Importing and Exploiting Parallel Execution Profile Information in the ParaWise Automatic Parallelization Environment

Pete Leggett[1], Steve Johnson[2] and Constantinos Ierotheou[1]

[1] Parallel Processing Research Group, University of Greenwich, London.
[2]Parallel Software Products

Abstract

An essential component of the ParaWise Automatic Parallelization Environment is the importation and exploitation of parallel performance execution profile information. To achieve this, an interface has been developed to enable information from Sun Studio Collect experiments to be imported into ParaWise. This information is used to indicate to the user which loops are inhibiting parallel performance and scalability. It is also used by the ParaWise Expert Assistant to automatically detect poorly performing loops and automatically identify loops that could more profitably be exploited in parallel, additionally providing recommendations for loops that would be better performed in serial and where alternative OpenMP loop schedules could be advantageous.

1 Introduction

The manual process of OpenMP [1] parallelization of an application code is a complex task, requiring expertise in a number of related areas if efficient and scalable code is to be produced. These skills include the detection of parallelism, the determination of variable privatization needed to legally exploit parallelism, the identification of where parallel regions can be merged, identifying where barrier synchronizations can be avoided, using tools to provide parallel execution profile information (such as Sun Studio [2], and Intel's VTune[3]), debugging OpenMP code (using tools such as the Totalview debugger [4]), along with tools, such as Intel's Thread Checker [5], to check the correctness of the OpenMP code before it can be reliably used in production. For real world application codes, consisting of tens of thousands of lines of source code and hundreds of routines, all these skills are tested and the management of complex interprocedural information can be extremely challenging. Very often, knowledge of the application code is vital in key decisions in the parallelization and so the code author is often also required to assist in this process.

This paper focuses on importing profile information into the ParaWise Automatic Parallelization Environment and how this information is used in the ParaWise browsers and to enhance the ParaWise Expert Assistant in making effective parallelization accessible to a wider audience.

2 The ParaWise Automatic Parallelization Environment

The aim of the ParaWise Automatic Parallelization Environment is to assist in all aspects of both the OpenMP and message-passing parallelization processes. The target users range from current parallelization experts with experience of OpenMP parallelization to code authors whose expertise lies in other areas of science and engineering and who do not wish to undertake the steep learning curve and significant effort required to manually produce efficient and scalable OpenMP code. It consists of a number of components that interact to simplify the parallelization processes as far as is possible.

Figure 1 shows the ParaWise automatic parallelization environment indicating all the components and how those components interact. The message passing and OpenMP directive generation components [6,7] have been developed over many years and are capable of producing efficient and scalable parallel code. The Expert Assistant [8] is designed to drive the interaction with the user, automatically identifying questions about the application code to be asked of the user in order to improve the performance of the generated parallel code. The Relative debugger [9] is still under development but is used to automatically identify where computations in the parallel and serial executions of the generated parallel code diverge, and subsequently interact with the Expert Assistant to identify the user decisions stored in the parallelization history that are potential causes of the erroneous computations. The final component, the parallel profiler, provides runtime performance information to the ParaWise user interface and the Expert Assistant to identify code sections that are important in improving efficiency and scalability.

