Parallelization of the DTU Scattering and Polymer Application Codes using the ParaWise/CAPO tools

Steve Johnson and Cos Ierotheou

Parallel Software Products Inc

TR-2005-09-01

Introduction

The ParaWise/CAPO automatic parallelization environment and the Sun Studio Performance Analyzer have been used to assist in the OpenMP parallelization of two codes Scattering and Polymer, provided by DTU, targeted at Sun Microsystems shared memory parallel systems. A series of parallel versions were created to evaluate the performance of the parallel codes based on differing sources of loop parallelism. The aim was to assess the quality of the parallelization from the environment, determine what interactions with the environment were needed and also what manual code alterations were necessary to obtain as efficient a parallelization as possible.

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. A version based on that of version 2 but with manual optimizations.

The initial parallelization of both codes was performed in a few hours following the standard stages of parallelization in the ParaWise/CAPO environment. These stages consist of loading the application source code into ParaWise and then performing a fully automatic inter-procedural value-based dependence analysis. This is followed by automatically generating OpenMP code via the CAPO module to produce the initial parallel versions.

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.

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.
All the results quoted in this report were obtained on the DTU HPC computer Issac which is part of a Sun Fire E25000 server split into two domains (Newton and Issac),

consisting of 72 x 1.35GHz UltraSparc IV processors. The compilers used were from the Sun Studio 10 suite.

Parallelization of the Scattering Application Code

The parallelization of the Scattering code was performed in a few hours following a number of standard stages in parallelization using the ParaWise/CAPO environment. This was followed by automatically generating OpenMP code via the CAPO module to produce an initial parallel version.

An example of the initial parallel version of the Scattering code is shown in Figure 1 where a loop in routine *HXP3C* has been selected and the privatization information and directive addition that was automatically determined is displayed.

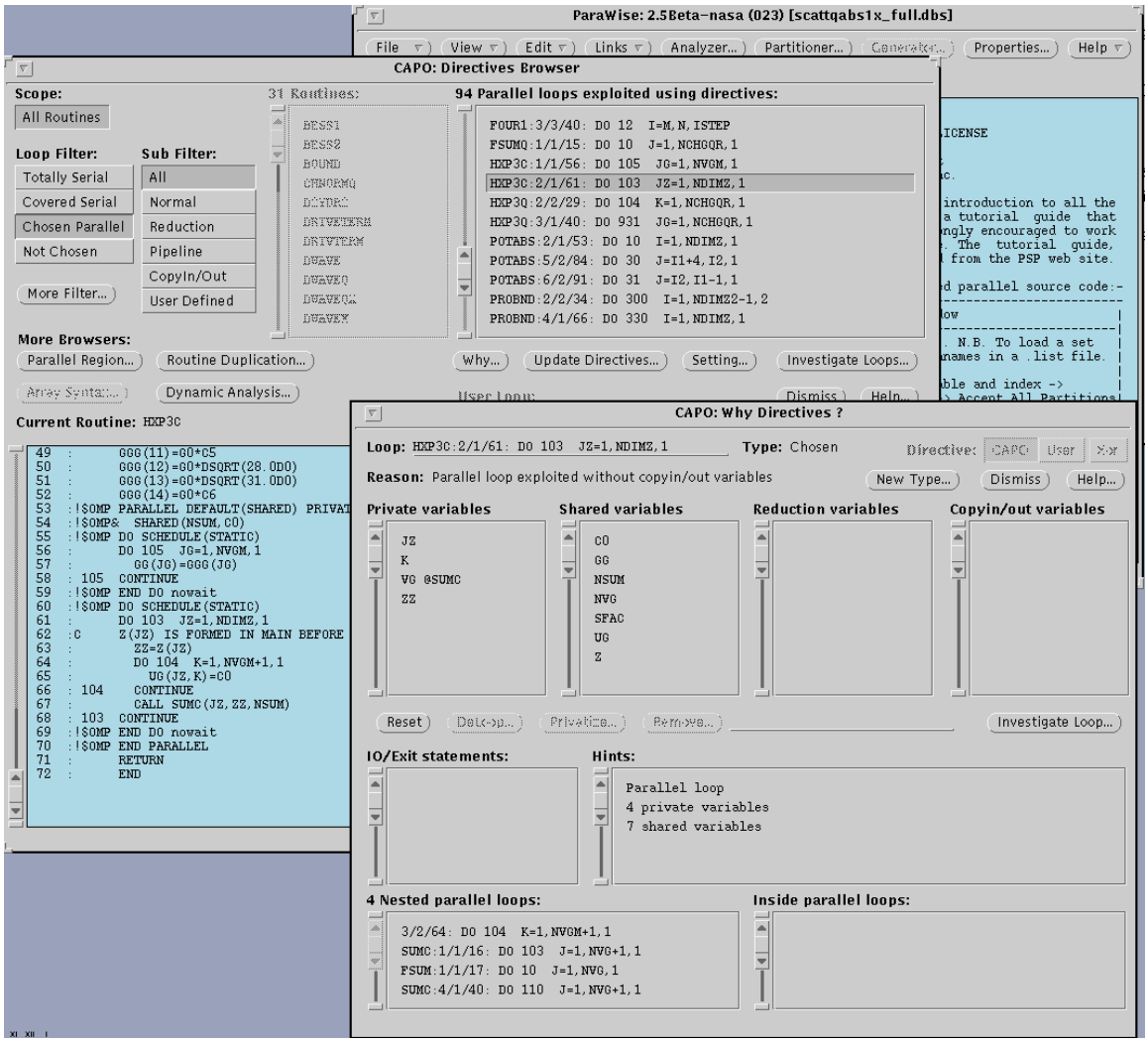

Figure 1 CAPO Directives browser showing CHOSEN PARALLEL loops from the initial parallel version with no user interaction with the automatically parallelized OpenMP code.

