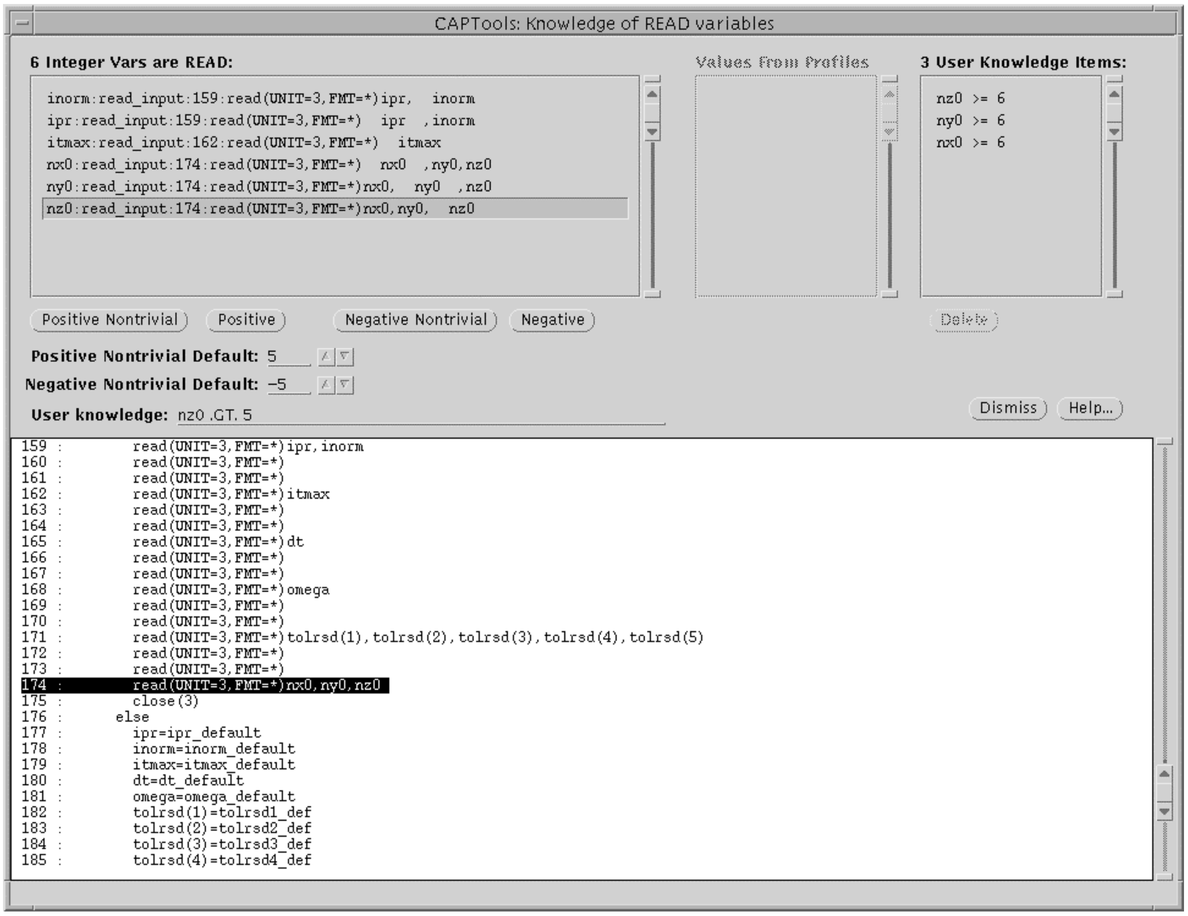
1. **Load file and enter user knowledge**. Click ***Load F77 Source*** in the **File** menu. Select All.list and click the $\boxed{\text{Load}}$ button. Select ***READ Knowledge*** from the **Edit** menu. In the **READ Knowledge** window, select variable `nx0` and click $\boxed{\text{Positive Nontrivial}}$, see Figure T2-1 on next page. Apply the same steps to variables `ny0` and `nz0`. These three variables define the number of grid points in each dimension. Making them positive nontrivial (> 5 in the current case) improves the quality of data dependence analysis.

2. **Perform the data dependence analysis**. After the user knowledge is entered, in the **Analyser** window select the $\boxed{\text{Full}}$ option and click $\boxed{\text{Analyse}}$. On an Indy R5000 workstation, the analysis process takes about 18 minutes.

3. **Save to database**. In the **File** menu, click ***Save Database***. Enter a filename for the database (lu_hp_full.dbs) and click $\boxed{\text{Save}}$.

4. **Browse directives**. In the **View** menu, click ***Directives*** to perform the directives analysis. The Directives browser will be popped up shortly. Select the $\boxed{\text{All Routines}}$ scope and browse through all loop filters. Pay attention to the serial loops (*Totally*, *Covered* and *Falsely*. For meanings of these loop types, refer to Section 3.2 in Appendix).

5. **Remove false dependences**. In the Directives browser window, select the $\boxed{\text{Totally Serial}}$ loop filter and the $\boxed{\text{All Routines}}$ scope. There are four loops listed under this category. Choose the first loop: `blts:1/1/35: do n=1,np,1` and click the $\boxed{\text{Why}}$ button. The **WhyDirectives** window as shown in Figure T2-2 will be popped up. As indicated in the window, the serialization of this loop is caused by loop-carried data dependences from two variables: `v` and `tv`. After inspecting the loop, the user realizes that this loop performs calculation for all points on a given hyper-plane (*i+j+k=constant*). Each point on one hyper-plane could be calculated independently, thus in parallel. However, indirect indexing was used to access data elements on the plane and these indices were calculated dynamically and not available at the data dependence analysis stage. Conservative decisions were made to keep these data dependences during the analysis. So, the user can safely remove these *false* dependences to enforce a parallel loop: using either the **DepGraph** window (in CAPTools) or the WhyDirectives window here (simpler). With the second method, select variable `v` and `tv` in the three lists (**True**, **Anti** and **Output**), click the $\boxed{\text{Remove}}$ button and click the $\boxed{\text{Apply}}$ button to confirm the action. Apply the same procedure to the second loop: `buts:1/1/35: do n=1,np,1`.



6 Integer Vars are READ:

```
inorm:read_input:159:read(UNIT=3,FMT=*)ipr,  inorm
ipr:read_input:159:read(UNIT=3,FMT=*)  ipr  ,inorm
itmax:read_input:162:read(UNIT=3,FMT=*)  itmax
nx0:read_input:174:read(UNIT=3,FMT=*)  nx0  ,ny0,nz0
ny0:read_input:174:read(UNIT=3,FMT=*)nx0,  ny0  ,nz0
nz0:read_input:174:read(UNIT=3,FMT=*)nx0,ny0,  nz0
```

3 User Knowledge Items:

```
nz0 >= 6
ny0 >= 6
nx0 >= 6
```

Positive Nontrivial Default: 5

Negative Nontrivial Default: -5

User knowledge: nz0 .GT. 5

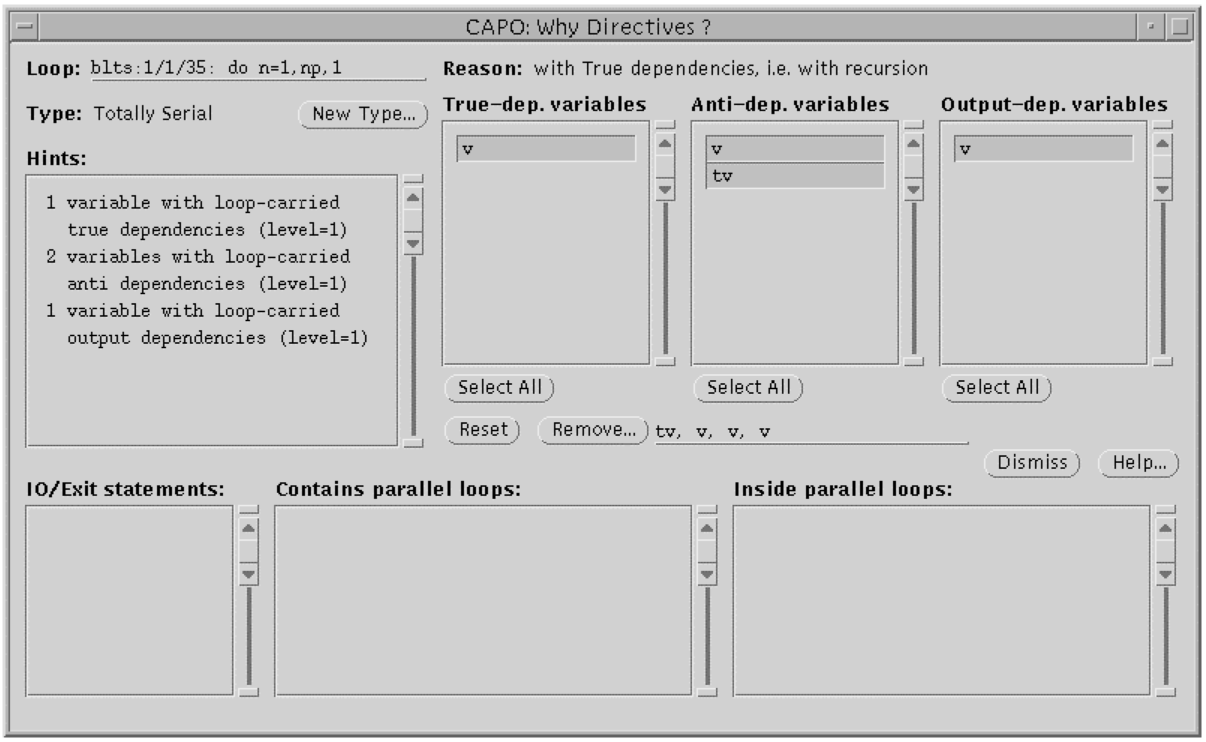
```
159 :         read(UNIT=3,FMT=*)ipr,inorm
160 :         read(UNIT=3,FMT=*)
161 :         read(UNIT=3,FMT=*)
162 :         read(UNIT=3,FMT=*)itmax
163 :         read(UNIT=3,FMT=*)
164 :         read(UNIT=3,FMT=*)
165 :         read(UNIT=3,FMT=*)dt
166 :         read(UNIT=3,FMT=*)
167 :         read(UNIT=3,FMT=*)
168 :         read(UNIT=3,FMT=*)omega
169 :         read(UNIT=3,FMT=*)
170 :         read(UNIT=3,FMT=*)
171 :         read(UNIT=3,FMT=*)tolrsd(1),tolrsd(2),tolrsd(3),tolrsd(4),tolrsd(5)
172 :         read(UNIT=3,FMT=*)
173 :         read(UNIT=3,FMT=*)
174 :         read(UNIT=3,FMT=*)nx0,ny0,nz0
175 :         close(3)
176 :       else
177 :         ipr=ipr_default
178 :         inorm=inorm_default
179 :         itmax=itmax_default
180 :         dt=dt_default
181 :         omega=omega_default
182 :         tolrsd(1)=tolrsd1_def
183 :         tolrsd(2)=tolrsd2_def
184 :         tolrsd(3)=tolrsd3_def
185 :         tolrsd(4)=tolrsd4_def
```

*Figure T2-1: The READ Knowledge window for entering initial user knowledge.*

In the Directives browser window, select loop filter Falsely Serial and sub-filter Privatization. Two loops are listed in this category. Choose the first loop: `jacld:1/1/160: do n=1,np,1` and click the Why button if the **WhyDirectives** window is not visible. A new set of variables is shown in the window, Figure T2-3. By the same token as above, the user selects those variables listed in the **Output-dep** list and applies Remove to delete the relevant loop-carried *Output* dependences. The variables in the **In/Out-dep** list were not selected because they are indeed used outside the current loop. If a variable is removed from the **In/Out-dep** list and kept in the **Output-dep** list, the variable would be *privatized*, which is not what we want here. Use the same procedure on the second loop: `jacu:1/1/160: do n=1,np,1`.

6. **Save new database and re-perform the directives analysis**. Once data dependences are modified, it is wise to save the results to a new database. In the **File** menu, click ***Save database***. Enter a filename for the database (lu_full_prune.dbs) and click Save. To re-perform the directives analysis with changes taking into account, click the Update Directives button in the **Directives** main window and Update to confirm the action. After the update, you will notice the four loops treated above are now listed in *Chosen* (parallel). CAPO automatically recognizes five reduction loops, two of them being array reductions.

7. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code***. Choose the ***Single Filename*** setting, enter a filename (lu_hp_omp.f) and click Save. The log file, lu_hp_omp.log, contains additional information and statistics for the parallelization process.



```
1 variable with loop-carried
  true dependencies (level=1)
2 variables with loop-carried
  anti dependencies (level=1)
1 variable with loop-carried
  output dependencies (level=1)
```

*Figure T2-2: The WhyDirectives window for a* Totally Serial *loop. It can be used to remove false dependences for the selected variables.*

To compile the OpenMP code on the SGI Origin2000, do

```
% f77 -o lu_hp_omp -O3 -mp lu_hp_omp.f
```

To execute the parallel code with 4 threads, do

```
% setenv OMP_NUM_THREADS 4
% ./lu_hp_omp
```

The output (for a class-W problem on 195MHz O2K) looks like:

```
Programming Baseline for NPB - LU Benchmark

Size:  33x 33x 33
Iterations: 300
Time step    1
...
     0.1161399311023E+02 0.1161399311023E+02 0.3074289103934E-13
Verification Successful

LU Benchmark Completed.
Class           =                        W
Size            =             33x 33x 33
Iterations      =                      300
Time in seconds =                    52.74
Mop/s total     =                   342.43
```

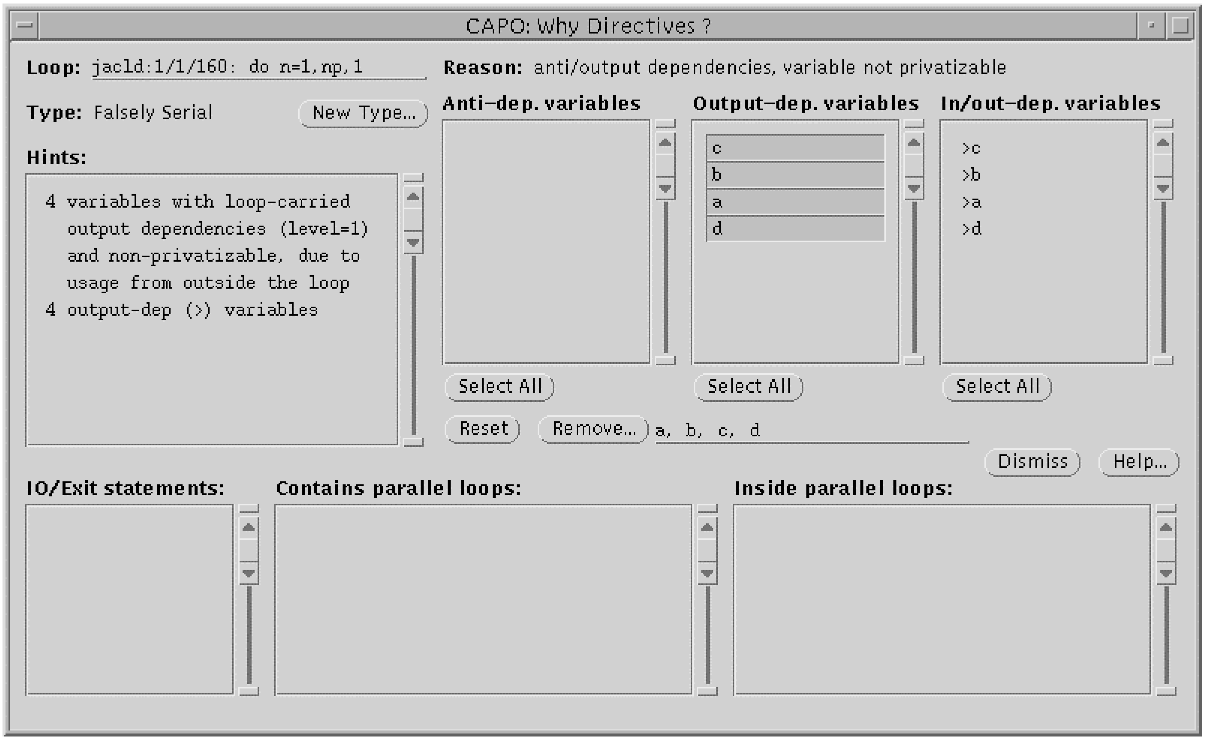


*Figure T2-3: The WhyDirectives window for a* Falsely Serial *loop. The loop-carried output dependences for variables* a,b,c,d *are selected for removal.*

The output from a single process execution looks like:

```
Programming Baseline for NPB - LU Benchmark

Size:  33x 33x 33
Iterations: 300
Time step    1
...
     0.1161399311023E+02 0.1161399311023E+02 0.3227238810597E-13
Verification Successful

LU Benchmark Completed.
Class           =                      W
Size            =              33x 33x 33
Iterations      =                    300
Time in seconds =                 155.97
Mop/s total     =                 115.80
```

We have a speedup of 2.96 on 4 CPUs for this particular problem. If the pipelined LU were used, the performance would be better (speedup of 3.32 on 4 CPUs). A version of the LU benchmark using the pipeline algorithm is included in directory **LU**. Parallelizing LU with CAPO is straightforward and similar steps as for parallelizing the hyper-plane LU can be followed. The difference is that the user does not even need to remove any false dependences when generating the OpenMP code (skip Steps 5 and 6). CAPO is able to automatically set up the parallel pipeline.

## Tutorial 3. NPB MG User-Defined Loop Type

This tutorial was included in Version 1.0 of CAPO to demonstrate how the user enforces loop type to improve the performance. This kind of interaction is not very often and can be done either within or outside CAPO. The outside interaction is often involved with direct change to the source code. In the following we first show the steps of parallelization without any change and then illustrate two ways of user manipulation to the source code.

The example is one of the benchmarks from the NAS Parallel Benchmark (NPB) suite. The benchmark, MG, uses the V-cycle multigrid algorithm to obtain an approximate solution to a discrete Poisson problem in three dimensions. The norm of the solution is calculated in each iteration to check for convergence. As was done in Tutorial 2, all the .f files are first listed in a single file: `All.list`.

### *Parallelization of the original code.*

1. **Perform the data dependence analysis**. Click ***Load F77 Source*** in the **File** menu. Select All.list and click the $\boxed{\text{Load}}$ button. In the **Analyser** window select the $\boxed{\text{Full}}$ option and click $\boxed{\text{Analyse}}$. On a 450 MHz Sun workstation, the analysis process takes about 20 minutes.

2. **Save to database**. In the **File** menu, click ***Save database***. Enter a filename for the database (mg_full.dbs) and click $\boxed{\text{Save}}$.

3. **Browse directives**. In the **View** menu, click ***Directives*** to perform the directives analysis. The Directives browser will be popped up shortly. Choose scope $\boxed{\text{All Routines}}$ and loop filter $\boxed{\text{Totally Serial}}$ and sub-filter $\boxed{\text{True Recursion}}$. Select loop: `norm2u3:1/1/27: do i3=2,n3-1` and click the $\boxed{\text{Why}}$ button. Figure T3-1 is what you will see afterwards. The loop nest (and two others inside) contains an `IF` statement which prevents the loop being recognized as a reduction loop over variable `rnmu`.[1] In order to be a valid reduction statement for OpenMP, the code needs to be modified (see Step 5). Without any change, this piece of code will be run in sequential.

4. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (mg_omp.f) and click $\boxed{\text{Save}}$. The log file, mg_omp.log, contains additional information and statistics for the parallelization process.

To compile the OpenMP code on the SGI Origin2000, do

```
% f77 -o mg_omp -O3 -mp mg_omp.f
```

To execute the parallel code with 8 threads, do

```
% setenv OMP_NUM_THREADS 8
% ./mg_omp
```

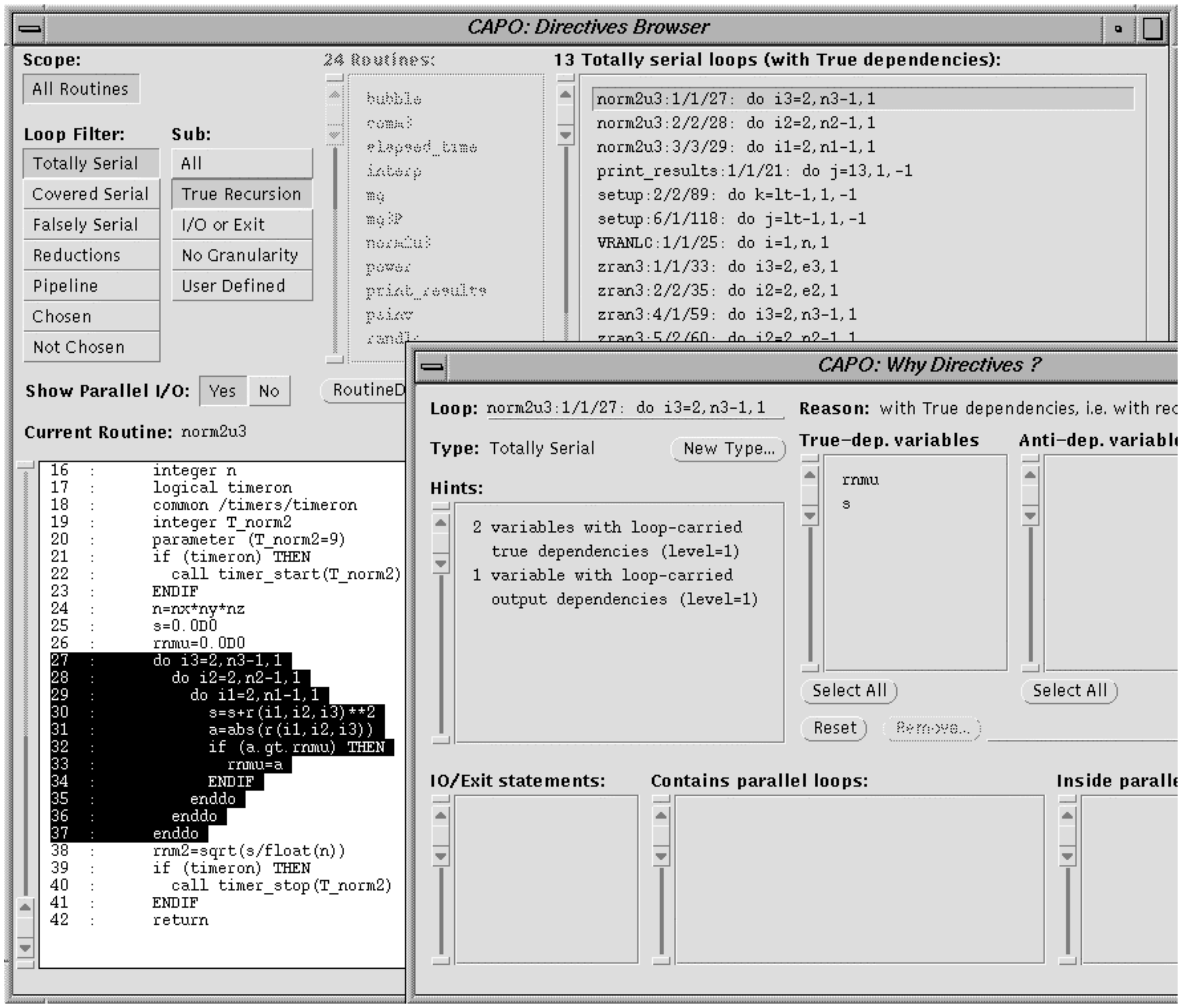
The output (for a class-A problem on 250MHz O2K) looks like:

```
Programming Baseline for NPB - MG Benchmark
...
```

[1] Due to the improvement in Version 1.1 of CAPO, the IF-type reduction is now automatically recognized. The described serial loops will no longer exist. But the concept of user interaction from this Tutorial is still valid.

```
VERIFICATION SUCCESSFUL
L2 Norm is    0.243336530907E-05
Error is      0.692805188218E-16

MG Benchmark Completed.
Class           =                  A
Size            =        256x256x256
Iterations      =                  4
Time in seconds =               6.65
Mop/s total     =             585.42
```

A single-CPU run of this code took 39.29 seconds. We have a speedup of 5.91 on 8 CPUs for this particular problem.


CAPO: Directives Browser

Scope: All Routines

Loop Filter: Totally Serial | Covered Serial | Falsely Serial | Reductions | Pipeline | Chosen | Not Chosen

Sub: All | True Recursion | I/O or Exit | No Granularity | User Defined

24 Routines:

bubble
comm3
elapsed_time
interp
mg
mg3P
norm2u3
power
print_results
psinv
rand

13 Totally serial loops (with True dependencies):

```
norm2u3:1/1/27:  do i3=2,n3-1,1
norm2u3:2/2/28:  do i2=2,n2-1,1
norm2u3:3/3/29:  do i1=2,n1-1,1
print_results:1/1/21:  do j=13,1,-1
setup:2/2/89:  do k=lt-1,1,-1
setup:6/1/118:  do j=lt-1,1,-1
VRANLC:1/1/25:  do i=1,n,1
zran3:1/1/33:  do i3=2,e3,1
zran3:2/2/35:  do i2=2,e2,1
zran3:4/1/59:  do i3=2,n3-1,1
zran3:5/2/60:  do i2=2,n2-1,1
```

Show Parallel I/O: [✓ Yes] No    RoutineD

Current Routine: norm2u3

```
      integer n
      logical timeron
      common /timers/timeron
      integer T_norm2
      parameter (T_norm2=9)
      if (timeron) THEN
        call timer_start(T_norm2)
      ENDIF
      n=nx*ny*nz
      s=0.0D0
      rnmu=0.0D0
      do i3=2,n3-1,1
        do i2=2,n2-1,1
          do i1=2,n1-1,1
            s=s+r(i1,i2,i3)**2
            a=abs(r(i1,i2,i3))
            if (a.gt.rnmu) THEN
              rnmu=a
            ENDIF
          enddo
        enddo
      enddo
      rnm2=sqrt(s/float(n))
      if (timeron) THEN
        call timer_stop(T_norm2)
      ENDIF
      return
```

CAPO: Why Directives ?

Loop: norm2u3:1/1/27:  do i3=2,n3-1,1

Reason: with True dependencies, i.e. with rec

Type: Totally Serial    New Type...

Hints:

```
2 variables with loop-carried
  true dependencies (level=1)
1 variable with loop-carried
  output dependencies (level=1)
```

True-dep. variables: rnmu, s

Anti-dep. variabl

Select All | Select All | Reset | Remove...

IO/Exit statements:

Contains parallel loops:

Inside paralle

*Figure T3-1: The window shows a serial loop in norm2u3, MG.*

Further improvement to the code can be made by parallelizing the loop in routine norm2u3 (the highlighted area in Figure T3-1). The operations inside the loop nest can be expressed as reductions with slight code modification. There are two ways to achieve the goal: modifying the serial code and re-performing the dependence analysis (Steps 5-7) or user enforcing loop type in the tool without re-analysis (Steps 8-9).

### *Modification of the serial code.*

5. **Modify the serial code**. The step involves directly modifying the serial code (mg.f) with an editor before the analysis. In routine `norm2u3`, change the IF statement

```
if (a.gt.rnmu) rnmu = a
```

to a form that can be expressed with reduction

```
rnmu = dmax1(rnmu, a)
```

Save the new version to mg2.f and create a new list file 'All2.list' to include mg2.f.

6. **Perform the data dependence analysis**. Click ***Load F77 Source*** in the **File** menu. Select All2.list and click the Load button. In the **Analyser** window select the Full option and click Analyse. Save the result to a database (mg2_full.dbs). Browse directives if you like (**View** → ***Directives***). You will notice the loop in routine `norm2u3` is now recognized as reduction.

7. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (mg2_omp.f) and click Save. The log file, mg2_omp.log, contains additional information and statistics for the parallelization process.

   Now you can compile and run the parallel code as described after Step 9.