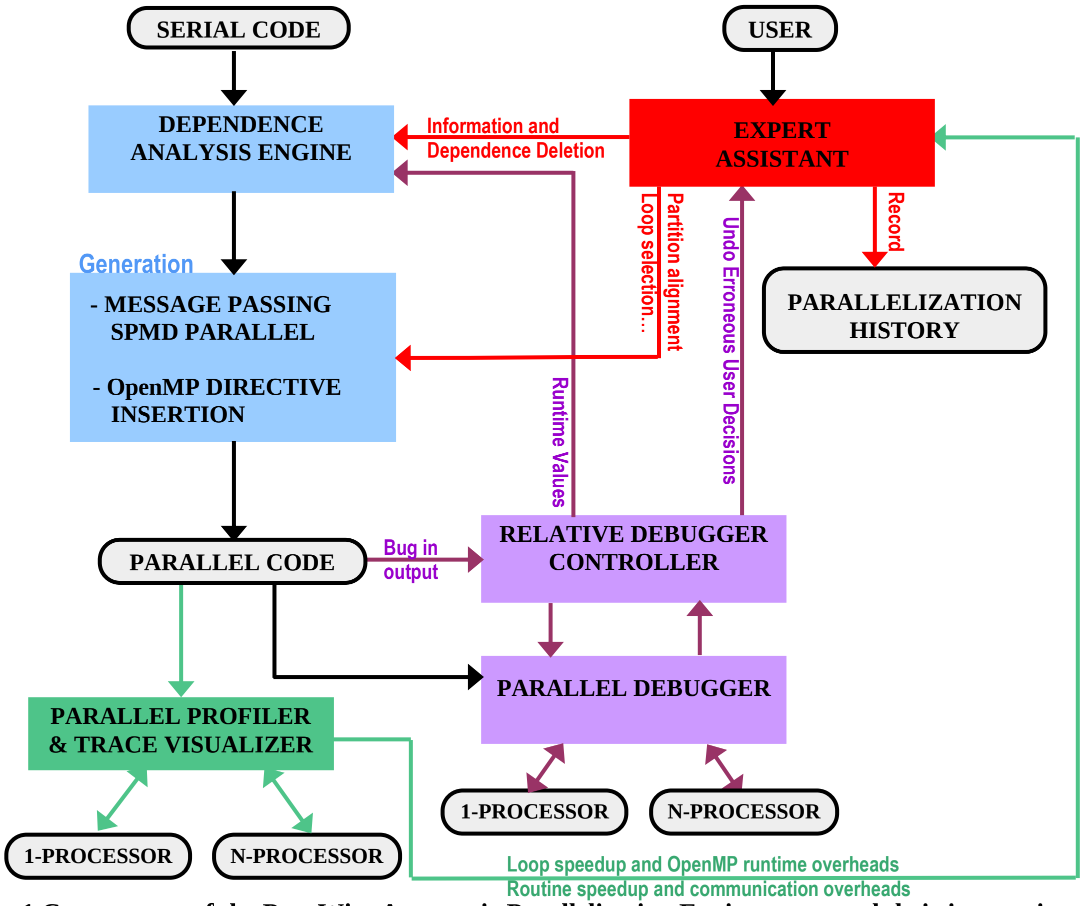

SERIAL CODE
USER
DEPENDENCE ANALYSIS ENGINE
Information and Dependence Deletion
EXPERT ASSISTANT
Loop selection...
Partition alignment
Undo Erroneous User Decisions
Record
PARALLELIZATION HISTORY
Generation
- MESSAGE PASSING SPMD PARALLEL
- OpenMP DIRECTIVE INSERTION
Runtime Values
PARALLEL CODE
Bug in output
RELATIVE DEBUGGER CONTROLLER
PARALLEL DEBUGGER
PARALLEL PROFILER & TRACE VISUALIZER
1-PROCESSOR
N-PROCESSOR
1-PROCESSOR
N-PROCESSOR
Loop speedup and OpenMP runtime overheads
Routine speedup and communication overheads

Figure 1 Components of the ParaWise Automatic Parallelization Environment and their interaction

To calculate useful metrics such as speed-up requires 1 and N processor run time execution data. The Sun Studio compiler and tool suite provides a number of utilities for collecting and examining experimental data from execution of the user's source code on any number of processors. In this paper, the Sun Studio *collect* and *er_print* tools [2] are used to collect and import runtime profile information into the ParaWise environment. The way this information is used in the ParaWise browsers and to further enhance the power of the Expert Assistant are detailed in later sections.

3 Importing Sun Studio Collect Experiments into ParaWise

The procedure for importing Sun Studio experiment data into ParaWise is as follows:-

1. Generate and save the OpenMP parallel code using ParaWise. We are assuming here that the original source code has already been parsed into ParaWise and both a dependence analysis and an initial OpenMP directives code generation was performed.
2. Compile this generated code using the Sun Studio compiler, e.g.
   ```
   % f90 -g -fast -xopenmp source_omp.f90 -o a.out
   ```
   The -g option is necessary to obtain source line information that is used later during the import.
3. The Sun Studio *collect* tool is then used to run the executable on 1 and N processors and generate two 'experiments' containing execution times for a serial and parallel run of the same executable, e.g.
   ```
   % setenv OMP_NUM_THREADS 1
   % collect -o proc1.er a.out
   % setenv OMP_NUM_THREADS 2
   % collect -o proc2.er a.out
   ```
4. The experiment data is then loaded into ParaWise using the import facility by specifying each experiment in turn together with whether the data is serial or parallel and the number of processors used.

Sun Studio provides the *er_print* experiment reporting tool [2] which can be run interactively or in batch mode to interrogate experiment data. The tool offers a number of different commands to output experiment information in a textual readable format. The import of an experiment into ParaWise is carried out by forking a sub-process of ParaWise which in turn executes the *er_print* tool, enabling the two processes to establish a Unix pipe connection connecting the stdin and stout of *er_print* with ParaWise file descriptors. ParaWise interrogates *er_print* as required, sending commands to *er_print* and receiving textual information back through the pipe. The main *er_print* commands used to import the data are *psummary*, *thread_list*, *thread_select*, *metrics* and *source*. *Psummary* is used as a checksum to validate that the experiment data matches the source code currently loaded into ParaWise and can therefore be imported successfully. The *thread_list* command is used to identify the correct threads within the experiment that corresponds to the execution of the user program. This is necessary since the experiment can also contain thread data for hidden 'system' threads used to initialise the OpenMP parallelization etc.

