Tools for OpenMP Application Development: The POST Project

L. Adhianto[a], F. Bodin[b], B. Chapman[c,a], L. Hascoet[d]
A. Kneer[e], D. Lancaster[a], I. Wolton[a], M. Wirtz[e]

[a] Electronics & Computer Science, University of Southampton, U.K.

[b] IRISA, University of Rennes, Rennes, France

[c] Computer Science, University of Houston, USA

[d] Simulog SA, Sophia Antipolis, France [e] Battelle GmbH, Eschborn, Germany

Abstract

OpenMP was recently proposed by a group of vendors as a programming model for shared memory parallel architectures. The growing popularity of such systems, and the rapid availability of product-strength compilers for OpenMP, seem to guarantee a broad take-up of this paradigm if appropriate tools for application development can be provided. POST is an EU-funded project that is developing a product, based on FORESYS from Simulog, which aims to reduce the human effort involved in the creation of OpenMP code. Additional research within the project focuses on alternative techniques to support OpenMP application development that target a broad variety of users. Functionality ranges from fully automatic strategies for novice users, the provision of parallelisation hints, and step-by-step strategies for porting code, to a range of transformations and source code analyses that may be used by experts, including the ability to create application-specific transformations. The work is accompanied by the development of OpenMP versions of several industrial applications.

Keywords: program analysis, restructuring, parallelisation, FORTRAN, program transformations, program development environment, OpenMP, expert system

1 Introduction

Most major hardware vendors market cost-effective parallel systems in which a modest number of processors share memory. Such shared memory parallel workstations and PCs (SMPs) are being increasingly deployed, not only as stand-alone computers, but also in workstation clusters. They are also used to populate the nodes of distributed memory machines such as IBM's SP-2, the Compaq/QSW AlphaServer SC and the QSW QM-1. The SGI Origin 2000 and HP SPP systems provide cache coherency across their SMP nodes and have very low latency. Users of such ccNUMA machines are encouraged to program the entire system as if it were a large SMP.

OpenMP was developed as a high-level programming model for creating applications that are able to exploit the processing power of SMPs, and it is also implemented on ccNUMA systems. It promises to provide ease of development together with good performance on this range of architectures. Although it is indeed much easier to write OpenMP code than, say, message-passing parallel code, it is not necessarily a straightforward task to create code that makes good use of cache and exploits a sufficient amount of parallelism. Some difficult porting problems remain. Interactive tools may not only help produce a high-quality translation of an existing program to an OpenMP version, they also have the potential to significantly reduce the human effort involved in the parallelisation process.

Within the EU-funded POST project (Programming with the OpenMP Standard), we are creating a programming environment to facilitate application development for the range of systems described above. When completed, the POST environment will be highly interactive and will cooperate with the user to develop code rather than simply providing a standardized automatic translation. POST helps the user analyze, and possibly modify, an existing program before adapting it using a conventional automatic OpenMP code generation strategy; yet it also applies alternative techniques, based in part upon examples of successful code parallelisation, to cooperate with the user in the identification and selection of a strategy for translating to OpenMP. Within this paper we describe the features available within the tools, and illustrate our approach with reference to OpenMP loop parallelisation, using code fragments from the application development work within the project. Given that OpenMP is not currently sufficient for the full range of machines described, we should point out that the the POST environment is also able to deal with combinations of MPI and OpenMP in a code.

This paper is organised as follows. In the next section we give a brief overview of the project tools and the applications that are being developed as part of the POST project. The application developers are supporting the tool creation work by providing feedback on their needs and suggesting functionality, as well as improvements to existing features.

In Section 3 we describe each of the components of the toolset, showing how they help solve porting problems encountered during the project application development. These range from the commercial reverse-engineering software product FORESYS from Simulog, at the heart of this development, to research activities that search for improved means of user support. Related work is briefly described in Section 4.

The paper concludes with a summary of related work and future prospects.

2 Project Overview

The toolset in POST is a highly interactive environment developed to support OpenMP parallelisation. It permits the user to study an existing program, obtain an automatically generated OpenMP code, manually insert OpenMP directives, or obtain analysis results from all or part of the program within a single environment.

