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:

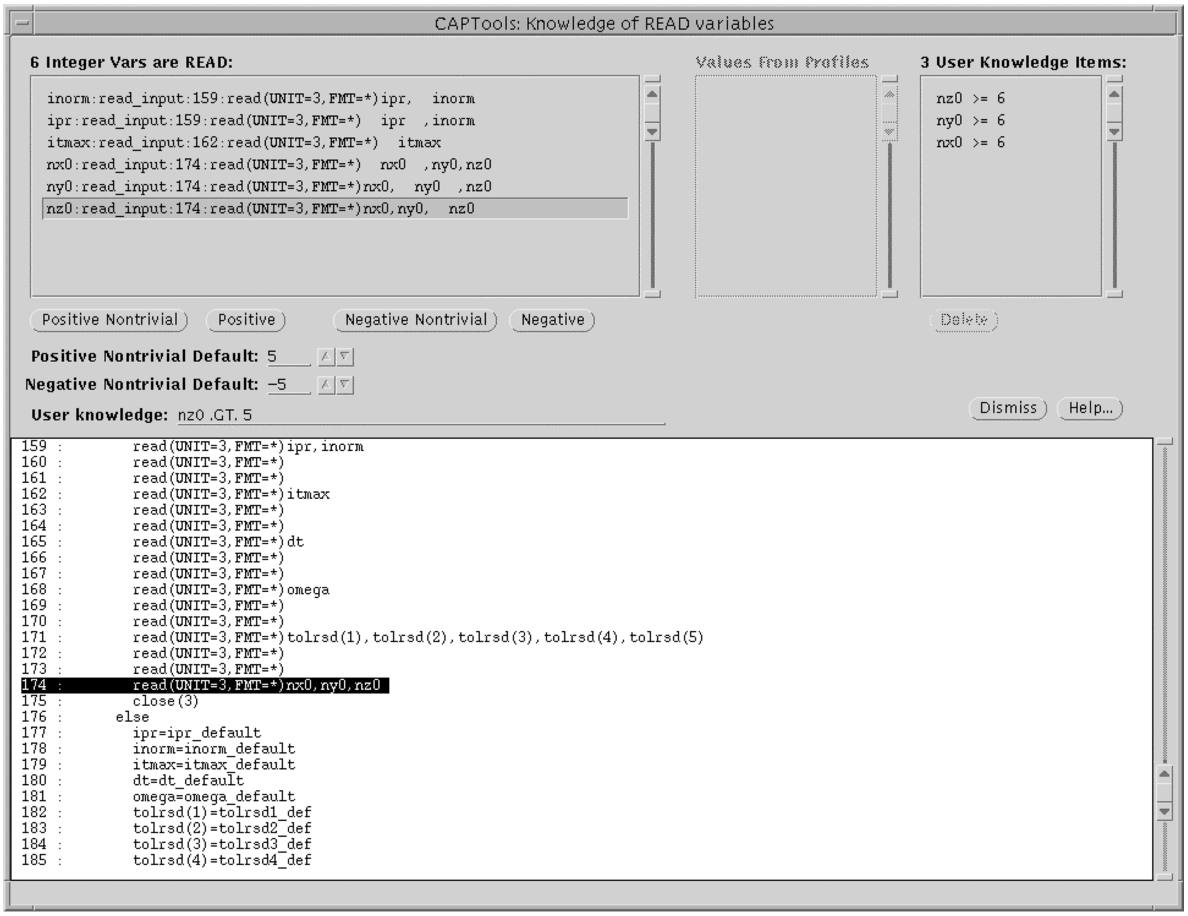
1. **Load file and enter user knowledge**. Click ***Load F77 Source*** in the **File** menu. Select All.list and click the Load button. Select ***READ Knowledge*** from the **Edit** menu. In the **READ Knowledge** window, select variable `nx0` and click Positive Nontrivial, see Figure T2-1 on next page. 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 Full option and click 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 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 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 Totally Serial loop filter and the 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 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 Remove button and click the 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

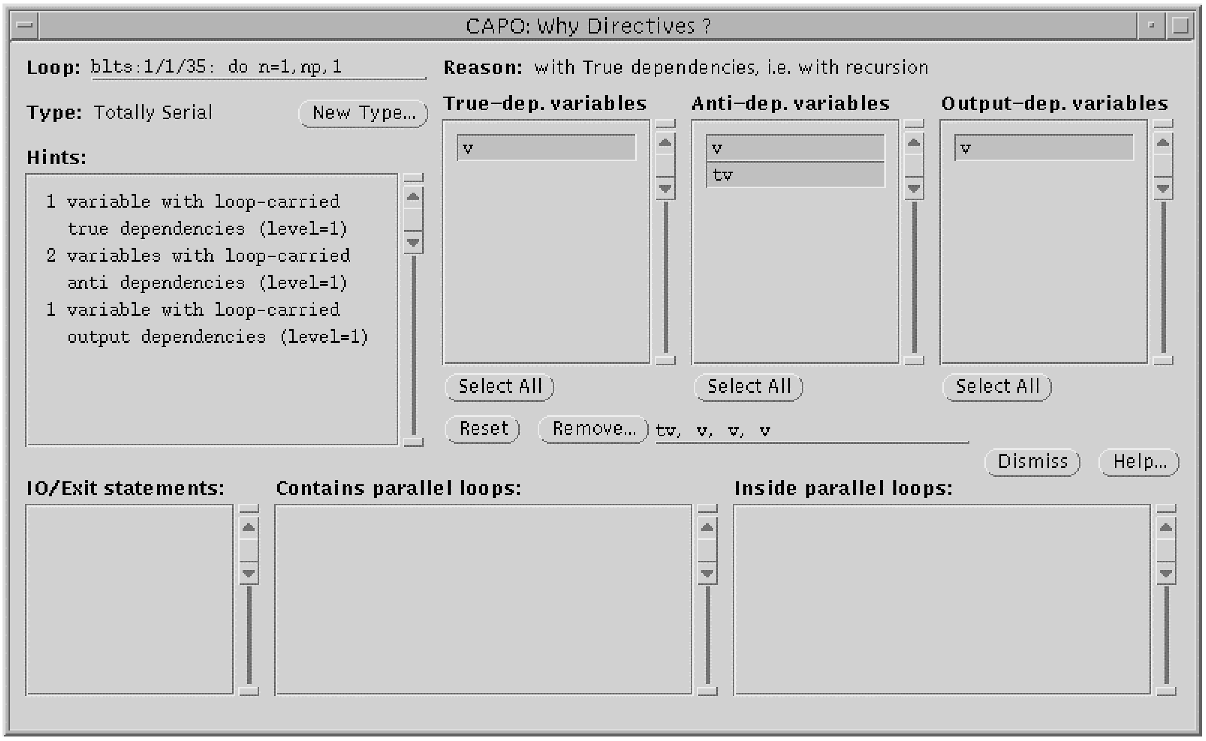
```
159 :         read(UNIT=3,FMT=*)ipr,inorm
160 :         read(UNIT=3,FMT=*)
161 :         read(UNIT=3,FMT=*)
162 :         read(UNIT=3,FMT=*)itmax
163 :         read(UNIT=3,FMT=*)
164 :         read(UNIT=3,FMT=*)
165 :         read(UNIT=3,FMT=*)dt
166 :         read(UNIT=3,FMT=*)
167 :         read(UNIT=3,FMT=*)
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.



Loop: blts:1/1/35: do n=1,np,1

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

Type: Totally Serial