Experimental data for each individual thread for each individual statement where data is available is imported into ParaWise using the *er_print thread_select* and *source* commands. The *thread_select* command tells *er_print* to select a particular thread's run time data, whilst the *source* command outputs all run time data for a particular Fortran routine. To correlate the imported *collect* run-time profile data with ParaWise routine, loop and statement data structures, a number of techniques are used to ensure that data is valid and correctly aggregated where necessary.

4 Basic Use of Profile Data

Once profile data has been imported, the user can examine the speedup and other metrics for routines and loops within the ParaWise browser windows. Speedup and other timing metrics are displayed in re-sizable columns in any relevant GUI list, i.e. lists showing routine names or loop heads. The lists can be easily sorted by any displayed metric by clicking on the list column. Columns can also be re-ordered or re-sized horizontally by dragging with the mouse.

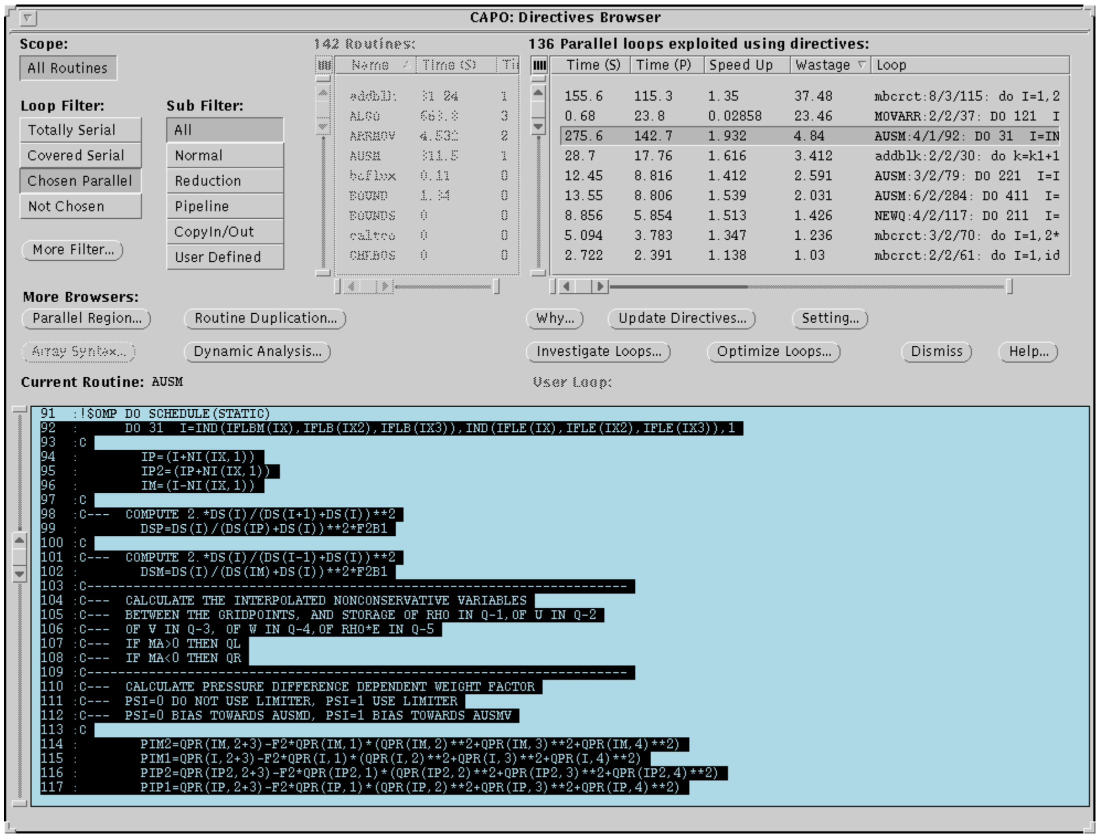
The basic requirement from imported execution profile data is to allow the user to identify poorly performing loops so that they can attempt to overcome the related issues. None of the standard metrics of serial time, parallel time and speedup provide a clear indication of where the user should focus their attention. What the user needs to know is which loops have the greatest potential to improve parallel performance. To this end, another metric, wastage, that indicates how much processor time is wasted in the loops, is used. We define wastage W, using the following formula:-

$$W = Tp - Ts / N$$

where N is number of parallel threads, Tp is parallel time and Ts is serial time.

A loop with a high wastage may be a loop with a high execution time that is exhibiting speedup, but not perfect speedup. Also, a loop with a lower execution time will still have a high wastage if it exhibits very poor speedup. Loops with high speedup will have a low wastage (as there is little more to be gained) and loops with a poor speedup but that only require a small amount of execution time have a low wastage as it will have little impact on performance even if perfect speedup could be achieved.

