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 1

1. General Information 2
2. Computer-Aided Parallelization Process 4
3. Producing Parallel Code with CAPO 10
4. Interacting with the Directives Browser 13
5. Other Features 18
6. Case Studies 21

References 25

Tutorials 26

Tutorial 1. A Simple Jacobi Code 27
Tutorial 2. NPB LU-hp Removing False Dependences 28
Tutorial 3. NPB MG User-Defined Loop Type 32
Tutorial 4. A CFD Application TEAMKE1 36
Tutorial 5. Mix of Message-Passing and OpenMP 44

Appendix 53

A1. Parameters for CAPO 54
A2. Messages and Symbols in the Log File 59
A3. CAPO Graphical User Interface 68
A4. CAPO Command Interface 87

USING CAPO

Contents

1. General Information 2

1.1. What is CAPO 2

1.2. Distribution and Contact Information 2

1.3. Installation and Execution 3

1.4. How to Use This Manual 3

2. Computer-Aided Parallelization Process 4

2.1. The OpenMP Programming Model 4

2.2. CAPTools 5

2.3. Generating OpenMP Directives 5

2.4. Data Dependence 6

3. Producing Parallel Code with CAPO 10

3.1. Prepare Serial FORTRAN Codes 10

3.2. Make Dependence Analysis 10

3.3. Inspect Loops and Optimize Directive Generation 11

3.4. Generate Parallel Code with Directives 12

3.5. Inspect the Generated Code and the Log Information 12

3.6. Compile and Run the Parallel Code 12

4. Interacting with the Directives Browser 13

4.1. Loop Classification 14

4.2. Browsing Different Types of Loops 15

4.3. Enforcing New Loop Type 17

4.4. Routine Duplication 17

5. Other Features 18

5.1. CAPO Parameters and Log Information 18

5.2. Automatic Code Transformation and Optimization 18

5.3. Command Interface and the Batch Mode 19

5.4. Parallel I/O 19

5.5. Mix of Message Passing and OpenMP 19

6. Case Studies 21

6.1. The NAS Parallel Benchmarks 21

6.2. ARC3D 22

6.3. OVERFLOW 23

References 25

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
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.

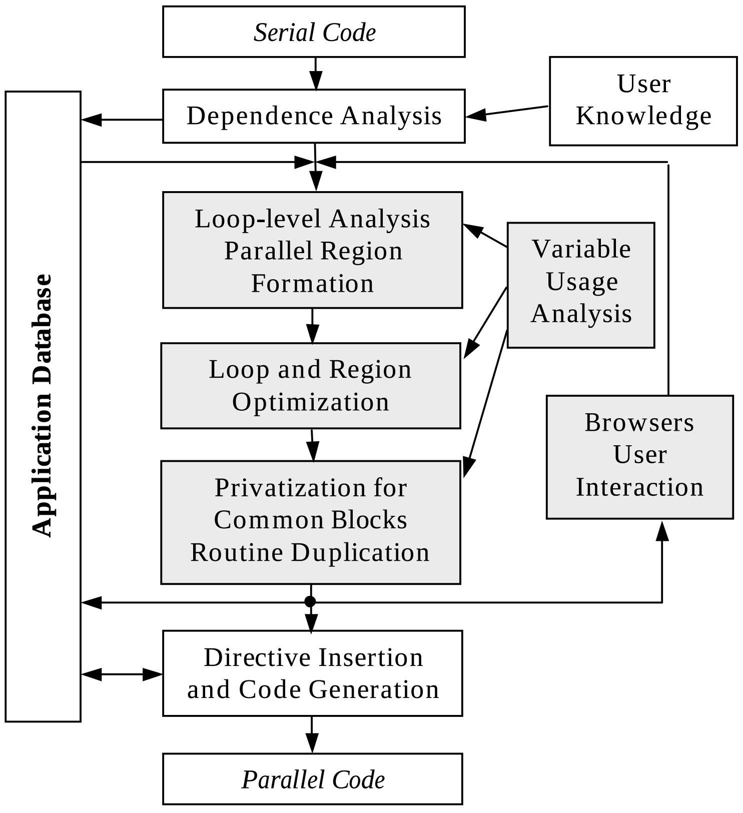

Serial Code
Dependence Analysis
User Knowledge
Application Database
Loop-level Analysis Parallel Region Formation
Variable Usage Analysis
Loop and Region Optimization
Browsers User Interaction
Privatization for Common Blocks Routine Duplication
Directive Insertion and Code Generation
Parallel Code

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.

