# Parallelization of the TFS multi-block code from RWTH Aachen using the ParaWise/CAPO tools

**Steve Johnson and Cos Ierotheou**

**Parallel Software Products Inc**

**TR-2005-09-02**

## *Introduction*

The ParaWise/CAPO automatic parallelization environment has been used to assist in the OpenMP parallelization of the TFS multi-block code targeted at Sun Microsystems shared memory parallel systems. A series of parallel versions have been created to evaluate the performance of the parallel code based on differing sources of loop parallelism. The aim was to assess the quality of the parallelization from the environment, determine what interaction with the environment is needed and also what manual code alterations are necessary. A parallel version of TFS that can scale to large numbers of processors is the ultimate goal of this work.

The versions of parallel code produced and assessed were:-

1. The initial version produced by ParaWise/CAPO with no user interaction.

2. An improved version where the ParaWise and CAPO browsers were used to investigate, add to and alter information to improve the parallelization.

3. An addition to the previous version where the outer loop of the *ijk* loop nests within a block (typically the '*k*' loop) is chosen to exploit parallelism.

4. A version where parallelism is exploited at the loops that iterate over the blocks in the multi-block code.

5. A hybrid exploiting both inter-block and intra-block parallelism, using nested OpenMP directives.

All the results quoted in this report were obtained on Aachen's four node SunFire E25000 consisting of 72 dual core UltraSPARC IV processors running at 1.05GHz, 288Gb memory, connected with Sun Fire Link network. The compilers used were from the Sun Studio 10 suite.

## *Production and Performance of the Initial Parallel Version*

The initial version was produced without any user interaction, just using the ParaWise interprocedural, value based dependence analysis, and the CAPO OpenMP code generation algorithms that use this dependence information to determine parallelism and variable scoping (i.e. if variables need to be PRIVATE).

The aim of CAPO is to perform as much application code computation as possible in parallel, so if all outer loops in a loop nest have dependencies that inhibit parallelism, any parallelism from inner loops in the loop nest is exploited. This can obviously have a detrimental effect on performance as the frequency of OpenMP runtime overheads can

become significant when exploiting inner loops. Most automatic compilers employ machine dependent metrics to determine at runtime if parallel execution is desirable, forcing serial execution when the OpenMP overhead is deemed to exceed the benefit of parallel computation within the loop. This is not at present used in CAPO so slowdown can be exhibited by some loops, but subsequent user interaction to uncover parallelism in outer loops and other features in CAPO can be used to avoid such cases.

The parallelism exploited in this parallelization (version 1) was mainly from within a block from one of the '*i*','*j*' or '*k*' loops that operate over the dimensions of a block (although not always the outermost of these loops) and also from grid level loops. The code was produced automatically by analysing the application code and generating the parallel OpenMP code all within the ParaWise/CAPO environment. Addition of privatization and reduction clauses, adding of 'nowait' clauses to avoid loop end synchronization when legal, generation of parallel regions containing as many parallel loops as possible in an interprocedural context are all performed automatically by CAPO. Examples of the automatically generated OpenMP code with no user interaction are shown in Figure 1 and Figure 2. Figure 1 shows a chosen parallel loop in routine *AUSM* and the lists of PRIVATE and SHARED variables determined by CAPO. It also shows the associated parallel region which is in routine *UPWIND* surrounding the call to *AUSM*. All loops in *AUSM* and another loop in *UPWIND* are inside this parallel region to promote efficient execution. Figure 2 shows a chosen parallel loop in routine *VISC*, again with the automatically determined set of PRIVATE and SHARED variables, along with the automatically generated parallel region which defines all the identified variables as PRIVATE and surrounds 8 parallel loops in total.