Expert users may write code replacement "transformations" (these are not necessarily legal code transformations in the conventional sense) in a special scripting language, and apply these as desired throughout the program. The insights used to restructure a loop may also be documented and saved, so that they may be made available to other users. We are creating a database of loop structures found in application codes, together with a description, as well as code, to record successful adaptation strategies. These can then be used to provide hints on possible code adaptations in a custom context-sensitive Help facility. The successful adaptations also serve as material for an experimental Case Based Reasoning system which aims to help the user recognise similar loop structures in other codes and provide assistance in transforming them.

A User Guidance Module is under development that will provide both extended transformation capabilities as well as guidance and explanation features to support the OpenMP parallelisation process. It is the subject of Section 3.3. The Guidance Module is invoked from within FORESYS, and is based upon the functionality of both FORESYS and TSF. Each of the components of the POST environment, FORESYS, TSF and the Guidance Module, provides a set of features to parallelise code under OpenMP; together, they provide options for beginners and advanced users, and parallelisation approaches that are able to handle source code at varying levels of complexity and with different levels of automation in the adaptation process.

The POST project includes three industrial partners who are parallelizing Fortran 77-based commercial strength codes using OpenMP. We have relied on these three efforts to provide us with insight into the requirements of OpenMP code developers, as well as with material to build one of the toolset's modules. All of the programs contain significant regions that are not amenable to conventional automatic parallelisation.

Comet from Battelle is a multi-purpose Computational Fluid Dynamics (CFD) solver that models combustion as well as flows and is well adapted for use with complex boundary geometries. It is a self-contained package incorporating mathematical models of a wide range of thermo-fluids and solids phenomena, in both compressible and incompressible flow regimes. The numerical routines are based on a flexible finite volume approach using unstructured meshes. Preliminary benchmarks for the OpenMP version indicate good scaling on the size of systems that have been targeted. We use examples of loops in the Comet code throughout the remainder of this paper to illustrate the components of our toolkit, their capabilities and their limitations.

The remaining applications are the legacy code TEFSI, which solves thermal problems involving the interaction between fluids and solids, with code portions that solve both the fluid-dynamic equations and the solid body heat conduction equation, and the CFD code EUL3D from ICASE, with applications in the aerospace industry. The former code uses a block-structured mesh and a semi-implicit Stone solver [25] that presents the key challenge for parallelisation. EUL3D [17] uses finite elements with a Euler solver. It uses adaptive meshing techniques on an unstructured grid. Many interesting parallelisation issues arise in consequence, some of which are also encountered in the Comet and TEFSI codes.

It is exceptionally difficult to provide tool support for the adaptation of unstructured grid codes and the porting methods used are (currently) not automatable. We have studied the strategies applied to these examples with the goal of being able to offer advice on parallelisation that will be encoded in the POST guidance module. We have identified commonalities in the loop structures and in successful adaptation strategies applied, and thus expect that insights gained from them will be useful for other application developers also.

3 The POST Development Environment

The POST program development environment is an interactive system which provides an integrated set of tools to support OpenMP application development. At the heart of the production environment is FORESYS [22], a commercial Fortran source code analysis and restructuring system. FORESYS is an interactive system that is used for reverse engineering and upgrading of existing software, and for validation and quality assurance of new code. It is frequently used to convert programs that use non-standard, or outdated, Fortran syntax into modern Fortran 77 or Fortran 95 code. FORESYS also has considerable functionality for creating MPI code. It incorporates semantic information about MPI functions, and performs a variety of syntax checks for MPI codes. It has been coupled with other parallel program development tools in the ESPRIT project FITS [8].

3.1 Manual and Fully Automatic Generation of OpenMP

FORESYS has been extended in several ways to support the development, verification and maintenance of OpenMP programs. The starting point may be either sequential or MPI code. In manual mode, the user has access to the results of FORESYS' analysis, in particular data dependence analysis, for part or all of the program. The built-in editor permits the user to update source text in response to the information shown. The automatic mode adapts the program, or a user-selected region of the program, without further user interaction.