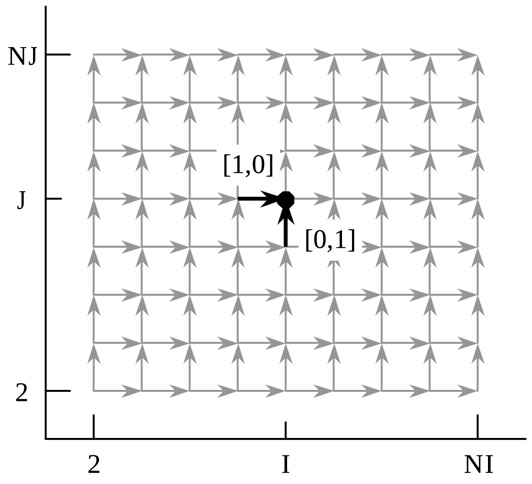


Figure 2: Dependence vectors formed in the loop iteration space.

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.

```
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".

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.

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 512MB RAM, 512KB Cache	Linux	10.5 mins
Intel PII, 300MHz 512MB RAM, 512KB Cache	Linux	16.4 mins
Sun UltraSparcII, 360 MHz 1GB RAM, 16KB L1, 4MB L2	Solaris	15.0 mins
Sun UltraSparcII, 300 MHz 2GB RAM, 16KB L1, 4MB L2	Solaris	17.6 mins
SGI R5K, 150 MHz 128MB RAM, 32KB L1	IRIX	71.4 mins
SGI R10K, 195MHz 512MB RAM, 32KB L1, 1MB L2	IRIX	26.4 mins
SGI R12K, 300MHz 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.

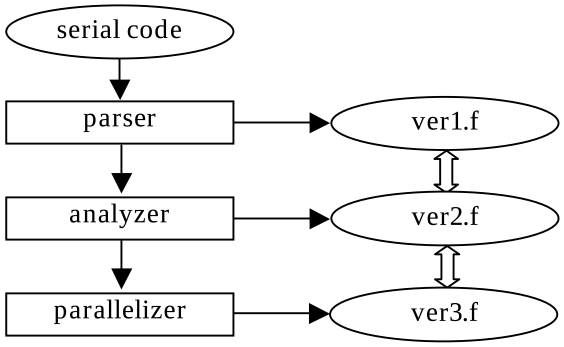


Figure 4: Compare source codes at different stages.

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.

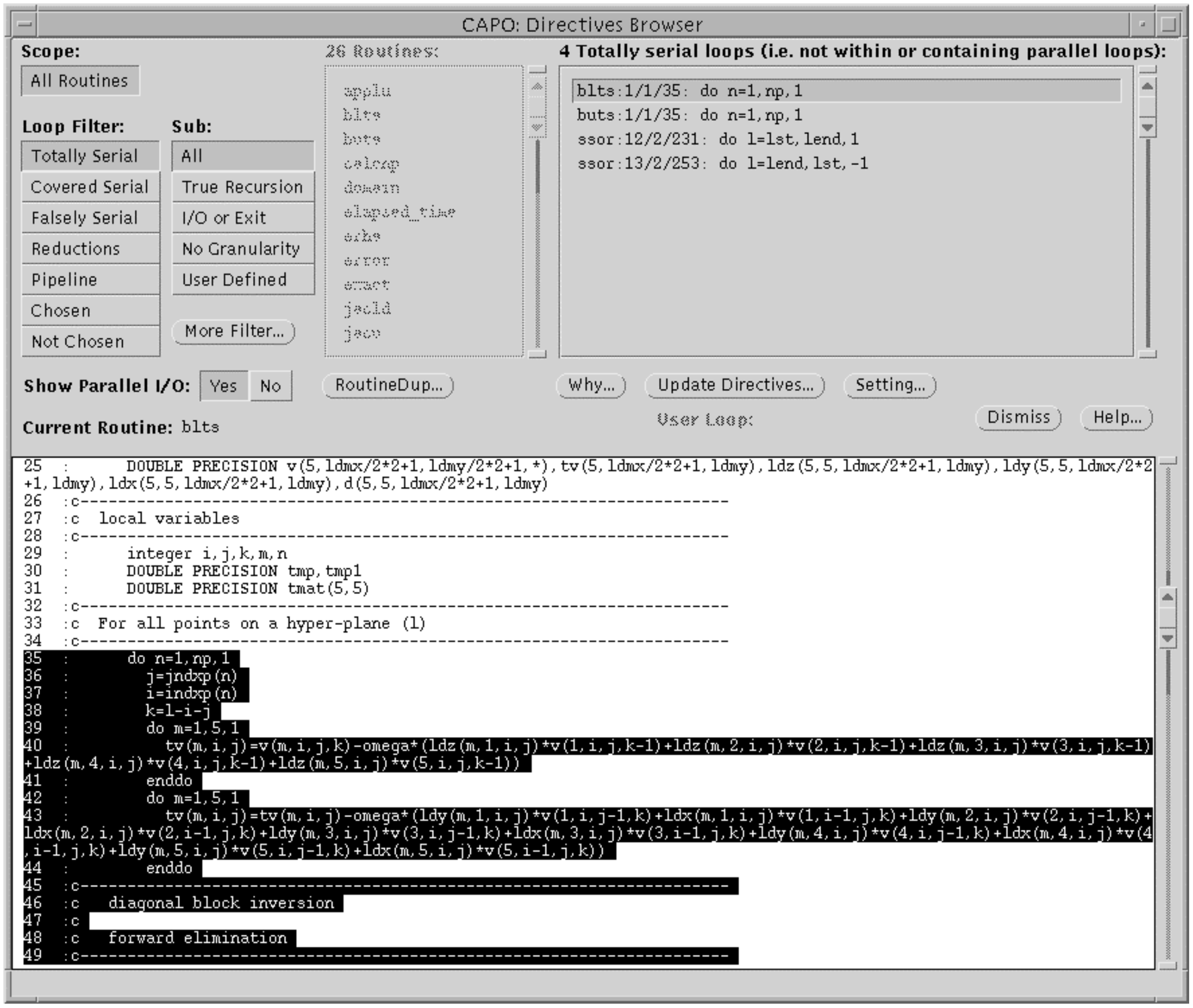
```
% f77 -o a.out -mp -O parallelcode.f
```

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.