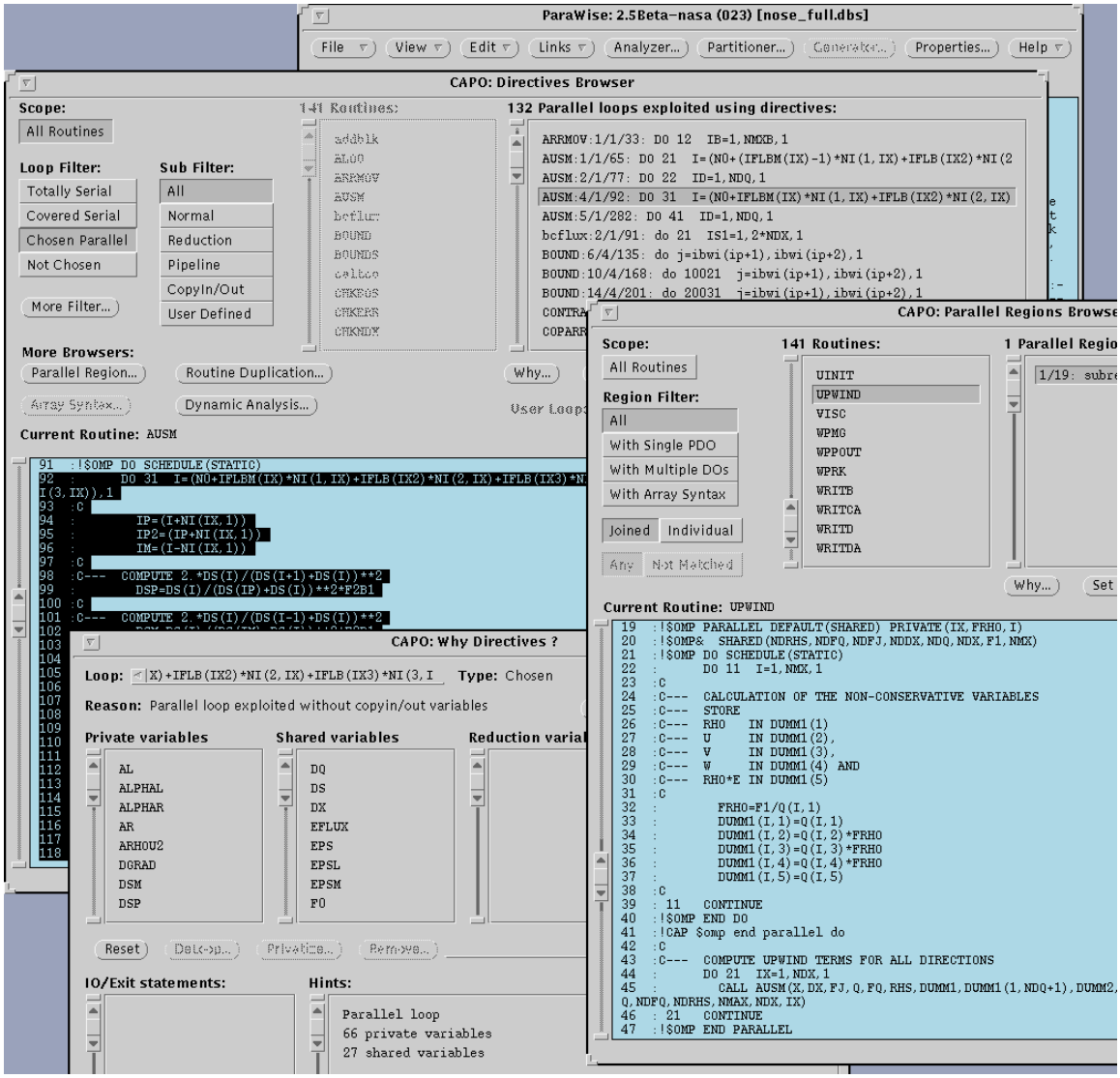


**Figure 1 CAPO showing a parallel loop in routine *AUSM* with the automatically determined PRIVATE and SHARED variables and the automatically set surrounding parallel region in calling routine *UPWIND* that also contains other parallel loops**

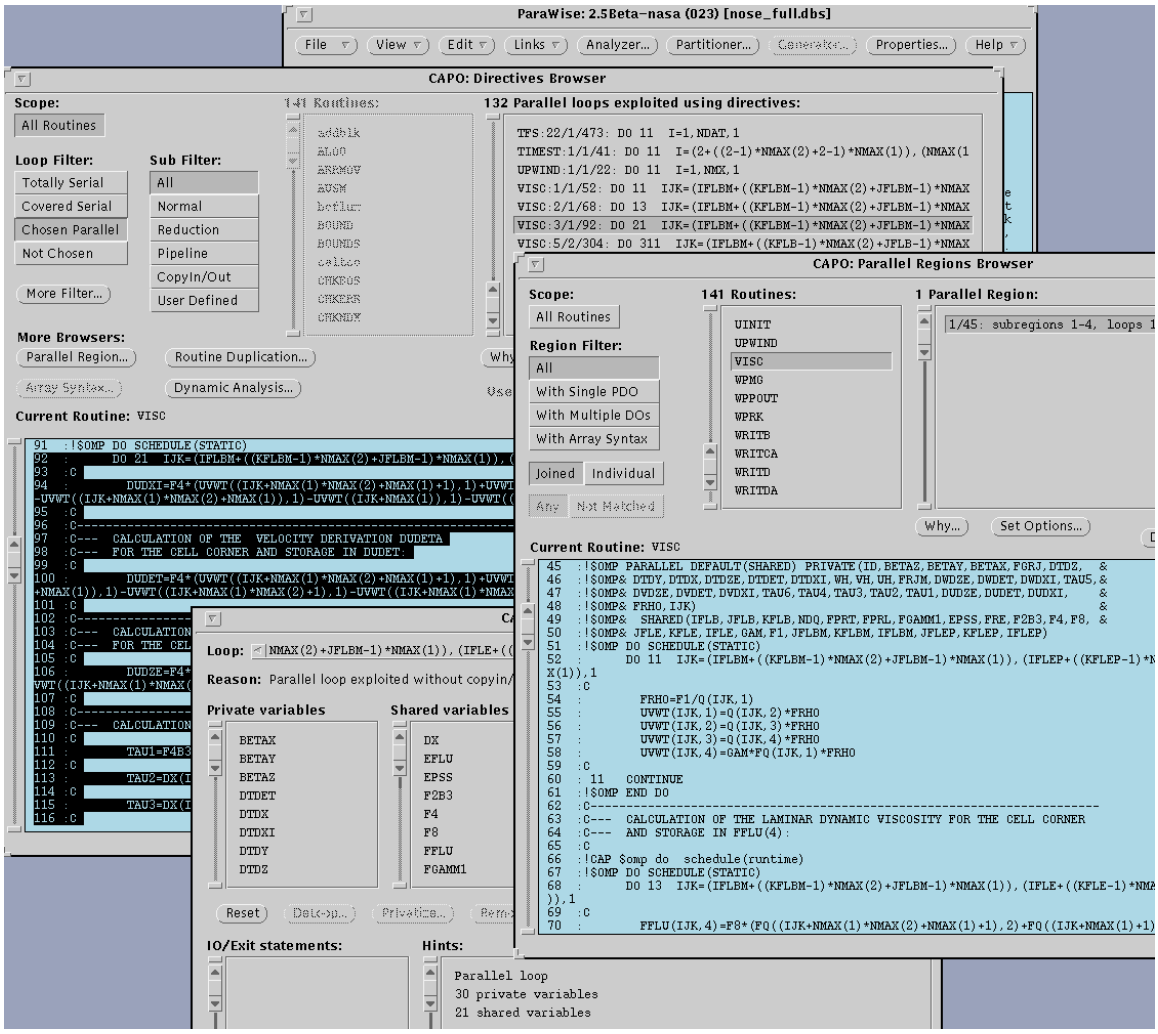


**Figure 2 CAPO showing a parallel loop in routine *VISC* with the automatically determined PRIVATE and SHARED variables and the automatically generated parallel region that surrounds all 8 parallel loops in this routine**

This first version did exhibit some limited speed improvement on some dedicated parallel systems, however, on the heavily loaded Sun Fire system at Aachen, parallel executions proved slower than the serial for the medium size problem input data.