The next phase relates to user inspection of the parallelization using the CAPO Directive browser to provide variable range and/or algorithmic data flow information to improve the accuracy of the dependence graph and enhance the available parallelism that is detected. Examination of loops listed as totally serial (i.e. loops that are executed on a single thread only with no outer loops or inner loops exploiting parallelism) and also loops listed as covered serial (where either an outer loop or some inner loop are parallel) is the first step.

In the case of the Scattering application code, most of the interaction relates to workspace arrays re-used in every iteration of a loop, where use of values calculated from a previous iteration of the loop or from before the loop have been detected and/or uses of the workspace after the loop have been detected. The inter-procedural, value based dependence analysis of ParaWise detects most cases where workspace is assigned and used only within an iteration of a loop, but complex index expressions and values whose nature is only known at runtime can prevent precise determination. The user can determine the non-existence of such dependencies either by knowledge of the algorithms implemented in the application code or by a detail examination of the data accesses involved.

For such loops, the types of dependencies that are relevant are:-

- True dependencies between iterations of the loop caused by an assignment in an iteration of the loop and a subsequent usage in a later iteration of that loop.
- Anti dependencies caused by a variable usage in an iteration where the used value is overwritten in a later iteration (where loop in/out dependencies also exist).
- Output dependencies caused by assignment in an iteration with a re-assignment of the same memory location in a later iteration (where loop in/out dependencies also exist).
- Loop in/out dependencies where values assigned before the loop are used in the loop or values assigned in the loop are used after the loop (where loop carried anti and/or output dependencies also exist).

Parallelism can be uncovered if the user can determine that the true dependencies listed by CAPO as serializing a loop do not exist. In addition, if no true dependence exists, parallelism can be uncovered when either anti and/or output dependencies are listed by CAPO as inhibitors to parallelism and are known not to exist (allowing the associated variables to be SHARED) or the related loop in/out dependencies are known not to exist (allowing the associated variables to be PRIVATE). Obviously, CAPO detects cases when FIRSTPRIVATE or LASTPRIVATE clauses can be used to allow parallelization, where these loops are classified as parallel in the CAPO browser.

Figure 2 shows the CAPO directive browser for a covered serial loop selected in routine *DWAVE* with the variables causing dependencies that currently inhibit parallel execution. An investigation of these variables and their dependencies using the ParaWise Dependence Graph browser revealed that they relate to workspace that is initialized in

every call to routine *D2YDR2* and are not used after the loop and therefore the TRUE and loop IN/OUT dependencies do not exist (and can be removed using the buttons in the Why Directive browser shown in Figure 2) allowing the variables to be privatized.