If manual parallelisation is selected, an interactive window is provided with which the user may insert OpenMP constructs into the source text. FORESYS feedback informs whether or not it has determined that a loop (level) is fully OpenMP parallelisable, which is the case if there are no cyclic dependences. It may also report that a loop nest is partly parallel, when an inner loop is fully parallel but the entire loop nest is not. The user has the option of applying loop fission to isolate parallel loop regions from those that are not parallel, or use insight into the application to insert parallel constructs where analysis has not been able to prove independence of iterations. Additional information on parallel (and partly parallel) loops includes a list of detected reductions, and lists of variables the analysis has deemed to be private or shared, together with the corresponding keyword. Where private variables are used after the loop nest, the need for the LASTPRIVATE clause is indicated.

The automatic translation process, if selected, is target machine-specific in that it is guided by a set of machine parameters that, among other things, provide a rough indication of the overheads incurred by parallelisation, and individual OpenMP constructs. These may help to determine whether there is any profit in parallelizing a given loop nest.

The diagram in Figure 1 shows an excerpt from the Comet code after automatic parallelisation by FORESYS. The cell-based loop highlighted in the text is typical of many such loops in the code, where individual cells are updated in a loop with no data dependences. For loops of this kind, a straightforward automatic OpenMP code generation is both appropriate and sufficient.

3.2 Additional Support for Experts: Code Replacement Based upon Patterns

The loop following the cell-based loop in Figure 1 presents more of a challenge. It is reproduced in Figure 2. This code is representative of a number of loops in the computationally intensive routines of the Comet code which are face oriented and conform to a structure often found in irregular computations. The potential dependences involving the array `wrk` prevent automatic parallelisation. Yet with some insight into parallelisation, it is possible to rewrite this in parallel form. A knowledgeable user, who is able to specify the necessary