## *Improving the Initial Parallel Version with User Interaction in CAPO*

The ParaWise/CAPO automatic parallelization environment was specifically designed with the understanding that user interaction is desirable and, often, essential for the production of effective parallel code. There are a wide spectrum of examples to reinforce this, in the case of the TFS code, the most obvious examples are the nature of the arrays used to index the *DA* array in the implementation of the multi-block mesh and the mesh dimension strides within a block that are calculated for each block as it is processed. It is obviously not possible for ParaWise to automatically determine the nature of the pointer

arrays, so the user must provide this information. For the mesh dimension strides within a block, the way this is implemented in TFS is beyond the current capabilities of ParaWise (i.e. calculated using the *IPROD1* function from products of the actual dimensions). The user can exploit their knowledge of the nature of these variables to address the parallelization inhibiters determined by CAPO as displayed in its browsers.

The types of loop to be examined by the user in CAPO are :-

- *Totally Serial* loops, where a loop is serial and is not nested within a parallel loop and also does not contain any parallel loops, and
- *Covered Serial* loops where a surrounding loop is parallel or some contained loops are parallel. Often, exploitation of parallelism in a *Covered Serial* loop that contains parallel loops is desirable, as it reduces the frequency of OpenMP runtime overheads.

There are several types of inhibitor to parallelism :-

- Loop carried True dependencies, where a value is assigned in an iteration of the loop and used in a subsequent iteration, obviously prohibit parallel execution.
- Loop carried Anti dependencies, usage of a value in an iteration where the used memory location is overwritten in a later iteration, but only if loop in/out dependencies also exist
- Loop carried Output dependencies, where the same memory location is assigned in several iterations, but only if loop in/out dependencies also exist
- Loop in/out dependencies, where values assigned before the loop are used by computations within the loop or where values assigned within the loop are used after the loop, but only if loop carried Anti or Output dependencies also exist and FIRSTPRIVATE or LASTPRIVATE clauses cannot satisfy the situation.

For the True dependence inhibitors, the user may be able to determine that no values are passed between iterations of the loop from either understanding of the implemented variable references or from knowledge of the algorithm being followed in that loop. For Anti and Output dependencies, it may be that those dependencies from one iteration of the loop to another do not exist (allowing the associated variable to be shared) or it may be that the uses after the loop actually receive their data from assignments after the loop and not from inside the loop (allowing the associated variable to be privatized), although it can also be the case that parallel execution is not legal as the inhibitors actually exist.

For this improvement of the initial parallel version of the TFS code, no information about the multi-block nature of the application was exploited so only intra-block parallelism was investigated. Most of the interaction related to the *ijk* loops within a block, particularly due to the lack of information about the dimension stride variables in linearized one-dimensional array references of the three-dimensional data. Typically, Anti and/or Output dependencies between iterations of the loop were set along with usages of the associated variable after the loop where it was the dependencies between iterations of the loop that should not exist. These inhibitors can be simply removed in the

CAPO Why Directive browser which details all inhibitors to parallelism for the selected loop. Details about the inhibitors can be examined using the full range of ParaWise browsers where, in particular, the Dependence Graph browser can be used to easily study in detail the interprocedural dependencies where assignments and usages exist down deep call trees from inside the loop.

In Figure 3, the *K* loop is currently serial as a loop carried output dependence for array *DUMM1* is set along with uses of *DUMM1* after the loop has completed. An investigation of the indices of *DUMM1* reveals that *NI* is computed in a call to *SETNMX* using the function *IPROD1* which prevented the ParaWise dependence analysis proving the non-existence of the output dependence. With a knowledge of the meaning of the *NI* array and/or an understanding of what the loop is doing as its part of the TFS algorithm, the user can remove the output dependence, allowing the loop to execute in parallel.