Sorting loops lists based on descending wastage then shows the loops that should be investigated at the head of the list. Figure 2 shows the ParaWise Directive browser with parallel loops sorted by decreasing wastage. The loop with the highest wastage has a significant runtime, but only shows a speedup of 1.35 out of 2, so improving this speedup would have the most significant affect on overall performance. The second loop has a very low serial execution time, but exhibits significant slowdown (due to OpenMP runtime overheads) in parallel, so improving or serializing this loop can have a large affect on overall performance. The third loop in the list has a very significant execution time which causes it to have a relatively high wastage despite the good speedup of 1.932 as achieving perfect speedup would improve overall performance.



142 Routines:

Name	Time (S)	Ti
addblk	31.24	1
ALGO	663.8	3
ARRMOV	4.532	2
AUSM	311.5	1
bcflux	0.11	0
BOUND	1.34	0
BOUNDS	0	0
caltco	0	0
CHKBOS	0	0

136 Parallel loops exploited using directives:

Time (S)	Time (P)	Speed Up	Wastage	Loop
155.6	115.3	1.35	37.48	mbcrct:8/3/115: do I=1,2
0.68	23.8	0.02858	23.46	MOVARR:2/2/37: DO 121 I
275.6	142.7	1.932	4.84	AUSM:4/1/92: DO 31 I=IN
28.7	17.76	1.616	3.412	addblk:2/2/30: do k=k1+1
12.45	8.816	1.412	2.591	AUSM:3/2/79: DO 221 I=I
13.55	8.806	1.539	2.031	AUSM:6/2/284: DO 411 I=
8.856	5.854	1.513	1.426	NEWQ:4/2/117: DO 211 I=
5.094	3.783	1.347	1.236	mbcrct:3/2/70: do I=1,2*
2.722	2.391	1.138	1.03	mbcrct:2/2/61: do I=1,id

Current Routine: AUSM

```
:!$OMP DO SCHEDULE(STATIC)
:      DO 31  I=IND(IFLBM(IX),IFLB(IX2),IFLB(IX3)),IND(IFLE(IX),IFLE(IX2),IFLE(IX3)),1
:C
:        IP=(I+NI(IX,1))
:        IP2=(IP+NI(IX,1))
:        IM=(I-NI(IX,1))
:C
:C---  COMPUTE 2.*DS(I)/(DS(I+1)+DS(I))**2
:        DSP=DS(I)/(DS(IP)+DS(I))**2*F2B1
:C
:C---  COMPUTE 2.*DS(I)/(DS(I-1)+DS(I))**2
:        DSM=DS(I)/(DS(IM)+DS(I))**2*F2B1
:C--------------------------------------------------------------------
:C---  CALCULATE THE INTERPOLATED NONCONSERVATIVE VARIABLES
:C---  BETWEEN THE GRIDPOINTS, AND STORAGE OF RHO IN Q-1,OF U IN Q-2
:C---  OF V IN Q-3, OF W IN Q-4,OF RHO*E IN Q-5
:C---  IF MA>0 THEN QL
:C---  IF MA<0 THEN QR
:C--------------------------------------------------------------------
:C---  CALCULATE PRESSURE DIFFERENCE DEPENDENT WEIGHT FACTOR
:C---  PSI=0 DO NOT USE LIMITER, PSI=1 USE LIMITER
:C---  PSI=0 BIAS TOWARDS AUSMD, PSI=1 BIAS TOWARDS AUSMV
:C
:        PIM2=QPR(IM,2+3)-F2*QPR(IM,1)*(QPR(IM,2)**2+QPR(IM,3)**2+QPR(IM,4)**2)
:        PIM1=QPR(I,2+3)-F2*QPR(I,1)*(QPR(I,2)**2+QPR(I,3)**2+QPR(I,4)**2)
:        PIP2=QPR(IP2,2+3)-F2*QPR(IP2,1)*(QPR(IP2,2)**2+QPR(IP2,3)**2+QPR(IP2,4)**2)
:        PIP1=QPR(IP,2+3)-F2*QPR(IP,1)*(QPR(IP,2)**2+QPR(IP,3)**2+QPR(IP,4)**2)
```

Figure 2 ParaWise Directive browser sorting loops by wastage

5 The ParaWise Expert Assistant

The ParaWise Expert Assistant can be used to automatically investigate why a loop has not been able to exploit parallel execution [8]. It is instigated by the user, either investigating a single loop or a list of loops, and presents a series of questions to the user that relate only to the nature of the application code. These include questions on constraints on variables read into the application code at runtime, questions on dataflow in the algorithms implemented in the application code and also questions about the importance of I/O statements (e.g. are they only for debugging purposes and so should not inhibit parallelization). The questions are derived from an investigation of dependencies in the application code that represent inhibitors to parallelism, with the investigation operating interprocedurally so that relevant questions for dependencies throughout the application code can be identified for a single loop. If the user can answer some of the questions then the Expert Assistant can store that information so that an automatic replay of the parallelization can exploit those decisions to generate a superior parallel version.