```
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
```

```
            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.

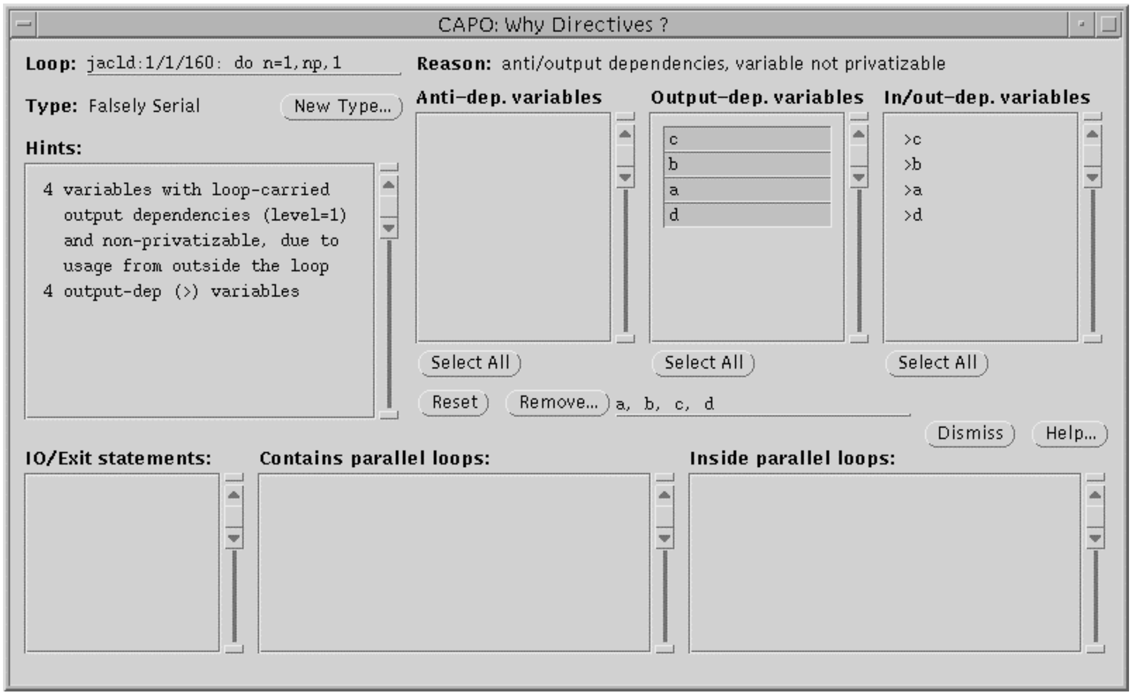
4.2. Browsing Different Types of Loops

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 (≥ 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.

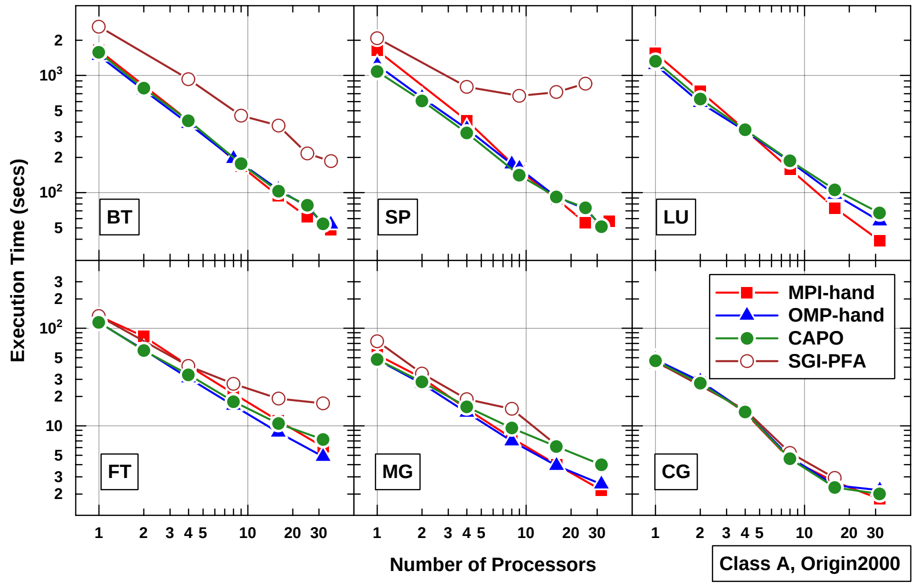


Figure 8: Comparison of the OpenMP NPB generated by CAPO with other parallel versions: MPI from NPB2.3, OpenMP by hand, and SGI-PFA.

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

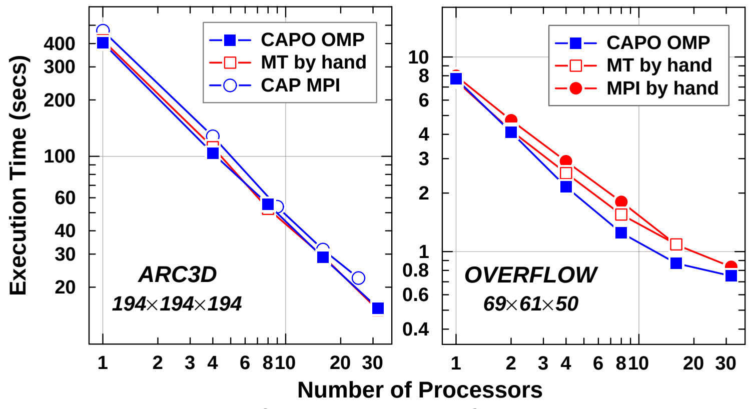


Figure 9: Comparison of execution times of CAPO generated parallel codes with hand-coded parallel versions for two CFD applications: ARC3D on the left and OVERFLOW on the right.

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.