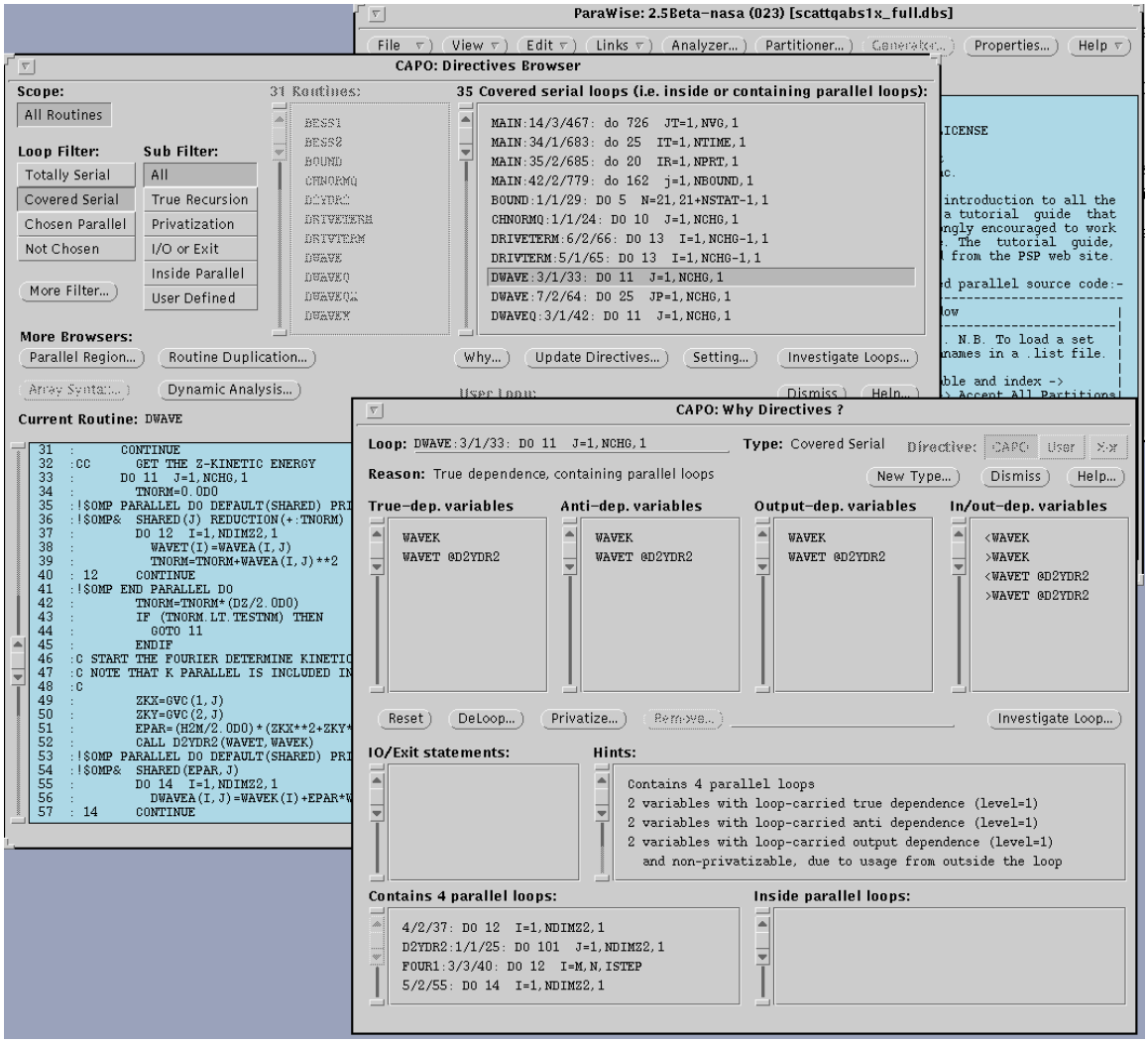

Figure 2 Example of a COVERED SERIAL loop displaying inhibitors of parallelization and contained parallel loops. Both *WAVEK* and *WAVET* are workspace arrays initialized in routine *D2YDR2*

The resultant parallel OpenMP code generated by CAPO showed improved performance over the originally generated version, but further improvements were needed to achieve significant speedup and scalability to larger numbers of processors.

An examination of the loops automatically selected for parallel execution by CAPO revealed that some of those loops performed only a small number of iterations at runtime and did not therefore provide a good source of parallelism. In almost all cases, other inner parallel loops in the associated loop nest had more significant numbers of iterations and therefore would provide a more profitable source of parallelism. To make CAPO select these more profitable loops, the chosen loops that perform only a few iterations were

marked in CAPO via the Why Directive, *New_Type* option as serial loops (as shown in Figure 3) so the contained parallel loops (*DO 26* and *DO 34*) would be chosen for parallel execution. The resulting parallel code showed significant speedup and some scalability, with a considerable improvement on the previous version.

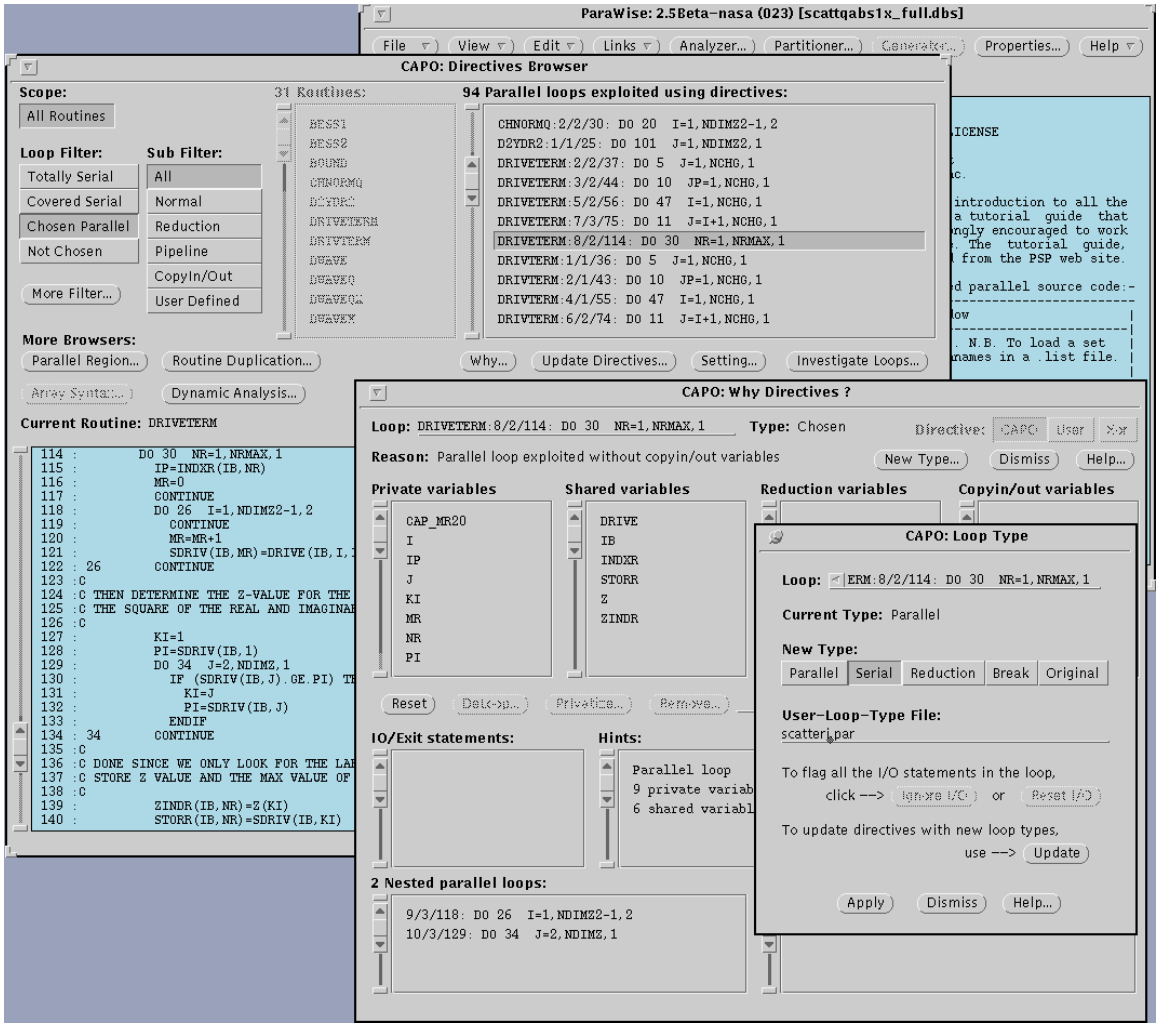

Figure 3 Using CAPO to change the chosen parallel loop forcing the currently selected loop to execute in serial so that the inner loops shown will be chosen for parallel execution