Currently, the user has to select the loop or loops to investigate. With the addition of execution profile data, the functionality of the Expert Assistant can be improved by, in particular, automatically identifying important loops that required investigation and ignore loop when it is clear that their parallel execution will not achieve scalability.

6 Enhancing the ParaWise Expert Assistant using Profile Data

To use the execution profile information to identify which loops can effectively exploit parallel execution (and also eliminate those loops that cannot be effective sources of OpenMP parallelism), a simple model of the time of a loop computation is used where estimates of time components can be extracted from imported timings. This model considers a loop that contains both parallel and serial loops, where this information can be used to determine if parallel operation of those contained loops or parallel operation of the loop under consideration itself is preferable. The simple timing model for a serial loop has components accumulated from inner loops (where i indicates the inner loop) :-

- Calculation time of parallel DO loops inside this loop (Tp_i)
- OpenMP overheads from parallel DO loops inside this loop (To_i)
- Calculation time of loops executing in serial inside this loop that could execute in parallel or that contain other loops that could execute in parallel (Ts_i)
- Calculation time of code not contained in any parallelizable loop (e.g. any inherently serial loop nest or code only nested in iterative loops, DO loops with exits and code not within any inner loops). Initially, any DO loop that does not contain an exit is not included here, all other loops are included. (Tc_i)

Figure 3 shows a loop nest where loop *I* is nested within an outer loop and contains a number of inner loops along with their time components.

```
DO OUTER=L1,H1                              → SERIAL LOOP (COULD BE PARALLEL)
        DO I=L2,H2                          → SERIAL LOOP (COULD BE PARALLEL)
                . . . .                     Tp=Tp1+Tp2+Tp3+Tp4=10.0,To=0.2,Ts=13.0,Tc=10.0
                . . . .
                DO INNER1=IL1,IH1           → PARALLEL LOOP
                        . . . .             Tp1=5.0.To1=0.1,Ts1=0,Tc1=0
                ENDDO
                DO INNER2=IL2,IH2           → DEFINITELY SERIAL LOOP (CONTAINS EXIT)
                        IF (C1) EXIT        Tp2=0.To2=0,Ts2=0,Tc2=10.0
                ENDDO
                DO INNER3=IL3,IH3           → ASSUMED SERIAL LOOP (COULD BE PARALLEL)
                        . . . .             Tp3=0.To3=0,Ts3=10.0,Tc3=0
                ENDDO
                DO INNER4=IL4,IH4           → ASSUMED SERIAL LOOP (COULD BE PARALLEL)
                        . . . .             Tp4=Tp41+Tp42=5.0.To4=To41+To42=0.1,Ts4=3.0,Tc4=0
                        . . . .
                        DO INNER41=IL41,IH41 → PARALLEL LOOP
                                . . . .     Tp41=5.0.To41=0.1,Ts41=0,Tc41=0
                        ENDDO
                        DO INNER42=IL42,IH42 → ASSUMED SERIAL LOOP (COULD BE PARALLEL)
                                . . . .     Tp42=0.To42=0,Ts42=3.0,Tc42=0
                        ENDDO
                ENDDO
        ENDDO
ENDDO
```

Figure 3 Loop nest showing parallel and serial loop time components and their accumulation to compute components for outer loops.

In this way, the estimated execution time of each loop can be calculated by summing the times from each of the inner loops. When a loop exhibits poor parallel performance, the determination of which of these factors is the main cause of this inefficiency can be used to direct the optimization process:-

- ❖ If *Ts* is the major term for a loop, then effort in exploiting parallelism in the inner serial loops may be advantageous.

- If *Tc* is the major term for a loop, then the only way to overcome this is to exploit parallelism between iterations of this loop or a loop that surrounds this loop.
- If *To* is the most dominant term for a loop, then an investigation of the overheads from inner parallel regions/loops may provide an improvement or, alternatively, parallelization of the loop itself or an outer loop would overcome this problem.
- If *Tp* is the most dominant term then, as long as the inner parallel loops exhibit high parallel speedup, then the performance for this loop as a whole may be deemed as acceptable, allowing the user's attention to be focused elsewhere.