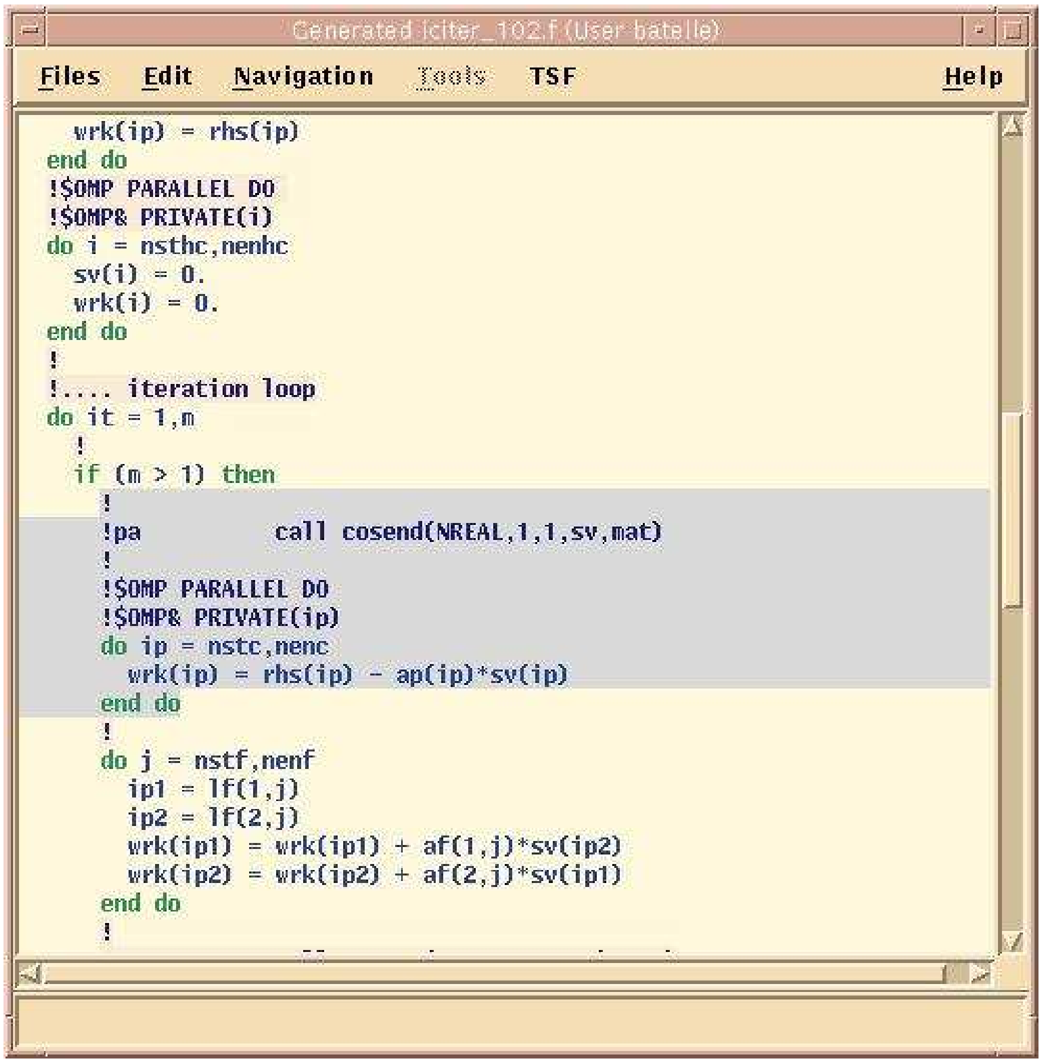

Figure 1: FORESYS OpenMP generated code

```
do j = nstart_face, nend_face
  ip1 = lf ( 1, j )
  ip2 = lf ( 2, j )
  wrk ( ip1 ) = wrk ( ip1 ) + af ( 1, j ) * sv ( ip2 )
  wrk ( ip2 ) = wrk ( ip2 ) + af ( 2, j ) * sv ( ip1 )
end do
```

Figure 2: Face-oriented loop that is not automatically parallelisable

adaptation, may employ another component of the POST environment, the TSF (Tool Set for Fortran) scripting system, to adapt such loops.

TSF is based upon the insight that many of the code modifications performed during parallelisation are highly repetitive in nature. The changes range from simple search and replace operations to complex program reorganisations, and the insertion of new constructs. Although some of these changes to the code can be specified in the form of automatable transformations, in many cases the required modifications are not general-purpose enough for inclusion into standard transformation libraries or for use in automated approaches to program adaptation. In many other cases, fully automatic transformation is currently not feasible at all. The TSF program transformation script system [4] has been developed to enable users to write their own custom transformations for repetitive tasks. Among the features available to a TSF user are commands to navigate a program's source code, to specify loops and statements, and to create, delete and modify program constructs, possibly – but not necessarily – based upon program analysis results. A number of "built-in" functions provide simple versions of several standard program transformations, and return information such as the level of a loop in a loop nest; functions are available to create dialog windows and accept user input.

TSF allows the user to select an existing fragment of code and generate a pattern from it. The resulting search pattern can then be generalised, either by the user, or by "built-in" abstractor functions. Fortran identifiers and labels which are matched by variables within the search pattern can be used in subsequent transformations of the matched code fragment by an associated script. The specific advantage of the TSF pattern matching features over conventional scripting languages, such as PERL, is its awareness of Fortran syntax.

A pattern generated to detect face-oriented loops, such as the one in Figure 2, is given in Figure 3. Although it is not particularly easy to read, it is easy to create this from the selected code using TSF utilities. Pattern matching variables may be used to match any constant or variable. If a given pattern variable is repeated in a pattern, the corresponding matches must be consistent within the matched code. Matched items can be referenced in the application of the script. We use several pattern variables from this pattern in the code adaptation script shown further below.

Once such a loop is detected, a custom transformation may be created to modify it. One strategy for parallelizing the loop shown above is to replace it by an equivalent nested loop, whose outer loop iterates over neighbours of cells, and whose inner loop iterates over the cells rather than faces. The inner loop can then be parallelised in a straightforward fashion as shown in Figure 4.