As most of the code was now being executed in parallel and no further parallel loops could be uncovered by user interaction in CAPO, the Sun Performance Analyzer was used to inspect the parallel execution on 1 and 2 processors. The information showed that most loops with significant runtime were speeding up reasonably efficiently on 2 processors, however, a few loops showed no speedup and a few others showed only a modest speedup. The loops that showed no speed improvement were inside parallel loops performing a significant number of iterations, implying load imbalance as a likely cause of this poor performance. This was confirmed by the large time attributed to OpenMP synchronizations on 2 processors shown in the Sun Performance Analyzer, and inspecting

the time attributed to each of the callers proved that load imbalance caused the poor performance of all the loops identified.

For the loops that showed no speed improvement, an inspection showed that most of the computational work occurred under certain conditions i.e. this only happened on the first processor of a 2 processor execution. To avoid this, a DYNAMIC schedule was applied to each of the poorly performing loops, producing a code that speeds up and scales fairly well and only wasted a small amount of time at OpenMP synchronizations when examined in the Sun Performance Analyzer.

The resultant parallel version showed a significant improvement on the previous version, but some loops still exhibited significant load imbalance when examined in the Sun Performance Analyzer. The imbalance in these loops is very severe with a few iterations performing a significant amount of computation and many other iterations performing no computation, so the DYNAMIC schedule did not achieve sufficient load balance. To overcome this, CAPO was used to force those loops to execute in serial so that inner loops could then operate in parallel and provide a better load balance.

The results for this version as run on the Sun Fire system Isaac at DTU are shown in Table 1. Reasonable speedup is being exhibited so another use of the Sun Performance Analyzer, this time on 8 threads, was used to see if the scalability could be improved.

Threads	1	4	8	16	20
Time(secs)	2875	777	466	283	272
Speedup		3.7	6.17	10.16	10.57

Table 1 Performance of Scattering code on Isaac

Examining this data revealed 4 loops with significant compute times that were not scaling well on 8 threads. Using CAPO to exploit parallelism from inner loops in two of these cases (loops 213 and 220 in the main program) alleviated their negative impact on performance. The other two loops (in routines *DWAVE* and *DWAVEQ*) were initializing arrays and were followed by loops that populated those arrays, where those loops had a DYNAMIC schedule preventing a NOWAIT clause from being used at the end of the initialization loop. To overcome this, the initialization was manually incorporated into the main DYNAMIC loop, avoiding the need for synchronization. The performance of the resultant parallel code is shown in Table 2. These minor improvements have increased scalability so that 20 or more processors can be effectively employed for this application code.

Threads	1	4	8	16	20
Time(secs)	2875	761	415	260	227
Speedup		3.78	6.75	11.05	12.67

Table 2 Performance of the improved scalability for the Scattering code on Isaac

Parallelization of the Polymer Application Code

An initial parallel version of the Polymer code was created after dependence analysis and CAPO directive generation. The performance of this initial version was poor and an inspection revealed that the loops chosen for parallel execution involved only a fairly small number of iterations and a fairly small amount of computation.

The investigation indicated that the only loops providing significant sources of parallelism were those in the routine *FORCE* operating over molecules. Of the four code sections, those relating to calls to routines *INTRAF*, *BNDDIH* and *FORSUB* are surrounded by two loops, one of which performs only a single iteration, so the other loop which processed molecules needs to operate in parallel. The other code section relates to the call to *INTERF* where a choice of loops exist, however, an algorithmic dependence relating to variable *NABO* prevents parallel execution of the outer of these loops (as shown in Figure 4) so only the inner loop is an obvious source of scalable parallelism.