```
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)
```

True-dep. variables: v

Anti-dep. variables: v, tv

Output-dep. variables: v

tv, v, v, v

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

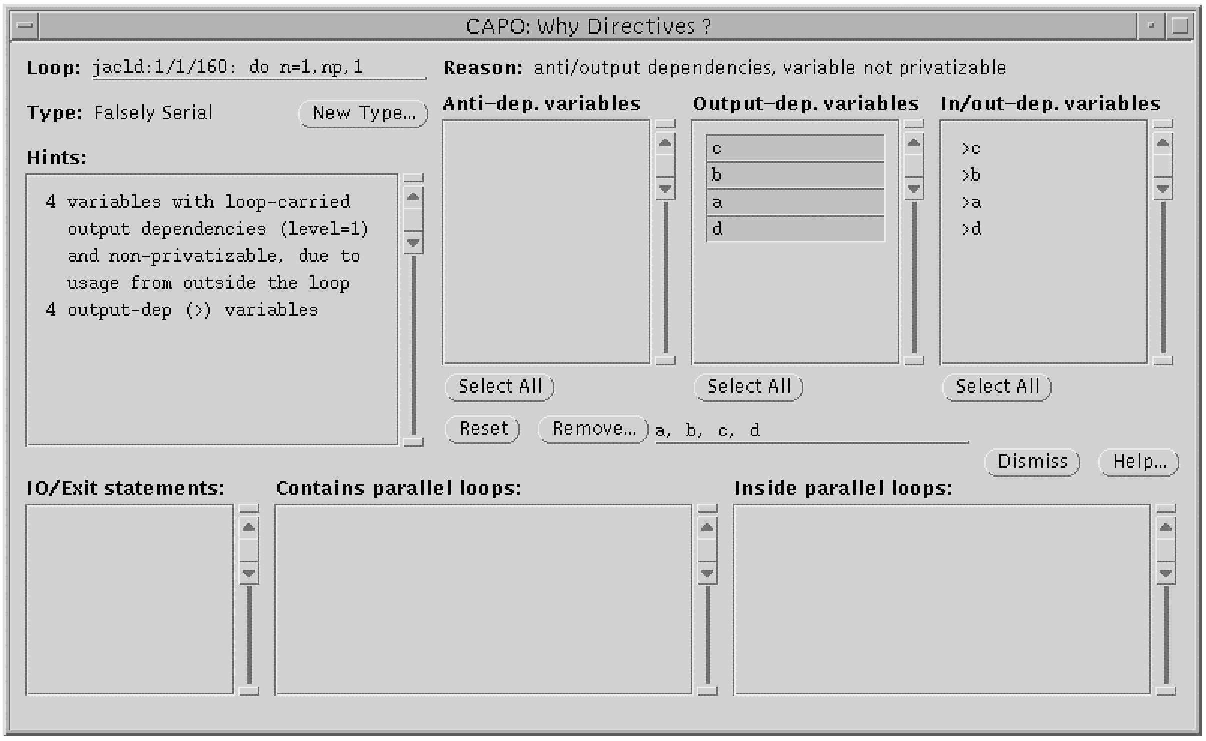
```
      % setenv OMP_NUM_THREADS 4
      % ./lu_hp_omp
```

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
```