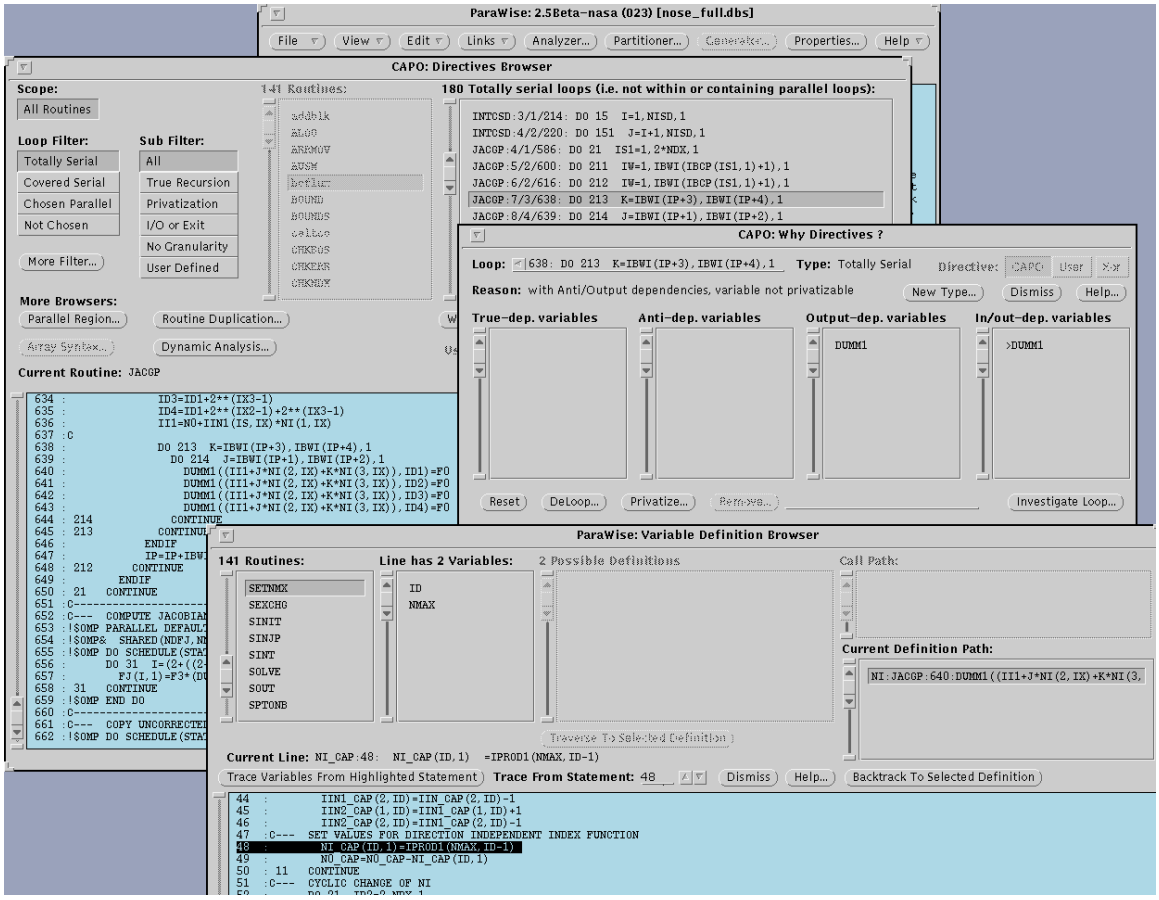


**Figure 3 Using CAPO to investigate serial loops where here the loop in *JACGP* is serial due to the variable *DUMM1* having both output and loop out dependencies. The definition of *NI* (an index of *DUMM1*) is also shown where it is computed in function *IPROD1***

Once the serial loops had been investigated and any alterations made, the parallel OpenMP code was generated. This version 2 did exhibit some speedup on small numbers of processors, but very little scalability.

## *Exploiting Parallelism at the Outer Loop Relating to a Mesh Dimension of a Block*

Although the user interaction in examining parallelism inhibitors was completed for the previous parallel version, it is often desirable to exploit parallelism in a mesh dimension as these can involve a reasonably large number of iterations, especially for larger problem sizes, and therefore can be a profitable source of parallelism. To implement this, the user can use the CAPO Why Directive browser *New_Type* option to force non-*ijk* loops that are currently in the Chosen Parallel category to execute in serial, allowing contained parallel *ijk* loops to be chosen. Figure 4 shows a loop in routine *BCFLUX* that is currently chosen to execute in parallel where it contains 2 loops that could execute in parallel. As the chosen loop exhibits poor parallel performance (performing very few iterations), the *New_Type* option is used to force serial execution and hence allows the contained parallel loops to operate in parallel, hopefully exhibiting superior performance.

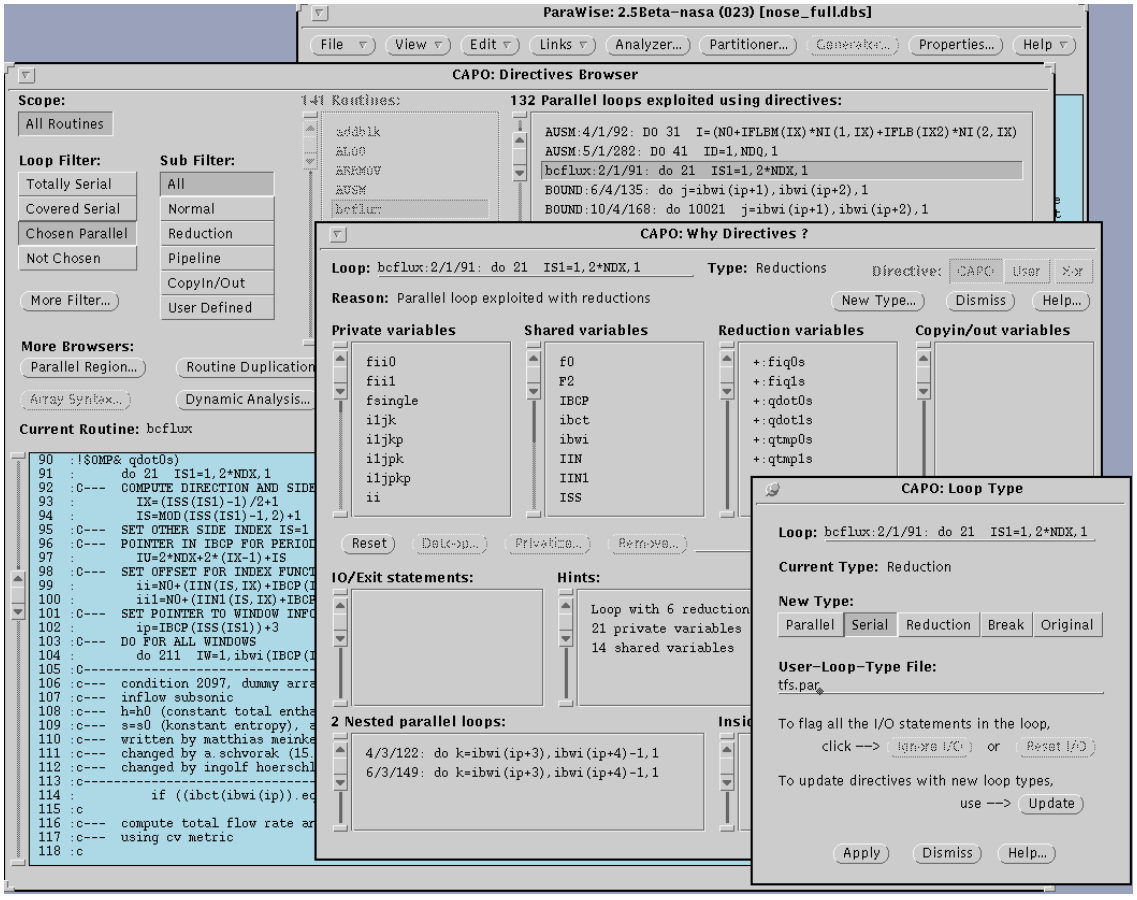


**Figure 4 Using the CAPO New_Type facility to force an inefficient loop to operate in serial allowing more efficient inner loops to be chosen for parallel execution**