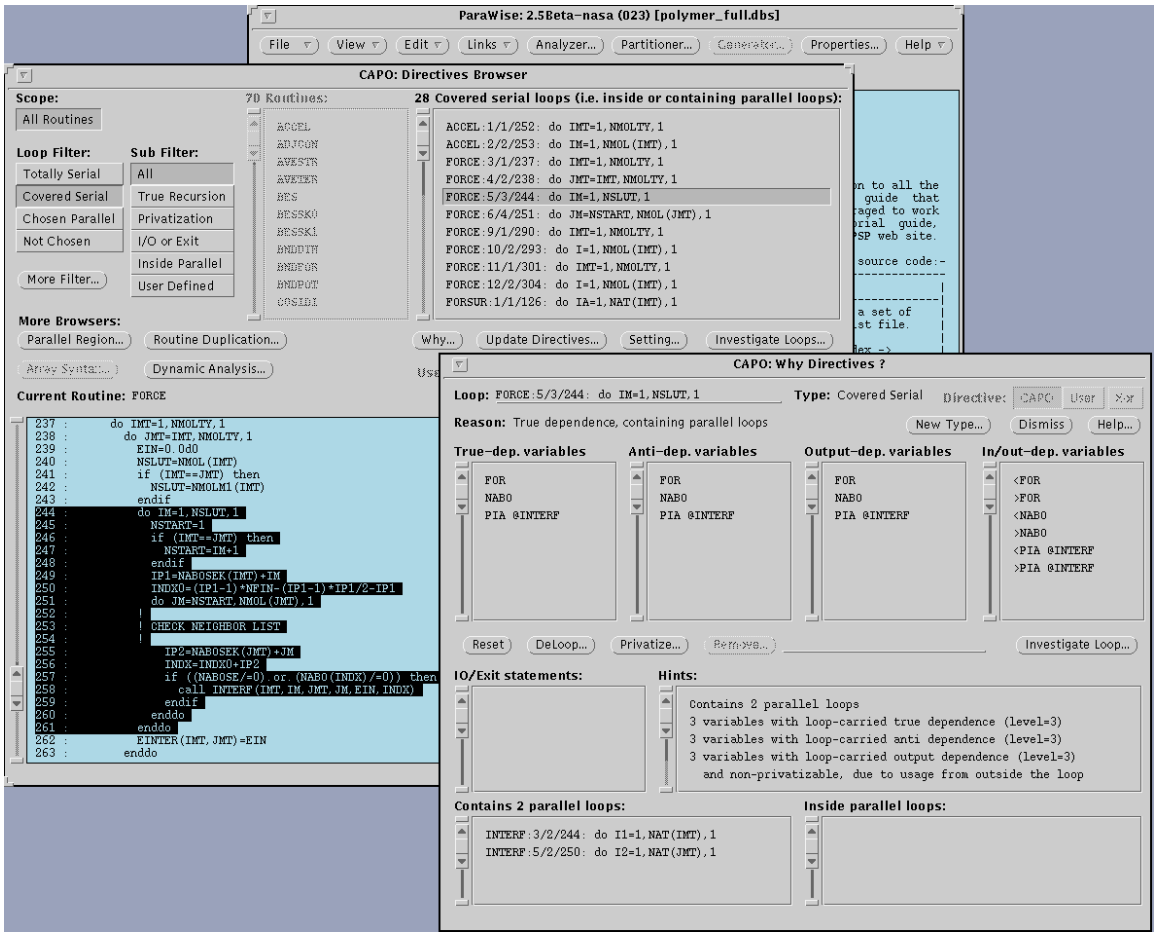

Figure 4 CAPO displaying inhibitors to parallel execution including variable *NABO* which prevents this loop from executing in parallel

Selecting this loop that surrounds *INTERF*, which is currently classed as covered serial in CAPO revealed that true, anti, output and loop in/out dependencies for variables *FOR* and *PIA* where preventing parallelism. An examination of the access to these arrays using the ParaWise Dependence Graph browser revealed that all accesses were within routine *INTERF* and are all summations adding contributions to the *FOR* and *PIA* arrays, as shown in Figure 5. This situation should have been recognized by CAPO, but the current version failed to detect the parallelism. The exploitation of this parallelism with OpenMP was therefore performed using CAPO with user interaction and also some manual alterations. These manual alterations concern the array size being summed and also the nature of the writes to those arrays (i.e. full or sparse).

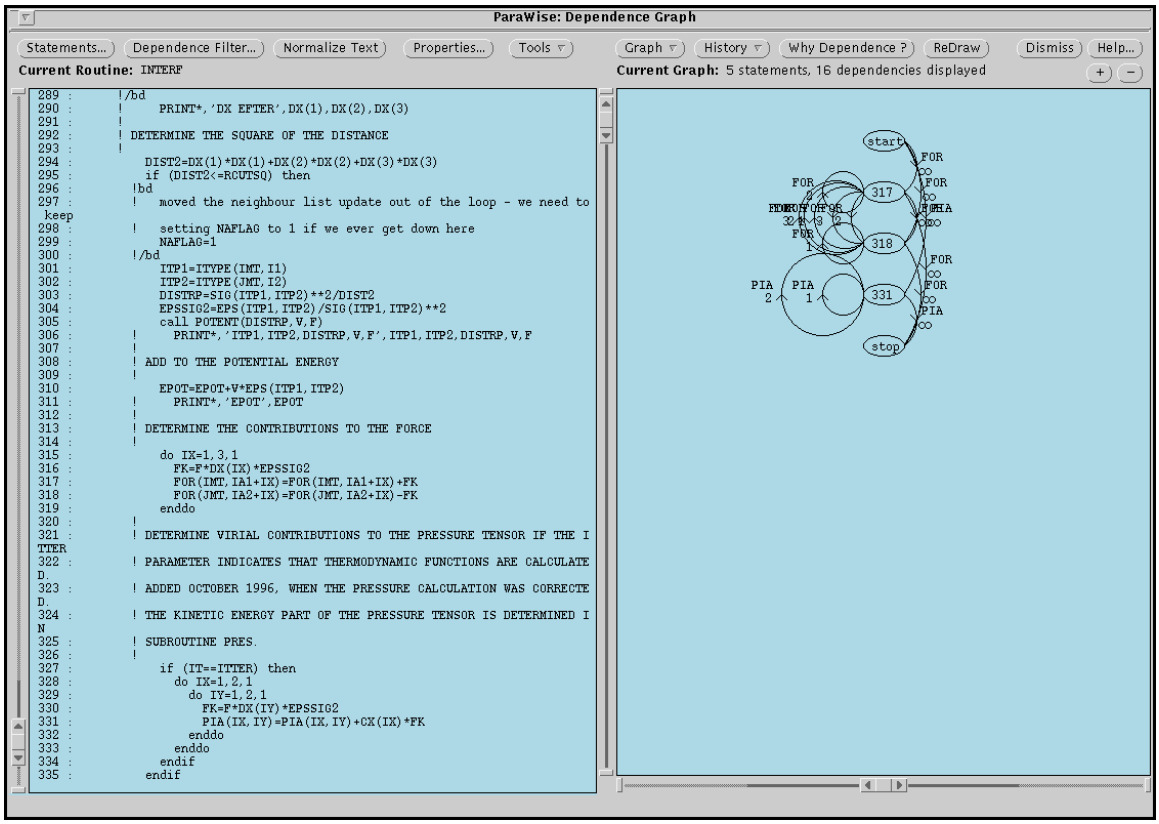

Figure 5 ParaWise Dependence Graph browser displaying dependencies of arrays *FOR* and *PIA* in routine *INTERF*

Within CAPO, the loop was selected and the Why Directive, *New_Type* option used. The loop type was set as a REDUCTION loop with both variables set as summation style reductions, as shown in Figure 6. The OpenMP code generated by CAPO was then edited manually.