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
```

Output-dep. variables: c, b, a, d

In/out-dep. variables: >c, >b, >a, >d

a, b, c, d

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 Load button. In the **Analyser** window select the Full option and click 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 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 All Routines and loop filter Totally Serial and sub-filter True Recursion. Select loop: `norm2u3:1/1/27: do i3=2,n3-1` and click the 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 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:

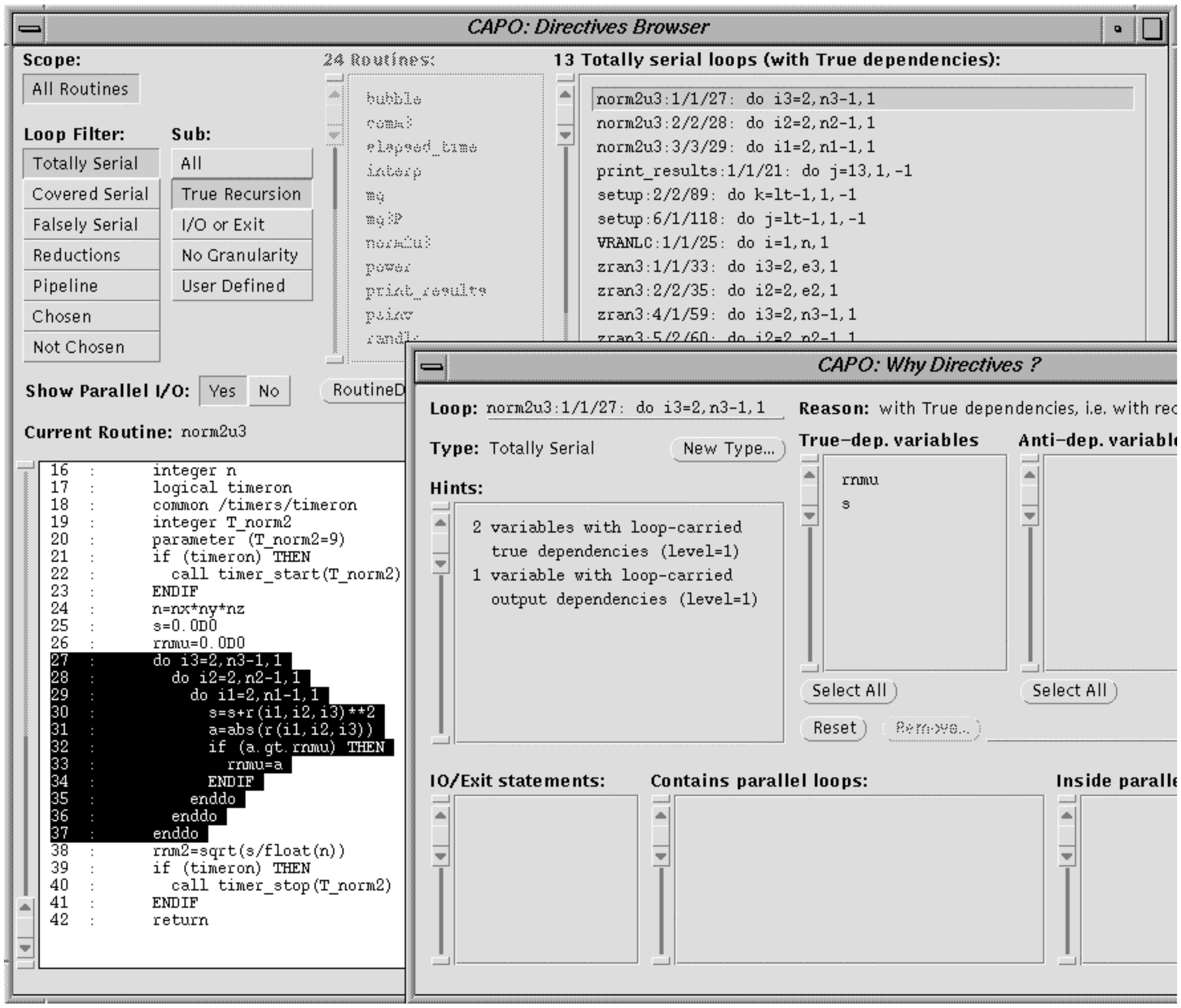
```
 Programming Baseline for NPB - MG Benchmark
 ...
```

[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.



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
```

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
```

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

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

Type: Totally Serial

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

True-dep. variables: rnmu, s

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 $\boxed{\text{Load}}$ button. In the **Analyser** window select the $\boxed{\text{Full}}$ option and click $\boxed{\text{Analyse}}$. 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 $\boxed{\text{Save}}$. 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 $\boxed{\text{New Type}}$ button. Right after the $\boxed{\text{Reduction}}$ setting is selected the **Reduction Operator** dialog box is shown up (see Figure T3-2). Select variable "`rnmu`" and intrinsic function "`max`", and push $\boxed{\text{Apply}}$ 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 $\boxed{\text{Update Directives}}$ 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 $\boxed{\text{Save}}$. 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)
   ```

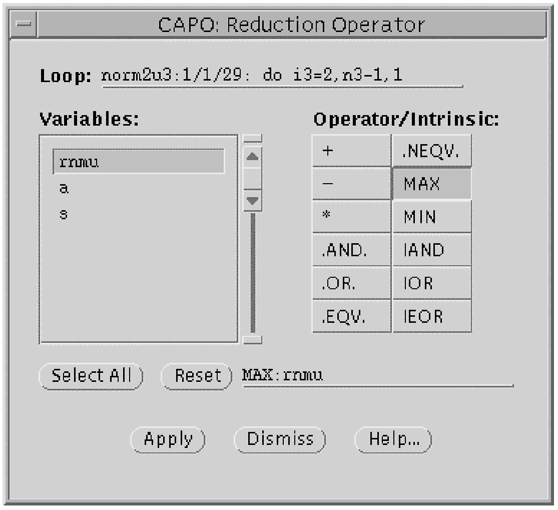


Figure T3-2: The ***Reduction Operator*** *dialog after the* $\boxed{\textit{Reduction}}$ *setting is selected.*

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.

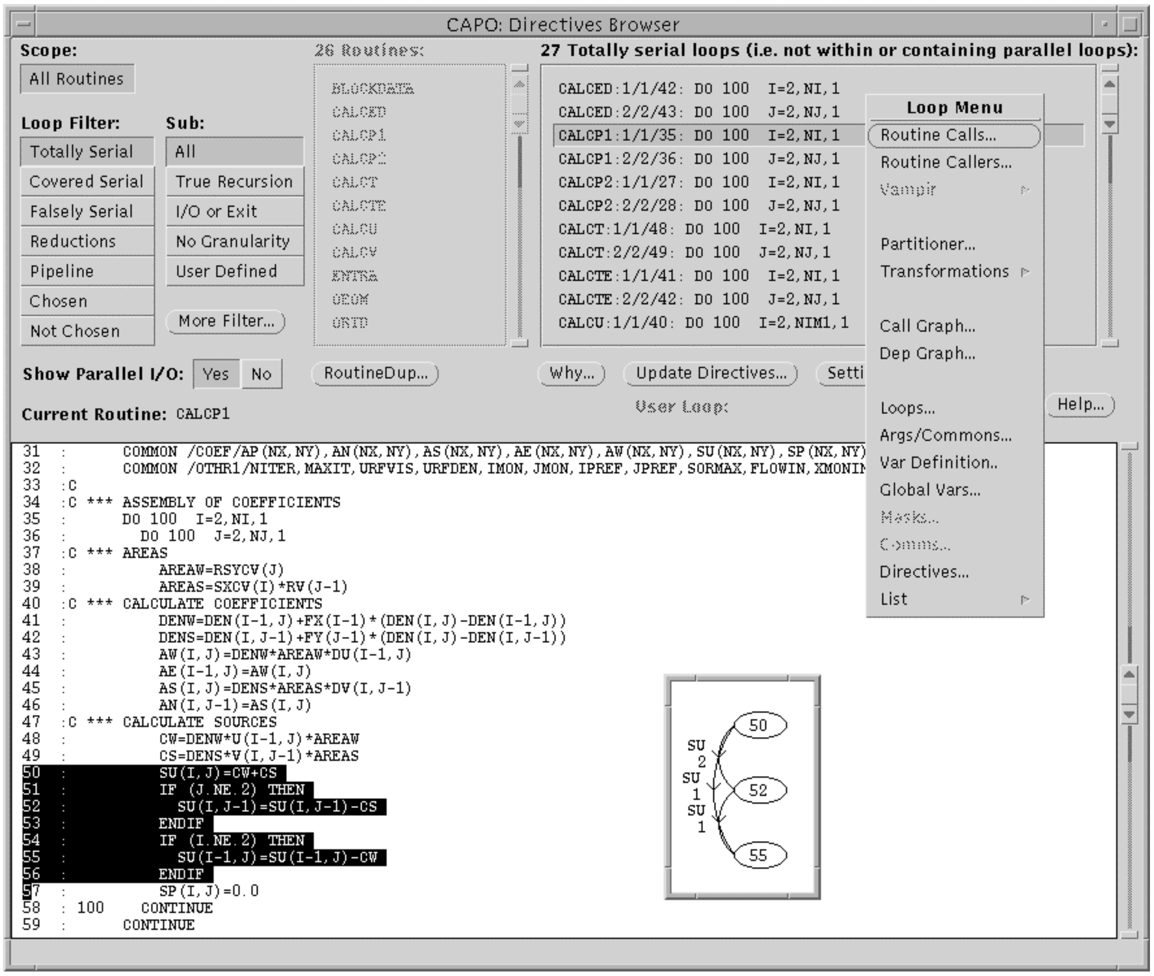
   Choose the loop "`CALCP1:1/1/35: DO 100 I=2,NI,1`" and click $\boxed{\text{Why}}$. The ***WhyDirectives*** window indicates the loop was serialized due to loop-carried dependences for variable `SU`. The ***DepGraph*** (activated from the right-mouse button **Loop Menu** over the selected loop) shows 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.*