### *User enforced loop type.*

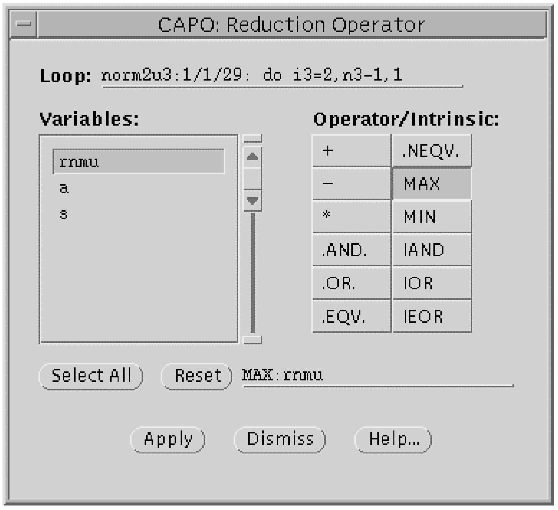
8. **Define a new loop type**. From the **File** menu, load in the database "mg_full.dbs" from the previous analysis. Perform Step 3. In the **WhyDirectives** window, click the New Type button. Right after the Reduction setting is selected the **Reduction Operator** dialog box is shown up (see Figure T3-2). Select variable "`rnmu`" and intrinsic function "`max`", and push Apply in the **Reduction Operator** dialog and in the **Loop Type** dialog. A new entry "`R[max:rnmu]`" is added to file "userloop.par" in the current working directoy. This is to inform CAPO to treat variable "`rnmu`" as a reduction variable besides other variables (such as "`s`"). Now in CAPO click Update Directives to re-perform the directives analysis, which will take into account the user-defined loop types from file "userloop.par."

9. **Save and change OpenMP code.** In the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (mg2_omp.f) and click Save. We need to do one last change in the generated OpenMP code: Use an editor, change in routine `norm2u3`

```
if (a.gt.rnmu) THEN
  rnmu=a
ENDIF
```

to an "OpenMP-compliant" form

```
rnmu = dmax1(rnmu, a)
```



Loop: norm2u3:1/1/29: do i3=2,n3-1,1

Variables: rnmu, a, s

MAX:rnmu

*Figure T3-2: The* ***Reduction Operator*** *dialog after the* *Reduction* *setting is selected.*

From either method, we should produce the same new parallel code (mg2_omp.f). Use the same process after Step 4 to compile and run the new code. The output from a run with 8 CPUs (for a class-A problem on 250MHz O2K) looks like:

```
Programming Baseline for NPB - MG Benchmark
...
VERIFICATION SUCCESSFUL
L2 Norm is   0.243336530907E-05
Error is     0.694753363997E-16

MG Benchmark Completed.
Class           =                    A
Size            =          256x256x256
Iterations      =                    4
Time in seconds =                 5.67
Mop/s total     =               686.60
```

The new code took 39.12 seconds on 1 CPU and 5.67 seconds on 8 CPUs, a speedup of 6.90 and 14% improvement over the first version.

# Tutorial 4. A CFD Application TEAMKE1

The sample code, teamke1, in this tutorial has been taken from one of the CAPTools' tutorials with a slight modification. This is a realistic application. It includes structures that may be encountered in many scientific applications. The example illustrates an incremental approach to achieve good performance with assistant from CAPO and other tools like SpeedShop (available on the Origin 2000 machine). These tools are used to pinpoint problematic code sections quickly so that the user can apply necessary changes.

## *Parallelization of the original code:* teamke1.f

1. **Perform the data dependence analysis**. Start CAPO, click ***Load F77 Source*** in the **File** menu. Select teamke1.f and click the $\boxed{\text{Load}}$ button. In the **Analyser** window select the $\boxed{\text{Full}}$ option and click $\boxed{\text{Analyse}}$. The analysis process takes only a few minutes.

2. **Save to database**. In the **File** menu, click ***Save Database***. Enter a filename for the database (teamke1_full.dbs) and click $\boxed{\text{Save}}$.

3. **Perform the directives analysis**. In the **View** menu, click ***Directives*** to perform the directives analysis. The Directives browser will be popped up shortly. Choose the $\boxed{\text{All Routines}}$ scope and browse through different loop filters. You will notice there are a quite number of *Totally Serial* loops (see Figure T4-1), which will limit the performance of this code. At this point, we only look into more details of the loop nest in routine `CALCP1`. The rest of the loops will be discussed in Step 5 and after.

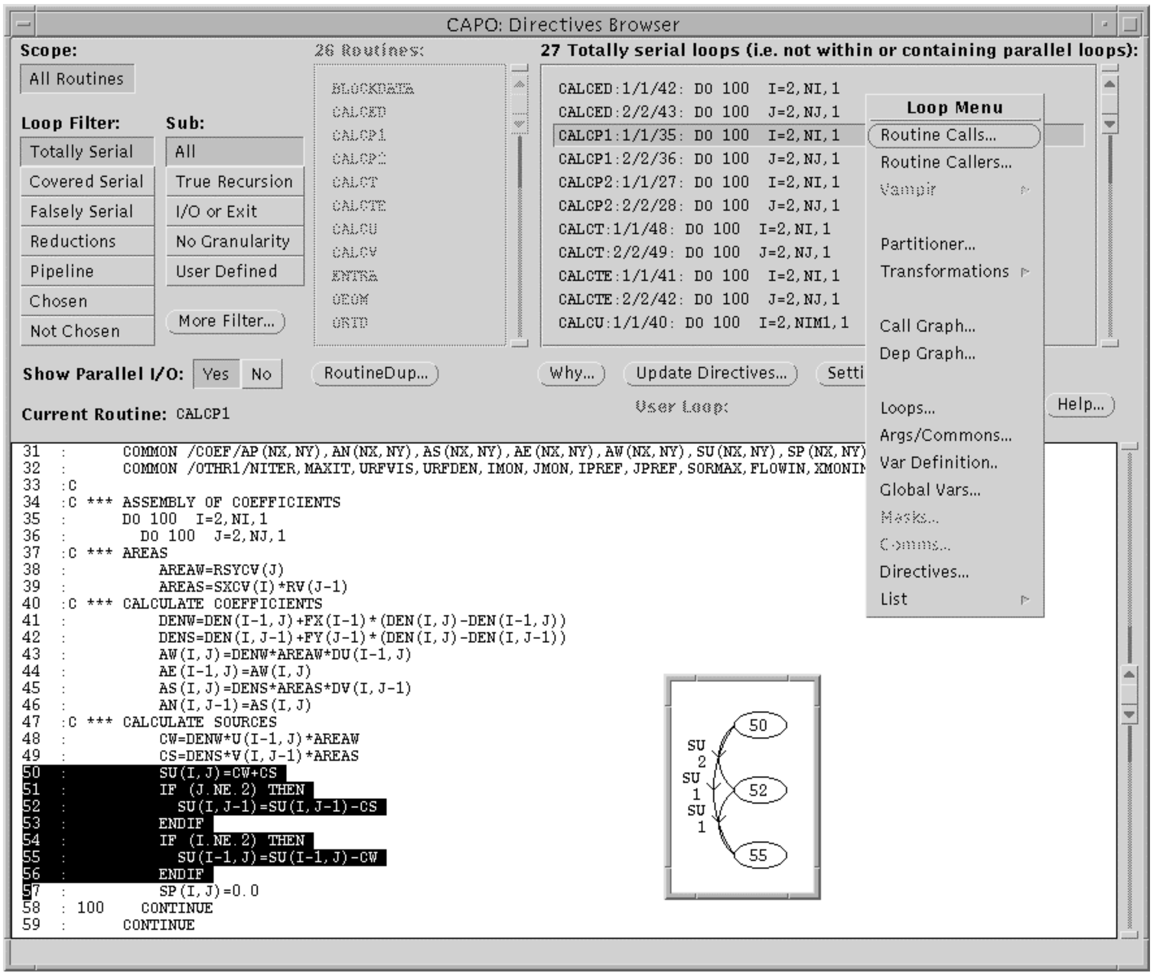
   Choose the loop "`CALCP1:1/1/35: DO 100 I=2,NI,1`" and click $\boxed{\text{Why}}$. The ***WhyDirectives*** window indicates the loop was serialized due to loop-carried dependences for variable `SU`. The ***DepGraph*** (activated from the right-mouse button **Loop Menu** over the selected loop) shows level-1 and level-2 dependences from statement 50 to 52 to 55 (see Figure T4-1). In particular the 52 → 55 dependence prevents even a pipeline being formed within the loop nests. In fact, we realize the add operation for variable `SU` in statements 52 and 55 is commutative, thus, the execution order of the two statements can be switched and the 52 → 55 dependence can be removed.

   In the DepGraph window, click the 52 → 55 dependence edge with the right-mouse button and load the "***Why Dependence?***" window (see Figure T4-2). Apply the $\boxed{\text{Remove This Dependence}}$ button and confirm the action. Save to a new database if you like. Click $\boxed{\text{Update Directives}}$ to re-perform the directives analysis and a pipeline is automatically recognized in routine `CALCP1`.

   Loop types are summarized here:

   25 *Totally Serial* loops
   10 *Reduction* loops
   1 *Pipeline* loop in routine `CALCP1`
   45 *Chosen* (parallel) loops

4. **Produce OpenMP code**. Without additional change, in the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (teamke1_omp.f) and click $\boxed{\text{Save}}$.



27 Totally serial loops (i.e. not within or containing parallel loops):

```
CALCED:1/1/42: DO 100  I=2,NI,1
CALCED:2/2/43: DO 100  J=2,NJ,1
CALCP1:1/1/35: DO 100  I=2,NI,1
CALCP1:2/2/36: DO 100  J=2,NJ,1
CALCP2:1/1/27: DO 100  I=2,NI,1
CALCP2:2/2/28: DO 100  J=2,NJ,1
CALCT:1/1/48: DO 100  I=2,NI,1
CALCT:2/2/49: DO 100  J=2,NJ,1
CALCTE:1/1/41: DO 100  I=2,NI,1
CALCTE:2/2/42: DO 100  J=2,NJ,1
CALCU:1/1/40: DO 100  I=2,NIM1,1
```

Current Routine: CALCP1

```
31  :       COMMON /COEF/AP(NX,NY),AN(NX,NY),AS(NX,NY),AE(NX,NY),AW(NX,NY),SU(NX,NY),SP(NX,NY)
32  :       COMMON /OTHR1/NITER,MAXIT,URFVIS,URFDEN,IMON,JMON,IPREF,JPREF,SORMAX,FLOWIN,XMONIN
33  :C
34  :C *** ASSEMBLY OF COEFFICIENTS
35  :       DO 100  I=2,NI,1
36  :         DO 100  J=2,NJ,1
37  :C *** AREAS
38  :           AREAW=RSYCV(J)
39  :           AREAS=SXCV(I)*RV(J-1)
40  :C *** CALCULATE COEFFICIENTS
41  :           DENW=DEN(I-1,J)+FX(I-1)*(DEN(I,J)-DEN(I-1,J))
42  :           DENS=DEN(I,J-1)+FY(J-1)*(DEN(I,J)-DEN(I,J-1))
43  :           AW(I,J)=DENW*AREAW*DU(I-1,J)
44  :           AE(I-1,J)=AW(I,J)
45  :           AS(I,J)=DENS*AREAS*DV(I,J-1)
46  :           AN(I,J-1)=AS(I,J)
47  :C *** CALCULATE SOURCES
48  :           CW=DENW*U(I-1,J)*AREAW
49  :           CS=DENS*V(I,J-1)*AREAS
50  :           SU(I,J)=CW+CS
51  :           IF (J.NE.2) THEN
52  :             SU(I,J-1)=SU(I,J-1)-CS
53  :           ENDIF
54  :           IF (I.NE.2) THEN
55  :             SU(I-1,J)=SU(I-1,J)-CW
56  :           ENDIF
57  :           SP(I,J)=0.0
58  : 100     CONTINUE
59  :       CONTINUE
```

*Figure T4-1: The Directives Browser window displaying* Totally Serial *loops in teamke1. The* **Loop Menu** *is used to activate the DepGraph (shown as inset) for the selected loop.*

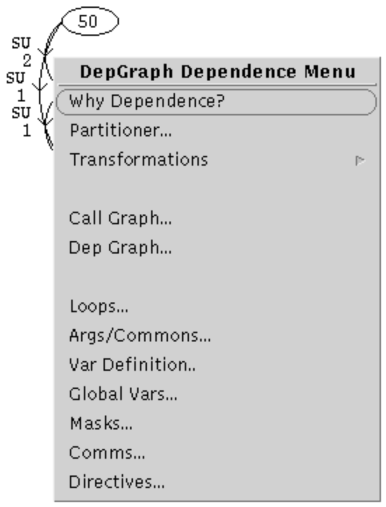


*Figure T4-2: The DepGraph Dependence Menu after clicking on a dependence edge.*

To compile the OpenMP code on the SGI Origin2000, do

```
    % f77 –o teamke1_omp –O2 –mp teamke1_omp.f
```

or use the supplied `Makefile`

```
    % make VERNO=1
```

To execute the parallel code with 4 threads, do

```
    % setenv OMP_NUM_THREADS 4
    % ./teamke1_omp < inp.dat > teamke1_omp.out.4
```

Use the `SpeedShop` tool available on the Origin 2000 to profile the code. For 1 CPU:

```
    % setenv OMP_NUM_THREADS 1
    % ssrun -pcsamp ./teamke1_omp < inp.dat > teamke1_omp.out.1
```

A sampling file named as "`teamke1_omp.pcsamp.m(pid)`" will be created. Here "(`pid`)" is a proper process id. Use the "`prof`" command to create the profile output:

```
    % prof teamke1_omp teamke1_omp.pcsamp.m(pid) > teamke1_omp.prof.1
```

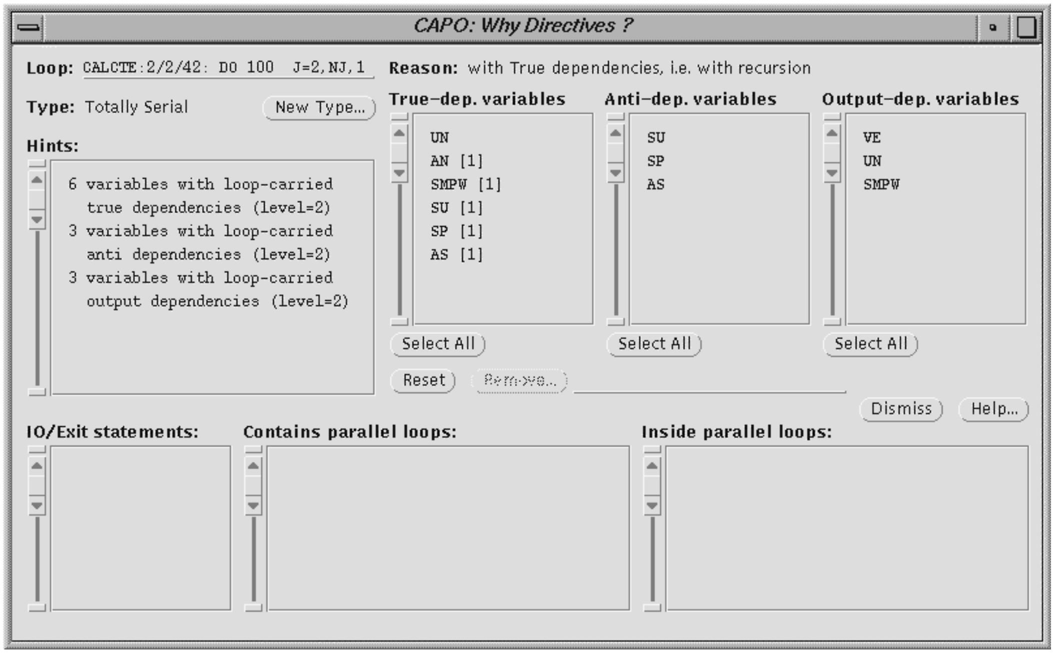
Follow the same procedure to obtain profile on 4 CPUs. The profile outputs for the key routines on 1 and 4 CPUs are compared in Table T4-1. "`ratio`" is 1-CPU time over 4-CPU time, or the speedup on 4 CPUs. The error of ratio is calculated from the statistical sampling error reported in the profile data. As we can see, except for two routines (`calcp1` and `props`), the major routines do not scale. The poor performance correlates with the ***Totally Serial*** loops indicated in Figure T4-1. These loops were executed sequentially. In order to improve the performance, we need to investigate and find a way to parallelize these loops.

*Table T4-1: Comparison of profile results for the first parallel version of teamke1. Time is given in seconds.*

| Function | 1CPU | 4CPUs | ratio | error |
|---|---|---|---|---|
| `LISOLV` | 16.18 | 16.89 | 0.958 | 0.033 |
| `CALCTE` | 9.53 | 9.06 | 1.052 | 0.049 |
| `CALCV` | 8.95 | 7.86 | 1.139 | 0.056 |
| `CALCU` | 8.58 | 7.58 | 1.132 | 0.056 |
| `CALCED` | 8.10 | 7.71 | 1.051 | 0.053 |
| `CALCT` | 7.10 | 6.47 | 1.097 | 0.060 |
| `calcp1` | 4.78 | 1.59 | 3.006 | 0.275 |
| `CALCP2` | 4.11 | 4.03 | 1.020 | 0.071 |
| `props` | 0.48 | 0.16 | 3.000 | 0.866 |
| `init` | 0.25 | 0.15 | 1.667 | 0.544 |
| `PRINT` | 0.06 | 0.20 | 0.300 | 0.140 |
| **Total** | **80.83** | **74.21** | **1.089** | **0.018** |

## *Version 2 – Code modification without change to the basic algorithm:*

5. **Inspect code sections**. Restart CAPO and load back teamke1_full.dbs (***Load Database*** in the **File** menu). In the **View** menu, click ***Directives*** to perform the directives analysis. In the **Directives browser** window, choose scope All Routines, loop filter Totally Serial and loop "`CALCTE:2/12/42: DO 100 J=2,NJ`". Click the Why button and the **WhyDirectives** window as shown in Figure T4-2 will be displayed. There are six variables with loop-carried true dependences, five of which have a determinable dependence vector length as indicated by "`[1]`". This is an indication of a potential pipeline loop if changes can be made to variable `UN` and two other variables `VE` and `SMPW` presented in the **Output-dep**. variable list.


CAPO: Why Directives ?

Loop: CALCTE:2/2/42: DO 100 J=2,NJ,1

Reason: with True dependencies, i.e. with recursion

Type: Totally Serial

Hints:

```
6 variables with loop-carried
  true dependencies (level=2)
3 variables with loop-carried
  anti dependencies (level=2)
3 variables with loop-carried
  output dependencies (level=2)
```

| True-dep. variables | Anti-dep. variables | Output-dep. variables |
|---|---|---|
| UN | SU | VE |
| AN [1] | SP | UN |
| SMPW [1] | AS | SMPW |
| SU [1] | | |
| SP [1] | | |
| AS [1] | | |

IO/Exit statements:

Contains parallel loops:

Inside parallel loops:

*Figure T4-3: The WhyDirectives window for a* Totally Serial *loop in teamke1.*

6. **Change scalar assignments**. Checking the code section in loop nests `I` and `J`, we realize that the dependences on scalar variables `UN` and `VE` were caused by the reuse of the assigned values from the previous `J` or `I` iteration in an `IF` statement. The dependences can be removed if we recalculate both variables at each `J` or `I` iteration.

   Start a text editor and load in teamke1.f. In subroutine `CALCTE` modify the assignment for `UN` from

```
    IF(J.NE.NJ)UN=0.5*(U(I,J)+U(I-1,J)+FY(J)*(U(I,J+1)+U(I-1,J+1)-
   >             U(I,J)-U(I-1,J)))
```

   to

```
    IF(J.NE.NJ)THEN
      UN=0.5*(U(I,J)+U(I-1,J)+FY(J)*(U(I,J+1)+U(I-1,J+1)-
   >     U(I,J)-U(I-1,J)))
    ELSE
      UN=0.5*(U(I,J-1)+U(I-1,J-1)+FY(J-1)*(U(I,J)+U(I-1,J)-
   >     U(I,J-1)-U(I-1,J-1)))
    ENDIF
```

   and for `VE` from

```
    IF(I.NE.NI)VE=0.5*(V(I,J)+V(I,J-1)+FX(I)*(V(I+1,J)+V(I+1,J-1)-
    >             V(I,J)-V(I,J-1)))
to
     IF(I.NE.NI)THEN
       VE=0.5*(V(I,J)+V(I,J-1)+FX(I)*(V(I+1,J)+V(I+1,J-1)-
    >     V(I,J)-V(I,J-1)))
     ELSE
       VE=0.5*(V(I-1,J)+V(I-1,J-1)+FX(I-1)*(V(I,J)+V(I,J-1)-
    >     V(I-1,J)-V(I-1,J-1)))
     ENDIF
```

Apply a similar modification to variables in three other routines. The changes are summarized:

| *Routine* | *Loop* | *Variable* | *Description* |
|---|---|---|---|
| CALCP2 | DO 100 J=2,NJ | SUS, SUW | Recalculate at each iteration |
| CALCTE | DO 100 J=2,NJ | VE, UN | |
| CALCU | DO 100 J=2,NJ | GAMN, DVDXN | |
| CALCV | DO 100 J=2,NJM1 | GAME | |

7. **Expand 1-D array to 2-D**. Variable SMPW is a 1-D working array throughout the program. In order to set up a pipeline of the J loop with the outer I loop, this array needs to be expanded to two dimensional. As an example, in routine CALCTE, change the declaration of SMPW from 1-D to 2-D, i.e. SMPW(NX) → SMPW(NX,NY). Then modify the following code section from

```
      CP=AMAX1(0.0,(SMPW(J)+CW))
      SMPW(J)=-CW-CS
      SMPW(J-1)=SMPW(J-1)+CS
```

to

```
      CP=AMAX1(0.0,(SMPW(I-1,J)+CW))
      SMPW(I,J)=-CW-CS
      SMPW(I,J-1)=SMPW(I,J-1)+CS
```

The initialization of SMPW is done in subroutine (entry) INIT. In this routine modify the declaration from SMPW(NX) to SMPW(NX,NY) and the assignment from SMPW(J)=0.0 to SMPW(I,J)=0.0.

Similar changes are made in several other places. The modifications on SMPW are summarized here:

| *Routine* | *Loop* | *Description* |
|---|---|---|
| CALCED | DO 100 J=2,NJ | Expand SMPW from 1-D to 2-D |
| CALCT | DO 100 J=2,NJ | Change declaration in the whole program |
| CALCTE | DO 100 J=2,NJ | |
| CALCU | DO 100 J=2,NJ | |
| CALCV | DO 100 J=2,NJM1 | |
| INIT | DO 951 J=1,NJ | |

All the modifications do not alter the basic algorithm, so the same run-time results should be expected. Save the modified code to a new file: teamke2.f.

8. **Perform code analysis**. Restart CAPO and load teamke2.f. Perform the **Full** data dependence analysis and save to teamke2_full.dbs. Start the Directives browser from the **View** menu and the ***Directives*** menu item. With the $\boxed{\text{All Routines}}$ scope browse through different loop filters. You will notice that the number of *Totally Serial* loops has been reduced from 25 to 13 with increase in the number of pipeline loops. Loop types are summarized here:

| | |
|---|---|
| 13 | *Totally Serial* loops (mainly in routine LISOLV) |
| 10 | *Reduction* loops |
| 7 | *Pipeline* loops |
| 45 | *Chosen* (parallel) loops |

9. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code*** and save to file teamke2_omp.f.

Compile and run the parallel code as before. The SpeedShop profile results for the new parallel code are summarized in Table T4-2. As one can see, the parallel performance of Version 2 has been improved in almost all routines except in routine `LISOLV`. `LISOLV` still executes serially and affects overall performance. The single CPU execution time increased slightly in comparison with the original version. This is because the recalculation of scalar variables in the new code costs slightly more time.

*Table T4-2: Comparison of profile results for the second parallel version. Time is given in seconds.*

| **Function** | **1CPU** | **4CPUs** | **ratio** | **error** |
|---|---|---|---|---|
| `LISOLV` | 16.14 | 18.00 | 0. 897 | 0.031 |
| `calcte` | 9.89 | 3.19 | 3.100 | 0.200 |
| `calcv` | 9.28 | 2.92 | 3.178 | 0.213 |
| `calcu` | 8.82 | 2.83 | 3.117 | 0.213 |
| `calced` | 8.76 | 2.87 | 3.052 | 0.208 |
| `calct` | 7.79 | 2.39 | 3.259 | 0.241 |
| `calcp1` | 5.04 | 1.75 | 2.880 | 0.253 |
| `calcp2` | 4.06 | 1.11 | 3.658 | 0.392 |
| `props` | 0.53 | 0.20 | 2.650 | 0.695 |
| `init` | 0.28 | 0.13 | 2.154 | 0.723 |
| `PRINT` | 0.14 | 0.26 | 0.538 | 0.178 |
| **Total** | **83.77** | **46.67** | **1.795** | **0.033** |

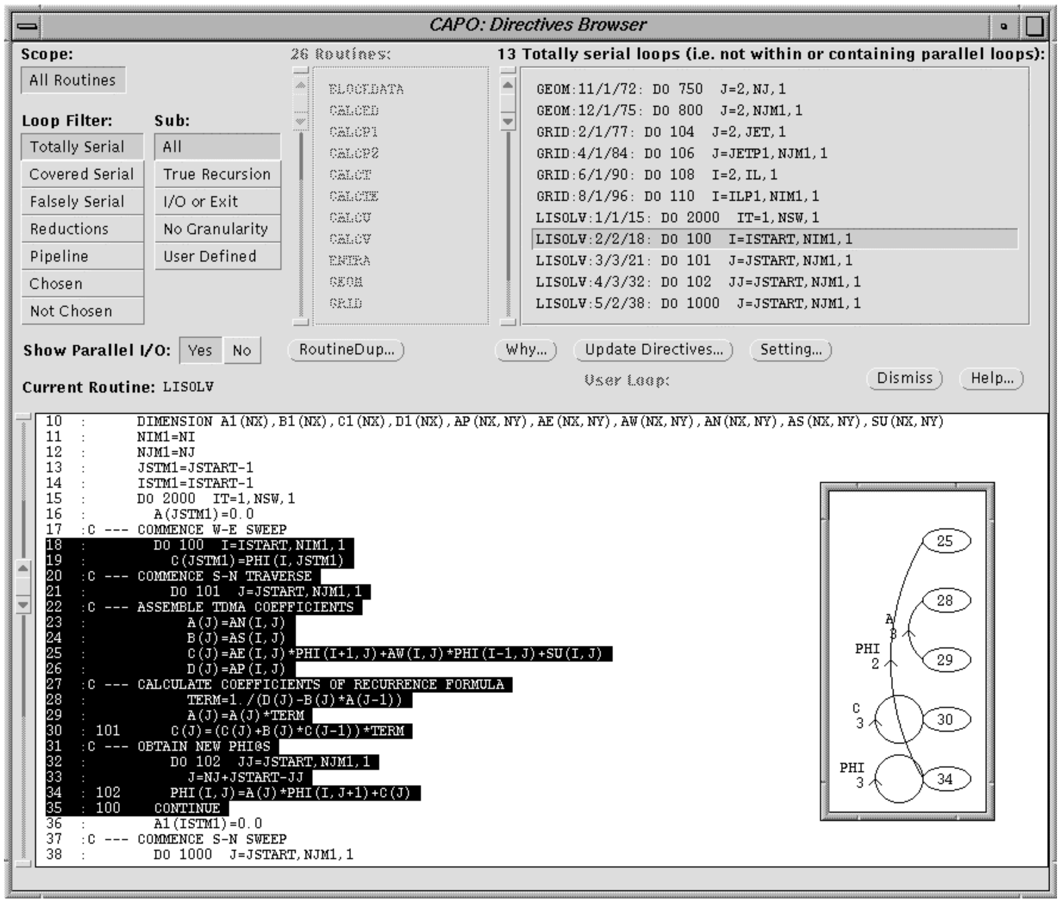
### *Version 3 – Change of algorithm in LISOLV:*

10. **Inspect code sections**. Restart CAPO and load back teamke2_full.dbs (***Load Database*** in the **File** menu). In the **View** menu, click ***Directives*** to perform the directives analysis. In the **Directives browser** window, choose scope All Routines, loop filter Totally Serial and loop "`LISOLV:2/2/18: DO 100 I=ISTART,NIM1`". Click the right mouse button to activate the **Loop Menu**. In the menu choose ***Dep Graph*** and the **DepGraph** window will show data dependences that serialize the loop (see Figure T4-4 and the inset): variable `PHI` at level 2 (loop `I`) and 3 (loop `J`) and variable `A`,`C` at level 3 (loop `J`). In loop `I`, variable `PHI` is used to calculate `A` and `C` and gets updated at each `I` iteration.

11. **Modify the algorithm**. We can use a more explicit algorithm in the `I` loop: Variables `A` and `C` are calculated for all the values of `I` before variable `PHI` is updated. The `I` loop then becomes parallel. The impact of such a change is mainly on the convergence speed of the underline algorithm. One may have to balance convergence rate and parallelization. In this case parallelization seems to be more important since it improves overall code performance.

    The modifications to the code involve expanding the dimensionality of `A` and `C` from 1-D to 2-D and splitting the `I` loop into two parts: the first part calculates `A` and `C` from `PHI` and the second part updates PHI. The modified code section is shown in Figure T4-4. Apply the same change to loop "DO 1000 J=JSTART, NJM1".