For a loop whose iterations are executed in parallel using an OpenMP directive, the time for that loop consists of the time for the slowest thread added to the OpenMP overhead (*To*) accrued in starting up and shutting down the parallel threads, load imbalance etc. Therefore, *Ts* and *Tc* are zero, *To* is estimated from timings and *Tp* is calculated using:-

$$Tp = \operatorname*{Max}_{j=1}^{NTHREADS} (T_j)$$

where T_j is the calculation time for thread j. A measure of the variation in the thread time is also stored to give an indication of load imbalance. The serial loop time is obviously directly measured. For nested loops, the components from each loop at the next inner level are combined to provide the overall time of the loop as shown in Figure 3.

$$T = \sum_{i=1}^{N} Tp_i + \sum_{i=1}^{N} To_i + \sum_{i=1}^{N} Ts_i + \sum_{i=1}^{N} Tc_i$$

where N is the number of inner loops at the next level. This process is recursive so that each outer loop at any level also has the four components that are summed from the components from inner loops.

Simple rules can then be used to determine which loop(s) in a loop nesting need to execute in parallel due to significant inherently non-parallelizable contained code (*Tc*). For serial loops that do not currently contain any significant non-parallelizable code, but do contain currently serial loops, investigation with the Expert Assistant in an attempt to uncover parallelism can be used to convert the serial loop to be a parallel loop and therefore alter its contributions so that *Tp* is set to *Ts*/*NTHREADS* and *Ts* is set to zero. If a loop currently executing in serial cannot be parallelized after the Expert Assistant investigation (including recursively investigating loops contained within that loop) then it is deemed to be an inherently serial loop and its *Tc* is set to *Ts* and *Ts* is reset to zero. As soon as *Tc* becomes significant, parallelism from inner loops cannot overcome this serial component so parallelism at this loop level (the *I* loop in Figure 3) must then be considered. This loop, in-turn, is examined using the Expert Assistant to attempt to allow it to execute in parallel where the process can continue processing outer loops (such as loop *OUTER* in Figure 3) until sufficient parallelism is achieved or all outer loops have been inspected.

Another measure from the parallel profile is the variation in the thread timing of a parallel loop. If the performance of the loop is poor and the timings are significantly different then a DYNAMIC OpenMP loop schedule can be tried to improve performance. When this is done the loop components are altered optimistically so that *To* is set to zero.

If such a loop already uses a DYNAMIC OpenMP loop schedule then, if it contains loops that could execute in parallel, the current loop itself is forced to execute in serial allowing parallelism to be exploited by the inner loops. In some cases, the recommendation of the Expert Assistant may be to perform the entire loop nest in serial as the OpenMP runtime overheads are too significant and no route to effective parallelization could be found.

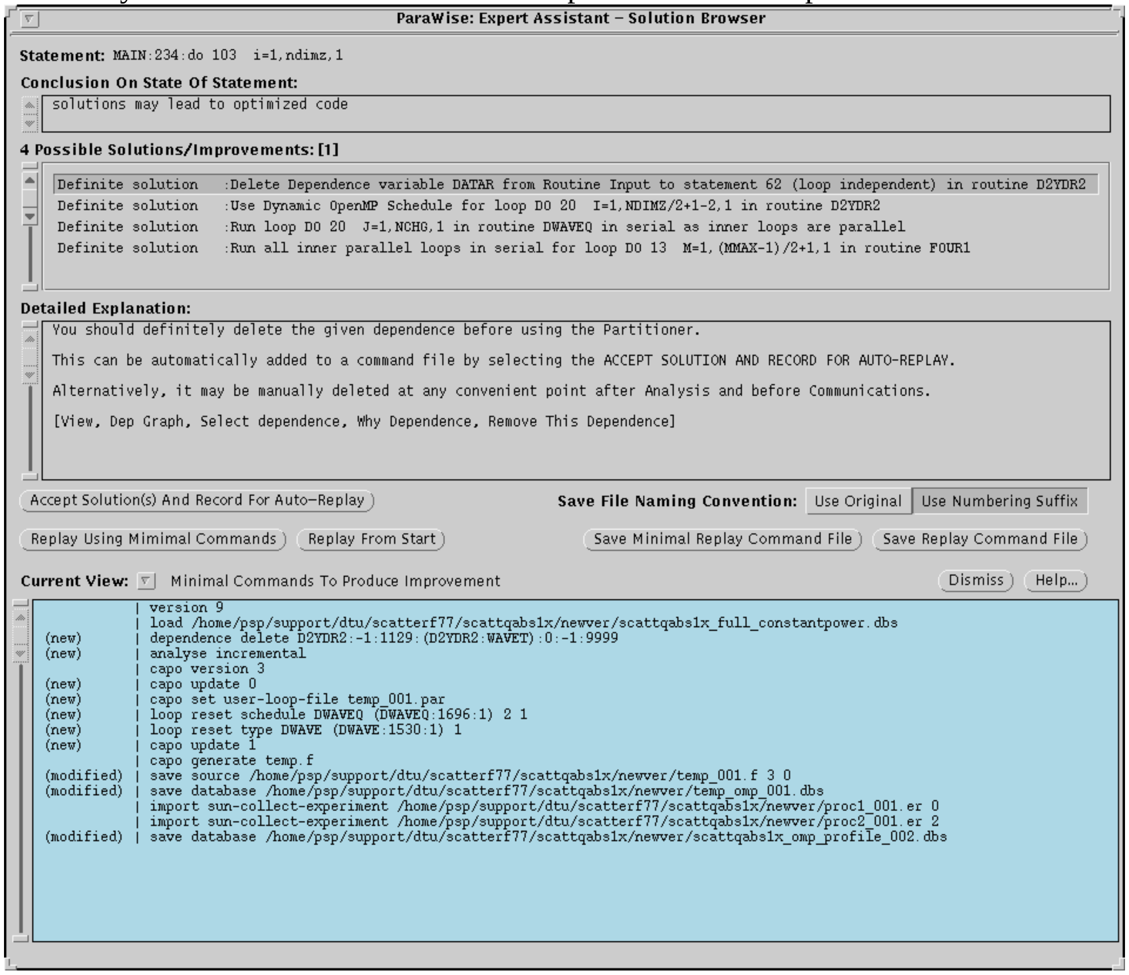
These decisions using profile information are all automatically made within the Expert Assistant so that the user is just presented with a series of questions about the variables and algorithms in the application code. The user's answers to these questions obviously affect the progress of the Expert Assistant investigation, but without any explicit direction from the user. Any loop that has trivial wastage is not considered for any investigation, reducing the number of questions presented to the user. As many loops in the application code are effectively parallelized in the first automatically parallelized version of the application code, those loops and all inner and outer loops in their loop nests are not considered for investigation and so no questions relating to them are presented to the user.