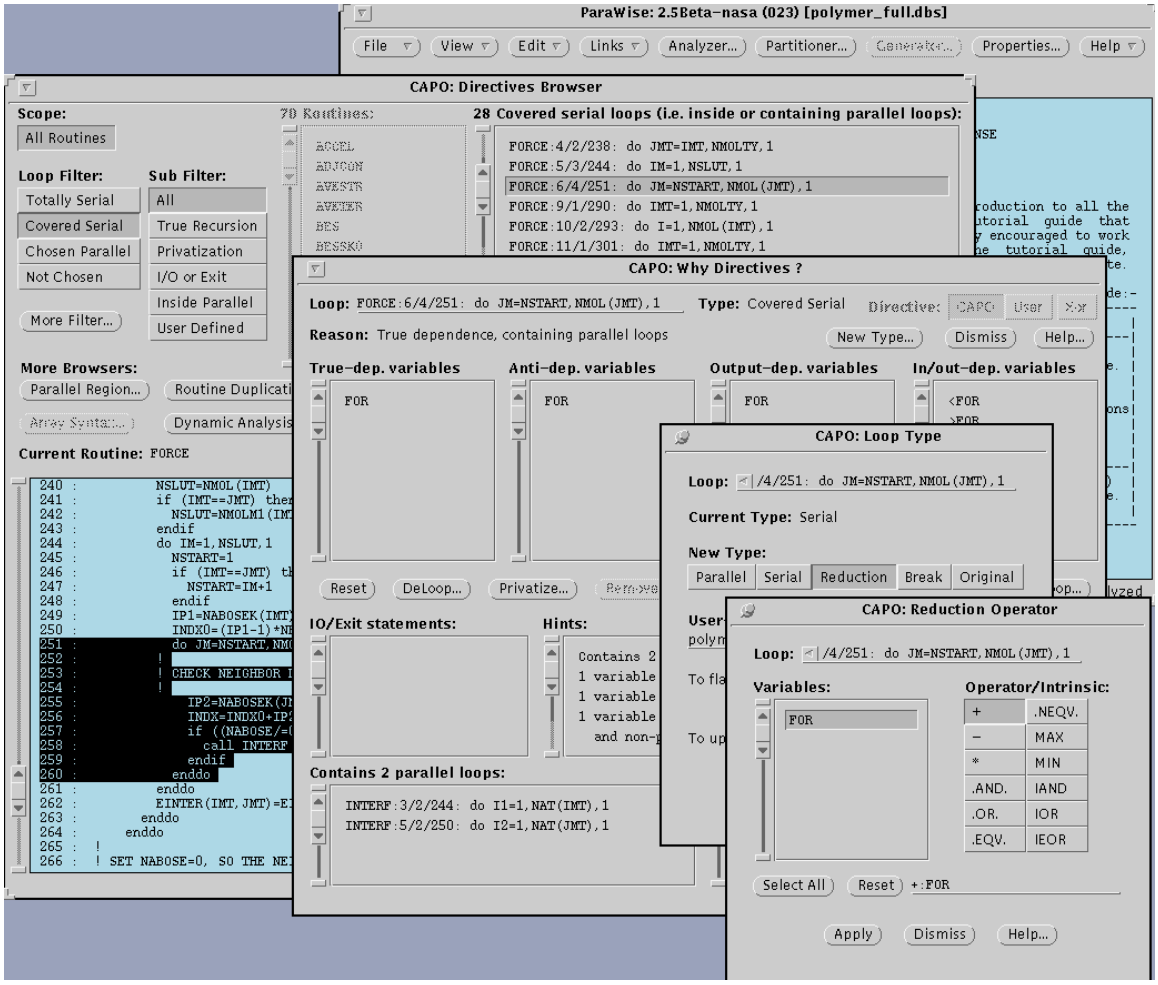

Figure 6 Using CAPO to force array *FOR* to be a summation style OpenMP reduction

The same technique was also used for the other loops in routine *FORCE* where for *INTRAF* and *BNDDIH*, only one or two other variables were listed in the CAPO Why Directive browser (relating to summations of a single value of an array). Evaluation of these variables could therefore by performed by using an OpenMP REDUCTION clause after being altered to use a dummy scalar in the routines that are used to set the array after the parallel region is complete. The final code section relating to routine *FORSUB* contained a larger number of inhibitors to parallelization, listed in CAPO, that all relate to the error handling and other library routines used. Almost all of these inhibitors were due to initialization of values on the first call to the associated routines. Removing these parallelization inhibitors causes each thread to initialize and use its own copy of these variables, although the full consequence of ignoring such dependencies has not been investigated. THREADPRIVATE directives were needed in these routines to cater for variables listed in SAVE statements to allow each thread to have its own private copy.

For the version using OpenMP to perform the array reduction of the *FOR* array, the initial version showed significant slowdown with most of the runtime spent summing the full declared range of the array, which far exceeds the range actually accessed in the

algorithm. To overcome this, the code was altered so that the declaration in routine *FORCE* related only to the accessed range.