Save the final code to teamke3.f



13 Totally serial loops (i.e. not within or containing parallel loops):

```
GEOM:11/1/72:  DO 750   J=2,NJ,1
GEOM:12/1/75:  DO 800   J=2,NJM1,1
GRID:2/1/77:  DO 104   J=2,JET,1
GRID:4/1/84:  DO 106   J=JETP1,NJM1,1
GRID:6/1/90:  DO 108   I=2,IL,1
GRID:8/1/96:  DO 110   I=ILP1,NIM1,1
LISOLV:1/1/15:  DO 2000   IT=1,NSW,1
LISOLV:2/2/18:  DO 100   I=ISTART,NIM1,1
LISOLV:3/3/21:  DO 101   J=JSTART,NJM1,1
LISOLV:4/3/32:  DO 102   JJ=JSTART,NJM1,1
LISOLV:5/2/38:  DO 1000   J=JSTART,NJM1,1
```

```
 :       DIMENSION A1(NX),B1(NX),C1(NX),D1(NX),AP(NX,NY),AE(NX,NY),AW(NX,NY),AN(NX,NY),AS(NX,NY),SU(NX,NY)
 :       NIM1=NI
 :       NJM1=NJ
 :       JSTM1=JSTART-1
 :       ISTM1=ISTART-1
 :       DO 2000  IT=1,NSW,1
 :         A(JSTM1)=0.0
 :C ---  COMMENCE W-E SWEEP
 :         DO 100  I=ISTART,NIM1,1
 :           C(JSTM1)=PHI(I,JSTM1)
 :C ---  COMMENCE S-N TRAVERSE
 :           DO 101  J=JSTART,NJM1,1
 :C ---  ASSEMBLE TDMA COEFFICIENTS
 :             A(J)=AN(I,J)
 :             B(J)=AS(I,J)
 :             C(J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
 :             D(J)=AP(I,J)
 :C ---  CALCULATE COEFFICIENTS OF RECURRENCE FORMULA
 :             TERM=1./(D(J)-B(J)*A(J-1))
 :             A(J)=A(J)*TERM
 : 101       C(J)=(C(J)+B(J)*C(J-1))*TERM
 :C ---  OBTAIN NEW PHI@S
 :           DO 102  JJ=JSTART,NJM1,1
 :             J=NJ+JSTART-JJ
 : 102       PHI(I,J)=A(J)*PHI(I,J+1)+C(J)
 : 100     CONTINUE
 :         A1(ISTM1)=0.0
 :C ---  COMMENCE S-N SWEEP
 :         DO 1000  J=JSTART,NJM1,1
```

*Figure T4-4: The Directive Browser window for* Totally Serial *loops in teamke2. The highlighted code section in routine* LISOLV *is to be modified to a more explicit form.*

```
 :       DIMENSION PHI(NX,NY),A(NX,NY),B(NX),C(NX,NY),D(NX)
 :       DIMENSION A1(NX),B1(NX),C1(NX),D1(NX),AP(NX,NY),AE(NX,NY),A
 :       NIM1=NI
 :       NJM1=NJ
 :       JSTM1=JSTART-1
 :       ISTM1=ISTART-1
 :       DO 2000  IT=1,NSW,1
 :C ---  COMMENCE W-E SWEEP
 :         DO 101  I=ISTART,NIM1,1
 :           A(I,JSTM1)=0.0
 :           C(I,JSTM1)=PHI(I,JSTM1)
 :C ---  COMMENCE S-N TRAVERSE
 :           DO 101  J=JSTART,NJM1,1
 :C ---  ASSEMBLE TDMA COEFFICIENTS
 :             A(I,J)=AN(I,J)
 :             B(J)=AS(I,J)
 :             C(I,J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
 :             D(J)=AP(I,J)
 :C ---  CALCULATE COEFFICIENTS OF RECURRENCE FORMULA
 :             TERM=1./(D(J)-B(J)*A(I,J-1))
 :             A(I,J)=A(I,J)*TERM
 : 101       C(I,J)=(C(I,J)+B(J)*C(I,J-1))*TERM
 :         CONTINUE
 :C ---  OBTAIN NEW PHI@S
 :         DO 100  I=ISTART,NIM1,1
 :           DO 102  JJ=JSTART,NJM1,1
 :             J=NJ+JSTART-JJ
 : 102       PHI(I,J)=A(I,J)*PHI(I,J+1)+C(I,J)
 : 100     CONTINUE
```

*Figure T4-5: The modified code section after loop* I *is split into two parts.*

12. **Perform code analysis**. Restart CAPO and load teamke3.f. Perform the **Full** data dependence analysis and save to teamke3_full.dbs. Start the Directives browser from the **View** menu and the ***Directives*** menu item. With the All Routines scope browse through different loop filters. You will notice that the number of *Totally Serial* loops has been reduced from 13 to 6 and these loops are in routines `GEOM` and `GRID`. Loop types are summarized here:

| | |
|---|---|
| 6 | *Totally Serial* loops |
| 10 | *Reduction* loops |
| 7 | *Pipeline* loops |
| 49 | *Chosen* (parallel) loops |

13. **Produce OpenMP code**. In the **File** menu, click ***Save OpenMP Directives Code*** and save to file teamke3_omp.f.

Compile and run the parallel code as before. The SpeedShop profile results for the final parallel code are summarized in Table T4-3. As one can see, the parallel performance of Version 3 has been improved over Version 2 and a reasonable speedup has been obtained. The single CPU execution time of routine `LISOLV` increased about 40% in comparison with the previous version but the parallel execution time decreased by a factor of 2.4 for 4 CPUs.

*Table T4-3: Comparison of profile results for the third parallel version. Time is given in seconds.*

| Function | 1CPU | 4CPUs | ratio | error |
|---|---|---|---|---|
| `lisolv` | 22.71 | 7.47 | 3.040 | 0.128 |
| `calcte` | 9.74 | 2.95 | 3.302 | 0.219 |
| `calcv` | 9.11 | 2.78 | 3.277 | 0.225 |
| `calced` | 8.89 | 2.55 | 3.486 | 0.248 |
| `calcu` | 8.74 | 2.64 | 3.311 | 0.232 |
| `calct` | 7.83 | 2.34 | 3.346 | 0.249 |
| `calcp1` | 4.87 | 1.80 | 2.706 | 0.236 |
| `calcp2` | 4.01 | 1.07 | 3.748 | 0.408 |
| `props` | 0.52 | 0.24 | 2.167 | 0.535 |
| `init` | 0.27 | 0.12 | 2.250 | 0.781 |
| `PRINT` | 0.05 | 0.37 | 0.135 | 0.064 |
| **Total** | **89.92** | **36.23** | **2.482** | **0.049** |

## Tutorial 5. Mix of Message-Passing and OpenMP

This tutorial demonstrates one way to generate a *hybrid* parallel code with CAPTools/CAPO. The parallelization is done at two levels: message-passing (MP) at one level and OpenMP at another. The example relies on the thread-safe feature introduced in MPI-2 and the success of execution depends on the implementation of a thread-safe MPI-2 library. We need to emphasize that the hybrid parallelization here is not the best way to achieve good performance for the currently selected code. We mainly like to illustrate that it is possible to produce a hybrid parallel code with the tools.

The example is one of the benchmarks from the NAS Parallel Benchmark (NPB) suite. The benchmark, BT, uses an implicit scheme to solve the Navier-Stokes equations in three dimensions. Within one time iteration the solver sweeps through each dimension successively. Each step has strong data dependences in the swept direction, but is completely parallel in the other two directions. The multi-level parallelization is achieved by first distributing the data in the `J` dimension for message passing and then applying directives on loops working on the `K` dimension. Small modification to the generated parallel code by hand is needed in order to work around an incompletion due to that the hybrid code generation is not really supported by the current tools.

The sequential version of the source code is in directory `BT-mix`. In order to load the code to CAPO, we list all the .f files in one file: `All.list`.

### *Parallelization with message-passing at the first level:*

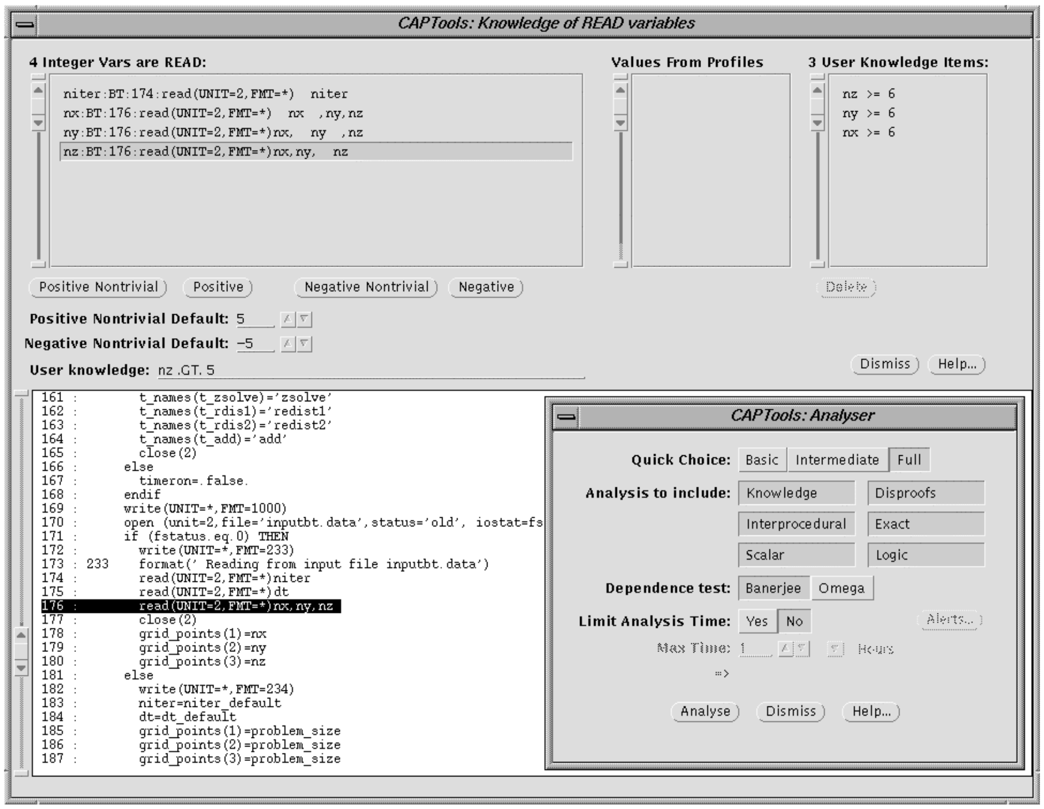
1. **Load source and enter user knowledge**. Click ***Load F77 Source*** in the **File** menu. Select All.list and click the Load button. Select ***READ Knowledge*** from the **Edit** menu. In the **READ Knowledge** window, select variable `nx` and click Positive Nontrivial, see Figure T5-1 on next page. Apply the same steps to variables `ny` and `nz`. These three variables define the number of grid points in each dimension. Making them positive nontrivial improves the quality of data dependence analysis in Step 2.

2. **Perform the data dependence analysis**. After the user knowledge is entered, in the **Analyser** window select the Full option and click Analyse. On a Sun Ultra-4 workstation, the analysis process took 12 minutes.

3. **Save to database**. In the **File** menu, click ***Save Database***. Enter a filename for the database (bt_full.dbs) and click Save.

4. **Partition data**. Launch the **Partitioner** from the CAPTools main window. Choose routine "`add`", array "`u`" and index "`3`" (see Figure T5-2) and click Generate Partition. This step creates a data distribution for array "`u`" on the 3rd index (the `J` dimension) and CAPTools also partitions automatically the relevant arrays throughout the program. Figure T5-3 shows the partitioning window after the process is finished. You will notice that array "`lhsb`" was left untouched. The next thing to do is to select this array, index 4 and perform another partitioning.

5. **Save to database**. Use the ***Save Database*** menu to save the partitioned data to bt_part_j.dbs.



4 Integer Vars are READ:

```
niter:BT:174:read(UNIT=2,FMT=*)  niter
nx:BT:176:read(UNIT=2,FMT=*)  nx  ,ny,nz
ny:BT:176:read(UNIT=2,FMT=*)nx,  ny  ,nz
nz:BT:176:read(UNIT=2,FMT=*)nx,ny,  nz
```

Values From Profiles

3 User Knowledge Items:

```
nz >= 6
ny >= 6
nx >= 6
```

Positive Nontrivial Default: 5

Negative Nontrivial Default: -5

User knowledge: nz .GT. 5

```
161 :          t_names(t_zsolve)='zsolve'
162 :          t_names(t_rdis1)='redist1'
163 :          t_names(t_rdis2)='redist2'
164 :          t_names(t_add)='add'
165 :          close(2)
166 :        else
167 :          timeron=.false.
168 :        endif
169 :        write(UNIT=*,FMT=1000)
170 :        open (unit=2,file='inputbt.data',status='old', iostat=fs
171 :        if (fstatus.eq.0) THEN
172 :          write(UNIT=*,FMT=233)
173 : 233      format(' Reading from input file inputbt.data')
174 :          read(UNIT=2,FMT=*)niter
175 :          read(UNIT=2,FMT=*)dt
176 :          read(UNIT=2,FMT=*)nx,ny,nz
177 :          close(2)
178 :          grid_points(1)=nx
179 :          grid_points(2)=ny
180 :          grid_points(3)=nz
181 :        else
182 :          write(UNIT=*,FMT=234)
183 :          niter=niter_default
184 :          dt=dt_default
185 :          grid_points(1)=problem_size
186 :          grid_points(2)=problem_size
187 :          grid_points(3)=problem_size
```

Quick Choice: Basic | Intermediate | [✓ Full]

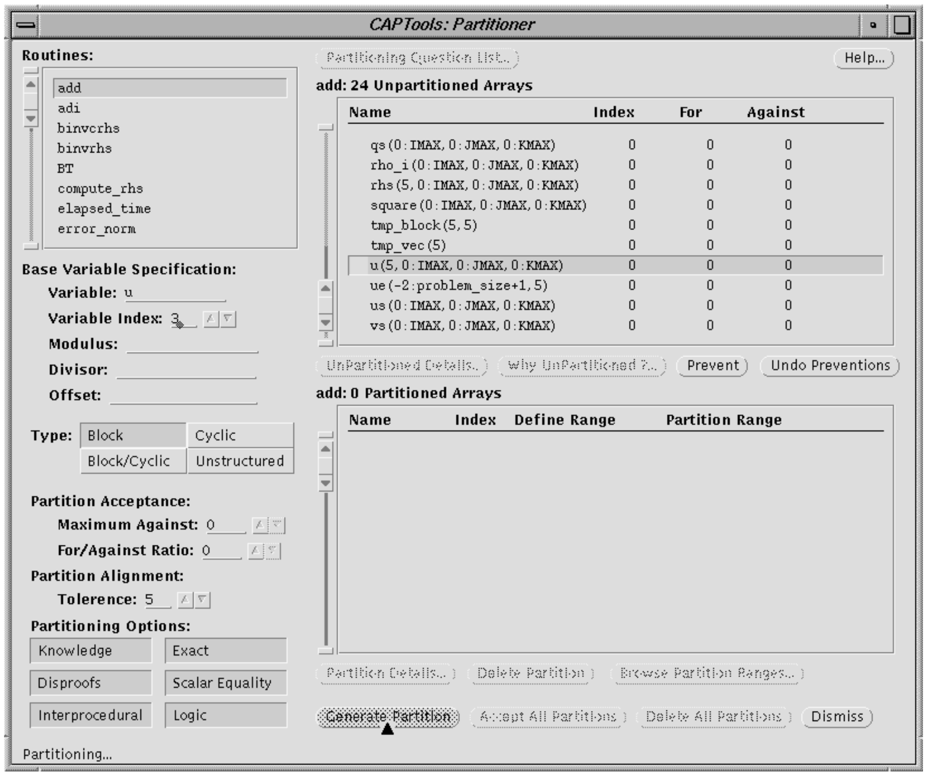
Analysis to include: Knowledge | Disproofs | Interprocedural | Exact | Scalar | Logic

Dependence test: Banerjee | Omega

Limit Analysis Time: Yes | No

Max Time: 1 Hours

Figure T5-1: The READ Knowledge window for entering user knowledge and the Analyser window.



Routines:

```
add
adi
binvcrhs
binvrhs
BT
compute_rhs
elapsed_time
error_norm
```

Base Variable Specification:

Variable: u

Variable Index: 3

Modulus:

Divisor:

Offset:

Type: Block | Cyclic | Block/Cyclic | Unstructured

Partition Acceptance:

Maximum Against: 0

For/Against Ratio: 0

Partition Alignment:

Tolerence: 5

Partitioning Options: Knowledge | Exact | Disproofs | Scalar Equality | Interprocedural | Logic

Partitioning...

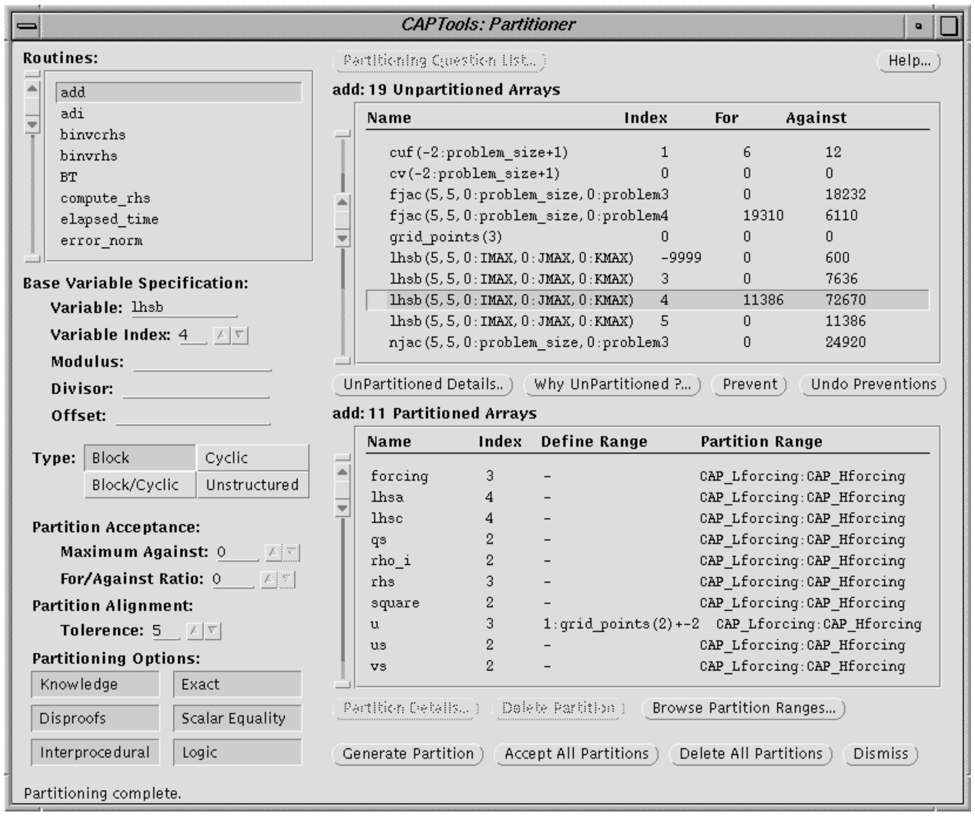
add: 24 Unpartitioned Arrays

| Name | Index | For | Against |
|---|---|---|---|
| qs(0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| rho_i(0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| rhs(5,0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| square(0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| tmp_block(5,5) | 0 | 0 | 0 |
| tmp_vec(5) | 0 | 0 | 0 |
| u(5,0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| ue(-2:problem_size+1,5) | 0 | 0 | 0 |
| us(0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |
| vs(0:IMAX,0:JMAX,0:KMAX) | 0 | 0 | 0 |

add: 0 Partitioned Arrays

| Name | Index | Define Range | Partition Range |
|---|---|---|---|

Figure T5-2: The Partitioner window for array partitioning: routine add, array u, index 3.



CAPTools: Partitioner

Routines:

- add
- adi
- binvcrhs
- binvrhs
- BT
- compute_rhs
- elapsed_time
- error_norm

Base Variable Specification:

Variable: lhsb

Variable Index: 4

Modulus:

Divisor:

Offset:

Type: Block | Cyclic | Block/Cyclic | Unstructured

Partition Acceptance:

Maximum Against: 0

For/Against Ratio: 0

Partition Alignment:

Tolerence: 5

Partitioning Options: Knowledge | Exact | Disproofs | Scalar Equality | Interprocedural | Logic

Partitioning complete.

add: 19 Unpartitioned Arrays

| Name | Index | For | Against |
|---|---|---|---|
| cuf(-2:problem_size+1) | 1 | 6 | 12 |
| cv(-2:problem_size+1) | 0 | 0 | 0 |
| fjac(5,5,0:problem_size,0:problem | 3 | 0 | 18232 |
| fjac(5,5,0:problem_size,0:problem | 4 | 19310 | 6110 |
| grid_points(3) | 0 | 0 | 0 |
| lhsb(5,5,0:IMAX,0:JMAX,0:KMAX) | -9999 | 0 | 600 |
| lhsb(5,5,0:IMAX,0:JMAX,0:KMAX) | 3 | 0 | 7636 |
| lhsb(5,5,0:IMAX,0:JMAX,0:KMAX) | 4 | 11386 | 72670 |
| lhsb(5,5,0:IMAX,0:JMAX,0:KMAX) | 5 | 0 | 11386 |
| njac(5,5,0:problem_size,0:problem | 3 | 0 | 24920 |

add: 11 Partitioned Arrays

| Name | Index | Define Range | Partition Range |
|---|---|---|---|
| forcing | 3 | - | CAP_Lforcing:CAP_Hforcing |
| lhsa | 4 | - | CAP_Lforcing:CAP_Hforcing |
| lhsc | 4 | - | CAP_Lforcing:CAP_Hforcing |
| qs | 2 | - | CAP_Lforcing:CAP_Hforcing |
| rho_i | 2 | - | CAP_Lforcing:CAP_Hforcing |
| rhs | 3 | - | CAP_Lforcing:CAP_Hforcing |
| square | 2 | - | CAP_Lforcing:CAP_Hforcing |
| u | 3 | 1:grid_points(2)+-2 | CAP_Lforcing:CAP_Hforcing |
| us | 2 | - | CAP_Lforcing:CAP_Hforcing |
| vs | 2 | - | CAP_Lforcing:CAP_Hforcing |

*Figure T5-3: Apply array partitioning on the second array:* `lhsb`*, index 4.*

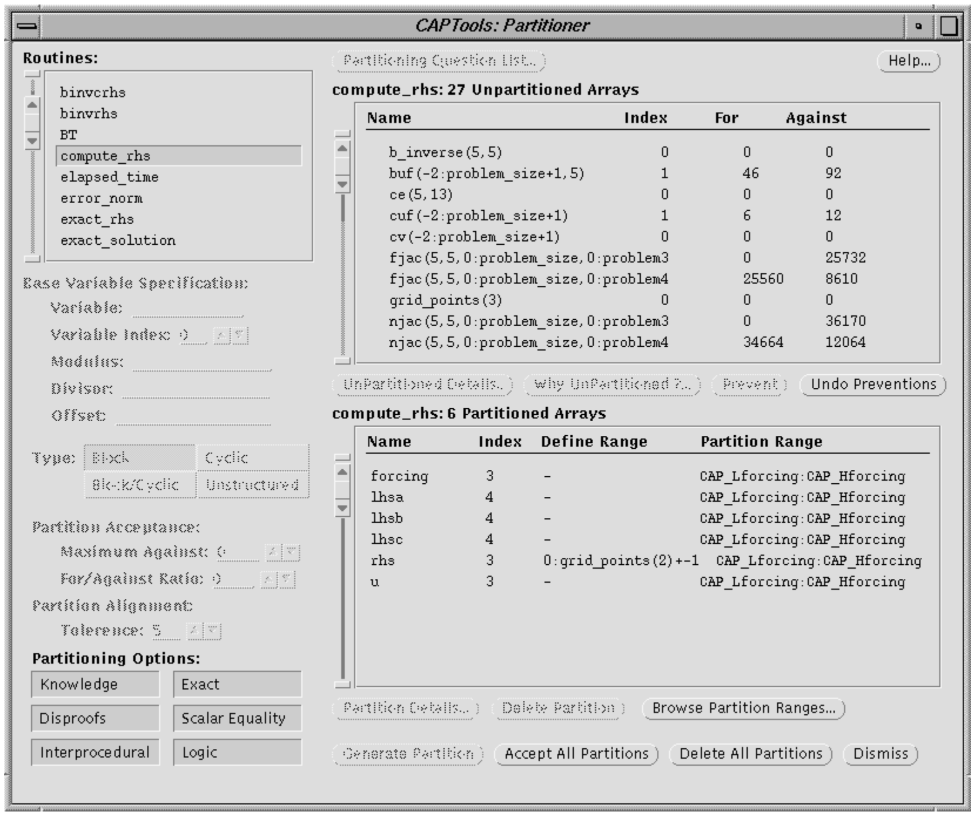
6. **Remove unwanted partitions**. If you use the result produced from Step 4 to generate message-passing code, you would notice that CAPTools place quite a few communication calls inside routine `COMPUTE_RHS`, which exchange boundary values of some of the working arrays (such as `qs`, `rho_i`...) for the partitioned dimension. These boundary values, in fact, can be calculated in the routine instead of being communicated from neighbors to improve the performance. This kind of improvement can be achieved within CAPTools by removing partitions on the relevant arrays (although it is not very obvious and intuitive). In the **Partitioner** window, select routine "`compute_rhs`". Select "`qs`" in the **Partitioned Array** list and click the Delete Partition button. Apply the same procedure to arrays: `rho_i`, `square`, `us`, `vs`, and `ws`. Figure T5-4 is what you will see after this process from which partitions on six arrays have been removed.

   Click the Accept All Partitions button.

7. **Generate masks and communications**. Start the **Code Generator** from the CAPTools main window. Choose 2 for **Min Slabs Per Processor**, which indicates at least 2 slabs in the partitioned direction to be used for the execution and reduces number of communications calls placed. Select Gather / Scatter for **Communication Type**. Click Generate Masks to start the mask generation and Calc & Gen Comms to generate communications. See Figure T5-5.

   At this point you could produce a pure message-passing program if you wish (the Generate & Save Final Code button). But we move onto next step.

8. **Save to database**. Use the ***Save Database*** menu to save the communication data to bt_comm_j.dbs.



CAPTools: Partitioner

Routines:

binvcrhs
binvrhs
BT
compute_rhs
elapsed_time
error_norm
exact_rhs
exact_solution

Base Variable Specification:
Variable:
Variable Index: 0
Modulus:
Divisor:
Offset:

Type: Block | Cyclic | Block/Cyclic | Unstructured

Partition Acceptance:
Maximum Against: 0
For/Against Ratio: 0
Partition Alignment:
Tolerence: 5

Partitioning Options:
Knowledge | Exact
Disproofs | Scalar Equality
Interprocedural | Logic

Partitioning Question List... | Help...

compute_rhs: 27 Unpartitioned Arrays

| Name | Index | For | Against |
|---|---|---|---|
| b_inverse(5,5) | 0 | 0 | 0 |
| buf(-2:problem_size+1,5) | 1 | 46 | 92 |
| ce(5,13) | 0 | 0 | 0 |
| cuf(-2:problem_size+1) | 1 | 6 | 12 |
| cv(-2:problem_size+1) | 0 | 0 | 0 |
| fjac(5,5,0:problem_size,0:problem3 | 0 | 0 | 25732 |
| fjac(5,5,0:problem_size,0:problem4 | 1 | 25560 | 8610 |
| grid_points(3) | 0 | 0 | 0 |
| njac(5,5,0:problem_size,0:problem3 | 0 | 0 | 36170 |
| njac(5,5,0:problem_size,0:problem4 | 1 | 34664 | 12064 |

UnPartitioned Details... | Why UnPartitioned ?... | Prevent | Undo Preventions

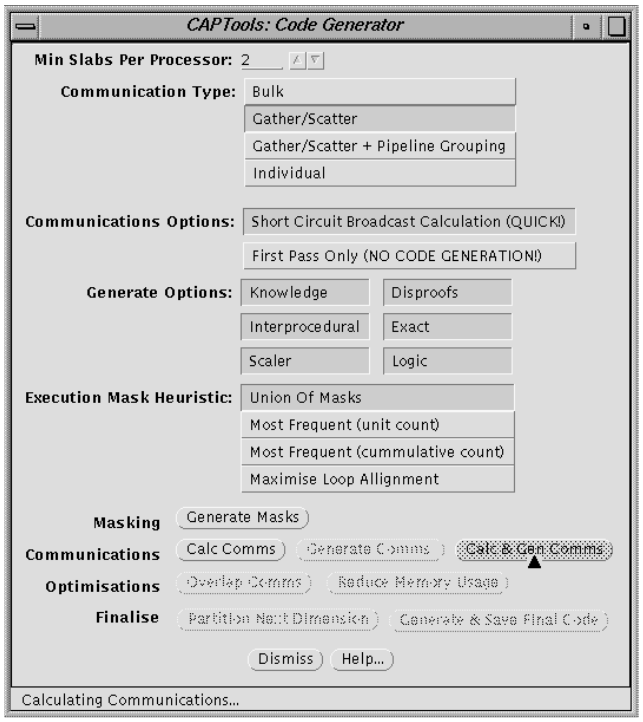
compute_rhs: 6 Partitioned Arrays

| Name | Index | Define Range | Partition Range |
|---|---|---|---|
| forcing | 3 | - | CAP_Lforcing:CAP_Hforcing |
| lhsa | 4 | - | CAP_Lforcing:CAP_Hforcing |
| lhsb | 4 | - | CAP_Lforcing:CAP_Hforcing |
| lhsc | 4 | - | CAP_Lforcing:CAP_Hforcing |
| rhs | 3 | 0:grid_points(2)+-1 | CAP_Lforcing:CAP_Hforcing |
| u | 3 | - | CAP_Lforcing:CAP_Hforcing |

Partition Details... | Delete Partition | Browse Partition Ranges...