The conclusions reached by the Expert Assistant that are presented for acceptance by the user include all those based on user responses as well as automatically reached conclusions based on the profile information. Figure 4 shows the Expert Assistant Solution Browser window including some automatic conclusions. The conclusions or suggestions are presented in the "Possible Solutions/Improvements" list. The user can select each suggestion individually and click the "Accept Solution(s) And Record For Auto-Replay" button to accept the proposed suggestion. Accepting a suggestion includes it in the automatic replay of the parallelization process so that the new version of the application will include all accepted alterations and exhibit superior parallel performance.

7 Case Study of Expert Assistant Using Parallel execution Profile Information

This extension to the Expert Assistant has been used in the parallelization of a real world application code from the Danish Technical University that models the scattering of helium particles. This code was previously parallelized using ParaWise to produce an efficient and scalable parallel version [11]. This exercise was to repeat the parallelization using the Expert Assistant exploiting parallel execution profile information to see how close the decisions and performance of the previous version could be matched.

The initial ParaWise parallelization was executed on 1 and 2 processors and the resultant Sun Collect experiments imported into ParaWise. The Optimize button on the Directive Browser window was then clicked to invoke the Expert Assistant and a series of 30 questions were then asked of the user, most of which could not be given positive responses. Two questions, however, could be given positive answers as the dataflow in question did not carry any values as another initialization provided all the values used in the algorithm, so the user's knowledge of the application code or examination of the relevant data accesses enabled a positive answer. These responses uncovered parallelism in an outer loop, where inefficient parallelism from inner loops had previously been used, and were presented to the user in the Expert Assistant Solution Browser. Additionally, automatic conclusions about serialization and using DYNAMIC OpenMP schedules were produced as shown in Figure 4. Accepting these conclusions and replaying the parallelization produced a far superior parallel version exhibiting some speedup and scalability when the resultant Sun collect experiments were imported into ParaWise.


ParaWise: Expert Assistant – Solution Browser
Accept Solution(s) And Record For Auto–Replay
Replay Using Mimimal Commands
Replay From Start
Save Minimal Replay Command File
Save Replay Command File
Dismiss
Help...

Statement: `MAIN:234:do 103  i=1,ndimz,1`

Conclusion On State Of Statement:

```
solutions may lead to optimized code
```

4 Possible Solutions/Improvements: [1]

```
Definite solution   :Delete Dependence variable DATAR from Routine Input to statement 62 (loop independent) in routine D2YDR2
Definite solution   :Use Dynamic OpenMP Schedule for loop DO 20  I=1,NDIMZ/2+1-2,1 in routine D2YDR2
Definite solution   :Run loop DO 20  J=1,NCHG,1 in routine DWAVEQ in serial as inner loops are parallel
Definite solution   :Run all inner parallel loops in serial for loop DO 13  M=1,(MMAX-1)/2+1,1 in routine FOUR1
```

Detailed Explanation:

```
You should definitely delete the given dependence before using the Partitioner.

This can be automatically added to a command file by selecting the ACCEPT SOLUTION AND RECORD FOR AUTO-REPLAY.

Alternatively, it may be manually deleted at any convenient point after Analysis and before Communications.

[View, Dep Graph, Select dependence, Why Dependence, Remove This Dependence]
```