This version exhibited significant speedup on dedicated parallel systems (albeit on small numbers of processors) indicating the potential to scale well on larger numbers of processors. When run on the Sun Fire system at Aachen, however, although speedup on

4, 8 and 16 processors was obtained, the performance was disappointing compared to that seen on a dedicated parallel system. The timings were also very erratic with significantly different timings being seen for different runs, indicating that the performance was very dependent on the overall load on the system. Using the Sun Performance Analyzer to determine the cause of this poor performance, the reasonably frequent inter-processor synchronization was, we believe, being affected by the machine load which was largely in calls to the routine *MBCRCT* which exhibited particularly poor speedup. An investigation revealed than the overhead was related to the block-block interactions (992 instances per call), where many block pairs have no connections but all involved one or more end of loop and end of parallel region synchronizations. The code generated by CAPO was then manually altered to move the parallel regions out of the loop over blocks in *MBCRCT* with only 3 synchronizations required for those block-block pairs that involve interaction, as shown in Figure 5. The investigation also revealed the initialization of the workspace array for every block pair required an additional synchronization and was also a major component in the overall runtime of the code. To reduce both the serial runtime and to reduce the often unnecessary synchronization, the code was altered to only initialize the workspace array in the first iteration (i.e. block 1 to block 2 interactions) and in the iteration immediately following those that used the workspace to implement a block-block interaction (other pairs do not write to the workspace so the initialization is not required), as shown in Figure 5. This reduced the serial runtime from 444 seconds to 390 seconds so most of the speedup results presented later in this report are based on the 390 serial runtime. The speedup and scalability of this version 3 of the TFS code are shown in Table 1.

```
!       set ipdm to 1 initially to force zeroing of workspace
        ipdm=1
c--- all multi block iterations
        do ib=1,nbp,1
c--- all neighbouring block iterations
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(IIN2_CAP,IIN1_CAP,IIN_CAP,       &
!$OMP& IFLEP_CAP,IFLBM_CAP,IFLE_CAP,IFLB_CAP,NI_CAP,IBP_CAP,N0_CAP,INB, &
!$OMP& IDSG,IPOINT) FIRSTPRIVATE(IPDM)
      . . . . . .
c--- reset dummy array with zero
          do inb=1,nbp,1
          if (ib.ne.inb) then
            if (ipdm.ne.0) then
!             only need to initialize if anything was added to buffer in
!             previous iteration
!$OMP DO
              do I=1,2*idsg,1
              da(ida+I-1)=f0
              enddo
!$OMP END DO
            endif
c--- get overlapping data from neighbouring blocks
            call setnmx(nmaxg(1,inb),ndx,inb,IIN2_CAP,IIN1_CAP,IIN_CAP, &
     &              IFLEP_CAP,IFLBM_CAP,IFLE_CAP,IFLB_CAP,NI_CAP,IBP_CAP&
     &              ,N0_CAP)
            call getblk(da(iag(niag,inb)),da(ida+idsg),nmx(inb),nag(niag&
     &      ),ndx,ib,inb,ibct(1,ib),nbct(ib),ibwi(1,ib),mbwi,ibcp(1,ib),&
     &      ipdm,ni_cap,n0_cap,iin1_cap)
            if (ipdm.ne.0) then
              call setnmx(nmaxg(1,ib),ndx,ib,IIN2_CAP,IIN1_CAP,IIN_CAP, &
     &                IFLEP_CAP,IFLBM_CAP,IFLE_CAP,IFLB_CAP,NI_CAP,     &
     &                IBP_CAP,N0_CAP)
!$OMP BARRIER
              call putblk(da(ida),da(ida+idsg),nmx(ib),nag(niag),ndx,ib,&
     &        inb,ibct(1,ib),nbct(ib),ibwi(1,ib),mbwi,ibcp(1,ib),       &
     &        NI_CAP,N0_CAP,IIN1_CAP)
!$OMP BARRIER
              call addblk(da(ipoint),da(ida),nmax(1),nmax(2),nmax(3),nag&
     &        (niag),ndx,ib,ibct(1,ib),nbct(ib),ibwi(1,ib),mbwi,ibcp(1, &
     &        ib))
!$OMP BARRIER
            endif
          endif
          enddo
!$OMP END PARALLEL
        enddo
```

**Figure 5 Code manually altered in routine *MBCRCT* to only require 3 barrier synchronizations in those block-block pairs that involve interaction. Also DA array is only initialized in the first iteration and only when the previous iteration involved interaction.**

| Threads | 1 | 4 | 8 | 16 |
|---|---|---|---|---|
| Time (secs) | 390 | 154 | 93.14 | 80.21 |
| Speedup | | 2.53 | 4.19 | 4.86 |

**Table 1 Performance of the parallel version exploiting loops for a dimension of the mesh within a block (for 3 iterations)**

Although some speedup and scalability are exhibited, the scalability is severely restricted. The most significant factor in this lack of scalability is still the *MBCRCT* routine where the amount of computation involved is still relatively small even when compared to the now greatly reduced OpenMP runtime overheads.

## *Exploiting Parallelism from the Block Loops*

In addition to the parallelism exploited from within a block, parallelism exists from the loops that operate over the blocks. To create a version that exploits inter-block parallelism, the previous *ijk*-loop parallel version was used as a starting point. As mentioned earlier, the multi-block nature of the code is implemented by integer arrays read into the application code at runtime, preventing dependence analysis from determining independence between the iterations of the block loops. As a result, the CAPO browsers are again used to produce most of the parallel version with manual code changes used to complete the parallel version.