Generate Partition | Accept All Partitions | Delete All Partitions | Dismiss

*Figure T5-4: The Partitioner window after partitions on six arrays were deleted.*



CAPTools: Code Generator

Min Slabs Per Processor: 2

Communication Type: Bulk | Gather/Scatter | Gather/Scatter + Pipeline Grouping | Individual

Communications Options: Short Circuit Broadcast Calculation (QUICK!) | First Pass Only (NO CODE GENERATION!)

Generate Options: Knowledge | Disproofs | Interprocedural | Exact | Scaler | Logic

Execution Mask Heuristic: Union Of Masks | Most Frequent (unit count) | Most Frequent (cummulative count) | Maximise Loop Allignment

Masking: Generate Masks
Communications: Calc Comms | Generate Comms | Calc & Gen Comms
Optimisations: Overlap Comms | Reduce Memory Usage
Finalise: Partition Next Dimension | Generate & Save Final Code

Dismiss | Help...

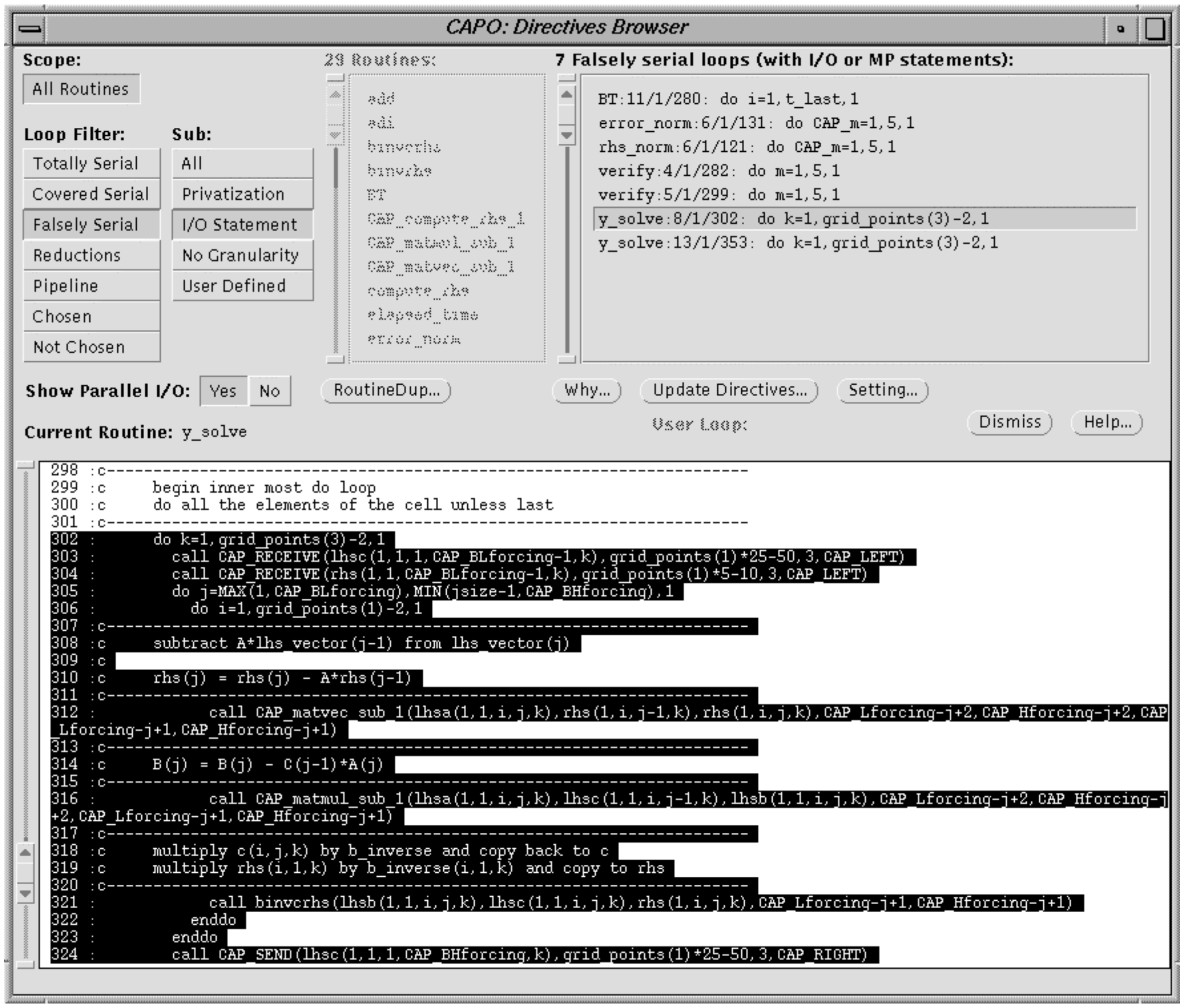
Calculating Communications...

*Figure T5-5: The Code Generator window for the final generation of message-passing code.*

### *Insertion of OpenMP directives at the second level:*

9. **Browse directives**. In the **View** menu, click ***Directives*** to perform the directives analysis. The Directives browser will be popped up shortly. Select the All Routines scope and browse through all loop filters. Pay attention to the serial loops (*Totally*, *Covered* and *Falsely*).

10. **Re-enforce new loop types**. In the Directives browser window, select the All Routines scope, the Falsely Serial loop filter and I/O Statement sub filter (Figure T5-6). There are two K loops listed under this category. Choose the first loop: `y_solve:8/1/302: do k=1,grid..` and click the Why button. The **WhyDirectives** window (see Figure T5-7) indicates that there are four MP (Message-Passing) calls (as part of the parallel pipelines) inside the K loop, which serialize the K loop. If nothing is done here, the inside I loop will be chosen for the second level parallelization with directives, which will not give a good performance.



7 Falsely serial loops (with I/O or MP statements):

```
BT:11/1/280: do i=1,t_last,1
error_norm:6/1/131: do CAP_m=1,5,1
rhs_norm:6/1/121: do CAP_m=1,5,1
verify:4/1/282: do m=1,5,1
verify:5/1/299: do m=1,5,1
y_solve:8/1/302: do k=1,grid_points(3)-2,1
y_solve:13/1/353: do k=1,grid_points(3)-2,1
```

Current Routine: y_solve

```
298 :c-------------------------------------------------------------------
299 :c     begin inner most do loop
300 :c     do all the elements of the cell unless last
301 :c-------------------------------------------------------------------
302 :       do k=1,grid_points(3)-2,1
303 :         call CAP_RECEIVE(lhsc(1,1,1,CAP_BLforcing-1,k),grid_points(1)*25-50,3,CAP_LEFT)
304 :         call CAP_RECEIVE(rhs(1,1,CAP_BLforcing-1,k),grid_points(1)*5-10,3,CAP_LEFT)
305 :         do j=MAX(1,CAP_BLforcing),MIN(jsize-1,CAP_BHforcing),1
306 :           do i=1,grid_points(1)-2,1
307 :c-------------------------------------------------------------------
308 :c     subtract A*lhs_vector(j-1) from lhs_vector(j)
309 :c
310 :c     rhs(j) = rhs(j) - A*rhs(j-1)
311 :c-------------------------------------------------------------------
312 :             call CAP_matvec_sub_1(lhsa(1,1,i,j,k),rhs(1,i,j-1,k),rhs(1,i,j,k),CAP_Lforcing-j+2,CAP_Hforcing-j+2,CAP
_Lforcing-j+1,CAP_Hforcing-j+1)
313 :c-------------------------------------------------------------------
314 :c     B(j) = B(j) - C(j-1)*A(j)
315 :c-------------------------------------------------------------------
316 :             call CAP_matmul_sub_1(lhsa(1,1,i,j,k),lhsc(1,1,i,j-1,k),lhsb(1,1,i,j,k),CAP_Lforcing-j+2,CAP_Hforcing-j
+2,CAP_Lforcing-j+1,CAP_Hforcing-j+1)
317 :c-------------------------------------------------------------------
318 :c     multiply c(i,j,k) by b_inverse and copy back to c
319 :c     multiply rhs(i,1,k) by b_inverse(i,1,k) and copy to rhs
320 :c-------------------------------------------------------------------
321 :             call binvcrhs(lhsb(1,1,i,j,k),lhsc(1,1,i,j,k),rhs(1,i,j,k),CAP_Lforcing-j+1,CAP_Hforcing-j+1)
322 :           enddo
323 :         enddo
324 :         call CAP_SEND(lhsc(1,1,1,CAP_BHforcing,k),grid_points(1)*25-50,3,CAP_RIGHT)
```

*Figure T5-6: The Directives Browser window for the* Falsely Serial *and* I/O Statement *type.*

In order to improve the performance, we can enforce a parallel type for the two K loops with an assumption that the MP calls are thread-safe. This is possible within the context of MPI-2. To define a new loop type, click the New Type button in the **WhyDirectives** window (Figure T5-7). Select new type Parallel and push Apply. A new entry is now added to file `userloop.par`.

Select the second K loop: `y_solve:13/1/353: do k=1,grid..` and click the New Type button. Again in the **LoopType** window choose new type Parallel and push Update. CAPO will save the new entry to file `userloop.par` and re-perform the directives analysis with the new loop types.

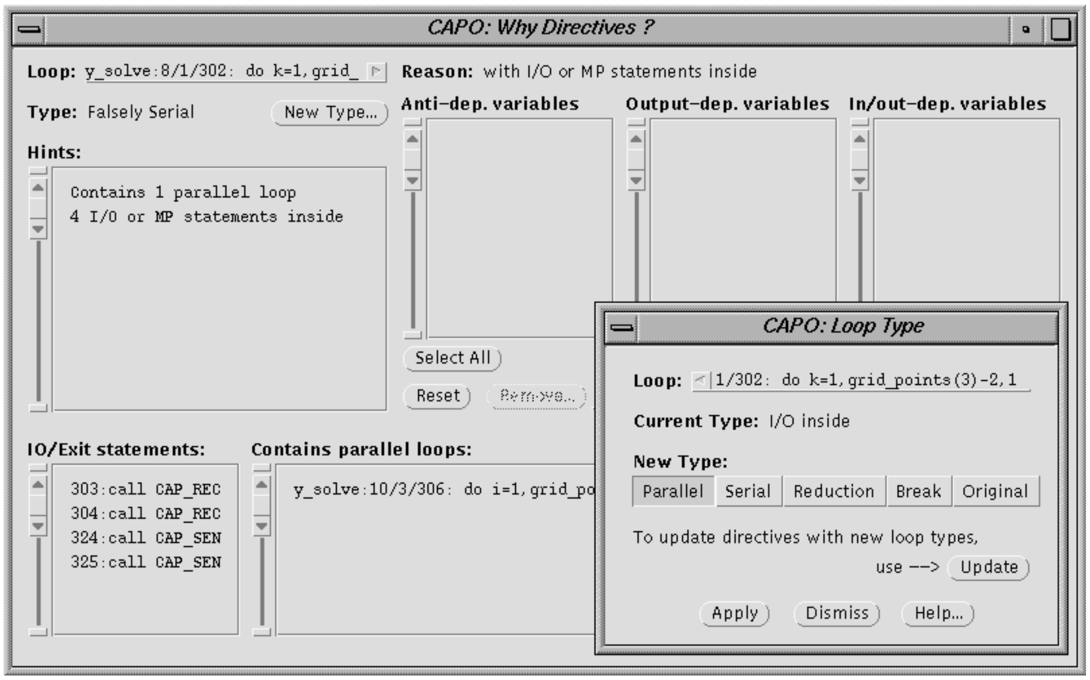


*Figure T5-7: The WhyDirectives window for the selected loop and the LoopType window for defining a new loop type.*

**11. Insert OpenMP directives**. In the **File** menu, click ***Save OpenMP Directives Code***. Enter a filename (bt_cap_j_omp.f) and click Save. By now you will have the first version of a hybrid BT code. The log file, bt_cap_j_omp.log, contains additional information and statistics for the parallelization process. You will see warnings on "*I/O or MP statements inside parallel region*". This is what we need to fix next.

### *Modification to the generated hybrid code:*

**12. Replace MP calls with thread-safe version**. As mentioned before, the current tool does not really support the generation of hybrid codes, but is merely used to assist such a process. The message-passing (MP) calls (`CAP_SEND`, `CAP_RECEIVE`...) placed inside the generated code by the tool are assumed to be used in a single-threaded environment. The supporting library, `CAPLIB`, is designed to run under a single-threaded environment as well. So in order to have the hybrid code working properly, we need to modify the message-passing calls inside parallel regions so that they can work safely under a multi-threaded environment. To achieve the goal, we will create a subset of the routines in `CAPLIB` to support multi-threading. These routines contain an additional field "*TAG*" in the argument for use with a specific thread. A sample implementation of the *thread-safe* MP routines used in this tutorial is included in file `caplib_thread.F`.

So we want to make a final touch to the generated code: replace several message-passing calls with the thread-safe version. Edit file bt_cap_j_omp.f with a text editor:

1) In subroutine Y_SOLVE, include the following two lines in the declaration

```
      integer omp_get_thread_num, myid
      external omp_get_thread_num
```

2) In subroutine Y_SOLVE, the third parallel region, change

```
   !$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(i,j,k)
```

to

```
   !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j,k,myid)
```

and add the following lines before "`do k=1,grid_points(3)-2,1`"

```
         myid = omp_get_thread_num()
   !$OMP DO
```

Now add a message tag to the four MP statements in the K loop by replacing

```
         CALL CAP_RECEIVE(...)
```

with

```
         CALL CAP_RECEIVE_TAG(...,2000+myid)
```

and

```
         CALL CAP_SEND(...)
```

with

```
         CALL CAP_SEND_TAG(...,2000+myid)
```

The tagged SEND and RECEIVE calls are from `caplib_thread.F` and the tag "2000+myid" is added to ensure the point-to-point communication between two threads with the same thread number. The offset "2000" in the tag is to avoid potential conflict with message tags internally used by CAPLIB, but the choice of the value is a bit of arbitrary.

Lastly, change

```
   !$OMP END PARALLEL DO
```

to

```
   !$OMP END DO NOWAIT
   !$OMP END PARALLEL
```

3) Apply the same changes as in 2) to the fifth parallel region in subroutine Y_SOLVE and save the modification.

## *Compile and run the hybrid code.*

In order to compile and run the hybrid code successfully, the following additions or installations are required:

1) The CAPLIB library from the CAPTools distribution. CAPLIB can be downloaded from http://captools.gre.ac.uk/.

2) A thread-safe extension to some of the routines in CAPLIB, which are supplied here in `caplib_thread.F` for MPI. One of the main things in the file is a dummy MPI_INIT() routine which just passes the call to MPI_INIT_THREAD(). The CAP_*_TAG routines are also in this file.

3) A thread-safe implementation of MPI-2 library that supports MPI_INIT_THREAD in level MPI_THREAD_MULTIPLE. Such an implementation is available from SGI's MIPSpro 7.3 compilers and MPT 1.4 toolkit.

We will use the supplied Makefile to compile the hybrid code on the SGI Origin2000. Modify the content of Makefile, in particular the value for CAPLIB. Then do

```
% make
```

which will create an executable "bt_cap_j_omp.1". To execute the parallel code with 3 MPI processes and 3 threads per MPI process, do

```
% setenv OMP_NUM_THREADS 3
% mpirun -np 3 ./bt_cap_j_omp.1 -top pipe3
```

The output (for a class-W problem on 195MHz O2K) looks like:

```
Thread support on Rank    0 =  3, number of threads =  3
Thread support on Rank    1 =  3, number of threads =  3
Thread support on Rank    2 =  3, number of threads =  3
PID      HOSTNAME     MPI_PROCNAME   UNIX_PID     BIN_NAME
  1      turing          turing      35973        bt_cap_j_omp.1
  2      turing          turing      35974        bt_cap_j_omp.1
  3      turing          turing      35979        bt_cap_j_omp.1

Programming Baseline for NPB - BT Benchmark

 Size:  24x 24x 24
 Iterations: 200    dt:   0.000800
 Time step    1
 ...
      5 0.1018045837718E+02 0.1018045837718E+02 0.4575047075825E-12
 Verification Successful

 BT Benchmark Completed.
 Class           =                        W
 Size            =                24x 24x 24
 Iterations      =                      200
 Time in seconds =                    11.66
 Mop/s total     =                   662.12
```

The execution time from a single process run is 84.69 seconds, so we have a speedup of 7.3 on 9 CPUs. You can run the code with different combinations of MPI processes and OpenMP threads, for example, to run with 2 MPI processes and 8 threads per MPI (2x8 = 16 CPUs):

```
% setenv OMP_NUM_THREADS 8
% mpirun -np 2 ./bt_cap_j_omp.1 -top pipe2
```

Table T5-1 on next page contains a collection of results from runs on two SGI Origin2000s: 195 (CPU type 195 MHz, 32Kb L1 and 4Mb L2 cache) and 300 (CPU type 300 MHz, 32Kb L1 and 8Mb L2 cache). **NP** stands for number of MPI processes and **NT** is the number of threads per MPI process. For a given number of CPUs, the hybrid code has a better performance when **NP** is close to **NT**. However, you also notice that "8x2" performs better than "4x4" or to say MPI is more preferable in this case.

*Table T5-1: Execution time (in seconds) and Mop/s (million floating point operations per second) of the hybrid BT code, obtained for the Class W (24x24x24) and with 1, 9 or 16 CPUs.*

| *195 MHz Origin2000, 1 or 9 CPUs* | | | | | |
|---|---|---|---|---|---|
| **NPxNT** | 1x9 | 3x3 | 9x1 | 1x1 | |
| **Time** | 14.26 | 11.66 | 12.26 | 84.69 | |
| **Mop/s** | 541.46 | 662.12 | 629.47 | 91.14 | |
| *300 MHz Origin2000, 16 CPUs* | | | | | |
| **NPxNT** | 1x16 | 2x8 | 4x4 | 8x2 | 16x1 |
| **Time** | 8.21 | 6.38 | 5.76 | 5.38 | 6.88 |
| **Mop/s** | 940.61 | 1210.05 | 1339.76 | 1433.53 | 1122.38 |

# APPENDIX

## Contents

# A1. Parameters for CAPO

The following describes parameters available in Version 1.1.

## A1.1. General

Parameters are referring to inputs that user can supply to control the behavior of directive generation in CAPO. There are default settings for all the parameters (see Section A1.3). Parameters can be defined from a file, environment variables, or the Setting box in the Directives Browser. Values from the parameter file or environment variables supersede any defaults. Values from the parameter file supersede environment variables. Changes from the Setting box (Section A3.6) in the Directives Browser are applied at last. Parameter setting can also be done from the CAPO command interface. See Section A4 for details.

## A1.2. The Parameter File

The parameter filename can be defined via the environment variable CAPO_PAR. The default filename is "`capo-inp.par`" in the current directory. An example of this file is given in Section A1.5.

Format of the parameter file:

| | |
|---|---|
| `'#'` | the sign starts a comment |
| `'key value'` | the pair defines an entry |

## A1.3. Parameter Keys and Possible Values

| *ENV_VARIABLE* | *KEY* | *DEFAULT* | *POSSIBLE VALUES* |
|---|---|---|---|
| CAPO_PAR | | capo-inp.par | |
| CAPO_LOG | log-file | on | (off on stdout) |
| CAPO_LOGNAME | log-file-name | codeoutput.log | |
| CAPO_LOGINFO | log-info | std | (min std more debug) |
| CAPO_PLOOP | loop-granularity | 6 | (0 1 2 ...) |
| CAPO_TYPE | directive-type | omp | (omp sgi sgix no) |
| CAPO_REGION | region-type | default | (loop bloop one join full) |
| CAPO_OPTIMIZE | optimize-type | o2 | (off on o2 o3) |
| CAPO_USERLOOP | user-loop-file | user-loop.par | |
| CAPO_DIRCLEAR | directive-clear | default-list | (off on filename) |
| CAPO_TPRIV | tpriv-directive | on | (off on) |
| CAPO_COMMENT | comment-type | f90 | (f77 f90) |
| CAPO_USEPARTI | use-parti-loop | no | (no yes) |
| CAPO_RDUPTYPE | rdup-type | region | (loop region) |
| CAPO_UNKSIZE | allow-unksize | false | (false true) |
| CAPO_PIO | allow-pio | no | (no incall write noread any) |

Description of the parameters:

- "`log-file`" type is one of

  | | |
  |---|---|
  | `off` | — Logging to file is off, only minimum messages are printed on screen |
  | `on` | — Information are logged to the log-file |
  | `stdout` | — Information are printed to stdout (screen) |

- "`log-file-name`" defines the name for a log file. If no name is defined, CAPO will use the output filename from the code generation to form a log filename. Contents of the log file are described in Section A2.

- "`log-info`" type is one of

  | | |
  |---|---|
  | `min` | — Only minimum information are logged or printed |
  | `std` | — Print standard set of log information |
  | `more` | — Print more detailed log information, including region and loop numbers in the final Fortran file |
  | `debug` | — Print debugging information, probably more than you want, including region and loop numbers in the final Fortran file |

- The loop granularity is based only on the loop iterations at this point. Future extension to include profile information can easily be added.

- Currently supported directive types are

  | | |
  |---|---|
  | `omp` | — Produce OpenMP directives (default) |
  | `sgi` | — Produce SGI native directives |
  | `sgix` | — Produce OpenMP directives with SGI extensions. Currently, only the '`NEST`' directive is supported |
  | `no` | — Do not insert directives in code generation (useful for comparison). |

- Different region types

  | | |
  |---|---|
  | `loop` | — consider only one loop for one region (no pipeline) |
  | `bloop` | — consider one block + one loop for one region (no pipeline) |
  | `one` | — consider one region (region not joined, no pipeline) |
  | `join` | — consider joined region (outer loop nesting, no pipeline) |
  | `full` | — consider full region (region joined and possible pipeline) |

  For SGI directives, only "loop" is allowed for the region type (region-type). The default region-type is "loop" for SGI and "full" for OMP.

- Optimization type is intended for possible improvements to be applied, such as loop granularity check, synchronization overhead reduction, and loop transformation. Currently an attempt to reduce synchronization at end-of-loop is implemented. Other optimizations are less defined and / or tested.

  | | |
  |---|---|
  | `off` | — Do not do any optimization |
  | `on` | — Try to reduce synchronization at end-of-loop |
  | `o2` | — Use logical disprove (slow sometime) for affinity comparison |
  | `o3` | — Perform additional optimization (such as loop transformation) before loop analysis and directive insertion. |

- User-defined loop types are read from a file that can be defined via environment variable `CAPO_USERLOOP` or "`user-loop-file`" entry in the parameter file. If a "`userloop.par`" file exists in the current working directory, this file will be taken if the other two methods are not used. The format of this file is:

```
# starts comment
#RoutineName LoopNumber NewType
routine_name loop_count S|P|R|B[options]
```

Entries are specified line-by-line. "Routine_name" is case insensitive. For a program without the main-routine name defined, "MAIN" can be used to indicate the main routine.

"loop_count" is the loop number counted from the beginning of a given routine. A negative "loop_count" indicates the loop (defined by -loop_count) will not be considered for automatic loop transformation.

Currently the following new loop types are supported:

| | |
|---|---|
| "S" | for serial |
| "P" | for parallel |
| "R" | for reduction |
| "B" | for break-type (e.g. so that a parallel region won't be formed around this loop). |

The "R" type can optionally be attached with

"[OPR:VAR]" or "[OPR:VAR()]" list

to indicate the reduction operator and the reduction variable, no space in-between. The second form indicates an array reduction.

- List of directives to be cleared can be read from a file or taken from the default list. The default list contains the following:

```
"cdir$",                /* Cray vector directive */
"cmic$",                /* Cray autotasking directive */
"c$par",                /* PCF (Parallel Computing Forum) directive */
"c$doacross", "c$&",    /* SGI multiprocessing directive */
"c$ ", "c$\t",
"c$omp",                /* OMP directive */
"c$sgi"                 /* SGI OMP extension */
```

The default setting is to use the above list. The 'clearing' action may be turned off by setting CAPO_DIRCLEAR to 'off'. Additional directives may be added to the default list by prefixing a '+' in front of the filename for CAPO_DIRCLEAR.

A dirclear-list file contains simply a list of directives (keywords) to be considered. A keyword should lead with one of ['C', '!', '*']. A '-' sign can be added to the front of a keyword to indicate the corresponding directive should not be cleared (i.e. keep its original form), otherwise, the directive will be commented out (cleared).

- The THREADPRIVATE directive will be generated by default. If the option is turned off via CAPO_TPRIV (=off), CAPO will use an alternative method to treat private variables used in a common block.

off — Use an alternative method to handle private variables
on — Try to produce THREADPRIVATE directives

- The comment type refers to the leading character to be used for directives. The 'C' character is for the f77 type and the '!' character is for the f90 type. Default is '!'.

- By default, if a loop is partitioned in a message-passing program, the loop will not be considered for directives (CAPO_USEPARTI=no). This is equivalent to a two-level parallelization. If a partitioned loop is intended for directives as well, CAPO_USEPARTI can be set to 'yes'. This would be a one-level parallelization with mixed type. The option is only meaningful when CAPTools is first used to generate message-passing program and CAPO is then applied to insert directives.

- Two types of routine duplication (RDUP) can be selected:

  | | |
  |---|---|
  | `loop` | — as the type for RDUP if a routine is used both inside and outside parallel loop(s). |
  | `region` | — as the (default) type for RDUP if a routine is used inside a parallel loop and inside parallel region but outside parallel loop. |

  The first option removes any nesting of parallel regions. The second option allows nested parallel regions in such a form that a parallel region can be nested inside a parallel loop but not inside a non-worksharing section of a parallel region.

- The environment variable `CAPO_UNKSIZE` controls how unknown-size private variable (USPV) is treated. A unknown-size variable has its last dimension declared as "*" or "1" in a subroutine and is in the routine argument list. By default, if an USPV is encountered, CAPO will take effort to adjust the size of the unknown dimension. If the size cannot be adjusted, the corresponding loop will be made serial. If `CAPO_UNKSIZE` is set to "`true`", the loop with USPV will not be made serial, instead, a warning will be printed so that the user can make manual change later on.