Using the Sun Performance Analyzer for a 2 thread execution revealed a significant amount of load imbalance for this version, related to the loops in routine *FORCE*. With the STATIC OpenMP loop distribution used by default, the significant time attributed to the OpenMP synchronization clearly indicated load imbalance, so the use of DYNAMIC scheduling for these loops significantly improved the parallel performance.

The performance of this version of the code is shown in Table 3 .

Threads	1	2	4	6
Time (secs)	186	120	101	109
Speedup		1.55	1.84	1.71

Table 3 Performance of Polymer code using OpenMP array reductions on Isaac for 2x1000 iterations of the medium problem size

The speedup and scalability are disappointing so further versions of the code were developed based on manually incorporating contributions for each thread into a final summed array. These versions also considered both full and sparse accesses to determine the most effective strategy.

A new array *FOR_CAP* is introduced with matching dimensions to the original *FOR* array but with an extra final dimension to hold the related thread number. This was needed to allow each thread to have its local copy of the *FOR* array whilst also allowing the main thread to sum those contributions. This array is SHARED in the parallel region and passed into *INTERF* with the final dimension set to the thread number. Inside *INTERF*, the *FOR_CAP* array has the same dimension as the original *FOR* array and all references to *FOR* are replaced with accesses of *FOR_CAP*. The contributions from each thread are combined after all loops surrounding *INTERF* are complete, looping over all thread contributions stored in *FOR_CAP* and finally adding into the original *FOR* array.

The most efficient way to implement this is dependent on the nature of the summation, if every element of the array is set then a simple full array summation is suitable whilst if the accesses are sparse then an index based sparse matrix storage technique is far more effective (storing the value and the related array indices on each thread). Figure 7 shows an example of the changes made to the updates to the array *FOR* to implement the sparse array version. The *for_updates* array stores the accessed indices and *for_nupdates* the number of access made. The *for_flags* array is used to determine when a new location is being accessed so that the required initialization can be made, where this array is effectively reset between iterations of the Polymer algorithm by increasing the *for_counter* scalar. For the full array version, only the section of the array accessed must be initialized to zero prior to the loop nest and summed after the loop nest (not the declared size as this far exceeds the accessed area and can dominate runtime as mentioned earlier). Versions of Polymer using both techniques have been created.

```
do IX=1,3,1
FK=F*DX(IX)*EPSSIG2
FOR(IMT,IA1+IX)=FOR(IMT,IA1+IX)+FK
FOR(JMT,IA2+IX)=FOR(JMT,IA2+IX)-FK
enddo
```

```
      do IX=1,3,1
      FK=F*DX(IX)*EPSSIG2
      if (for_flags(imt,IA1+IX).ne.for_counter) then
        first touch so zero it now
        FOR_CAP(IMT,IA1+IX)=0
        for_flags(imt,IA1+IX)=for_counter
        for_nupdates=for_nupdates+1
        for_updates(for_nupdates,1)=IMT
        for_updates(for_nupdates,2)=IA1+IX
      endif
      FOR_CAP(IMT,IA1+IX)=FOR_CAP(IMT,IA1+IX)+FK
      if (for_flags(JMT,IA2+IX).ne.for_counter) then
        first touch so zero it now
        FOR_CAP(JMT,IA2+IX)=0
        for_flags(JMT,IA2+IX)=for_counter
        for_nupdates=for_nupdates+1
        for_updates(for_nupdates,1)=JMT
        for_updates(for_nupdates,2)=IA2+IX
      endif
      FOR_CAP(JMT,IA2+IX)=FOR_CAP(JMT,IA2+IX)-FK
      enddo
```

Figure 7 Original updates of *FOR* and manually conversion to sparse updates, initializing and storing index values for the first touch

Using the Sun Performance Analyzer showed that the compute time for the combination of the contributions to the *FOR* array from each thread becomes more significant as the number of threads increases. To reduce this overhead, two alternative algorithms for the parallel combination were tried. Firstly, a binary tree style summation was tried where pairs of threads combine their contributions until the global value is reached. The second version just used the threads to perform the loop adding contributions from one thread in parallel, but adding the contribution from each thread one at a time. Timing proved the second, simpler version was more efficient, where the overhead is constant as processors are added. This result is as expected since the time required for the binary tree version is proportional to $log_2(P)*N$ and for the linear version, time is proportional to $(N/P)*P=N$ where P is the number of threads and N is the size of the array. So as long as N is sufficiently large to outweigh the OpenMP runtime overheads, the linear version is superior.

Table 4 shows the performance of the full array access version of the code. The equivalent version assuming a sparse array access, but otherwise following the same parallelization as the full array version, produces the results in Table 5.

Threads	1	4	8
Time (secs)	186	86	91
Speedup		2.16	2.04

Table 4 Performance of Polymer code assuming a full array on Isaac for 2x1000 iterations of the medium problem size

Threads	1	4	8	10	16
Time (secs)	211	88	73	73	80
Speedup		2.4	2.89	2.89	2.64

Table 5 Performance of Polymer code using sparse storage techniques on Isaac for 2x1000 iterations of the medium problem size

The results show that, although the full array summation version is superior to the OpenMP array reduction version, it is not as scalable as the sparse array technique version. The overheads of the sparse array version are clearly seen in serial, and on 4 threads the full array version is slightly faster than the sparse array version. The 8 thread run, however, is far inferior and shows a performance fall for the full array version when compared to the 4 thread run. Examining the nature of the contributions to the *FOR* array in the 8 thread version showed between 20% and 30% of the contributions from all threads were not set, so the sparse array assumption is advantageous in compute time for accumulating the overall values. This advantage grows as more processors are used so the sparse array version clearly provides far more scalability than the other versions.