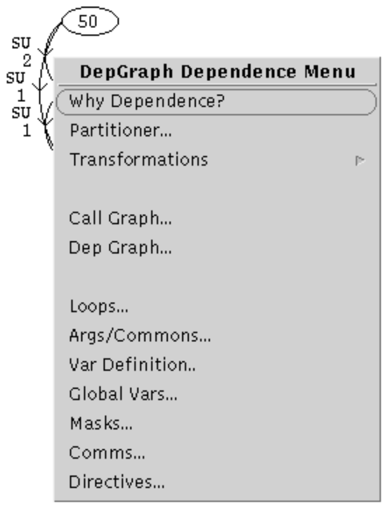


Figure T4-2: The DepGraph Dependence Menu after clicking on a dependence edge.

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
```

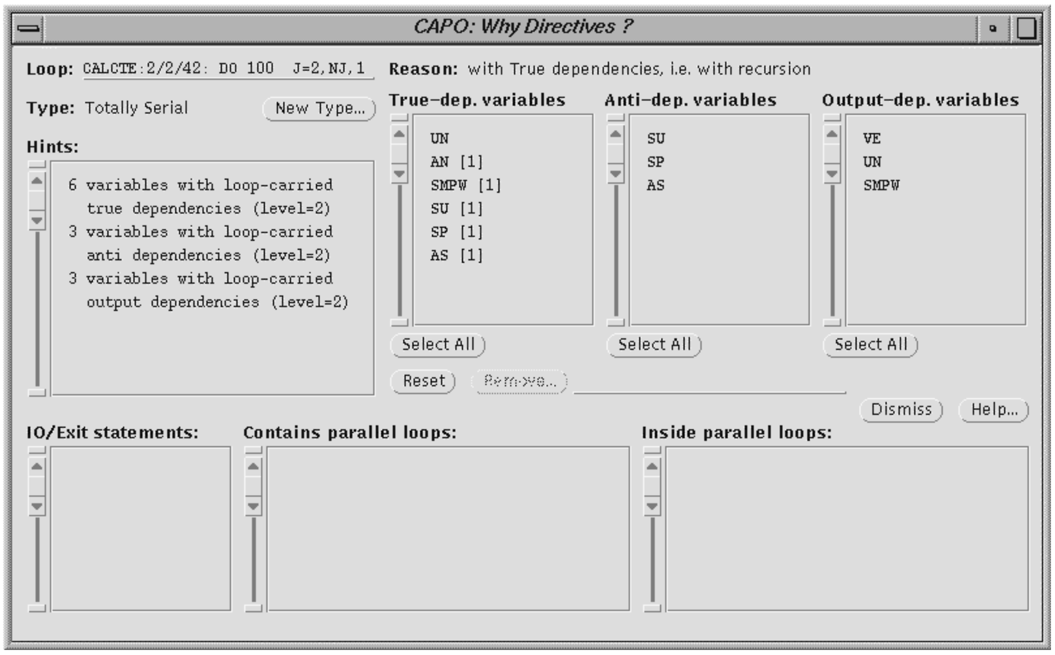
Follow the same procedure to obtain profile on 4 CPUs. The profile outputs for the key routines on 1 and 4 CPUs are compared in Table T4-1. "`ratio`" is 1-CPU time over 4-CPU time, or the speedup on 4 CPUs. The error of ratio is calculated from the statistical sampling error reported in the profile data. 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.



```
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
CALCTE	DO 100 J=2,NJ	VE, UN	iteration
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 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**