- By default I/O statements are not allowed in the dynamic extent of parallel loops. However, one can exploit certain degrees of parallel I/O with `CAPO_PIO`.

  | | |
  |---|---|
  | `no` | — no I/O statements in the dynamic extent of a loop (default). |
  | `incall` | — no I/O in the current scope of a loop, but allowed inside subroutine calls. |
  | `write` | — allow "`WRITE(*,*)`", i.e. write to the standard output. |
  | `noread` | — no `READ`, but allow any `WRITE`. |
  | `any` | — allow any type of I/O statements. |

## A1.4. Parameters for Debugging Purpose

The following parameters are only available from the Setting box (Section A3.6) in the Directives browser. By default, all these parameters are enabled. The Setting box can be used to disable them for debugging purpose.

| | |
|---|---|
| `Generate-NOWAIT` | — enable/disable the `NOWAIT` directive |
| `Transform-Induction-Loop` | — enable/disable induction loop treatment |
| `Handle-Array-Reduction` | — enable/disable array reduction |
| `Remove-Old-Directives` | — enable/disable removing old directives |
| `Apply-UserLoop-Type` | — enable/disable applying userloop types |
| `Setup-Pipeline-Loop` | — enable/disable pipeline loop |

## A1.5. Sample Parameter File

```
# env: CAPO_PAR
# Parameters for CAPTools-based Parallelizer with OpenMP (CAPO)
# They apply to version 1.1

# env: CAPO_LOG
# defines if log-information is wanted
log-file                on      (off on stdout)
# env: CAPO_LOGNAME
# defines log-file name when log-file = on
log-file-name                   (default: codeoutput.log)

# env: CAPO_LOGINFO
# defines type of information to be logged
log-info                std     (min std more debug)

# env: CAPO_PLOOP
# defines granularity (min. no. of iters.) for parallel loops
loop-granularity        6       (0 1 2 ...)

# env: CAPO_TYPE
# defines type of directives to be produced
directive-type          omp     (omp sgi sgix no)

# env: CAPO_REGION
# defines type of parallel regions to be considered
region-type             full    (loop bloop one join full)

# env: CAPO_OPTIMIZE
# defines optimization type for parallel regions
optimize-type           o2      (off on o2 o3)

# env: CAPO_USERLOOP
# defines the file name for user-defined loop types
user-loop-file                  (default: user-loop.par)

# env: CAPO_DIRCLEAR
# defines the file name for directives to be cleared
directive-clear         Default (off on filename)

# env: CAPO_TPRIV
# switches on/off the generation of THREADPRIVATE
tpriv-directive         on      (off on)

# env: CAPO_COMMENT
# chooses a comment type for directives
comment-type            f90     (f77 f90)

# env: CAPO_USEPARTI
# uses partitioned loops for directives
use-parti-loop          no      (no yes)

# env: CAPO_RDUPTYPE
# defines routine duplication type
rdup-type               region  (loop region)

# env: CAPO_UNKSIZE
# allows unknown-size variables
allow-unksize           false   (false true)

# env: CAPO_PIO
# allows parallel I/O
allow-pio               no      (no incall write noread any)
```

# A2. Messages and Symbols in the Log File

By default, the process of automatic insertion of directives is logged to the log-file "`code-output.log`". Information in this file may be examined after directives are added. There are three main sections in the log file, as outlined in the following subsections. Depending on the log-info type as described in Section A1, different levels of information details may be logged. In general, the log-info type controls:

**1)** `min` — only minimum amount of information, such as WARNING and INFO messages,

**2)** `std` — information from `min`, plus summary for each routine and each region,

**3)** `more` — information from `std`, plus more detailed results for each loop and each region,

**4)** `debug` — information from `more`, plus additional debug information that are probably too much for an ordinary user.

In the case of "`more`" and "`debug`", additional labels (region# and loop#) are added as comments for parallel loops in the generated parallel code. Regions and loops are labeled within a given routine, sequentially.

## A2.1. Classification of Loops

The first section lists the analysis of loops in all routines from the dependence information. For a given routine a loop is labeled with its sequence number, the group number and the loop-nesting level. The group number is defined as a sequence number for a loop-nest group at a given nesting level. Loops are classified as parallel, serial, or possible pipeline. For a parallel loop, it is further tested for granularity and is indicated if a parallel directive is to be added, provided the loop is not nested inside another parallel loop. For a serial loop, the reason of serialization as well as the first variable that causes the loop to be serialized is given. The causes of loop serialization include loop-carried dependences (true, anti and output), I/O statement inside, and breaking out of the loop. A pipeline loop is a serial loop with only loop-carried true dependences and determinable dependence vectors (see Section 2.4 for definition). The basic information for loops is as the following:

```
Routine: ROUTINE_NAME
  Loop # (loop_variable), group #, level #: parallel/serial
       TYPE? Reason for serial...
```

"`TYPE?`" is one of types from the loop type list:

```
   "REDU", "NPAR", "PAR", "IO", "LVAR", "SER", "ANTI", "PIPE",
   "BRK", "UPIPE", "PAREG", "INDU", "INPLP", "RDINP", "GRAN", "PARTI"
```

As an example, part of the analysis for three routines in NPB-LU is given here (with `log_info` set to `MORE`).

```
Routine: BUTS
 Loop 1 (J), group 1, level 1: parallel, granularity - ok
        PAR-> directives to be added for the loop <1,1>
 Loop 2 (I), group 1, level 2: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 Loop 3 (M), group 1, level 3: parallel, granularity - no
 Loop 4 (J), group 2, level 1: serial
```

```
        PIPE? true dependence, pipeline loop? dvector: V[0,0,-1,0]
 Loop 5 (I), group 2, level 2: serial
        PIPE? true dependence, pipeline loop? dvector: V[0,-1,0,0]
 Loop 6 (M), group 2, level 3: parallel, granularity - no
 Loop 7 (M), group 2, level 3: parallel, granularity - no
 *** Total number of loops: 7, parallel: 5, serial: 2, directive: 1
Routine: JACU
 Loop 1 (J), group 1, level 1: parallel, granularity - ok
        PAR-> directives to be added for the loop <1,1>
 Loop 2 (I), group 1, level 2: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 *** Total number of loops: 2, parallel: 2, serial: 0, directive: 1
...
Routine: SSOR
 Loop 1 (I), group 1, level 1: serial
        ANTI? loop carried output or non-exact anti dependence: ELAPSED
 Loop 2 (I), group 2, level 1: serial
        ANTI? loop carried output or non-exact anti dependence: ELAPSED
 Loop 3 (ISTEP), group 3, level 1: serial
        BRK? break out of the loop or comm-call inside the loop
 Loop 4 (K), group 3, level 2: parallel, granularity - ok
        PAR-> directives to be added for the loop <2,1>
 Loop 5 (J), group 3, level 3: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 Loop 6 (I), group 3, level 4: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 Loop 7 (M), group 3, level 5: parallel, granularity - no
 Loop 8 (K), group 3, level 2: serial
        SER? loop carried true dependence: ELAPSED
 Loop 9 (K), group 3, level 2: serial
        SER? loop carried true dependence: ELAPSED
 Loop 10 (K), group 3, level 2: parallel, granularity - ok
        PAR-> directives to be added for the loop <2,2>
 Loop 11 (J), group 3, level 3: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 Loop 12 (I), group 3, level 4: parallel, granularity - ok
        INPLP? no directive, loop inside a parallel loop
 Loop 13 (M), group 3, level 5: parallel, granularity - no
 *** Total number of loops: 13, parallel: 8, serial: 5, directive: 2

>>>> Grand total: num_routines 25, num_loops 157
           loops: parallel 145, serial 12, directive 30
```

The label for a parallel loop with directive to be added (`PAR->`) is given as <`level,group`> pairs. In the case of a serial loop only one variable is listed for the cause of serialization. For a potential pipeline loop, the dependence vector for the first related variable is given, as the case of `V[0,0,-1,0]` for loop 4 (J) in routine BUTS.

The user-defined loop types are applied after the loop classification. Therefore, it is user's responsibility to ensure the correctness of user-supplied loop types.

## A2.2. Construction and Optimization of Parallel Regions

This section contains first the summary from the pass-two analysis of all the routines in the outer-most loop level to decide if directives need to be added in a routine. Routines are traversed on their call sequences. A `<yes>` or `<no>` flag is marked for each analyzed routine to indicate the addition of directives in the routine. A routine may need to be duplicated if it is called both inside and outside a parallel loop and will contain directives in itself.

```
Routine: ROUTINE_NAME <yes/no/inploop/noploop>
```

| | |
|---|---|
| `<yes>` | — routine is added with directives for parallel loops |
| `<no>` | — routine has no directives |
| `<inploop>` | — routine is called inside a parallel loop |
| `<noploop>` | — routine has no parallel loop, but may contain potential pipeline loops |

A sample result from the analysis of NPB-LU looks like the following.

```
Routine: APPLU <yes>
Routine: READ_INPUT <no>
Routine: DOMAIN <no>
Routine: SETCOEFF <no>
Routine: SETBV <yes>
Routine: SETIV <yes>
Routine: ERHS <yes>
Routine: SSOR <yes>
Routine: TIMER_CLEAR <no>
Routine: JACLD <yes>
Routine: BLTS <yes>
Routine: JACU <yes>
Routine: BUTS <yes>
Routine: RHS <yes>
Routine: TIMER_START <no>
Routine: L2NORM <yes>
Routine: TIMER_STOP <no>
Routine: ELAPSED_TIME <no>
Routine: WTIME <no>
Routine: ERROR <yes>
Routine: EXACT <no>
Routine: PINTGR <yes>
Routine: VERIFY <no>
Routine: PRINT_RESULTS <no>
Routine: TIMER_READ <no>
>>> Total routines: 25, checked: 24, with directives: 13
    in/outside ploop: 0, in/with ploop: 0, no ploop: 12
    Total directive loops: 30, effective: 30, in ploop: 0
```

The last line of the statistics indicates how many loops can be put with directives, how many of them are really added with directives, and how many of them are nested inside other loops with directives.

Next is to construct parallel regions based on the loop information. A parallel region includes at least one parallel loop or pipeline loop with possible basic blocks in the beginning of the loop. No nested parallel loops are considered at this point. Two neighboring regions can be joined together if no codes other than comments or nops (such as `continue`) exist between the two regions. Individual regions are labeled sequentially within a routine. For each region a number is included in () to indicate the end (or last) region of a joined area of regions. For disjointed regions, the end region is the same as the region itself. Additional information included for a region are: loops in the region and type of the region. Regions are also summarized for a routine as "`region-type-summary`."

```
Region-type:
   one ploop      — containing exactly one parallel loop (no pipeline)
   +prev-block    — one parallel loop plus any preceded basic blocks
   sub ploop      — one or more parallel loops nested at different levels
   pipeline       — potential pipeline
   <default>      — region with joined neighbors

Region-type-summary:
   DEFAULT        — routine contains normal parallel regions
   PIPE           — routine is part of a pipeline region
   UPIPE          — routine contains potential pipeline regions
```

Sample outputs from the analysis of NPB-LU:

```
Region-in-Routine: BUTS
 region-type-summary: UPIPE
 Parallel region 1 (2): loops [1-3]
 Parallel region 2 (2): loops [4-7]
 *** Total number of regions: 2, joined regions: 1
Region-in-Routine: JACU
 region-type-summary: DEFAULT
 Parallel region 1 (1): loops [1-2] one ploop
 *** Total number of regions: 1, joined regions: 1
Region-in-Routine: SSOR
 region-type-summary: DEFAULT
 Parallel region 1 (1): loops [4-7] one ploop
 Parallel region 2 (2): loops [10-13] one ploop
 *** Total number of regions: 2, joined regions: 2
```

Once the initial regions are determined, routines are then checked for possible pipeline regions across routines. If such a region is identified, the pipeline-loop limit is checked against all other parallel loops in the same pipeline region for alignment. If a discrepancy is found, a message will be printed out as either "not the same limit" or "low-high limit swapped." In the first case, the suggested pipeline operation may produce incorrect run-time result and further check of this generated code is needed. In the second case CAPO automatically swaps the loop limit to ensure the consistence. If pipeline loops are not desirable, set the environment variable `CAPO_REGION` to "`join`."

For LU, routines `BUTS` and `JACU` were identified to be part of a pipeline region in routine `SSOR` and information was generated as follows.

```
Region-in-Routine: BUTS
 region-type-summary: PIPE
 pipeloop: DO J=JEND,JST,-1 (BUTS)
 thisloop: DO J=JEND,JST,-1 (BUTS)
   same limit
Region-in-Routine: JACU
 region-type-summary: PIPE
 pipeloop: DO J=JEND,JST,-1 (BUTS)
 thisloop: DO J=JST,JEND,1 (JACU)
   low-high limit swapped!
Region-in-Routine: SSOR
 region-type-summary: DEFAULT
 Parallel region 1 (1): loops [4-7] one ploop
 Parallel region 2 (2): loops [8-8] pipeline
 Parallel region 3 (3): loops [9-9] pipeline
 Parallel region 4 (4): loops [10-13] one ploop
 *** Total number of regions: 4, joined regions: 4

>>>> Grand total: routines 25, regions 34, joined regions 26
```

Parallel regions are further optimized for removal of end-of-loop synchronization (use the 'NOWAIT' construct). Although more conservative approach is taken, careful examination of NOWAIT is still needed. For example, one should pay attention to the WARNING messages on "EndLoop-Sync required/re-enforced." If any problem occurs, one can always switch the optimization off (setenv CAPO_OPTIMIZE off).

For LU, this is the summary after region optimization:

>>>> Total number of syncs removed: 7, in 4 routines (13 checked)

## A2.3. Insertion of Directives in Routines

There are four functions performed in this stage:

- clearing any old directives if CAPO_DIRCLEAR is not off (Section A1.3),
- searching for threadprivate common blocks and inserting the THREADPRIVATE directive if CAPO_TPRIV is not off,
- duplicating routines if needed, and
- inserting region/loop-level directives.

Information resulted from these four actions are not fed back to the Directives Browser except for presented as directives in the source code. Thus, once directives are inserted, the Directives Browser should not be used to do further changes.

A threadprivate common block is the one that have all its variables used as private (including copyin) for all the parallel regions in the whole program. It means even a single instance of a non-private usage of a variable can prevent the common block from becoming threadprivate. In the debug mode, causes of a common block being determined as threadprivate or shared can be examined (see Section A2.4 for details). Normally messages are printed for identified threadprivate common blocks and routines that contain them. An example is given here.

```
T_PRIV common blocks:
 -/WORK_1D/-18: SP SET_CONSTANTS EXACT_RHS INITIALIZE ADI TXINVR X_SOLVE
       NINVR Y_SOLVE PINVR Z_SOLVE LHSINIT TZETAR ADD VERIFY ERROR_NORM
       COMPUTE_RHS RHS_NORM
 -/WORK_LHS/-18: SP SET_CONSTANTS EXACT_RHS INITIALIZE ADI TXINVR X_SOLVE
       NINVR Y_SOLVE PINVR Z_SOLVE LHSINIT TZETAR ADD VERIFY ERROR_NORM
       COMPUTE_RHS RHS_NORM

>>> THREADPRIVATE directive added for 2 common blocks in 18 routines
```

Warnings may be printed for those common blocks that may potentially be threadprivate:

```
WARNING! SSOR... region 4, loop 8
```

```
      /CJAC/ Type conflict: old SHARED, new PRIV - use SHARED
```

It indicates that in routine SSOR all variables in common block /CJAC/ are used as private in region 4, but the common block is shared in other places. One can trace further for where the common block is shared in the debug mode.

Directives are added by annotating the call graph and using the parallel region information obtained in A2.2. The call paths are printed as the insertion is progressing. Any routine is only visited one time.

```
Routine: APPLU
Routine: APPLU->SETCOEFF
Routine: APPLU
Routine: APPLU->SETBV
Routine: APPLU
Routine: APPLU->SETIV
Routine: APPLU
Routine: APPLU->ERHS
Routine: APPLU
Routine: APPLU->SSOR
Routine: APPLU->SSOR->RHS
Routine: APPLU->SSOR->RHS->TIMER_START
Routine: APPLU->SSOR->RHS->TIMER_START->ELAPSED_TIME
Routine: APPLU->SSOR->RHS->TIMER_START->ELAPSED_TIME->WTIME
Routine: APPLU->SSOR->RHS->TIMER_START->ELAPSED_TIME
Routine: APPLU->SSOR->RHS->TIMER_START
Routine: APPLU->SSOR->RHS
Routine: APPLU->SSOR->RHS->TIMER_STOP
Routine: APPLU->SSOR->RHS
Routine: APPLU->SSOR
Routine: APPLU->SSOR->L2NORM
INFO! Array reduction variable replaced with local critical in region 1 -
        SUM() --> SUM_CAP1()
Routine: APPLU->SSOR
Routine: APPLU->SSOR->JACLD
Routine: APPLU->SSOR
Routine: APPLU->SSOR->BLTS
Routine: APPLU->SSOR
WARNING! Potential memory conflict for shared variable in region <2,1> -
ELAPSED
Routine: APPLU->SSOR->JACU
Routine: APPLU->SSOR
Routine: APPLU->SSOR->BUTS
Routine: APPLU->SSOR
WARNING! Potential memory conflict for shared variable in region <3,1> -
ELAPSED
Routine: APPLU
Routine: APPLU->ERROR
INFO! Array reduction variable replaced with local critical in region 1 -
        ERRNM() --> ERRNM_CAP1()
Routine: APPLU
Routine: APPLU->PINTGR
Routine: APPLU
Routine: APPLU->VERIFY
Routine: APPLU
```

WARNINGs for "...variable used after a parallel region," "potential memory conflict," and INFOs on the changes made to routine arguments should be examined carefully. These are just warnings, may or may not cause any programming errors. The warnings are the cases where CAPO are uncertain of decision making and user needs to inspect the generated code at the pointed places for verification. The parallel region is labeled as `<region_number, parallel_loop_number>` pairs in the call path right preceding the warning message.

Meanings of keywords in the WARNING message:

| | |
|---|---|
| "variable" | — a variable used in the current routine scope |
| "common-variable" | — a variable used outside the current scope, e.g. through COMMON blocks or SAVE statements in a subroutine |
| "Shared" | — variable shared in the current region |
| "Plocal" | — potential private variable in the current region |
| "Control" | — variable with multiple control paths, i.e. variable could be updated either inside or outside the current region |
| "I/O statement" | — routine called inside a parallel region contains i/o (OPEN,READ,WRITE,CLOSE) statements |
| "STOP statement" | — routine called inside a parallel region contains STOP/PAUSE statements |
| "Potential memory conflict" | — for shared variable that can cause memory conflict in a parallel region |

If a private variable in a parallel region is updated via a `COMMON` block in a subroutine, CAPO tries to privatize such a variable by adding it to the subroutine's argument list and renaming the original variable in the `COMMON` block of the subroutine. CAPO will generate the following INFO messages in this process:

```
New argument () added to CALL OTHER_ROUTINE():# in ROUTINE_NAME
New symbol () added to the argument list of ROUTINE_NAME
Common block /cblk/ duplicated for ROUTINE_NAME
```

CAPO performs a code transformation automatically for a reduction variable that is an array element. The corresponding message is like:

```
Array reduction variable replaced with scalar in region # -
      OLD_ARRAY_ELEMENT --> NEW_SCALAR_VARIABLE
```

## A2.4. Debug Information

More information will be logged if `CAPO_LOGINFO` is set to "`debug`." These are useful for debugging CAPO. Some of the information are included here for reference only.

- UserLoop information for user-defined loop types

  ```
  Userloop: Defined loop # in routine ROUTINENAME - newtype
  ```

  "`newtype`" is one of (S, P, R, B) as mentioned in Section A1.3.

- List of old directives to be cleared

- Summary of loop type with list of all dependence vector deltas for pipeline loops

- Three tests during region formation

```
Mem-Conflict check for region #R, loops #L-#L...
   Conflict variables: <var,var...>
Shared-Array check for region #R, loops #L-#L...Assigned <Symbol>
IO-Statement check for region #R, loops #L-#L...
   I/O or Reduction in routine <RoutineName>
```

- List of symbols and types in each region

`TYPE`

| | |
|---|---|
| `Private` | — Local (privatizable) variable |
| `Reduction` | — Scalar reduction variable |
| `ArrayReduction` | — Array reduction variable |
| `Shared` | — Shared variable |
| `LastPrivate` | — Usage in and after the region |
| `FirstPrivate` | — Usage in and before the region |
| `CopyInOut` | — Shared but no or no proof of loop-variable dependence |
| `ThreadPrivate` | — Used in a threadprivate common block |
| `UnknownType` | — Type not defined yet |

`CONTROL`

| | |
|---|---|
| `No-Control` | — Symbol not in a control dependence |
| `Control-Dep` | — Symbol in a control dependence |

`SCOPE`

| | |
|---|---|
| `In-Scope` | — Symbol defined in the current routine |
| `Not-in-Scope` | — Symbol not defined in the current routine (defined via common block or save statement) |
| `Not-in-Use` | — Symbol passed into a subroutine but not used in the subroutine |

`DTYPE:DEPTH` (printed in `[.:.]`)

| | | |
|---|---|---|
| `IO` | `-1,` | `Routine` Input/Output |
| `NT` | `0,` | Non-exact True |
| `NA` | `1,` | Non-exact Anti |
| `NO` | `2,` | Non-exact Output |
| `ET` | `3,` | Exact True |
| `EA` | `4,` | Exact Anti |
| `EO` | `5,` | Exact Output |
| `CT` | `6,` | Control |
| `UN` | `7,` | Unknown type |

`Depth = 0` for loop-independent dependence

- List of routine call types, indicating the usage of a routine inside/outside parallel regions/loops. Five bits are used:

| | |
|---|---|
| `bit1 [0x01]` | called outside parallel region |
| `bit2 [0x02]` | called inside paregion but outside parallel loop |
| `bit3 [0x04]` | called inside parallel loop |
| `bit4 [0x08]` | called outside parallel loop (= bit1 \| bit2) |
| `bit5 [0x10]` | called inside parallel region |

- Information on updating duplicated routines

```
Replace call to DROUTINE with CAP_DROUTINE in ROUTINE
Removed ROUTINE from the calledby list of DROUTINE
Added   ROUTINE to the calledby list of CAP_DROUTINE
```

- List of symbols and affine expressions for testing loop limits (such as in the removal of end-of-loop synchronizations)

```
HOME (LOOP-VAR-EXPR, #hits) Low <EXPR> High <EXPR> [A1:INDX,A2:INDX..]
     (LOOP-VAR-EXPR, #hits) Low <EXPR> High <EXPR> [B1:INDX,B2:INDX..]
OTHER (NONLOOP-EXPR, #hits) [C1:INDX,C2:INDX..]
      (NONLOOP-EXPR, #hits) [D1:INDX,D2:INDX..]
```

  Here <EXPR> is a symbolic expression, `A,B,C,D` are array names, `INDX` is the relevant array index. The lists are for both source and sink.

- Summary of fields associated with the ploopinfo data struct, mainly for development purpose.

```
 Loop   Lvar    D/L     Type    G WP IP Nest Flag
Routine: ROUTINE_NAME
    #   var     ?/?     TYPE?   ?  ?  ? n/cn [321]
```

|  |  |
|---|---|
| `'Loop'` | — the loop number in a routine |
| `'Lvar'` | — the loop variable name |
| `'D'` | — the nesting level of the outermost DO loop containing this loop |
| `'L'` | — the nesting level of the loop |
| `'Type'` | — one of type strings given in Section A2.1 |
| `'G'` | — the loop granularity flag (internal info only) |
| `'WP'` | — '1' containing parallel loop, '0' without parallel loop |
| `'IP'` | — '1' inside parallel loop, '0' not inside parallel loop |
| `'n'` | — this loop nest flag (containing nested parallel loop) |
| `'cn'` | — child loop nest flag (part of nested parallel loops) |
| `'Flag'` | — three bits for internal usage only |

- Symbols and their types in common blocks (for testing threadprivate). Meanings of symbol types:

```
[U] — Unset
[P] — Private
[R] — Reduction
[A] — ArrayReduction
[S] — Shared (RW)
[s] — Shared (Readonly)
[L] — LastPrivate
[F] — FirstPrivate
[C] — CopyInOut
```

- Methods used in determining the declaration size of unknown-size variables

```
[NOT ]IDENTICAL SIZE, method 1 (caller declaration) used
MAX(e1,...), MIN(e1,...), method 2|3 (access range in routine) used
NO method - variable NOT safe - <var>
```

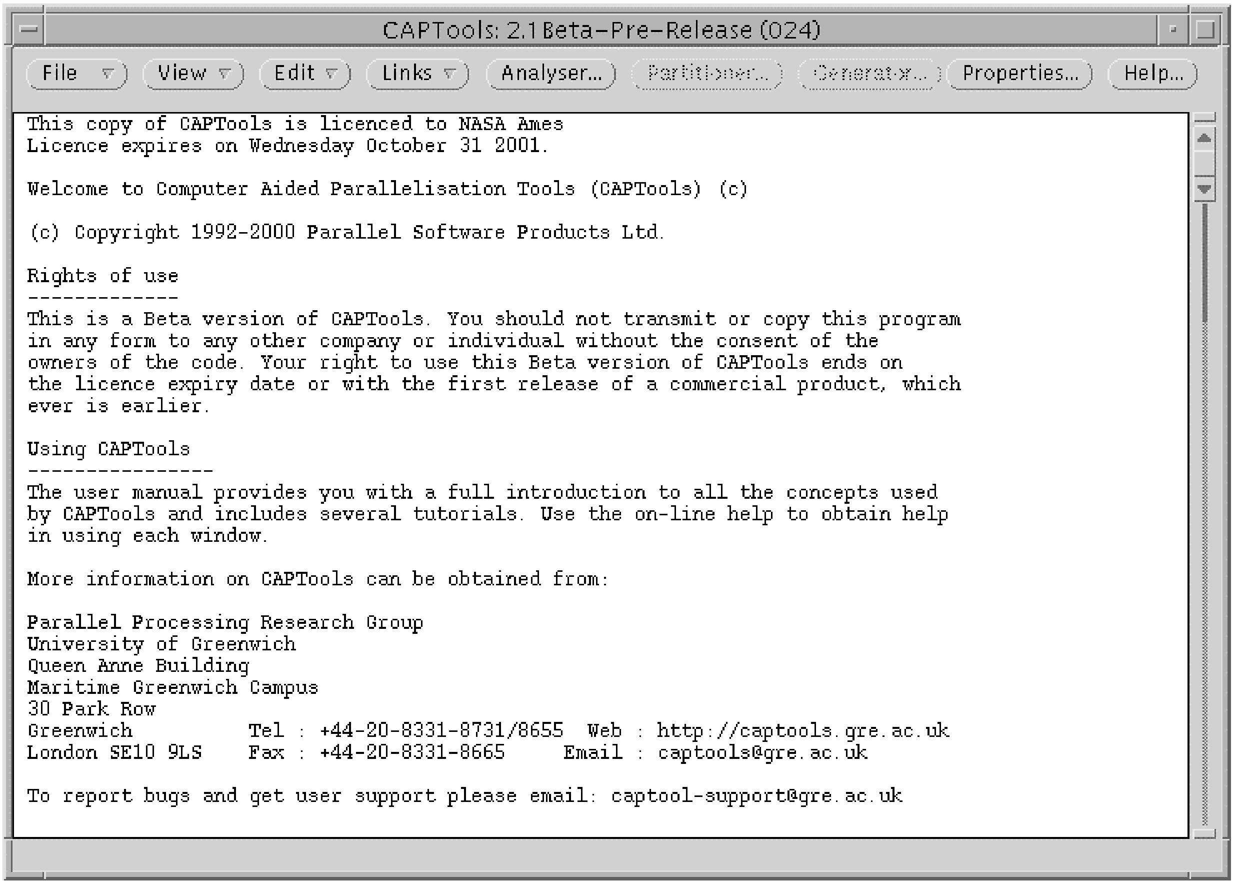
# A3. CAPO Graphical User Interface

CAPO is currently integrated into CAPTools as a component to generate OpenMP directives. For CAPO-enabled CAPTools, additional items have been added to the **File**, **View** and **Edit** menus of the CAPTools main window to access the CAPO graphical user interface (GUI).

The CAPO GUI is also referred to as the *Directives Browser*. It provides an easy way for user to access information generated during the directives analysis and insertion. The browser consists of several information windows and dialog boxes as given in the following sections. It also provides hookups to the CAPTools GUI tools, such as DepGraph browser, Variable Definition browser, etc., so that one can easily navigate and interact with the parallelization process.