This parallelisation strategy is too complex for implementation via standard automatic

```
(do ?v258 ?label (bounds (name . ?j) (name . ?nstf) (name . ?nenf) ?v259)
(l_stat
(ass (name . ?i1) (vardim (name . ?lf) (l_exp (int_cst . 1) (name . ?j))))
(ass (name . ?i2) (vardim (name . ?lf) (l_exp (int_cst . 2) (name . ?j))))
(ass (vardim (name . ?wrk) (l_exp (name . ?i1))) (add (vardim (name . ?wrk)
(l_exp (name . ?i1))) (mul (vardim (name . ?af) (l_exp (int_cst . 1) (name . ?j)))
(vardim (name . ?sv) (l_exp (name . ?i2))))))
(ass (vardim (name . ?wrk) (l_exp (name . ?i2))) (add (vardim (name . ?wrk)
(l_exp (name . ?i2))) (mul (vardim (name . ?af) (l_exp (int_cst . 2) (name . ?j)))
(vardim (name . ?sv) (l_exp (name . ?i1))))))
(labstat ?label (continue))))
```

Figure 3: A TSF pattern to search for face-oriented loops

```
!$OMP PARALLEL PRIVATE(I)
do i = 1, max_neighbours
!$OMP DO PRIVATE(J,IP1,IP2)
  do iom = nstart_cell(i), nend_cell(i)
    j   = iface(iom)
    ip1 = lf ( 1, j )
    ip2 = lf ( 2, j )
    wrk ( ip1 ) = wrk ( ip1 ) + af ( 1, j ) * sv ( ip2 )
    wrk ( ip2 ) = wrk ( ip2 ) + af ( 2, j ) * sv ( ip1 )
  end do
!$OMP END DO
end do
!$OMP PARALLEL
```

Figure 4: Restructured and parallelised face-oriented loop

transformation techniques. Moreover, its realisation in the Comet code makes use of a set of data structures that do not appear in the original loop, although they do already exist in the source code. However, it is possible to use a TSF pattern to identify such loops in the program, and to develop a corresponding script to implement the required restructuring (once the user is satisfied that this is appropriate).

Figure 5 shows the script used to transform the loop into the parallel form shown in Figure 4. The comments (preceded by //) in the script should be sufficient to give a flavour of how the script works. The first line in the script sets the TSF variable `stat` to the currently selected code fragment (as determined by the search pattern in this case). The application of the `Undo:Save` operation enables the user to undo the results of the application of the script (via a button in the TSF window) if it does not produce the desired result. In our example we use the values of the pattern variables `?j,?i1,?i2` from the above pattern to obtain the names of the private Fortran variables `j,ip1,ip2` in the OpenMP PRIVATE clause.

3.3 More Help for Non-Experts: User Guidance Module for Parallelisation

The use of TSF presupposes user insight into the desired adaptation process. Where this is not the case, alternative means of support may be needed, especially if the performance obtained by an initial automatic parallelisation is unsatisfactory.

POST provides a User Guidance Module, for which a prototype is being created at the University of Southampton, to help in such situations. This module is intended to combine several approaches to parallelisation within a single user interface (Figure 6). It not only aims to provide help for novice users; its additional analyses and more broadly applicable transformation options can also be utilised by experts. The Guidance Module is invoked from within FORESYS, after that system has completed its initial analysis of the input program. The module performs a data locality analysis in addition to FORESYS' (interprocedural) data flow and data dependence analyses. It is able to report on, and explain, the results of all of these analyses. The module provides

- *step-by-step instructions on how to create a parallel program* under OpenMP, both in a *tutorial* mode and in an application mode, where a user is expected to complete each step with respect to the given input program,
- an *assisted semi-automatic parallelisation mode* in which the user is provided with a suggested parallelisation strategy, and a set of transformations that may be alternatively selected and applied,
- an *extended Help facility* that provides feedback on the appropriateness of parallelisation strategies, on the usage of transformations, and on why (if appropriate) a transformation cannot be applied in the given context, as well as
- *access to related parallelisation problems* stored in the module's database, together with the solution to the problem specified in the stored version.