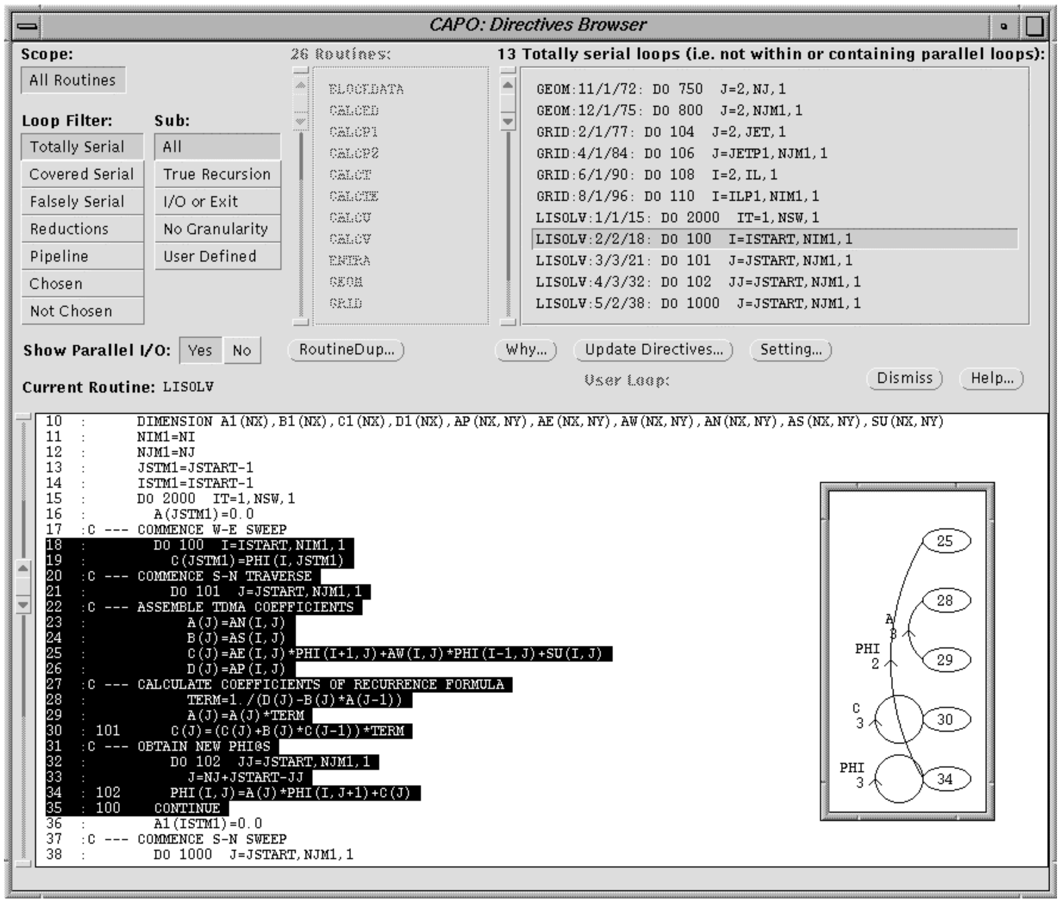
Version 3 – Change of algorithm in LISOLV:

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



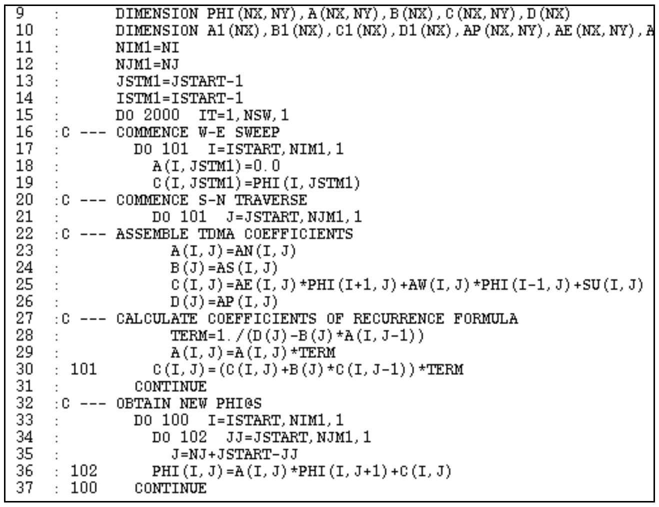
13 Totally serial loops (i.e. not within or containing parallel loops):

```
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
```