## A3.1. CAPTools Main Window

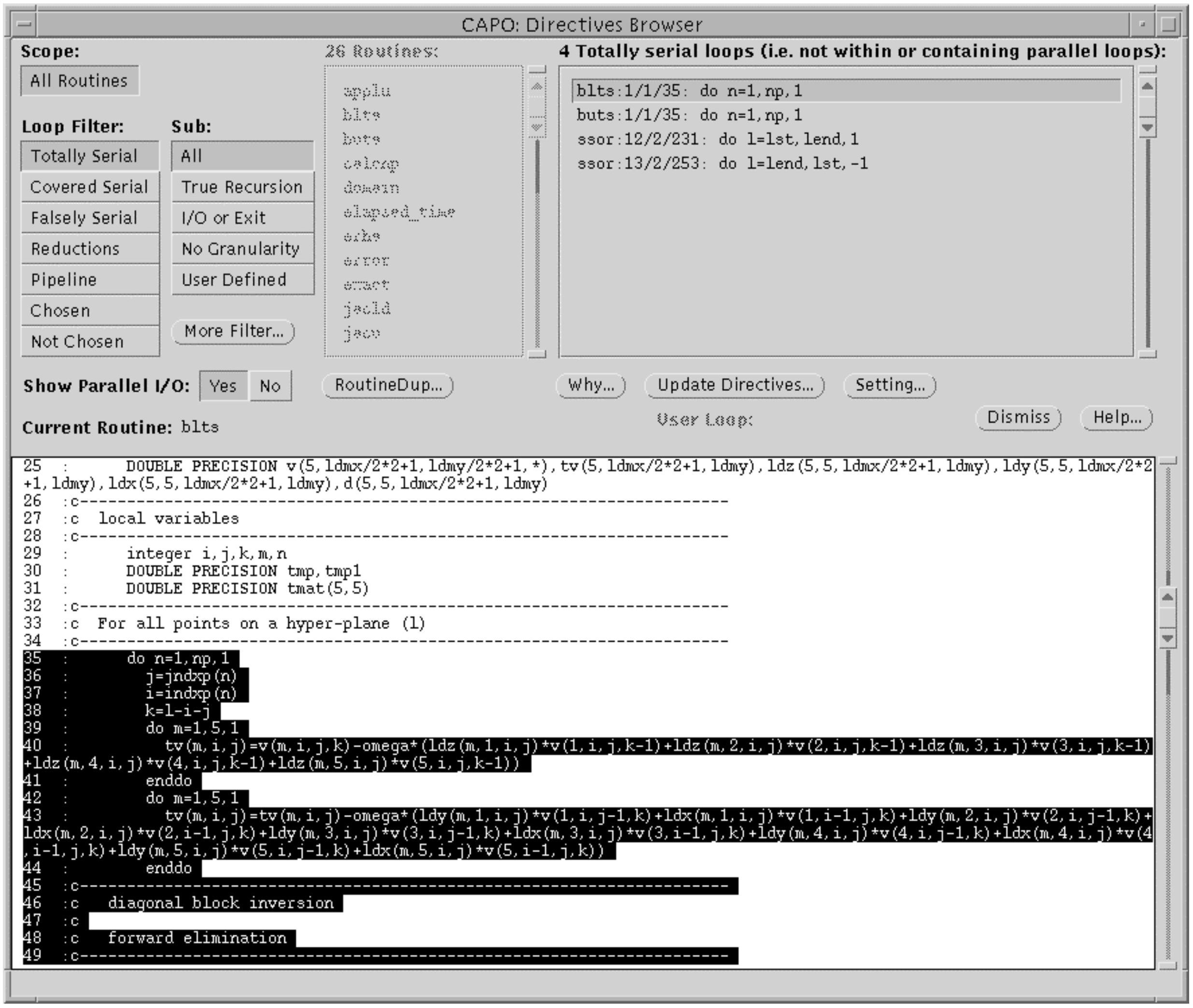
This is the main GUI window the user will see after CAPO/CAPTools is started. The CAPO GUI (the Directives Browser) is started from the **View** (***Directives***) menu after a source file or a database file is loaded from the **File** menu. A summary of CAPO hookups to CAPTools is given in Section A3.12.



```
This copy of CAPTools is licenced to NASA Ames
Licence expires on Wednesday October 31 2001.

Welcome to Computer Aided Parallelisation Tools (CAPTools) (c)

 (c) Copyright 1992-2000 Parallel Software Products Ltd.

Rights of use
-------------
This is a Beta version of CAPTools. You should not transmit or copy this program
in any form to any other company or individual without the consent of the
owners of the code. Your right to use this Beta version of CAPTools ends on
the licence expiry date or with the first release of a commercial product, which
ever is earlier.

Using CAPTools
----------------
The user manual provides you with a full introduction to all the concepts used
by CAPTools and includes several tutorials. Use the on-line help to obtain help
in using each window.

More information on CAPTools can be obtained from:

Parallel Processing Research Group
University of Greenwich
Queen Anne Building
Maritime Greenwich Campus
30 Park Row
Greenwich           Tel : +44-20-8331-8731/8655  Web : http://captools.gre.ac.uk
London SE10 9LS     Fax : +44-20-8331-8665     Email : captools@gre.ac.uk

To report bugs and get user support please email: captool-support@gre.ac.uk
```

## A3.2. Directives Browser Main Window

The main window of the Directives browser is activated by **View–>*Directives…*** from the CAPTools main window (see Sections A3.1 and A3.12) after a source or database is loaded in. It presents information from the first two phases of the directives analysis (before directives are added). It is organized around loop types and is an entry point for other browser windows, such as **WhyDirectives** and **RoutineDuplication**. Once directives are generated (via ***Save OpenMP Directives Code***), the Directives browser should not be used to do further changes.



4 Totally serial loops (i.e. not within or containing parallel loops):

```
blts:1/1/35:  do n=1,np,1
buts:1/1/35:  do n=1,np,1
ssor:12/2/231:  do l=lst,lend,1
ssor:13/2/253:  do l=lend,lst,-1
```

Current Routine: blts

```
25   :       DOUBLE PRECISION v(5,ldmx/2*2+1,ldmy/2*2+1,*),tv(5,ldmx/2*2+1,ldmy),ldz(5,5,ldmx/2*2+1,ldmy),ldy(5,5,ldmx/2*2
+1,ldmy),ldx(5,5,ldmx/2*2+1,ldmy),d(5,5,ldmx/2*2+1,ldmy)
26   :c-------------------------------------------------------------------
27   :c  local variables
28   :c-------------------------------------------------------------------
29   :       integer i,j,k,m,n
30   :       DOUBLE PRECISION tmp,tmp1
31   :       DOUBLE PRECISION tmat(5,5)
32   :c-------------------------------------------------------------------
33   :c  For all points on a hyper-plane (l)
34   :c-------------------------------------------------------------------
35   :       do n=1,np,1
36   :         j=jndxp(n)
37   :         i=indxp(n)
38   :         k=l-i-j
39   :         do m=1,5,1
40   :           tv(m,i,j)=v(m,i,j,k)-omega*(ldz(m,1,i,j)*v(1,i,j,k-1)+ldz(m,2,i,j)*v(2,i,j,k-1)+ldz(m,3,i,j)*v(3,i,j,k-1)
+ldz(m,4,i,j)*v(4,i,j,k-1)+ldz(m,5,i,j)*v(5,i,j,k-1))
41   :         enddo
42   :         do m=1,5,1
43   :           tv(m,i,j)=tv(m,i,j)-omega*(ldy(m,1,i,j)*v(1,i,j-1,k)+ldx(m,1,i,j)*v(1,i-1,j,k)+ldy(m,2,i,j)*v(2,i,j-1,k)+
ldx(m,2,i,j)*v(2,i-1,j,k)+ldy(m,3,i,j)*v(3,i,j-1,k)+ldx(m,3,i,j)*v(3,i-1,j,k)+ldy(m,4,i,j)*v(4,i,j-1,k)+ldx(m,4,i,j)*v(4
,i-1,j,k)+ldy(m,5,i,j)*v(5,i,j-1,k)+ldx(m,5,i,j)*v(5,i-1,j,k))
44   :         enddo
45   :c-------------------------------------------------------------------
46   :c   diagonal block inversion
47   :c
48   :c   forward elimination
49   :c-------------------------------------------------------------------
```

**Scope** [setting]: selects one routine or all routines for loop listing.

**Routines** [list]: a list of routines that can be selected for loop listing.

**Loops** [list]: a list of loops under the selected routine/loop filters. To activate the **WhyDirectives** window through the $\boxed{\text{Why…}}$ button, a loop needs to be selected.

**Loop Filter** [list]: provides a way to focus on a particular type of loops, mainly serial or parallel, as described in details in Section A3.3.

**Sub** [list]: sub-loop filter to be combined with the loop filter to provide finer control of loop selection.

**More Filter** [button]: activates the **Loop Variable Filter** window to perform even finer loop selection (Section A3.3.1).

**Show Parallel I/O** [setting]: controls the way that a loop with I/O statements inside is displayed. By default (*Yes*), loops with potential parallel I/O are classified as parallel although parallel I/O with directives is not supported at this point.

**RoutDup** [button]: activates the **RoutineDuplication** window (Section A3.5).

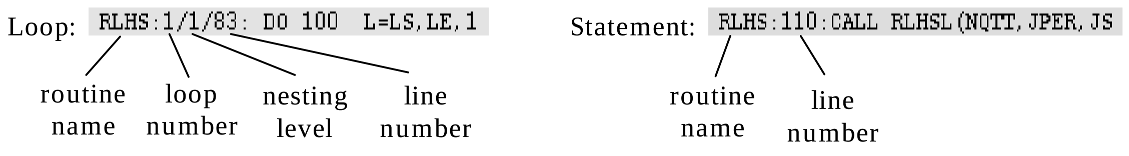
**Why** [button]: activates the **WhyDirectives** window (Section A3.4) after a loop is selected.

**Update Directives** [button]: activates the **Update** dialog box (Section A3.9) to re-perform the directives analysis, usually after settings are changed.

**Setting** [button]: activates the **Setting** window (Section A3.6) to reset parameters for CAPO. The window may also be launched from **Edit**–>***Directives Setting...*** in the CAPTools main window.

**Current Routine** [textpane]: displays the source of a selected routine or a routine in which a selected loop is located. The selected loop nest is highlighted.

***How a loop or a statement is labeled***:



```
RLHS:1/1/83: DO 100  L=LS,LE,1
RLHS:110:CALL RLHSL(NQTT,JPER,JS
```

## A3.3. Loop Filters and Sub-filters

Definitions of basic loop types:

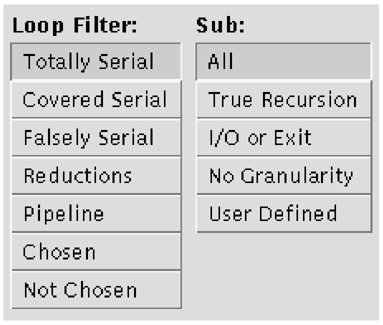
**Serial loop** — a loop *with* loop-carried TRUE dependence from data flow, ANTI/OUTPUT dependence from non-privatizable variables, I/O statements, and/or exit statements.

**Parallel loop** — a loop *without* loop-carried TRUE dependence from data flow, ANTI/OUTPUT dependence from non-privatizable variables, I/O statements, and exit statements. Such a loop can be executed in parallel.

**Reduction loop** — a loop, other than one or more reduction operations, that can be executed in parallel.

**Pipeline loop** — a loop that contains loop-carried TRUE dependences with determinable, non-negative dependence vectors (see definition in Section 2.4). The loop can potentially be used to set up a parallel pipeline with an outer loop.

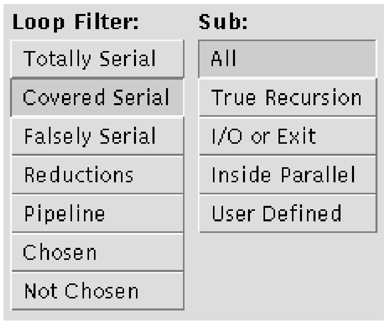
**Distributed loop** — one of Parallel loop, Reduction loop or Pipeline loop.



**Loop Filter:** ***Totally Serial* —**
serial loop with loop-carried TRUE dependence, containing no distributed loop and not nested inside other distributed loop. The code section in the loop will be executed sequentially.

**Sub-filter:**

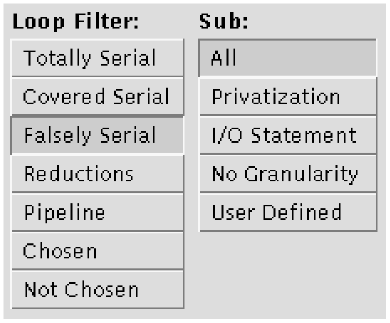
| | |
|---|---|
| ***True Recursion*** | — no I/O or exit statements |
| ***I/O or Exit*** | — with I/O and/or exit statements |
| ***No Granularity*** | — one or no iteration |
| ***User Defined*** | — user-defined serial loop |



**Loop Filter:** ***Covered Serial* —**
serial loop with loop-carried TRUE dependence, containing distributed loop or nested inside other distributed loop. The code section in the loop will partially or completely be executed in parallel.

**Sub-filter:**

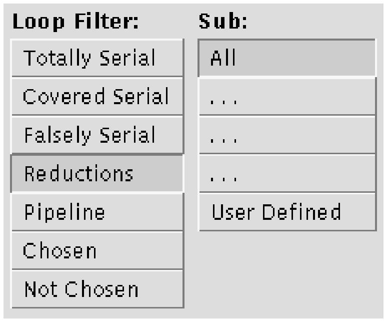
| | |
|---|---|
| ***True Recursion*** | — no I/O or exit statements |
| ***I/O or Exit*** | — with I/O and/or exit statements |
| ***Inside Parallel*** | — inside other parallel loops |
| ***User Defined*** | — user-defined serial loop |



**Loop Filter:** ***Falsely Serial* —**
serial loop without loop-carried TRUE dependence, but containing ANTI/OUTPUT dependence from non-privatizable variables. Loop may contain distributed loops for parallel execution.

**Sub-filter:**

| | |
|---|---|
| ***Privatization*** | — due to non-privatizable variables |
| ***I/O Statement*** | — with I/O statements but no nested parallel loops |
| ***No Granularity*** | — no granularity and no nested parallel loops |
| ***User Defined*** | — user-defined serial loop |

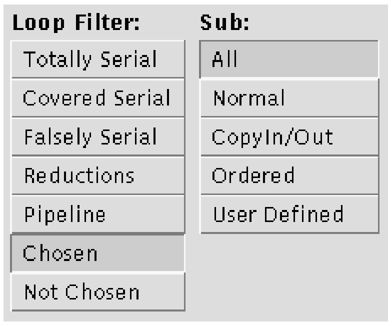


**Loop Filter:** ***Reductions* —**
loop with one or more reduction operations which can be executed as parallel reductions.

**Loop Filter:** ***Pipeline* —**
A pipeline loop as part of a parallel pipeline working with an outer loop.

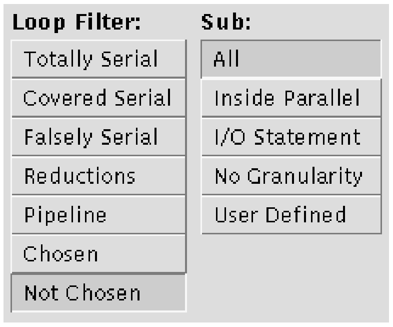
**Sub-filter:**

| | |
|---|---|
| ***All*** | — all loops with reductions/pipeline |
| ***User Defined*** | — user-defined reduction loop |



**Loop Filter:** ***Chosen (Parallel)*** **—**
parallel loop chosen for distribution with directives. The code section in the loop will be executed in parallel.

| | | |
|---|---|---|
| **Sub-filter:** | ***Normal*** | — regular parallel loop |
| | ***CopyIn/Out*** | — with copyin/copyout variables |
| | ***Ordered*** | — with ordered code section |
| | ***User Defined*** | — user-defined parallel loop |

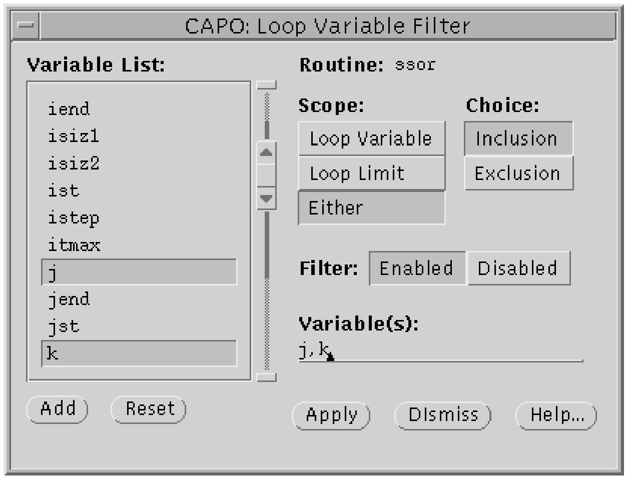


**Loop Filter:** ***Not Chosen (Parallel)*** **—**
parallel loop not chosen due to other parallel loop(s) already been chosen. The loop is either inside other distributed loop or contains distributed loops.

| | | |
|---|---|---|
| **Sub-filter:** | ***Inside Parallel*** | — inside other parallel loops |
| | ***I/O Statement*** | — with I/O statements |
| | ***No Granularity*** | — parallel but no granularity |
| | ***User Defined*** | — user-defined parallel loop |

### A3.3.1. Loop Variable Filter Window

The Loop Variable Filter Window controls even finer selection of loops in conjunction with the main loop filter and sub filter. The filter applies to variables used in loop heads.



**Routine** [label]: indicates the currently selected routine.

**Variable List** [list]: contains a list of variables used in the loop heads of the current routine.

**Scope** [setting]: controls the scope of variables.

| | |
|---|---|
| Loop Variable | — variables from loop iteration |
| Loop Limit | — variables from loop high-low limit |
| Either | — either of the above two cases |

**Choice** [setting]: controls the filtering effect.

| | |
|---|---|
| Inclusion | — show loops when variables appear |
| Exclusion | — show loops when variables do not appear |

**Filter** [setting]: disables or enables the loop variable filter.

**Variable(s)** [textfield]: contains a list of the currently filtered variables.

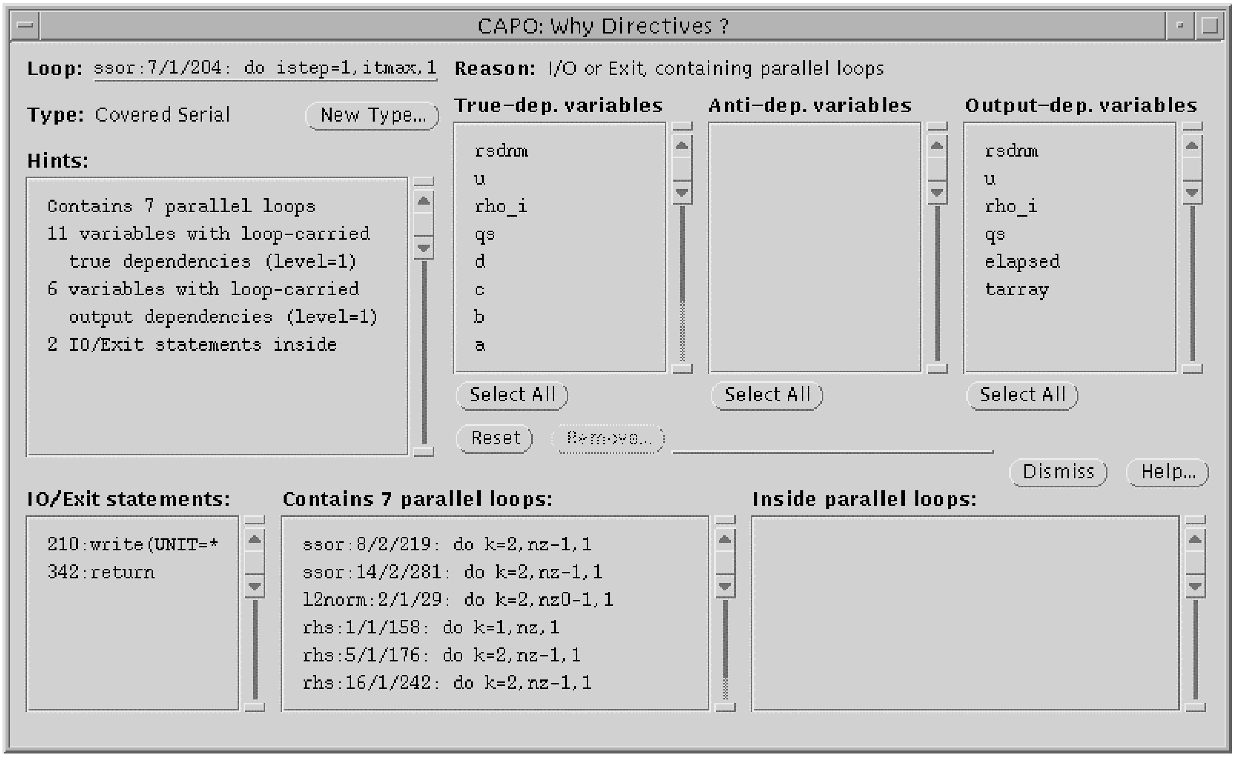
**Add** [button]: adds the selected variables in the ***Variable List*** to the filtered variable list.

**Reset** [button]: resets variable selection.

**Apply** [button]: applies the current filter to the display.

## A3.4. WhyDirectives Window

The **WhyDirectives** window is displayed for a selected loop after the Why… button is clicked in the **Directives** main window. It presents detailed information for the selected loop, in particular, reasons and hints on why the loop was classified as serial or parallel. The window can be used to remove false dependences identified by the user and to redefine the loop type. Depending on the current loop type, the three variable lists may show different types of variables and the two loop lists may present different information. The displayed window is for a loop of the *Covered Serial* type.



**Loop:** ssor:7/1/204: do istep=1,itmax,1 **Reason:** I/O or Exit, containing parallel loops

**Type:** Covered Serial

**Hints:**

```
Contains 7 parallel loops
11 variables with loop-carried
  true dependencies (level=1)
6 variables with loop-carried
  output dependencies (level=1)
2 IO/Exit statements inside
```

**True-dep. variables**

```
rsdnm
u
rho_i
qs
d
c
b
a
```

**Anti-dep. variables**

**Output-dep. variables**

```
rsdnm
u
rho_i
qs
elapsed
tarray
```

**IO/Exit statements:**

```
210:write(UNIT=*
342:return
```

**Contains 7 parallel loops:**

```
ssor:8/2/219: do k=2,nz-1,1
ssor:14/2/281: do k=2,nz-1,1
l2norm:2/1/29: do k=2,nz0-1,1
rhs:1/1/158: do k=1,nz,1
rhs:5/1/176: do k=2,nz-1,1
rhs:16/1/242: do k=2,nz-1,1
```

**Inside parallel loops:**

<u>**The following items are common for All Loop Types.**</u>

**Loop** [textfield]: currently selected loop with routine name and loop labels (see the end of Section A3.3).

**Type** [textfield]: loop type as described in Section A3.3.

**Reason** [textfield]: one sentence summarizing why the loop was classified to its type.

**Hints** [textarea]: more detailed summary of the usage of the relevant variables in the loop and whether the loop contains I/O statements, exit statements, etc.

**New Type** [button]: activates the **New Loop Type** dialog box (Section A3.7).

**Select All** [button]: selects all variables in the corresponding variable list.

**Reset** [button]: deselects all variables in the variable lists.

**Remove** [button]: activates the **Variable Removal** dialog box (Section A3.10) for the selected variables.

**IO/Exit statements** [list]: list of I/O and exit statements in the selected loop nest.

**The following list is common for Totally Serial and Covered Serial.**

**True-dep. variables** [list]: list of variables causing loop-carried TRUE dependences, *removable*. An "[x]" followed a variable indicates the dependence vector length for this variable.

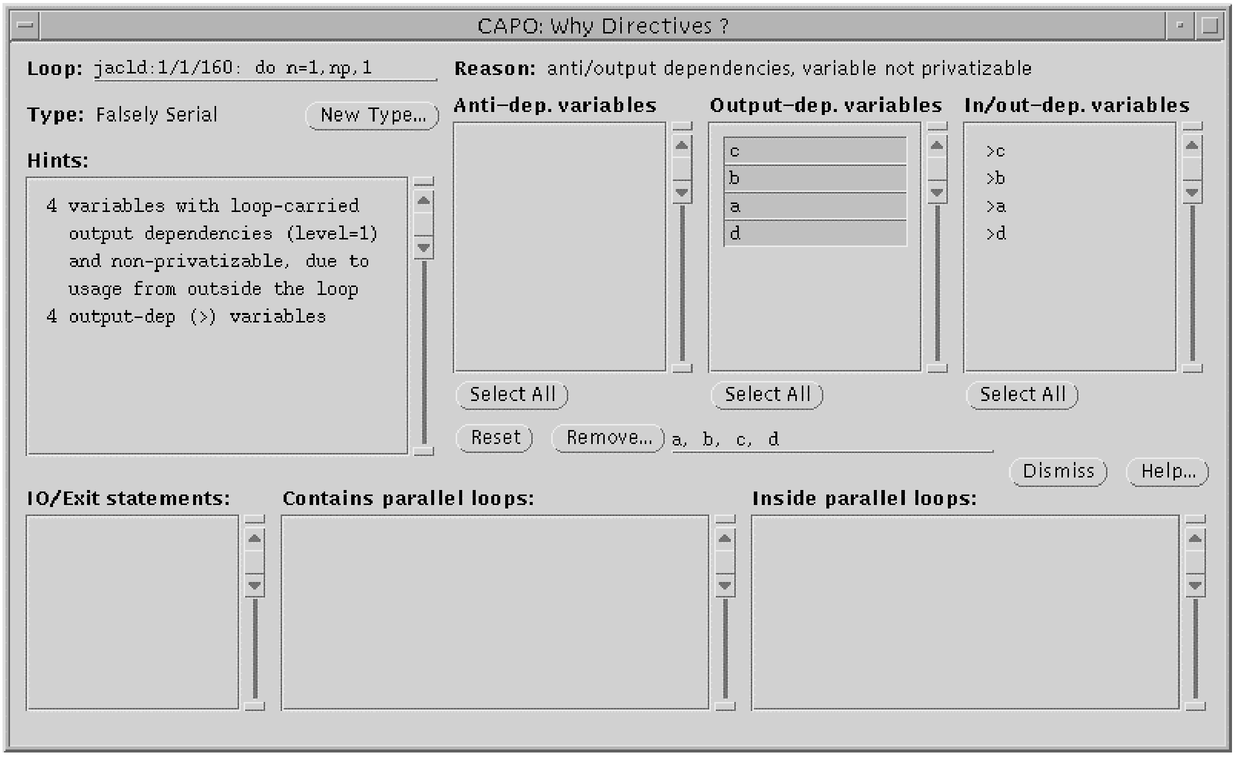
**The following lists are common for Totally Serial, Covered Serial and Falsely Serial.**

**Anti-dep. variables** [list]: list of variables causing loop-carried ANTI dependences and the variables cannot be privatized, *removable*.

**Output-dep. variables** [list]: list of variables causing loop-carried OUTPUT dependences and the variables cannot be privatized, *removable*.

**Contains parallel loops** [list]: list of parallel loops that are nested inside the current loop.

**Inside parallel loops** [list]: list of parallel loops that contain the current loop.



Loop: jacld:1/1/160: do n=1,np,1
Reason: anti/output dependencies, variable not privatizable
Type: Falsely Serial
Hints:

```
4 variables with loop-carried
  output dependencies (level=1)
  and non-privatizable, due to
  usage from outside the loop
4 output-dep (>) variables
```

Anti-dep. variables:
Output-dep. variables: c, b, a, d
In/out-dep. variables: >c, >b, >a, >d
Remove...: a, b, c, d
IO/Exit statements:
Contains parallel loops:
Inside parallel loops:

The above window is for a *Falsely Serial* loop.

**The following list is for Falsely Serial.**

**In/out-dep. variables** [list]: list of variables that have TRUE data dependences from the outside the loop, *removable*. A “<” sign in front of a variable indicates loop entry dependence on this variable, while a “>” sign indicates loop exit dependence on this variable.