Save File Naming Convention: Use Original [✓ Use Numbering Suffix]

Current View: Minimal Commands To Produce Improvement

```
           | version 9
           | load /home/psp/support/dtu/scatterf77/scattqabs1x/newver/scattqabs1x_full_constantpower.dbs
(new)      | dependence delete D2YDR2:-1:1129:(D2YDR2:WAVET):0:-1:9999
(new)      | analyse incremental
           | capo version 3
(new)      | capo update 0
(new)      | capo set user-loop-file temp_001.par
(new)      | loop reset schedule DWAVEQ (DWAVEQ:1696:1) 2 1
(new)      | loop reset type DWAVE (DWAVE:1530:1) 1
(new)      | capo update 1
           | capo generate temp.f
(modified) | save source /home/psp/support/dtu/scatterf77/scattqabs1x/newver/temp_001.f 3 0
(modified) | save database /home/psp/support/dtu/scatterf77/scattqabs1x/newver/temp_omp_001.dbs
           | import sun-collect-experiment /home/psp/support/dtu/scatterf77/scattqabs1x/newver/proc1_001.er 0
           | import sun-collect-experiment /home/psp/support/dtu/scatterf77/scattqabs1x/newver/proc2_001.er 2
(modified) | save database /home/psp/support/dtu/scatterf77/scattqabs1x/newver/scattqabs1x_omp_profile_002.dbs
```

Figure 4 The ParaWise Expert Assistant Solution Browser , showing conclusions from user decisions and also conclusions automatically formed based on imported profile information.

To further investigate the scalability of the parallel version, a parallel execution on 8 threads was performed and the experiment information produced was loaded into ParaWise. A repeat of the Optimization using the Expert Assistant led to only automatic conclusions being reached, some of which relate to scalability. When the conclusions are accepted and the parallelization is automatically repeated, significant and scalability are displayed. Continuing the process to larger numbers of processors, further scalability inhibitors can be automatically detected and addressed.

The final parallel version produced using the Expert Assistant coupled with parallel execution profile information was close to the previously parallelized version. The loops chosen for parallel execution are identical in the two versions and DYNAMIC schedules are applied to the same loops in both. Only the final manual optimization used in the previous parallelization were not included in the new version, so it attained a reasonable speedup of 10.57 on 20 processors.

This demonstrates that an effective parallelization can be produced using the Expert Assistant including profile information without requiring any insight into parallelism or OpenMP of the user. The user's role was just to perform the Sun collect experiments, import the experiments, click the Optimize button, answer a series of questions about the application code, accept conclusions and finally press the replay button to automatically repeat the parallelization.

8 Conclusion

Serial and parallel execution run time profile data has been imported into ParaWise and used to identify and target loops where efficient parallelization is most important. The use of profile data has been shown to not only allow the user to focus their efforts more efficiently, but more importantly, it can be used by the ParaWise Expert Assistant to automatically identify the most effective loops in a nest to parallelize, skipping the investigation of loops as soon as their parallel execution is deemed as ineffective in achieving efficient parallel execution and scalability. It also offers optimizations such as DYNAMIC scheduling that are not possible without profile information.

In future work, we expect to expand on the use of profile data in ParaWise to identify and investigate other OpenMP runtime overheads (such as barrier synchronizations when either parallel region merger or the use of a NOWAIT has not been deemed possible) as well as adding facilities to import from other run time instrumentation data formats.

9 References

1 OpenMP, http://www.openmp.org.
2 SUN Studio, http://developers.sun.com/prodtech/cc/index.jsp.
3 Vtune, http://www.intel.com/vtune/.
4 Totalview, http://www.etnus.com/.
5 Thread Checker, http://www.intel.com/.
6 E W Evans, S P Johnson, P F Leggett, M Cross "Automatic and effective multi-dimensional parallelisation of structured mesh based codes.", Parallel Computing 26(6): pp 677-703, 2000.
7 H Jin, M Frumkin, J Yan "Automatic generation of OpenMP directives and its application to computational fluid dynamics codes" International symposium on High Performance Computing, Tokyo, Japan, 2000, LNCS 1940, pp 440-456, 2000.
8 S Johnson, E Evans, H Jin and C Ierotheou, "The ParaWise Expert Assistant – Widening Accessibility to Efficient and Scalable Tool Generated OpenMP Code" Proceedings of WOMPAT 2004, Houston, Texas, USA, May 2004.
9 G Matthews, R Hood, H Jin, S Johnson and C Ierotheou "Automatic Relative Debugging of OpenMP Programs", Proceedings of EWOMP 2003, Aachen, Germany, September 2003.
11 Steve Johnson and Cos Ierotheou, Parallelization of the DTU Scattering and Polymer Application Codes using the ParaWise/CAPO tools. Technical Report TR-2005-09-01, http://www.parallelsp.com, 2005.