Current Routine: LISOLV

```
10  :       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)
11  :       NIM1=NI
12  :       NJM1=NJ
13  :       JSTM1=JSTART-1
14  :       ISTM1=ISTART-1
15  :       DO 2000   IT=1,NSW,1
16  :         A(JSTM1)=0.0
17  :C --- COMMENCE W-E SWEEP
18  :         DO 100   I=ISTART,NIM1,1
19  :           C(JSTM1)=PHI(I,JSTM1)
20  :C --- COMMENCE S-N TRAVERSE
21  :           DO 101   J=JSTART,NJM1,1
22  :C --- ASSEMBLE TDMA COEFFICIENTS
23  :             A(J)=AN(I,J)
24  :             B(J)=AS(I,J)
25  :             C(J)=AE(I,J)*PHI(I+1,J)+AW(I,J)*PHI(I-1,J)+SU(I,J)
26  :             D(J)=AP(I,J)
27  :C --- CALCULATE COEFFICIENTS OF RECURRENCE FORMULA
28  :             TERM=1./(D(J)-B(J)*A(J-1))
29  :             A(J)=A(J)*TERM
30  : 101       C(J)=(C(J)+B(J)*C(J-1))*TERM
31  :C --- OBTAIN NEW PHI@S
32  :           DO 102   JJ=JSTART,NJM1,1
33  :             J=NJ+JSTART-JJ
34  : 102       PHI(I,J)=A(J)*PHI(I,J+1)+C(J)
35  : 100     CONTINUE
36  :         A1(ISTM1)=0.0
37  :C --- COMMENCE S-N SWEEP
38  :         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:

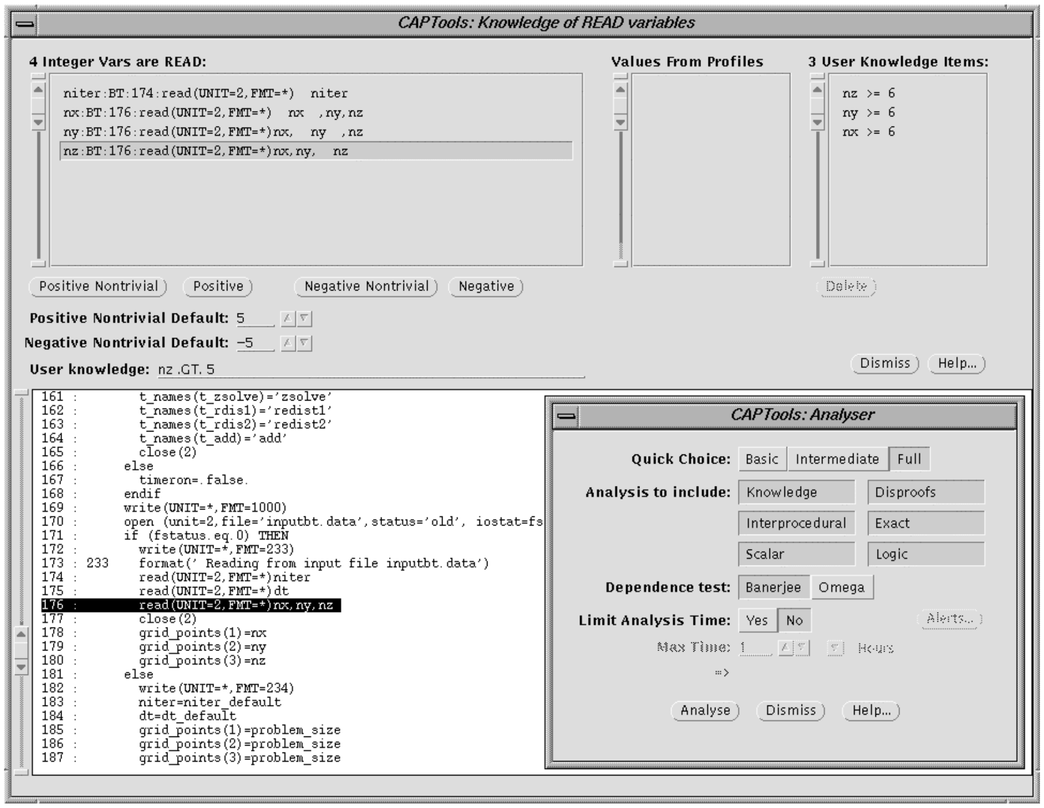
1. **Load source and enter user knowledge**. Click ***Load F77 Source*** in the **File** menu. Select All.list and click the Load button. Select ***READ Knowledge*** from the **Edit** menu. In the **READ Knowledge** window, select variable `nx` and click Positive Nontrivial, see Figure T5-1 on next page. 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

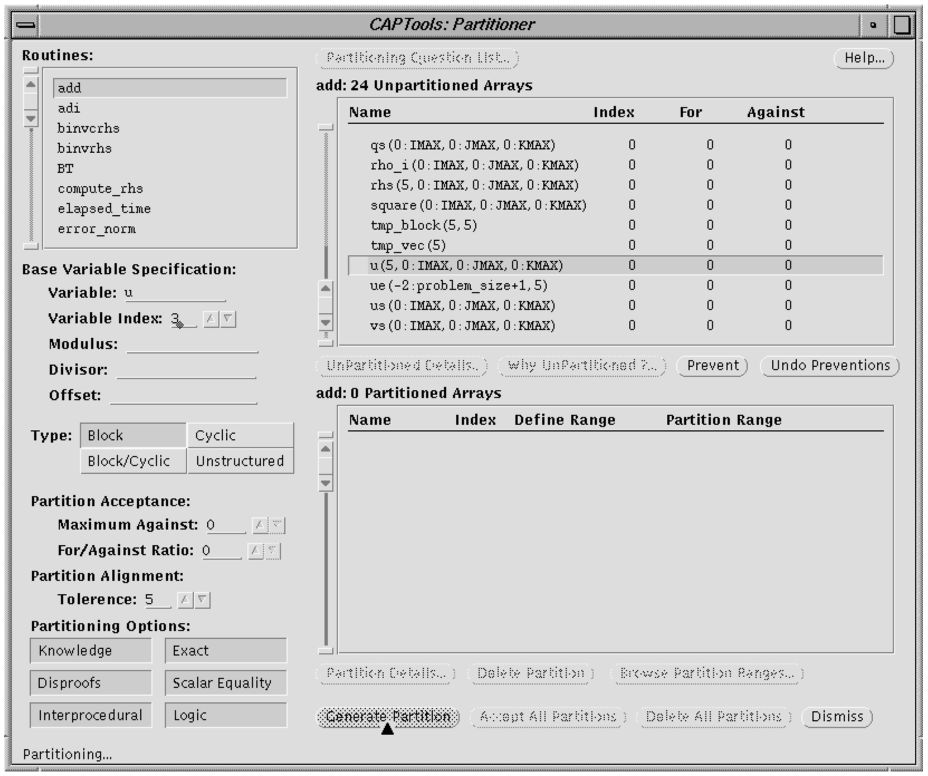
Analysis to include: Knowledge | Disproofs | Interprocedural | Exact | Scalar | Logic

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...

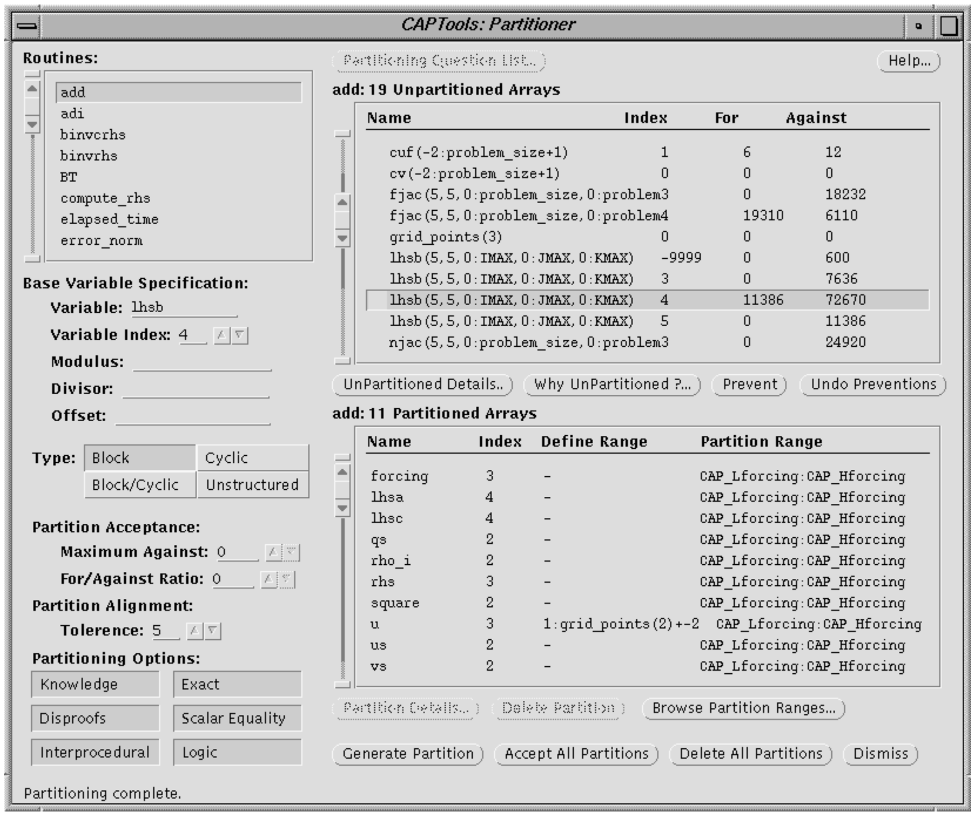
add: 24 Unpartitioned Arrays

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.*



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

Partition Acceptance:

Maximum Against: 0
For/Against Ratio: 0

Partition Alignment:

Tolerence: 5

Partitioning Options:

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:problem3	0	0	18232
fjac(5,5,0:problem_size,0:problem4	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:problem3	0	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

Partitioning complete.

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

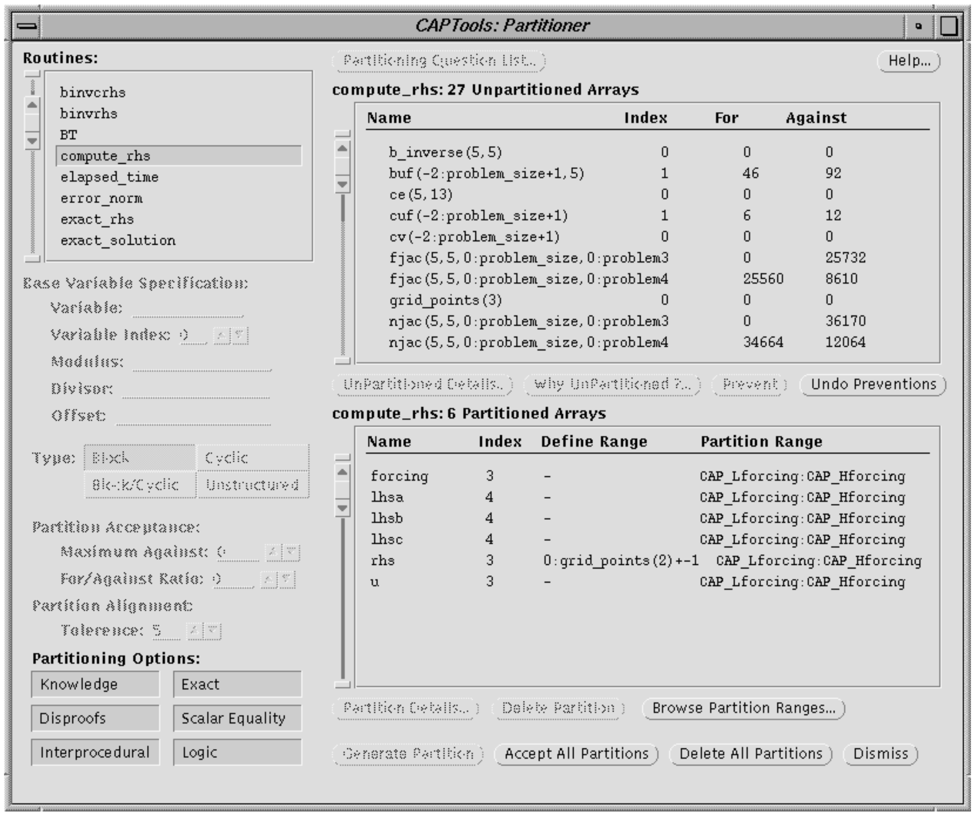
6. **Remove unwanted partitions**. If you use the result produced from Step 4 to generate message-passing code, you would notice that CAPTools place quite a few communication calls inside routine `COMPUTE_RHS`, which exchange boundary values of some of the working arrays (such as `qs`, `rho_i`...) for 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
Help...
Partitioning Question List...
Variable:
Variable Index: 0
Modulus:
Divisor:
Offset:
Type: Block Cyclic Blk-k/Cyclic Unstructured
Maximum Against: 0
For/Against Ratio: 0
Tolerence: 5
UnPartitioned Details.. Why UnPartitioned ?.. Prevent Undo Preventions
Partition Details... Delete Partition Browse Partition Ranges...
Generate Partition Accept All Partitions Delete All Partitions Dismiss

Routines:
binvcrhs
binvrhs
BT
compute_rhs
elapsed_time
error_norm
exact_rhs
exact_solution

Base Variable Specification:

Partition Acceptance:

Partition Alignment:

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

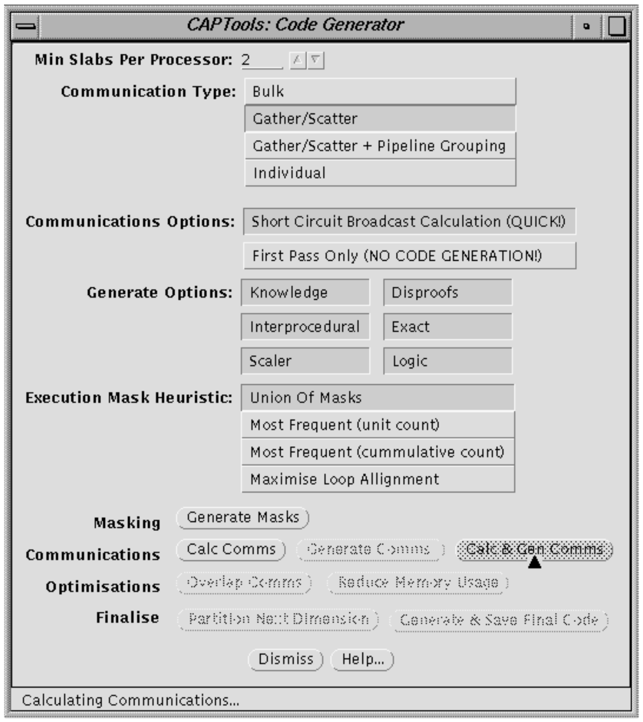
compute_rhs: 27 Unpartitioned Arrays

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	25732
fjac(5,5,0:problem_size,0:problem4		25560	8610
grid_points(3)	0	0	0
njac(5,5,0:problem_size,0:problem3		0	36170
njac(5,5,0:problem_size,0:problem4		34664	12064

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

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


CAPTools: Code Generator
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...

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

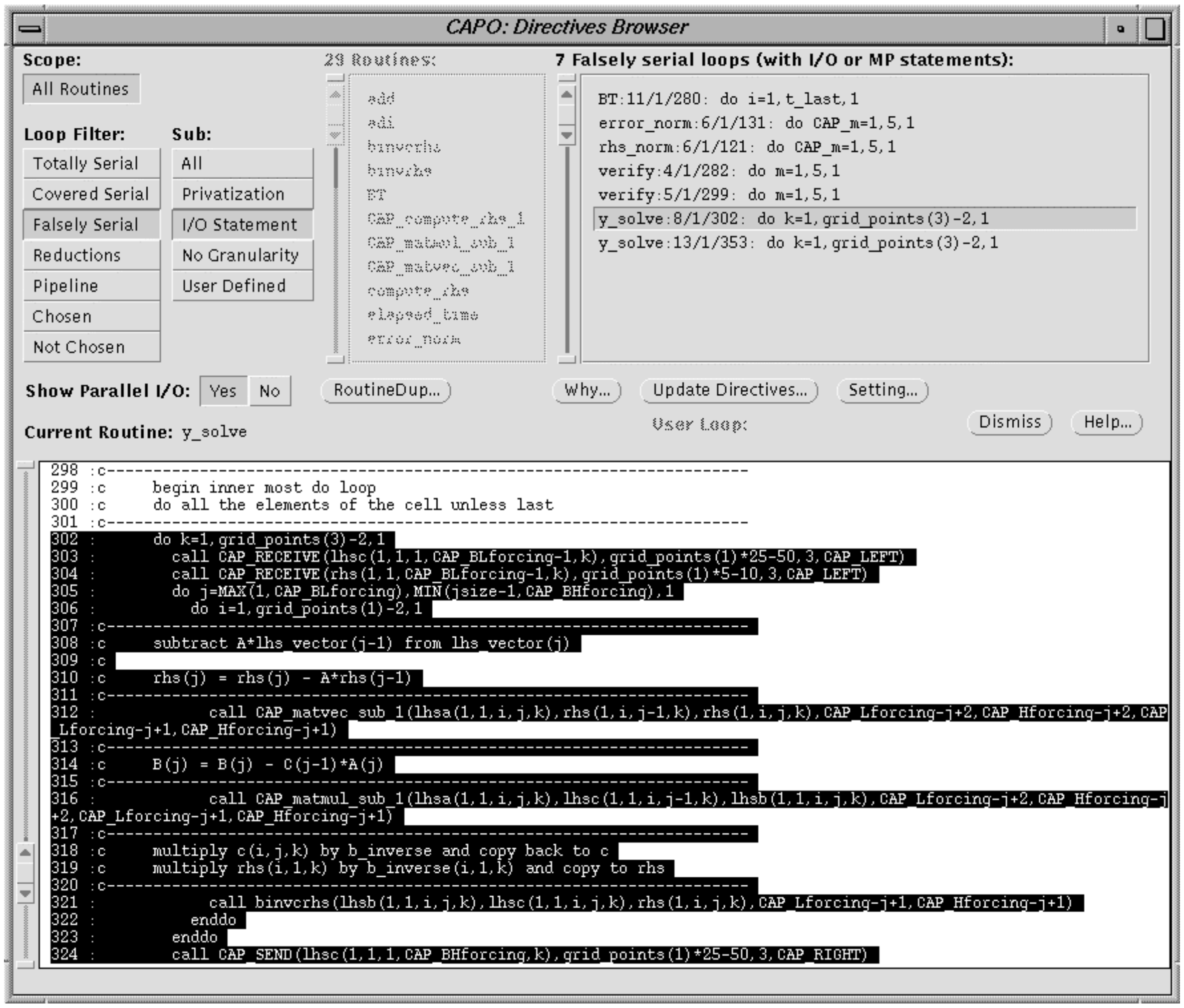
Calculating Communications...

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
```