**The following lists are common for Reductions, Pipeline, Chosen, and Not Chosen.**

**Private variables** [list]: list of privatizable variables in the loop nest, *not removable*.

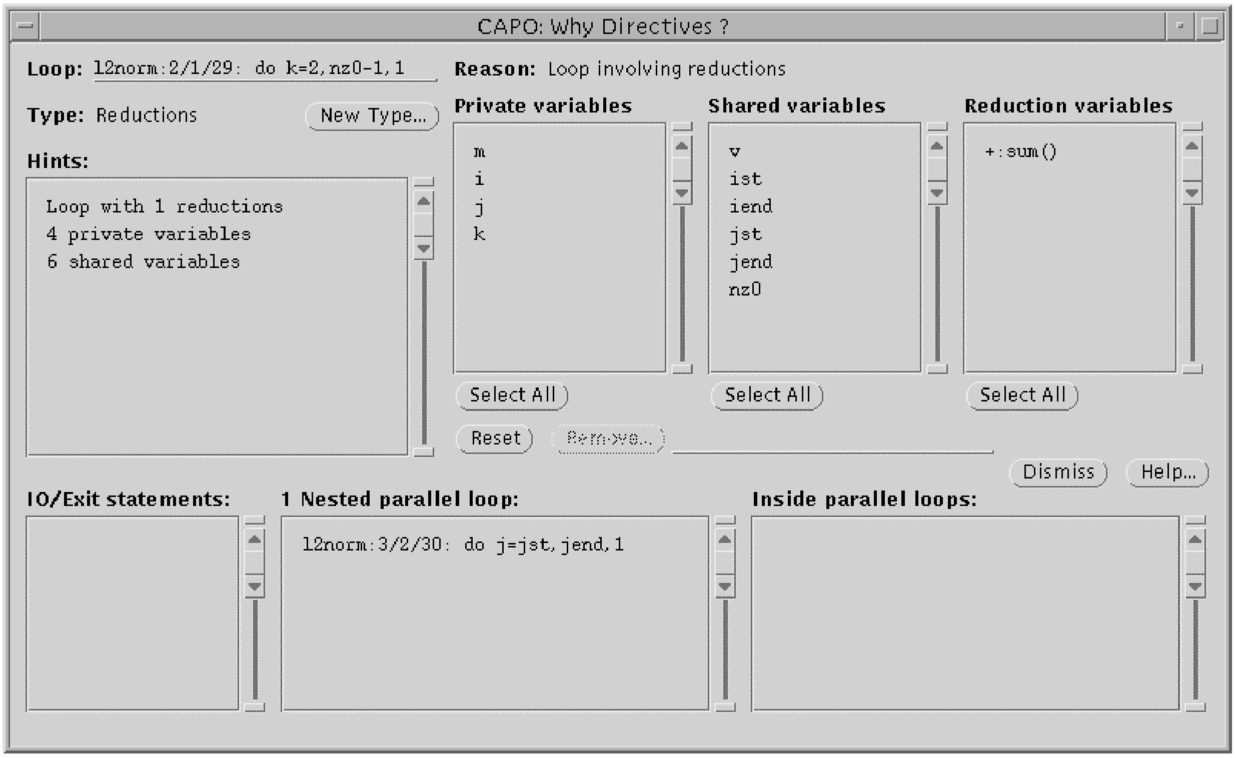
**Shared variables** [list]: list of shared variables in the loop nest, *not removable*.

**Nested parallel loops** [list]: list of secondary parallel loops that are nested inside the current loop.

**Inside parallel loops** [list]: list of parallel loops that contain the current loop (except for Pipeline).

**The following list is only for Reduction Loop.**

**Reduction variables** [list]: list of variables for reductions in the loop nest, *not removable*. Reduction variables are preceded with labels indicating reduction operators or intrinsic functions. A “()” after a variable indicates an array reduction.



**Loop:** 12norm:2/1/29: do k=2,nz0-1,1

**Reason:** Loop involving reductions

**Type:** Reductions

**Hints:**

```
Loop with 1 reductions
4 private variables
6 shared variables
```

| Private variables | Shared variables | Reduction variables |
|---|---|---|
| m | v | +:sum() |
| i | ist | |
| j | iend | |
| k | jst | |
| | jend | |
| | nz0 | |

**IO/Exit statements:**

**1 Nested parallel loop:**

```
12norm:3/2/30: do j=jst,jend,1
```

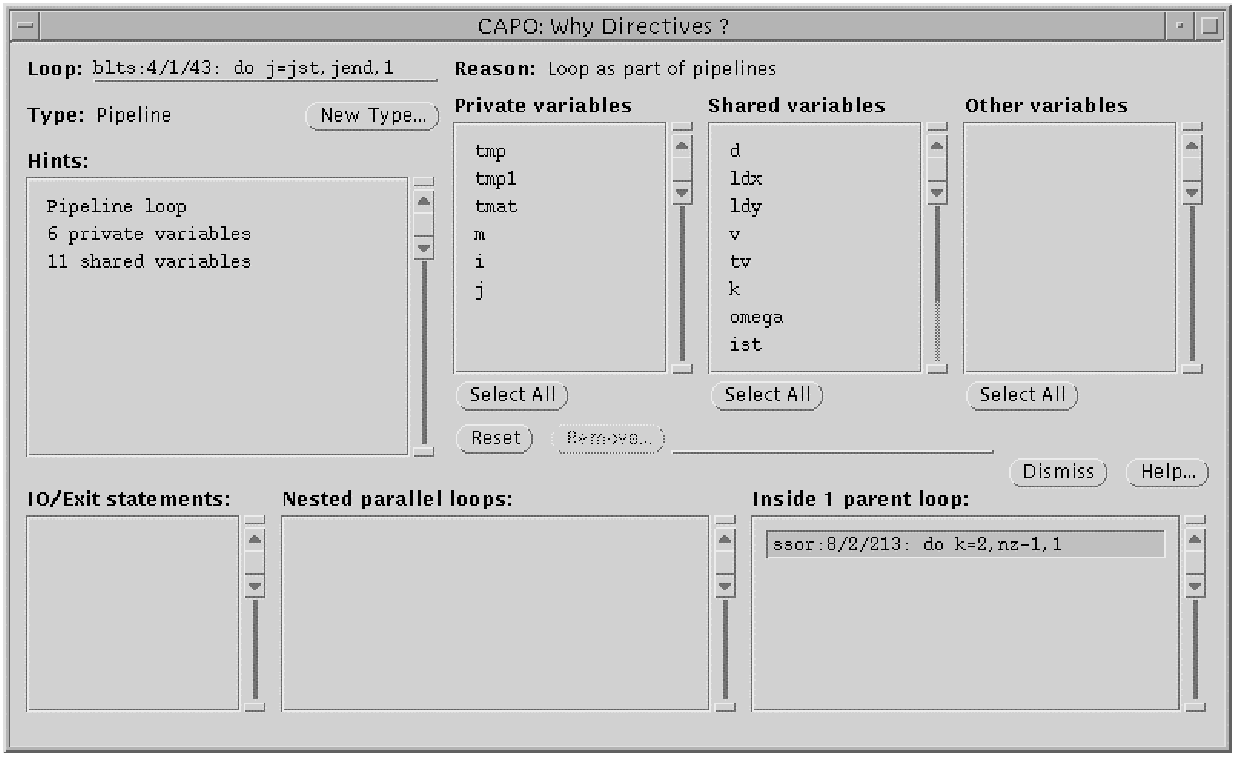
**Inside parallel loops:**

The above window is for a *Reduction* loop with reduction array variable “`sum()`.” A reduction operator or intrinsic is one of those defined in Section A3.8 or `IMAX/IMIN` (`MAX/MIN` expressed with an `IF` statement block).

**The following lists are only for Pipeline Loop.**

**Inside parent loops** [list]: list of loops that are nested above the current pipeline loop to form pipelines. Appropriate synchronization directives and statements will be inserted at the code generation. A parent loop is usually a serial loop without I/O and exit statement inside.

**Other variables** [list]: list of variables other than private and shared, such as **CopyIn/CopyOut** variables, *not removeable*.



```
Loop: blts:4/1/43:  do j=jst,jend,1      Reason: Loop as part of pipelines
Type: Pipeline
```

Hints:

```
Pipeline loop
6 private variables
11 shared variables
```

Private variables: tmp, tmp1, tmat, m, i, j

Shared variables: d, ldx, ldy, v, tv, k, omega, ist

Inside 1 parent loop:

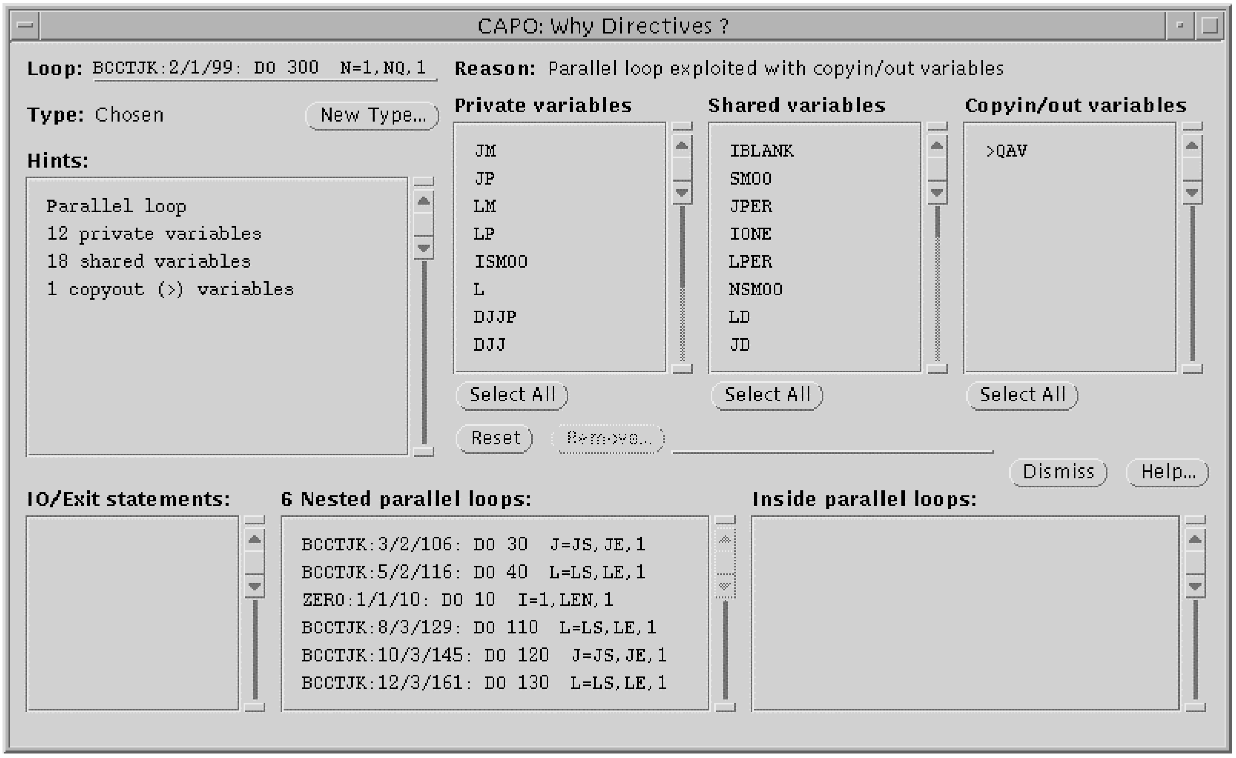
```
ssor:8/2/213:  do k=2,nz-1,1
```

The above window is for a *Pipeline* loop with the parent loop highlighted.

**The following lists are only for Chosen Parallel Loop.**

**Copyin/out variables** [list]: list of variables that will be declared as **CopyIn** (FIRSTPRIVATE, marked by "<") and/or **CopyOut** (LASTPRIVATE, marked by ">") due to potential conflict in updating the same memory location and the variable(s) having usage outside the loop. It might arise, for example, from an induction variable that is assigned before the loop and used after the loop. It could also indicate a programming bug.

**Controlled variables** [list]: list of variables that will be placed inside an "ORDERED" code section. These variables are usually inside IF conditional statements and the corresponding assignments need to be executed in a designated order as is in sequential.


CAPO: Why Directives ?

Loop: BCCTJK:2/1/99: DO 300 N=1,NQ,1
Reason: Parallel loop exploited with copyin/out variables

Type: Chosen

Hints:
```
Parallel loop
12 private variables
18 shared variables
1 copyout (>) variables
```

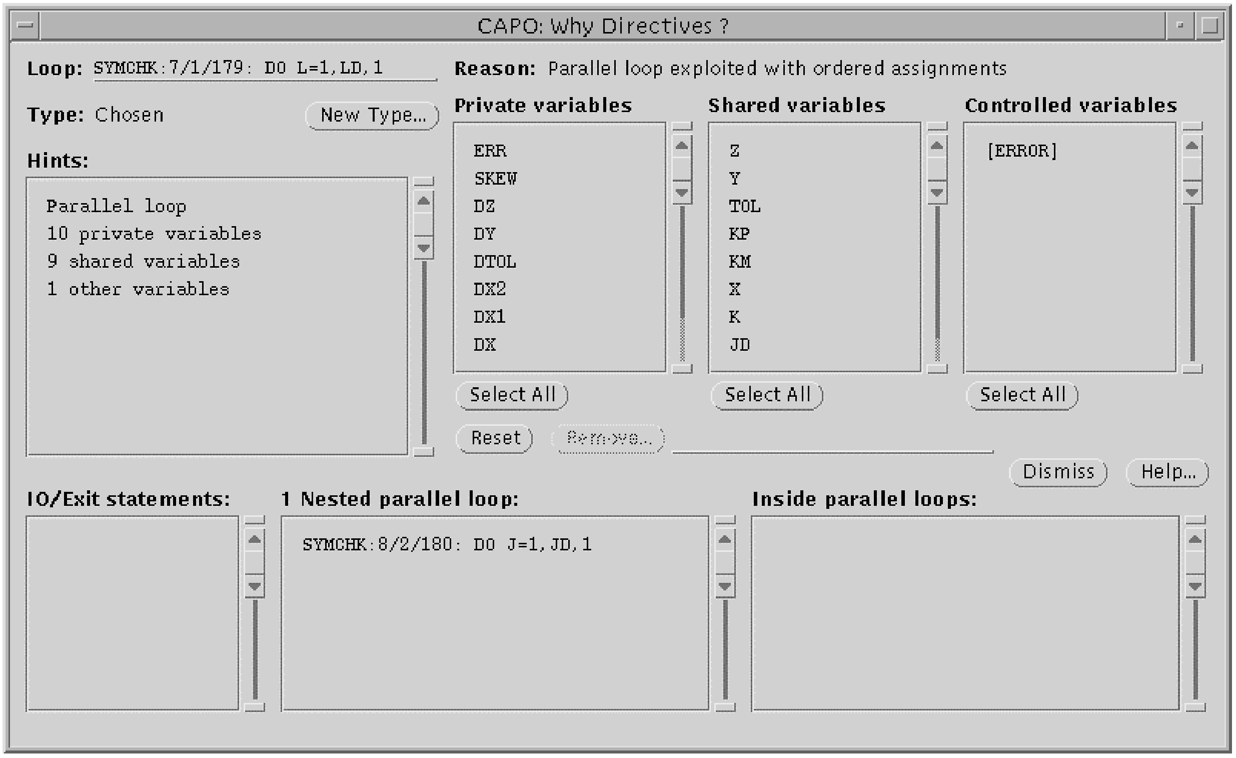
| Private variables | Shared variables | Copyin/out variables |
|---|---|---|
| JM | IBLANK | >QAV |
| JP | SMOO | |
| LM | JPER | |
| LP | IONE | |
| ISMOO | LPER | |
| L | NSMOO | |
| DJJP | LD | |
| DJJ | JD | |

IO/Exit statements:

6 Nested parallel loops:
```
BCCTJK:3/2/106: DO 30  J=JS,JE,1
BCCTJK:5/2/116: DO 40  L=LS,LE,1
ZERO:1/1/10: DO 10  I=1,LEN,1
BCCTJK:8/3/129: DO 110  L=LS,LE,1
BCCTJK:10/3/145: DO 120  J=JS,JE,1
BCCTJK:12/3/161: DO 130  L=LS,LE,1
```

Inside parallel loops:

The above windows is for Chosen parallel loop with Copyin/out variables.


CAPO: Why Directives ?

Loop: SYMCHK:7/1/179: DO L=1,LD,1
Reason: Parallel loop exploited with ordered assignments

Type: Chosen

Hints:
```
Parallel loop
10 private variables
9 shared variables
1 other variables
```

| Private variables | Shared variables | Controlled variables |
|---|---|---|
| ERR | Z | [ERROR] |
| SKEW | Y | |
| DZ | TOL | |
| DY | KP | |
| DTOL | KM | |
| DX2 | X | |
| DX1 | K | |
| DX | JD | |

IO/Exit statements:

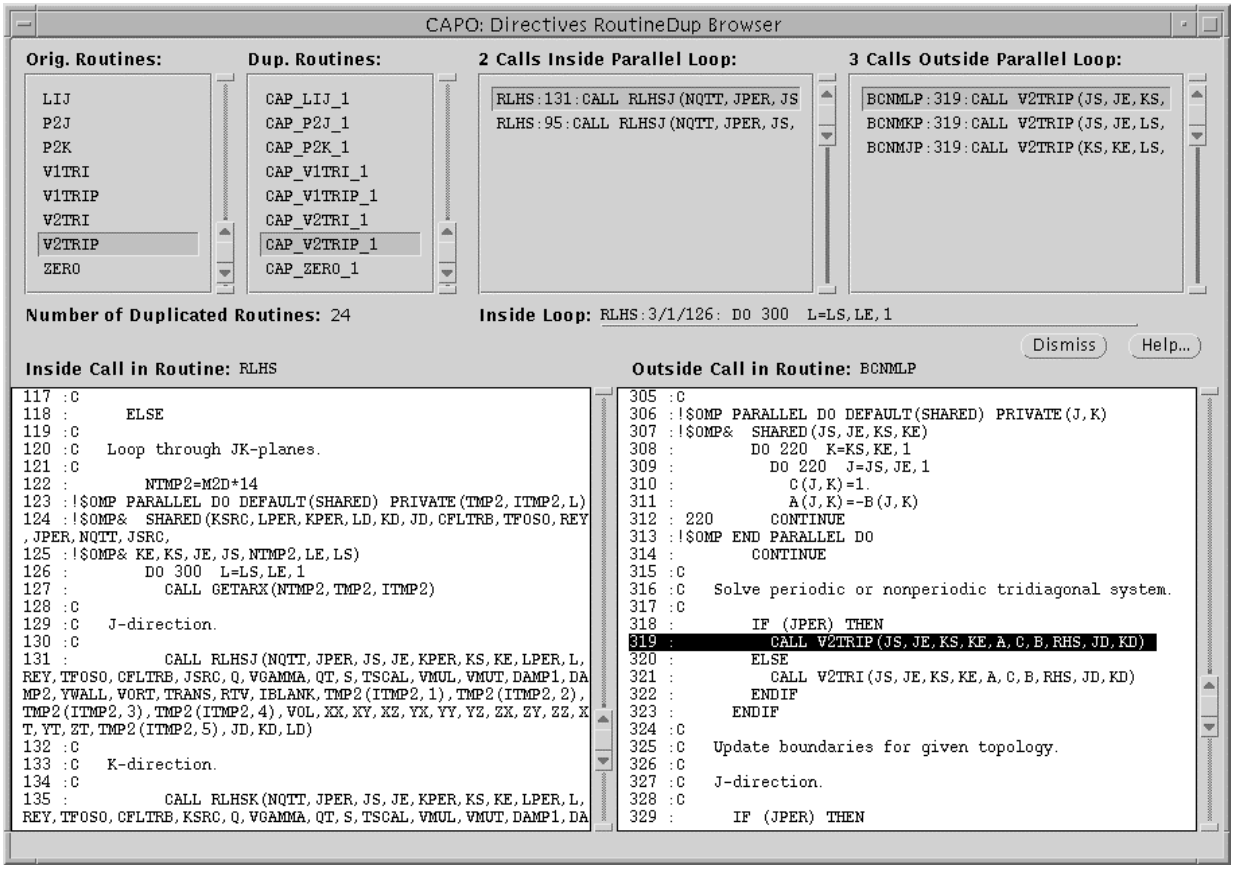
1 Nested parallel loop:
```
SYMCHK:8/2/180: DO J=1,JD,1
```

Inside parallel loops:

The above window is for Chosen parallel loop with Controlled variables.

## A3.5. Routine Duplication Browser

The **RoutineDuplication** window is used for browsing routines that are to be or were duplicated to avoid usage conflict of directives. The window is activated from the RoutDup... button in the Directives browser main window.



**Orig. Routines:**

LIJ
P2J
P2K
V1TRI
V1TRIP
V2TRI
V2TRIP
ZERO

**Dup. Routines:**

CAP_LIJ_1
CAP_P2J_1
CAP_P2K_1
CAP_V1TRI_1
CAP_V1TRIP_1
CAP_V2TRI_1
CAP_V2TRIP_1
CAP_ZERO_1

**2 Calls Inside Parallel Loop:**

```
RLHS:131:CALL RLHSJ(NQTT,JPER,JS
RLHS:95:CALL RLHSJ(NQTT,JPER,JS,
```

**3 Calls Outside Parallel Loop:**

```
BCNMLP:319:CALL V2TRIP(JS,JE,KS,
BCNMKP:319:CALL V2TRIP(JS,JE,LS,
BCNMJP:319:CALL V2TRIP(KS,KE,LS,
```

**Number of Duplicated Routines:** 24

**Inside Loop:** RLHS:3/1/126: DO 300 L=LS,LE,1

**Inside Call in Routine:** RLHS

```
117 :C
118 :      ELSE
119 :C
120 :C   Loop through JK-planes.
121 :C
122 :        NTMP2=M2D*14
123 :!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(TMP2,ITMP2,L)
124 :!$OMP&  SHARED(KSRC,LPER,KPER,LD,KD,JD,CFLTRB,TFOSO,REY
,JPER,NQTT,JSRC,
125 :!$OMP& KE,KS,JE,JS,NTMP2,LE,LS)
126 :        DO 300  L=LS,LE,1
127 :          CALL GETARX(NTMP2,TMP2,ITMP2)
128 :C
129 :C   J-direction.
130 :C
131 :          CALL RLHSJ(NQTT,JPER,JS,JE,KPER,KS,KE,LPER,L,
REY,TFOSO,CFLTRB,JSRC,Q,VGAMMA,QT,S,TSCAL,VMUL,VMUT,DAMP1,DA
MP2,YWALL,VORT,TRANS,RTV,IBLANK,TMP2(ITMP2,1),TMP2(ITMP2,2),
TMP2(ITMP2,3),TMP2(ITMP2,4),VOL,XX,XY,XZ,YX,YY,YZ,ZX,ZY,ZZ,X
T,YT,ZT,TMP2(ITMP2,5),JD,KD,LD)
132 :C
133 :C   K-direction.
134 :C
135 :          CALL RLHSK(NQTT,JPER,JS,JE,KPER,KS,KE,LPER,L,
REY,TFOSO,CFLTRB,KSRC,Q,VGAMMA,QT,S,TSCAL,VMUL,VMUT,DAMP1,DA
```

**Outside Call in Routine:** BCNMLP

```
305 :C
306 :!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(J,K)
307 :!$OMP&  SHARED(JS,JE,KS,KE)
308 :        DO 220  K=KS,KE,1
309 :          DO 220  J=JS,JE,1
310 :            C(J,K)=1.
311 :            A(J,K)=-B(J,K)
312 : 220      CONTINUE
313 :!$OMP END PARALLEL DO
314 :        CONTINUE
315 :C
316 :C   Solve periodic or nonperiodic tridiagonal system.
317 :C
318 :        IF (JPER) THEN
319 :          CALL V2TRIP(JS,JE,KS,KE,A,C,B,RHS,JD,KD)
320 :        ELSE
321 :          CALL V2TRI(JS,JE,KS,KE,A,C,B,RHS,JD,KD)
322 :        ENDIF
323 :      ENDIF
324 :C
325 :C   Update boundaries for given topology.
326 :C
327 :C   J-direction.
328 :C
329 :      IF (JPER) THEN
```

**Orig. Routines** [list]: list of original routines to be duplicated.

**Dup. Routines** [list]: list of duplicated routines. Before code generation, this list will be empty. After code generation, the list is filled with new routines that have one-to-one correspondence to the original routines. The matched (original, duplicated) routine pairs are selected concurrently.

**Number of Duplicated Routines** [numeric]: as it says.

**Calls Inside Parallel Loop** [list]: list of call statements (to a selected original routine) that are inside parallel loop(s).

**Calls Outside Parallel Loop** [list]: list of call statements (to a selected duplicated routine) that are outside any parallel loop.

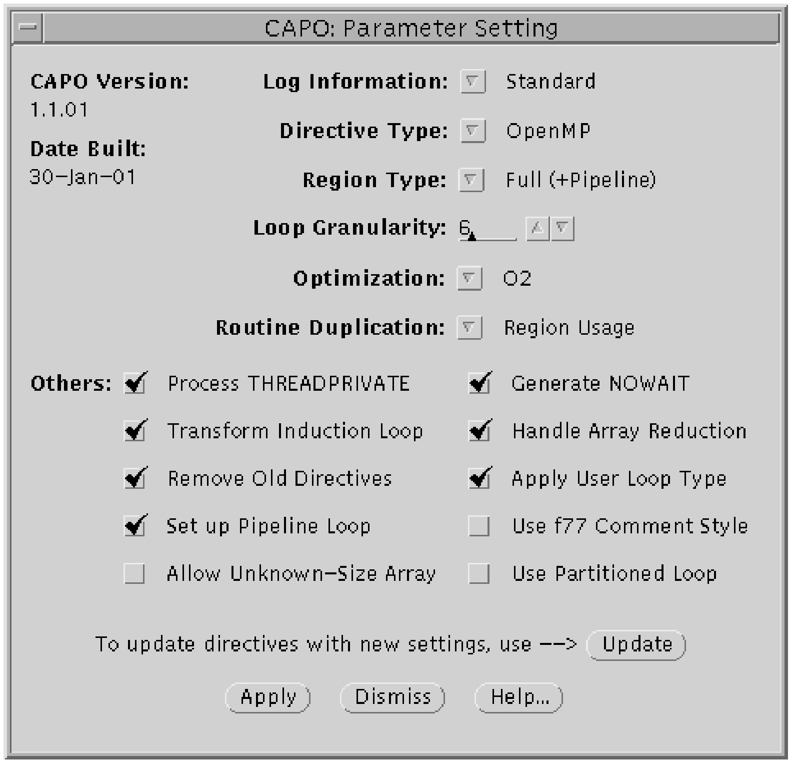
**Inside Loop** [textfield]: the loop that contains the selected call statement to an original routine.

**Inside Call in Routine** [textpane]: the source for the corresponding loop for **Inside Loop**. The textpane is also used for displaying source code for the selected original routine.

**Outside Call in Routine** [textpane]: the source around the selected call statement from the **Call Outside Parallel Loop** list. The textpane is also used for displaying source code for the selected duplicated routine.

## A3.6. Parameter Setting Window

A default setup for the **Parameter Setting** window is displayed on the left. It is launched from either the Setting… button in the Directives main window or the **Edit** → ***Directives Setting…*** in CAPTools main window. The window is used to reset parameters for CAPO to control the directives analysis and generation. The available parameters and their values are described in Section A1.


CAPO: Parameter Setting

CAPO Version: 1.1.01
Date Built: 30-Jan-01

Log Information: Standard
Directive Type: OpenMP
Region Type: Full (+Pipeline)
Loop Granularity: 6
Optimization: O2
Routine Duplication: Region Usage

Others:
- [✓] Process THREADPRIVATE
- [✓] Generate NOWAIT
- [✓] Transform Induction Loop
- [✓] Handle Array Reduction
- [✓] Remove Old Directives
- [✓] Apply User Loop Type
- [✓] Set up Pipeline Loop
- [ ] Use f77 Comment Style
- [ ] Allow Unknown-Size Array
- [ ] Use Partitioned Loop

To update directives with new settings, use --> Update

**CAPO Version**: the current version number of CAPO.

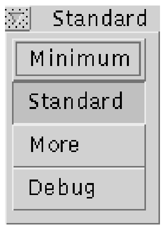
**Date Built**: date on which the current version of CAPO was built.

**Update** [button]: re-performs directives analysis with the current parameters.

**Apply** [button]: applies the current parameter setting without performing the directives analysis.

**Loop Granularity** [numeric]: the minimum number of iterations in a loop for the consideration as a distributed loop. If the number is 0 or if the number of iterations cannot be evaluated, there will be no check on the granularity for the loop.

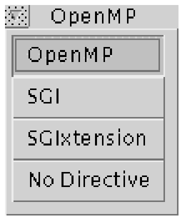
For detailed information on settings and checks, see Section A1.3 and Section A2. The following briefly describes each setting and check box in the window.



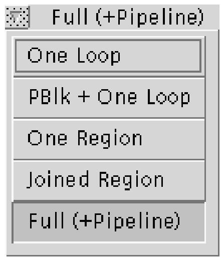
**Log Information** [setting]:

***Minimum*** — minimum log information, such as warning and info messages,
***Standard*** — “Minimum” information plus statistics for loops and regions,
***More*** — “Standard” information plus more detailed loop and region information,
***Debug*** — “More” information plus much more for debugging purpose.

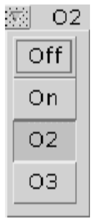
For both ***More*** and ***Debug***, loop and region labels are inserted in the generated source code.



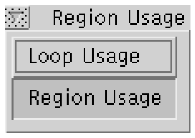
**Directive Type** [setting]:
- ***OpenMP*** — generate OpenMP directives (default),
- ***SGI*** — generate SGI native directives,
- ***SGIxtension***— generate OpenMP directives with SGI extensions,
- ***No Directive***— create source file without directives.