Only block loops that did not contain I/O were considered and examined in the CAPO browser. All the relevant loops involved the *DA* array with True, Anti, Output and loop in/out dependencies as inhibitors to parallelism. Additionally, in many loops a few other variables were listed as parallelism inhibitors. For these other variables, many inhibitors can be removed as either dependencies between iterations are known not to exist or loop in/out dependencies are known not to exist. A few variables were involved in reduction style operations (e.g. summations) and these can be set in the CAPO Why Directive browser *New_Type* option, as shown in Figure 6.

Some of the routines called in the block loops contained code controlled by a condition to only execute for the first block. These related, for example, to initializing variables that are produced from summations of contributions from every block. To facilitate parallel execution, these initialization were manually moved to before the block loop.

For the *DA* array inhibitors, although the dependencies between iterations of the block loop for call arguments relating to field data of the TFS multi-block mesh should not exist, some arguments using *DA* related to workspace in the called routine which was re-used by every block. Those sections of *DA* used for mesh data must be SHARED as they are used throughout the application code. However, the sections of *DA* used for workspace need to be PRIVATE due to the re-use for every block and where they are not used after the loop completes. The ParaWise Dependence Graph and Arguments browser were used to determine which call arguments relate to workspace to assist the process, but the code alterations required to handle the shared and private sections of *DA* were implemented manually after generation of code from CAPO.

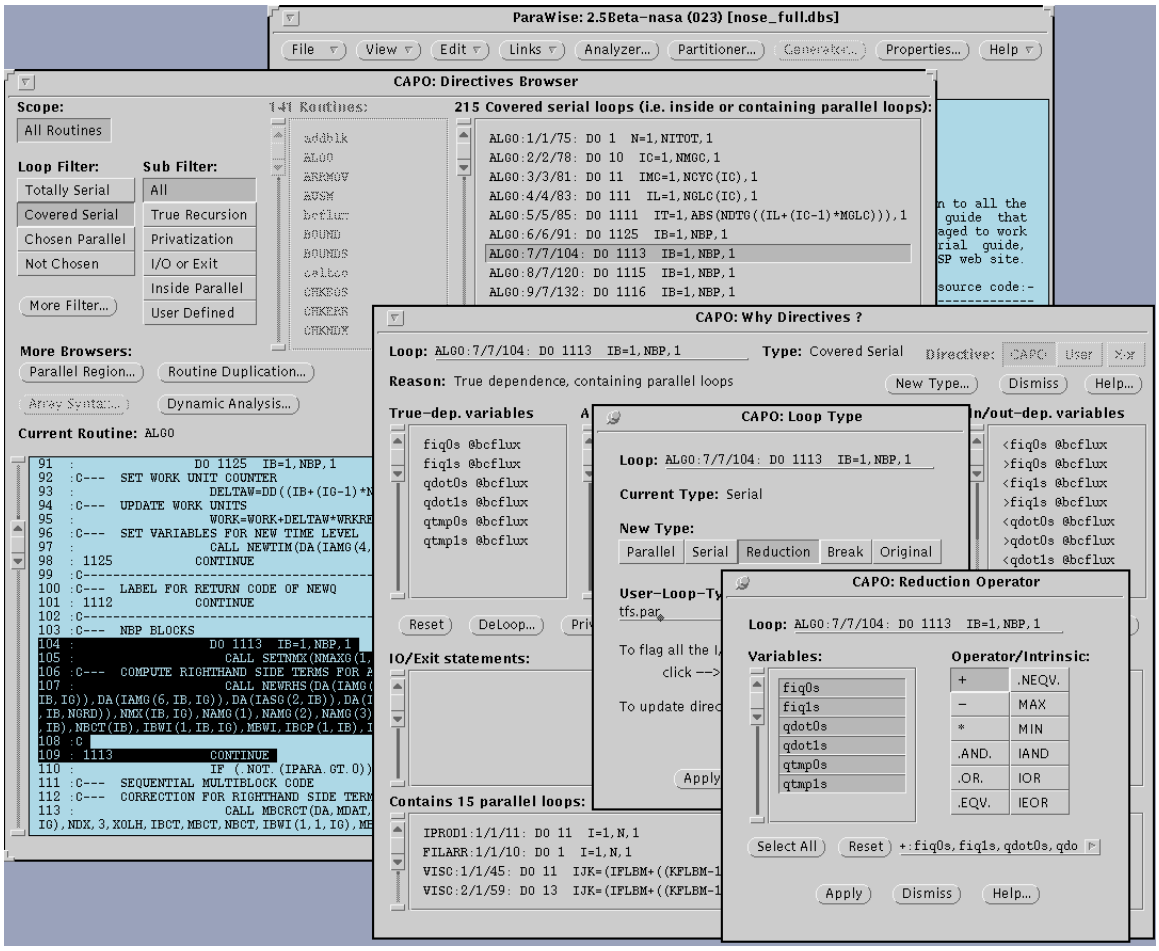


**Figure 6 Forcing CAPO to treat the selected covered serial loop as parallel with the selected variables being evaluated in an summation style OpenMP reduction**