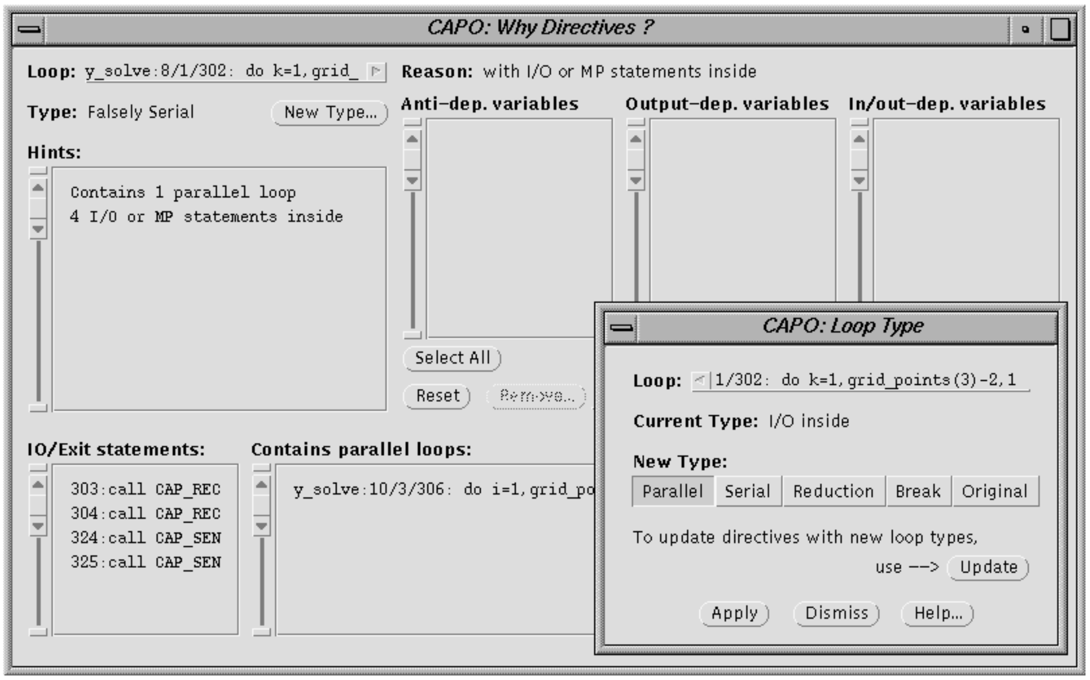
Current Routine: y_solve

```
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.



```
CAPO: Why Directives ?
Loop: y_solve:8/1/302: do k=1,grid_        Reason: with I/O or MP statements inside
Type: Falsely Serial                       Anti-dep. variables   Output-dep. variables   In/out-dep. variables
Hints:
  Contains 1 parallel loop
  4 I/O or MP statements inside
IO/Exit statements:          Contains parallel loops:
  303:call CAP_REC             y_solve:10/3/306: do i=1,grid_po
  304:call CAP_REC
  324:call CAP_SEN
  325:call CAP_SEN

CAPO: Loop Type
Loop: 1/302: do k=1,grid_points(3)-2,1
Current Type: I/O inside
New Type:
  Parallel  Serial  Reduction  Break  Original
To update directives with new loop types,
  use --> Update
```

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