**Region Type** [setting]:
- ***One Loop*** — only one loop for one region,
- ***Pblk + One Loop*** — one pre-block plus one loop for one region,
- ***One Region*** — regions are not joined,
- ***Joined Region*** — regions are joined, no pipeline consideration,
- ***Full Region*** — consider joined region and possible pipeline (default).



**Optimization** [setting]:
- ***Off*** — do not do any optimization,
- ***On*** — try to reduce synchronization at end-of-loop,
- ***O2*** — use logical disprove (slow sometime) for affinity comparison,
- ***O3*** — enable additional optimization (such as automatic loop transformation) before directive insertion.



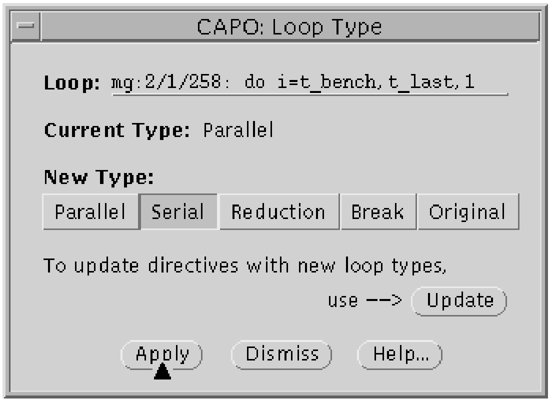
**Routine Duplication** [setting]:
- ***Loop Usage*** — routine duplicated if it is used both inside and outside parallel Loops (no nested parallel region),
- ***Region Usage*** — routine duplicated if it is used inside a parallel loop and inside parallel region but outside parallel loop (allow nested parallel region).

**Others** [checkbox]:

| | |
|---|---|
| ***Process THREADPRIVATE*** | — enable / disable the THREADPRIVATE directive |
| ***Generate NOWAIT*** | — enable / disable the NOWAIT directive |
| ***Transform Induction Loop*** | — enable / disable induction loop treatment |
| ***Handle Array Reduction*** | — enable / disable array reduction |
| ***Remove Old Directives*** | — enable / disable removing old directives |
| ***Apply UserLoop Type*** | — enable / disable applying userloop types |
| ***Setup Pipeline Loop*** | — enable / disable pipeline loop |
| ***Use f77 Comment Style*** | — use f77 (not checked) or f90 (checked) comment style |
| ***Allow Unknown-Size Array*** | — enable / disable the use unknown-size array in PRIVATE |
| ***Use Partitioned Loop*** | — enable / disable partitioned loop for directives |

## A3.7. User Loop Type Window

The loop type window is used to redefine a loop type manually. It is displayed for a selected loop by clicking on the New Type button in the **WhyDirectives** window.



Loop: mg:2/1/258: do i=t_bench,t_last,1

Current Type: Parallel

New Type:

To update directives with new loop types,
use --> Update

**Loop** [textfield]: print of the selected loop.

**Current Type** [textfield]: the current loop type.

**Update** [button]: saves the newly defined loop type to the `userloop.par` file and re-performs the directives analysis with the new setting.

**Apply** [button]: saves the newly defined loop type to the `userloop.par` file but does not re-perform the directives analysis.

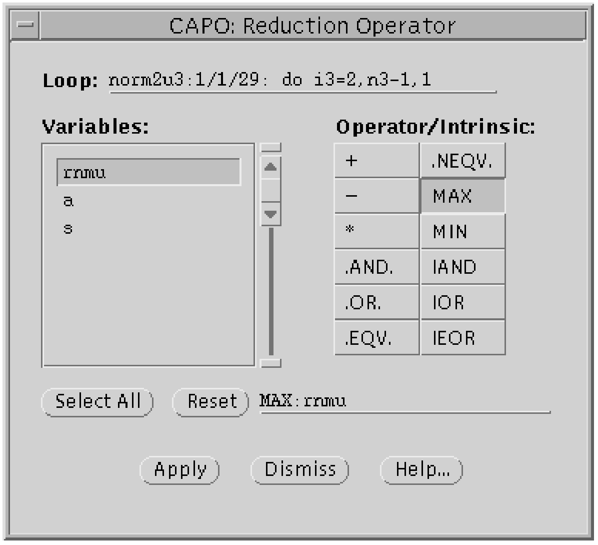
**New Type** [setting]: one of the selectable types.

***Parallel*** – a parallel loop
***Serial*** – a serial loop
***Reduction*** – a parallel loop with reduction. The Reduction setting may activate an additional dialog box: **Reduction Operator** (See Section A3.8).
***Break*** – a serial loop excluded from any parallel region
***Original*** – the type originally set by CAPO.

An un-selectable type indicates a type that cannot be converted to from the current type.

## A3.8. Reduction Operator Dialog

This is a dialog box to select an option (or options) for user-defined reduction loop type. The option specifies reduction operators/intrinsics and variables as part of the entry in the userloop.par file. See Section A1.3 for the description of the userloop.par file.



Loop: norm2u3:1/1/29: do i3=2,n3-1,1

rnmu
a
s

MAX:rnmu

The dialog box is activated only if the *Reduction* setting in the LoopType window is selected and there exist potential reducible variables detected in the loop by CAPO.

**Loop** [textfield]: print of the selected loop.

**Variables** [list]: list of variables that can potentially be selected as reduction variables, *selectable*.

**Operator/Intrinsic** [setting]: one of the defined reduction operators or intrinsic functions.

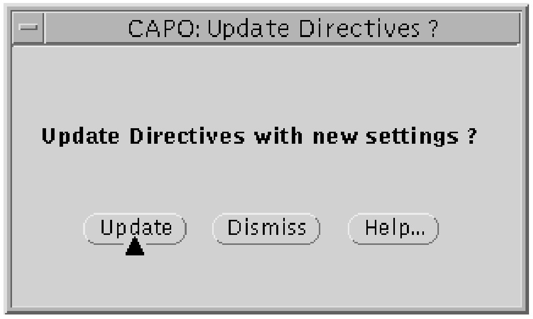
**Select All** [button]: selects all the variables in the variable list.

**Reset** [setting]: resets any previous selection. The textfield on the right lists the selected **Operator/Intrinsic** and variables.

**Apply** [button]: creates an [operator/intrinsic:variables] combination and add to the option list for the currently selected loop. The option and user-loop type are only stored to the userloop.par file when the Apply or Update button in the **LoopType** window is pressed.

## A3.9. Updating Directives Dialog

This is a dialog box for confirming the analysis of directives with new settings. It is popped up after the Update button in the **Directives browser** main window is pushed.

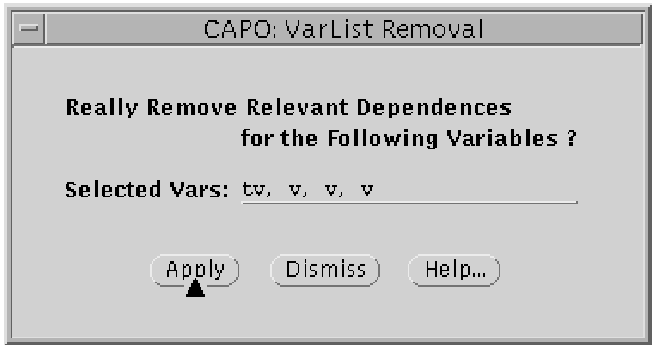


Update Directives with new settings ?

**Update** [button]: performs the directives analysis, including loop and region level analysis, without generating directives. The dialog will be disabled after the OpenMP directives code is generated.

## A3.10. Variable Removal Confirmation Dialog

The dialog is used for confirming the removal of dependences for selected variables and types. The variables and types are determined in the **WhyDirectives** window and the dialog box is activated by pushing the Remove button. This box provides a shortcut to the **DepGraph** for quickly deleting false dependences.



Really Remove Relevant Dependences for the Following Variables ?

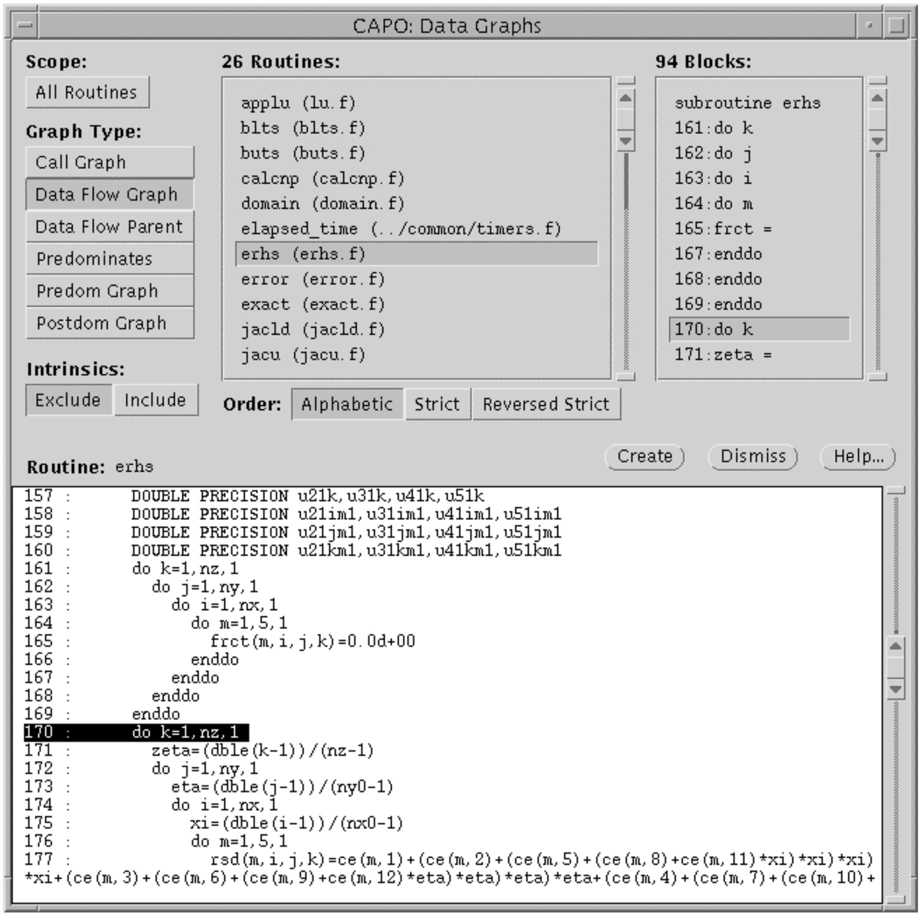
Selected Vars: tv, v, v, v

**Selected Vars** [textfield]: list of selected variables from the **WhyDirectives** window (Section A3.4). A variable listed multiple times indicates it is selected from multiple variable lists in the **WhyDirectives** window.

**Apply** [button]: applies the removal action.

## A3.11. Data Graph Window

The **Data Graph** window is used to create graphs for development purpose. It may have little use to a typical user, but is included for reference. The window is activated from ***View→Data Graph*** in the CAPTools main window. If the “Data Graph” menu item is not present, try to start CAPO with the [-capodg] option.



```
applu (lu.f)
blts (blts.f)
buts (buts.f)
calcnp (calcnp.f)
domain (domain.f)
elapsed_time (../common/timers.f)
erhs (erhs.f)
error (error.f)
exact (exact.f)
jacld (jacld.f)
jacu (jacu.f)
```

```
subroutine erhs
161:do k
162:do j
163:do i
164:do m
165:frct =
167:enddo
168:enddo
169:enddo
170:do k
171:zeta =
```

Routine: erhs

```
157 :       DOUBLE PRECISION u21k,u31k,u41k,u51k
158 :       DOUBLE PRECISION u21im1,u31im1,u41im1,u51im1
159 :       DOUBLE PRECISION u21jm1,u31jm1,u41jm1,u51jm1
160 :       DOUBLE PRECISION u21km1,u31km1,u41km1,u51km1
161 :       do k=1,nz,1
162 :         do j=1,ny,1
163 :           do i=1,nx,1
164 :             do m=1,5,1
165 :               frct(m,i,j,k)=0.0d+00
166 :             enddo
167 :           enddo
168 :         enddo
169 :       enddo
170 :       do k=1,nz,1
171 :         zeta=(dble(k-1))/(nz-1)
172 :         do j=1,ny,1
173 :           eta=(dble(j-1))/(ny0-1)
174 :           do i=1,nx,1
175 :             xi=(dble(i-1))/(nx0-1)
176 :             do m=1,5,1
177 :               rsd(m,i,j,k)=ce(m,1)+(ce(m,2)+(ce(m,5)+(ce(m,8)+ce(m,11)*xi)*xi)
*xi+(ce(m,3)+(ce(m,6)+(ce(m,9)+ce(m,12)*eta)*eta)*eta)*eta+(ce(m,4)+(ce(m,7)+(ce(m,10)+
```

**Scope** [setting]: defines the scope of the routine list.

**Graph Type** [setting]: chooses from one of the predefined graph types.

**Intrinsics** [setting]: excludes or includes intrinsic functions in the routine list and in the graph.

**Routines** [list]: list of routines (name of the file containing a routine).

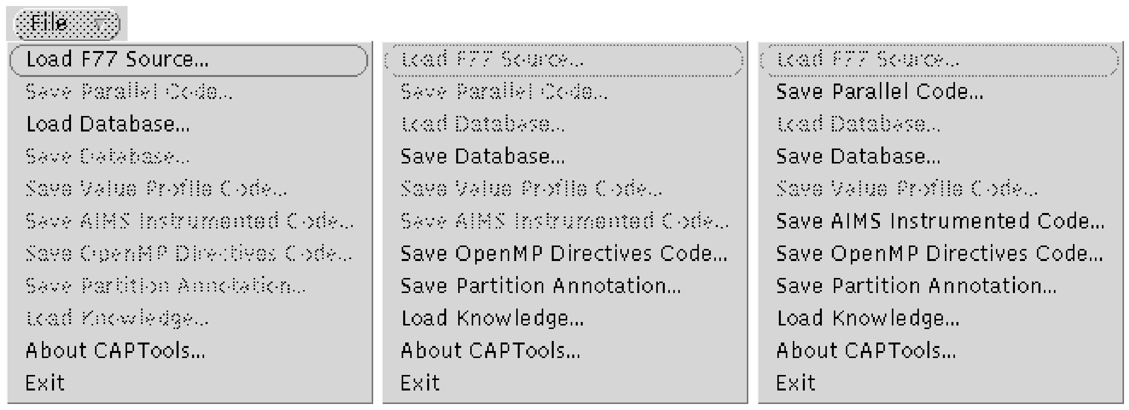
**Order** [setting]: defines the way routines are listed (Alphabetic, Strict, Reversed Strict).

**Blocks** [list]: list of basic program blocks in the selected routine.

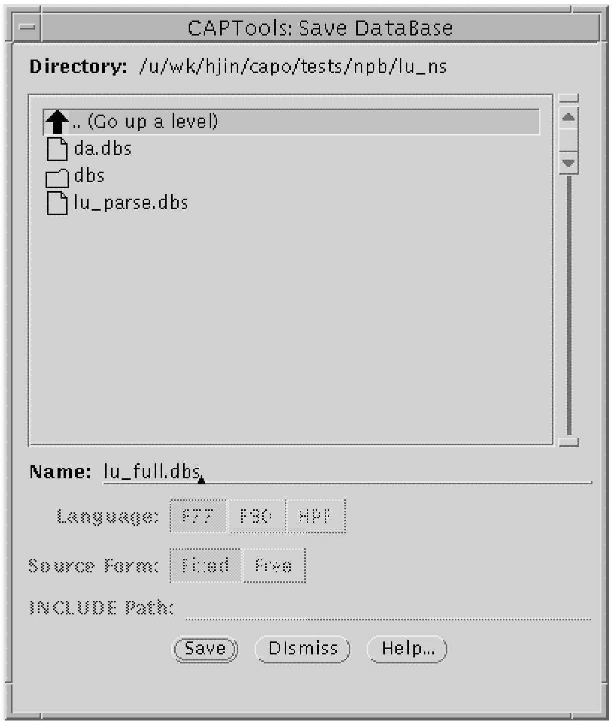
**Create** [button]: creates a graph for the selected routine and / or block (currently *xvcg* is used to display the graph).

## A3.12. Hookups to CAPTools

For CAPO-enabled CAPTools, additional items are added to the **File** (***Save OpenMP Directives Code***), **View** (***Directives***) and **Edit** (***Directives Setting***) menus in the CAPTools main window (Section A3.1). The menu items that are relevant to directives generation are summarized here.


File
Load F77 Source...
Save Parallel Code...
Load Database...
Save Database...
Save Value Profile Code...
Save AIMS Instrumented Code...
Save OpenMP Directives Code...
Save Partition Annotation...
Load Knowledge...
About CAPTools...
Exit

Before source is loaded After source is loaded After communication is generated



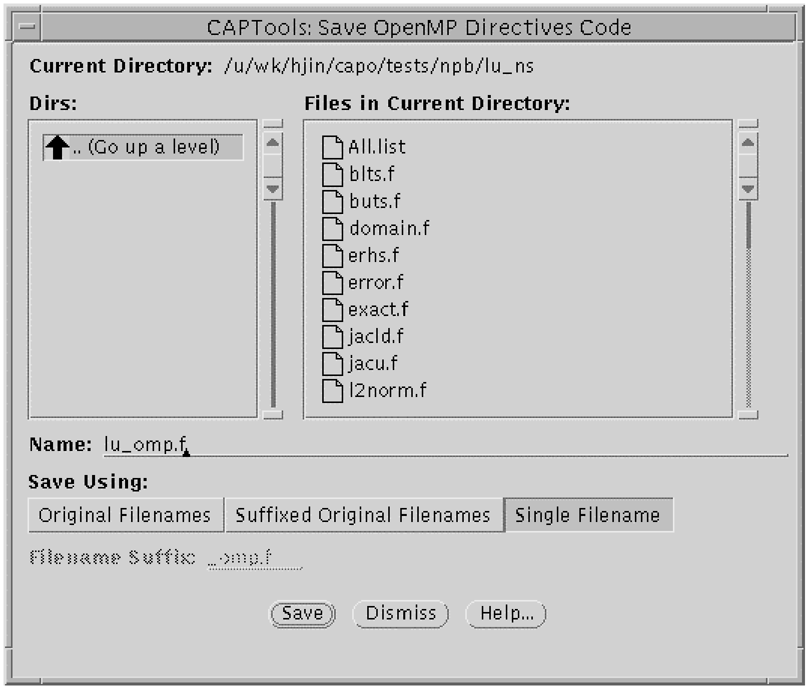
The **Save Database** dialog box.

The **File** menu:

**Load F77 Source** [entry]: loads Fortran 77 source (.f or .list file).

**Load Database** [entry]: loads a previously saved database (.dbs file).

**Save Database** [entry]: saves the current analysis result to a database. As of CAPO Version 1.1, the directives analysis result is not yet saved to the database. But the inserted directives are saved.


CAPTools: Save OpenMP Directives Code

Current Directory: /u/wk/hjin/capo/tests/npb/lu_ns

Dirs:
.. (Go up a level)

Files in Current Directory:
All.list
blts.f
buts.f
domain.f
erhs.f
error.f
exact.f
jacld.f
jacu.f
l2norm.f

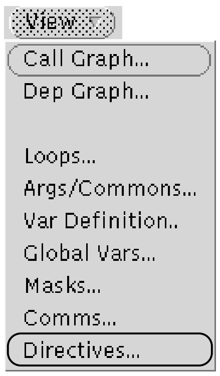
Name: lu_omp.f

Save Using: Original Filenames | Suffixed Original Filenames | [✓ Single Filename]

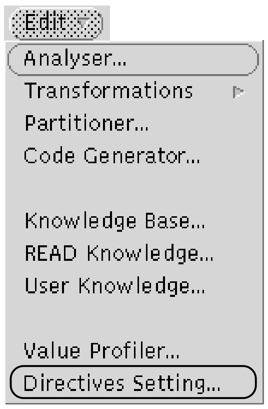
Filename Suffix: _omp.f

**Save OpenMP Directives Code** [entry]: performs the directives analysis if it has not been done and generates OpenMP directives. The code can be saved to multiple files or to a single file.

The **Save OpenMP Directives Code** dialog box.

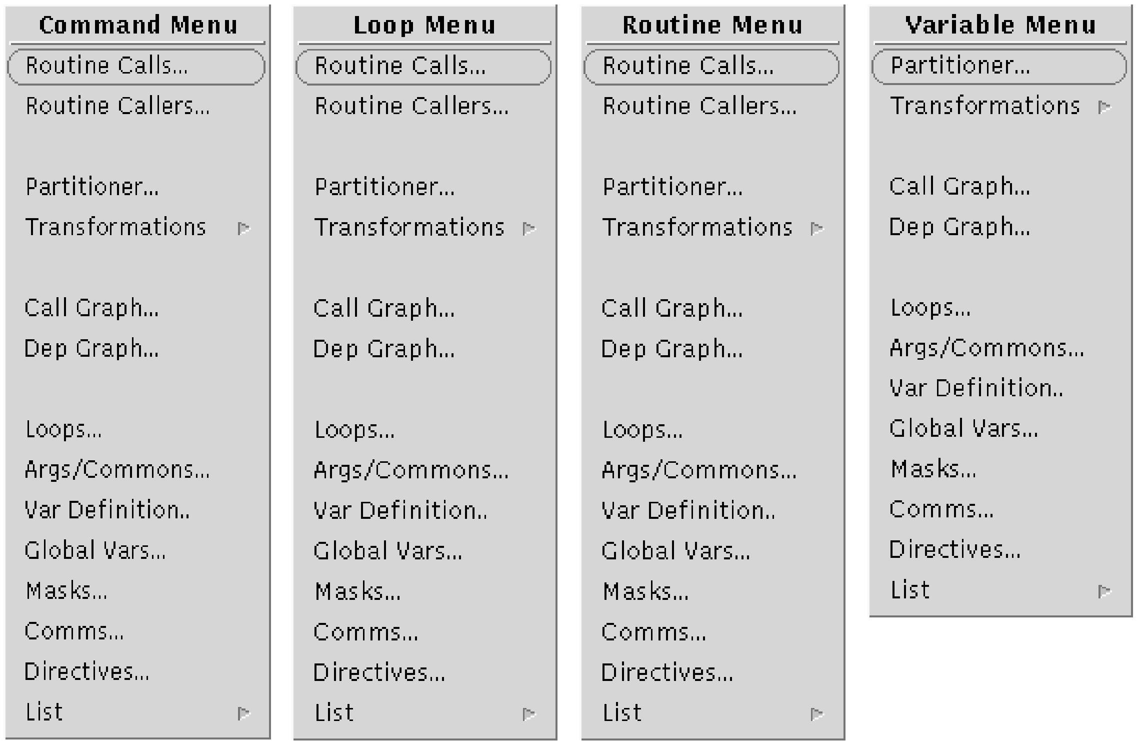


The **View** menu:
**Directives** [entry]: activates the Directives browser, which performs the directives analysis (if not yet done) and presents information on directives.



The **Edit** menu:
**Directives Setting** [entry]: activates the Setting dialog box as given in Section A3.6. It can be used to set up parameters for CAPO before the the directives analysis is performed.

The following popup menus are hookups to various tools from selected lists or items in a GUI window, usually activated with a right-mouse-button click.



**Command Menu** [popup]: for a selected statement.

**Loop Menu** [popup]: for a selected loop.

**Routine Menu** [popup]: for a selected routine.

**Variable Menu** [popup]: for a selected variable.

# A4. CAPO Command Interface

The command interface for CAPO is available in Version 1.1 and works closely with the CAPTools command interface. It provides a way to access the functionality of GUI components without starting the GUI. It serves as a means to record actions (to a log file) as a result of any user GUI activities so that these actions can be played back later. The commands in the command interface are usually recorded to a log file or a command file with

```
capo –logfile capo_run.cmd
```

and played back with

```
capo [-batch] capo_run.cmd.
```

The second line with the `[-batch]` option can be used to start a CAPO session in a batch mode.

The command interface for CAPO is different from the command-line version of CAPO, which takes simply the database as input and creates the Fortran output:

```
capo –capoc [-options] database.dbs output.f.
```

This stand-alone version is mostly for testing purpose. The command interface is the preferred method.

## A4.1. Commands for the Command Interface

CAPO commands start with the keyword "`capo`" to distinguish them from CAPTools commands.

**Main commands:**

`load <file.dbs>`
- Load database file

`capo version 1`
- Define CAPO command version

`capo removedep <routine> <variable> <loop_number> <dtype> <fc> [<drout>]`
- Remove loop-related data dependences
  * `routine` - routine name
  * `variable` - relevant variable in the routine
  * `loop_number` - loop to be considered
  * `dtype` - dependence type: 1 for loop-carried TRUE dependences
    2 for TRUE dependences from outside loop
    3 for loop-carried ANTI dependences
    4 for loop-carried OUTPUT dependences
  * `fc` - `1` father list, `2` child list, `0` both lists
  * `[drout]` – optional field to define routine in which the variable is actually declared (if it is different from <`routine`>)

`capo update [0/1]`
- Perform directives analysis with the new setting
  `'0'` for initial analysis, `'1'` for new update

`capo passtwo`

- Re-perform the pass-two analysis

`capo generate [<file.f>]`
- Generate OpenMP directives. `<file.f>` is used to define the logfile name, i.e. <`file.log`>. If <`file.f`> is not given, "`capo-info.log`" is assumed for the logfile name.

`save source <file.f> 3 0`
- Save source code to `<file.f>`
  `'3'` indicates a single file

("`load`" and "`save`" are two CAPTools commands. See A-4.2 for details.)

**Parameter setting commands:**

`capo set log-file on/off/stdout`
- Turn on/off information logging, default is on

`capo set log-file-name <filename>`
- Define log filename, default is `"capo-info.log"`

`capo set log-info minl/std/more/debug`
- Select log information type, default is `std`

`capo set loop-granularity <value>`
- Set loop granularity threshold, default value = `6`

`capo set directive-type omp/sgi/sgix/no`
- Select directive type, default is `omp`

`capo set optimize-type off/o1/o2/o3`
- Set the optimization type, default is `o2`

`capo set user-loop-file <filename>`
- Define user loop file, default is `"userloop.par"`

`capo set directive-clear off/on/<filename>`
- Turn on/off old directive clearing, default is `on`
  A `<filename>` is used to define a new set of directives

`capo set comment-type f77/f90`
- Set the comment type for directive, default is `f90`

`capo set use-parti-loop yes/no`
- Allow the partitioned loop for directive, default is `no`

`capo set rdup-type loop/region`
- Select the routine duplication type, default is `region`

`capo set allow-pio no/incall/write/noread/any`
- Allow parallel I/O type, default is `no`

**Setting commands for debugging purpose:**

`capo set mflag <mflag_value>`
- Define the module flag
  `<mflag_value>` can be `<number>/<m1:m2..>` with `[+-]` sign

`capo set region-type default/loop/bloop/one/join/full`
- Set a region type, default is `full`

`capo set tpriv-directive on/off`
- Turn on/off the generation of `THREADPRIVATE`, default is `on`

`capo set allow-unksize true/false`
- Allow the use of unknown-size private variables, default is `false`

`capo set have-pipeloop true/false`
- Generate pipeline loop, default is `true`

`capo set have-induc true/false`
- Treat parallel induction loop, default is `true`

`capo set have-arreduc true/false`
- Treat array reduction, default is `true`

`capo set have-nowait true/false`
- Generate the `NOWAIT` directive, default is `true`

`capo set apply-userloop yes/no`
- Apply user defined loop types, default is `yes`

`capo set apply-dirclear yes/no`
- Apply old directive clearing, default is `yes`

## A4.2. Other CAPTools Commands Useful for CAPO

`version 2`
- Define CAPTools command version

`load <file.f/file.list/file.dbs>`
- Load source / database file

`save database <file.dbs>`
- Save to database

`save source <dir/suffix/file.f> <1/2/3> 0`
- Save source with type `1, 2 or 3`
  Type `1`: Save to original files, `<dir>` is required for directory name
  Type `2`: Save to original files with `<suffix>`, `<dir/suffix>` required
  Type `3`: Save to a single file with file name `<file.f>`

```
set exact on
set scaler on
set knowledge on
set disproofs on
set interprocedural on
set logic on
```
- Settings for the analysis power

`add read knowledge applu:76:((nx-5 .GT. 0))`
- Define read user knowledge

`analyse`
- Perform dependence analysis

## A4.3. An Example of "capo_run.cmd"

```
version 2
load applu_full.dbs
capo version 1
capo set log-file-name applu_omp.log
capo update 0
capo removedep setbv u 1 4 0
capo removedep setbv u 3 4 0
capo removedep setbv u 5 4 0
capo update 1
capo generate
save source applu_omp.f 3 0
```

To use the command file, do `"capo -batch capo_run.cmd"`.