The OpenMP standard requires variables to be either SHARED or PRIVATE and also requires specific definition of the size of any PRIVATE arrays as the memory allocation for slave threads need this size. Inside the subroutines called within the block loops, the renaming of various sections of *DA* allow simple expression of SHARED or PRIVATE variables. Outside of this, in the loop containing the block loop, no renaming is available. One solution is to introduce new arrays for the workspace and alter the call arguments so that the *DA* array is not used, as shown in Figure 7. The disadvantages with this approach are that the memory requirements are increased and that the size of the workspace needs to be determined, either for each block loop in turn or the maximum of all the block loops that use the workspace. As the target for the parallel TFS code is for it to execute efficiently on large numbers of processors, the extra memory with this approach only relates to the master thread as all slave threads will allocate the workspace for their private copies anyway. The size of the workspace required can be determined once during the execution of the code based on the maximum requirement for any block in the mesh, although the current implementation just uses a set workspace size.

```
      double precision dumm1(5000001),dumm2(5000001)
      . . . . .
      . . . . .
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(IIN2,IIN1,IIN,NI,IBP,N0,IB,    &
!$OMP&  DUMM1,DUMM2)                                                      &
!$OMP&  SHARED(MBWI,NGRD,NDX,IG,NBP) SCHEDULE(DYNAMIC)
                    DO 1115  IB=1,NBP,1
                    CALL SETNMX(NMAXG(1,IB,IG),NDX,IB,IIN2,IIN1,IIN,NI, &
     &              N0,IBP)
C---  COMPUTE VARIABLES FOR A NEW TIME LEVEL
                    CALL NEWQ(DA(IAMG(1,IB,IG)),DA(IAMG(2,IB,IG)),DA(    &
     &              IAMG(3,IB,IG)),DA(IAMG(4,IB,IG)),DA(IAMG(5,IB,IG)),  &
     &              DA(IAMG(6,IB,IG)),DA(IASG(2,IB)),DA(IASG(3,IB)),DA(  &
     &              IAMG(7,IB,IG)),DUMM1,DUMM2,NMX(IB,IG),NAMG(1),       &
     &              NAMG(2),NAMG(3),NAMG(4),NAMG(5),NAMG(6),NASG(2),     &
     &              NASG(3),NAMG(7),NMAXG(1,IB,IG),NDX,IBCT(1,IB),       &
     &              NBCT(IB),IBWI(1,IB,IG),MBWI,IBCP(1,IB),IR,IIN2,IIN1,&
     &              IIN,NI,IBP,N0)
C
 1115               CONTINUE
!$OMP END PARALLEL DO
```

**Figure 7 Parallel block loop with the workspace arrays in the call to routine *NEWQ* replaced by new variables *DUMM1* and *DUMM2***

For block loops that contain I/O that were not considered for parallelism, the *ijk*-loops within a block are considered instead. Additionally, the block-block interaction loop in routine *EXCHNG* has dependencies that force serial execution (where the results of this loop differ significantly if these dependencies are ignored).

This version 4 of the application does display some speedup on the Sun Fire system at Aachen as shown in Table 2. (Note that this version did not have the improvement in workspace initialization in *MBCRCT* mentioned earlier). Obviously, as the size of blocks and the corresponding workload for each block varies significantly (a 20+ times difference between quickest and slowest blocks was measured), a static schedule for the block loops was inefficient as compared to using a dynamic schedule. In this case, as the test meshes provided contain only 32 blocks, scalability is restricted to 32 processors, but for efficiency with the variations in block workload, around 8 processors (or a proportion of the number of blocks allowing a few blocks per processor) are needed to allow reasonable load balance.

| **Threads** | **1** | **4** | **8** | **16** |
|---|---|---|---|---|
| **Time (seconds)** | **444** | **153.9** | **143.3** | **108.46** |
| **Speedup** | | **2.88** | **3.1** | **4.09** |

**Table 2 Performance of the parallel version exploiting parallelism in loops processing separate blocks (for 3 iterations) using a DYNAMIC schedule**

## *Hybrid Inter and Intra Block Parallel Version*

The final version 5 is needed due to the restriction on scalability fundamentally present in the inter-block parallel version. The relatively disappointing performance on the Aachen

Sun system from the intra-block parallel version seems to prevent this being a source of efficient performance on large numbers of processors. A hybrid of the two versions using nested OpenMP parallelism should allow a more efficient parallel execution from a small number of processors exploiting inter-block parallelism to be multiplied by the performance of the intra-block parallelism on a small number of processors within each of those original threads.

This hybrid version was implemented by combining the two previous parallel versions of the code. Care had been taken with the placement of parallel regions in the intra block version to ensure parallel regions were kept inside block loops with the extension to this hybrid version in mind. The one significant piece of extra work required in this merge was that the use of THREADPRIVATE directives for common blocks was not possible in the hybrid version as, for example, THREADPRIVATE data was set in a call to *SETNMX* made inside the outer (inter block) parallel region, so those extra threads created for the inner (intra block) region used un-initialised values. In this case these variables are passed into the necessary routines as additional arguments and then defined as PRIVATE for the outer (inter block) parallel region, but shared by the subsequent inner (intra block) parallel region.

Another requirement was to turn nested OpenMP parallelism on and off as required by the parallelism in the code. For most of the application code, where the outer block loop is parallel and inner mesh dimension loops are also parallel, OpenMP nested parallelism is enabled and the number of threads set to the square root of the total number of threads requested by the user (as both levels are given the same number of threads). For the code sections containing only a single level of parallelism, the number of threads is set to be the number requested by the user.

As expected, the results for the hybrid version were superior to either of the previous parallel versions, as shown in Table 3. The runtime is still decreasing as more processors are added up to at least 25 processors. The overheads on larger numbers of processors relate to OpenMP runtime overheads and load imbalance between different blocks. Contributions to this are, for example, the poorly performing block-interaction loop in *EXCHNG* that must execute in serial (so that all threads are employed for the contained *ijk* loops), with the additional impact of one or two small code sections that were left as serial because the OpenMP overheads led to slowdown when parallelism was exploited. These code sections (such as in routine *SEBNUL* which exploits inter-block level parallelism, but no intra-block level parallelism) only represented a tiny proportion of runtime in serial, but on larger numbers of processors their impact on speedup is more significant (as predicted by Amdahl's law). A 16 thread (4x4) timeline from the Sun Performance Analyzer is shown in Figure 8 that clearly shows these performance limitations.

| Threads | 1 | 4 | 9 | 16 | 25 |
|---|---|---|---|---|---|
| Time (secs) | 390 | 151.39 | 78.53 | 52.79 | 44.79 |
| Speedup | | 2.57 | 4.97 | 7.39 | 8.71 |