On closer inspection of the timings for these versions, it was seen that the overhead of the OpenMP DYNAMIC schedule algorithm was having a significant affect on performance, particularly on 16 threads. Testing a STATIC schedule and DYNAMIC schedules for varying chunk sizes for the routines in FORCE produced a dramatic improvement in runtime. Once this was identified, the next phase was a detailed tuning of the parallel code using the timeline display of the Sun Studio Performance Analyzer. Inspecting the causes of idle time in the timeline identified a few poorly performing loops, typically with little contained computation and only a few iterations. Also, the initialization of the *FOR* array in routine *FORCE* could be placed in the same parallel region as the calls to *INTERF* and a nowait could legally be used as the references were to *FOR_CAP* and not *FOR* in the code sections following initialization. The results for this optimized version are shown in Table 6. Results for a much larger data set with 224 C32 molecules are shown in Table 7. For the medium size data set, the best performance was obtained using a DYNAMIC schedule with a chunk size of 6. For the large test cases, the STATIC schedule provided the best performance with the DYNAMIC schedule with a small chunk size producing very poor performance on higher numbers of threads.

Threads	1	4	8	12	16
Time (secs) STATIC	186	105	72	69	68
Time (secs) DYNAMIC,1		72	63	63	62
Time (secs) DYNAMIC,6		70	60	59	59
Best Speedup		2.66	3.1	3.15	3.15

Table 6 Performance of Polymer code using sparse storage techniques on Isaac for 2x1000 iterations using a range of schedules of the medium problem size (DYNAMIC,6 was most successful)

Threads	1	4	8	12	16
Time (secs) STATIC	297	105	74	65	57
Time (secs) DYNAMIC,1		104	86	106	138
Best Speedup		2.86	4.01	4.57	5.21

Table 7 Performance of Polymer code using sparse storage techniques on Isaac for 1000 iterations using a range of schedules of the large problem size (STATIC was most successful)

These results show far superior speedup and scalability to the previous timings. As a final investigation the dependence of variable *NABO* in routine *FORCE* mentioned earlier that inhibited parallelization of the outer loop surrounding the calls to *INTERF* was re-visited. After the investigation of the related variables and runtime values, its influence was removed allowing parallelism in that outer loop to be exploited. For this version, a DYNAMIC schedule proved superior to STATIC and all non-unit DYNAMIC chunk size versions (i.e. better than DYNAMIC,2 etc.). The results for this version are shown in Tables 8 and 9 and demonstrate some speedup and scalability, particularly for the large test case. These final versions all incur overheads that proved to be less detrimental to performance than alternative strategies with their associated overheads. The OpenMP runtime overheads, load imbalance and the extra computation required for the sparse array approach all lessen performance, but provide better speedup and scalability than all other alternatives investigated here.

Threads	1	4	8	12	16
Time (secs)	186	85	60	50	52
Speedup		2.19	3.1	3.72	3.58

Table 8 Performance of Polymer code using sparse storage techniques for the outer loop parallelism in FORCE on Isaac for 2x1000 iterations using a DYNAMIC schedule of the medium problem size

Threads	1	4	8	12	16
Time (secs)	297	110	73	58	50
Speedup		2.7	4.01	5.12	5.94

Table 9 Performance of Polymer code using sparse storage techniques for the outer loop parallelism in FORCE on Isaac for 1000 iterations using a DYNAMIC schedule of the large problem size

Summary of Experience of using ParaWise/CAPO for the Scattering and Polymer Codes

For both application codes, the ParaWise/CAPO tool was used to perform most of the parallelization and provide a large amount of vital information in the improvment of those codes.

For the Scattering code, with the exception of the final minor code alterations, the entire parallelization was performed within ParaWise/CAPO, most of it being performed completely automatically relying on the inter-procedural, value based dependence analysis of ParaWise and the inter-procedural OpenMP parallelization performed by CAPO. The user interaction only related to a small proportion of the routines and loops in the application code and then only on a very small number of variables as the influence of the vast majority of variables on parallelism had been automatically discounted using dependence information. Using the Sun Performance Analyzer then identified which loops performed poorly in parallel allowing the user to simply force selection of other

loops that ParaWise/CAPO already knew were parallel using the CAPO browsers, or to change the scheduling algorithm used in the OpenMP loop distribution.

For the Polymer code, the deficiency in the CAPO array reduction detection algorithm (which is currently being addressed), requires user intervention at an earlier stage than should be necessary in future versions. Even then, the CAPO browsers provide a mechanism to force the related loops to operate in parallel with the associated variables evaluated using the OpenMP array REDUCTION facility. The remaining work related to optimizing the parallel code in an attempt to minimize the OpenMP runtime overheads and load imbalance, as the fairly small compute times in the parallel loops could easily be overwhelmed by these overheads. The index range of the arrays being considered in the summation is vital to achieve reasonable performance, where the default in OpenMP of using the declared range was not suitable in this case. Not relying on the OpenMP REDUCTION proved beneficial and usage of sparse array storage techniques becomes essential as larger numbers of threads are employed. As with the Scattering code, only a small proportion of the Polymer code required consideration by the user with most of the code being processed automatically.

Lessons on producing an efficient and scalable parallel Polymer code for Sun parallel systems have been gained from this experience. The basic loop parallel code produced by ParaWise/CAPO, with a small amount of user interaction, exploited the required parallelism and will lead to a reasonable performance when the parallel loops involve a larger amount of computation. It should be possible to include some of these sparse array techniques into the CAPO code generation in future versions.

Most of the time spent in this work was in debugging manually altered code for the Polymer application, as it is very easy to make mistakes in manual OpenMP code. No debugging was needed for the Scattering code, enabling its parallelization and tuning to be completed in a few hours.