**Table 3 Performance of the hybrid parallel version exploiting parallelism between blocks and within blocks (for 3 iterations)**

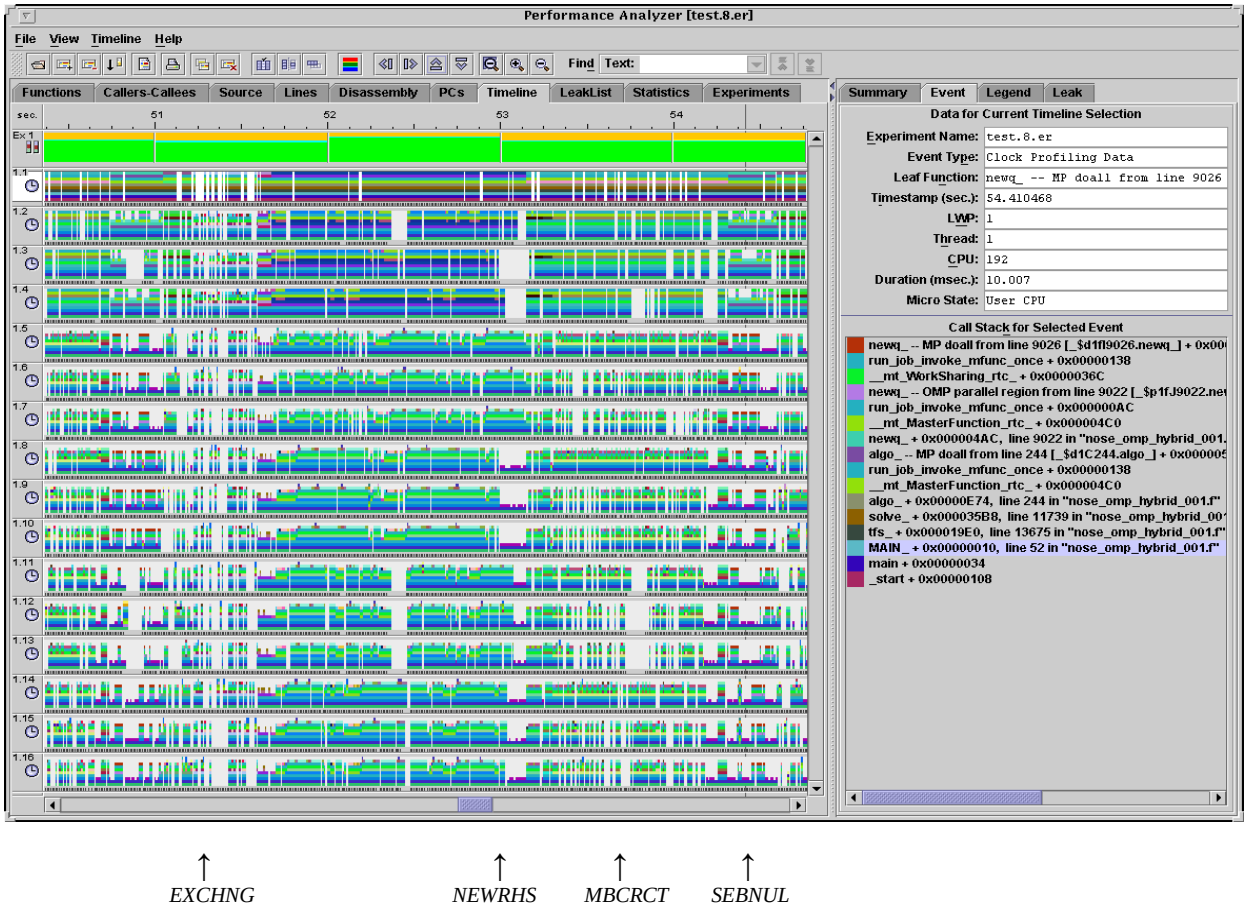


**Figure 8 Sun Performance Analyzer timeline for the 16 thread (4x4) hybrid OpenMP TFS code. The effect of the non-parallelizable loop in routine *EXCHNG*, the load imbalance between blocks for calls to *NEWRHS*, the effect of routine *MBCRCT* and the serial execution of *SEBNUL* are all highlighted.**

## *Summary of Experience of Using the ParaWise/CAPO Environment for the Parallelization of the TFS Application Code*

The ParaWise/CAPO environment was used to perform most of the parallelization of the TFS code, and greatly assisted the subsequent manual tuning also used. Additionally, the

information provided by the Sun Studio Performance Analyzer was essential in achieving reasonable speedup and scalability.

Most of the loops and routines in the code were automatically parallelized without any user attention, using the inter-procedural, value based dependence analysis provided by ParaWise, along with the power of the code generation algorithms provided by the CAPO module. Most of the work in improving the parallelism detected and in selecting the most profitable loops for parallel execution was performed within the environment, allowing this to be achieved in a short time period (a few hours) without any need for debugging the generated parallel code. The Sun Performance Analyzer was used to determine which loops were performing poorly focusing the user's effort in the CAPO browsers to the crucial code sections.

For the mesh dimension intra block parallel version, manual intervention was only required in the final tuning phase, where the amount of computation in the parallel loops was small when compared to the related OpenMP overheads.

For the inter block parallel version, most of the parallelization was performed within the ParaWise/CAPO environment. The use of integer arrays that are read at runtime as pointers into another large array prevented the dependence analysis from detecting the parallelism, however, using the CAPO browsers allow generation of parallel code for the block loops. Additionally, the requirement of OpenMP to define variables as PRIVATE or SHARED forces alteration to the calls using sections of the *DA* array as part of the mesh and workspace. Currently, this needs to be performed manually, although an algorithmic approach that could be automated in CAPO may be possible in future versions.

Most of the time spent in this work was in debugging manually altered code in the implementation of the inter block parallel version and, particularly, the tuning of the *MBCRCT* routine.