When adapting the program's loops, the user may request that the tool propose a strategy. Depending on the nature of the loop nest, this may be a comparatively straightforward adaptation, or the response may involve a search for similar loop nests within

```
SCRIPT face_job()
  // set variable stat to current selection
  stat:=$csel
  IF(stat.VARIANT=="do") THEN
    // Save previous code for undo operation
    tmp:=Undo:Save(stat.PARENT,TRUE,TRUE)
    // Change loop bounds
    new_bounds:=PARSEEXPR("iom=nstart_cell(i), nend_cell(i)")
    stat.BOUND <- new_bounds
    // add reordering statement
    reorder := PARSESTAT("      j=iface(iom)")
    PASTE(1,reorder,stat.BODY)
    // Add neighbour loop around copy of modified original loop
    inner_loop := COPY(stat)
    n_loop := PARSESTAT("
       do i=1, max_neighbours
       %s
       end do",inner_loop)
    IF(!n_loop) THEN
      ERROR(1, "Creation of n-loop failed")
    ENDIF
    // Add OpenMP directives
    privates := "PRIVATE(" + TOSTRING(?j) + "," + TOSTRING(?i1)
    + "," + TOSTRING(?i2) + ")"
    INSERTCOMMENT(n_loop.BODY,"$OMP DO " + privates,"insert","last")
    INSERTCOMMENT(n_loop.LAST,"$OMP END DO")
    INSERTCOMMENT(n_loop,"$OMP PARALLEL PRIVATE(i)","insert","last")
    following := $csel.SKIP
    INSERTCOMMENT(following,"$OMP END PARALLEL","insert","first")
    stat <- n_loop
  ELSE
    PRINT "Do loop not selected"
  ENDIF
ENDSCRIPT
```

Figure 5: A TSF script to restructure face-oriented loops

the stored set of parallelisation problems. If the application of a strategy is prevented by some feature of the program, the user may request additional feedback to explain the problem. A range of user responses to that feedback are possible, including assertions of (data) independence, direct manual application of transformations or additional searches for examples of parallelisation that may shed insight into the porting problem. A range of useful loop transformations, that generalise those available in FORESYS, are provided to aid this adaptation process. The explanation facility associated with them attempts to explain not only how each transformation is applied, but also why a specific transformation cannot be used in a given context. This may include some hints on how the user might be able to remedy the situation.

The User Guidance Module attempts to deal with two issues, in particular, that are not well covered by existing automatic parallelisation systems:

- the optimisation of OpenMP code during parallelisation to ensure the efficient use of the memory hierarchy, and
- the parallelisation of those loop nests for which automatic parallelisation is either inadequate or not possible without information from the user and/or a preliminary restructuring of the code.

To achieve the first objective, the module exploits the dependency analysis and built-in transformations already available within FORESYS and TSF (see Sections 3.1 and 3.2) in combination with input from the user. This information is used to propose a coherent strategy for applying a set of relatively standard transformations (loop fusion, loop interchange, loop tiling and loop distribution), to restructure and permute a given loop nest that will promote efficient use of the memory hierarchy within OpenMP parallel loops. The code may be profiled during the information-gathering process.

The user may request an automated application of the proposed strategy, or may request feedback and a suggested strategy from the tool. In this latter case, several standard transformations may be applied manually, or the user may request information on why certain transformations cannot be used. Analysis results may be displayed.

We began this work by adapting an algorithm proposed by McKinley [16], in which the loops of a loop nest are analyzed to determine which of them potentially provides the best cache re-use. This may be used to rank loops according to their potential for optimizing cache. An attempt is made to reorder the loops accordingly: if this is not possible, an approximation to this order is sought. A subsequent reordering (or strip-mining) may be needed in order to ensure that the outermost loop is parallel.

Unfortunately, this strategy has not been particularly helpful for the POST project codes, most of whose loops are either straightforward to parallelise or are not amenable to this approach at all. Loop nests are typically not deep, so that where the algorithm is applicable, the major problem is to determine an appropriate trade-off between cache optimisation in the inner loop and parallelism in the outer loop.

But many key loops are not amenable to automatic parallelisation. Even in the block-structured TEFSI code, we find several crucial loops where the data structures do not permit any useful automatic parallelisation. For each code, some sophisticated insight into how the code is structured, and how it can be restructured, has been necessary before the parallelisation of critical routines could be achieved. Having derived this insight, it has been

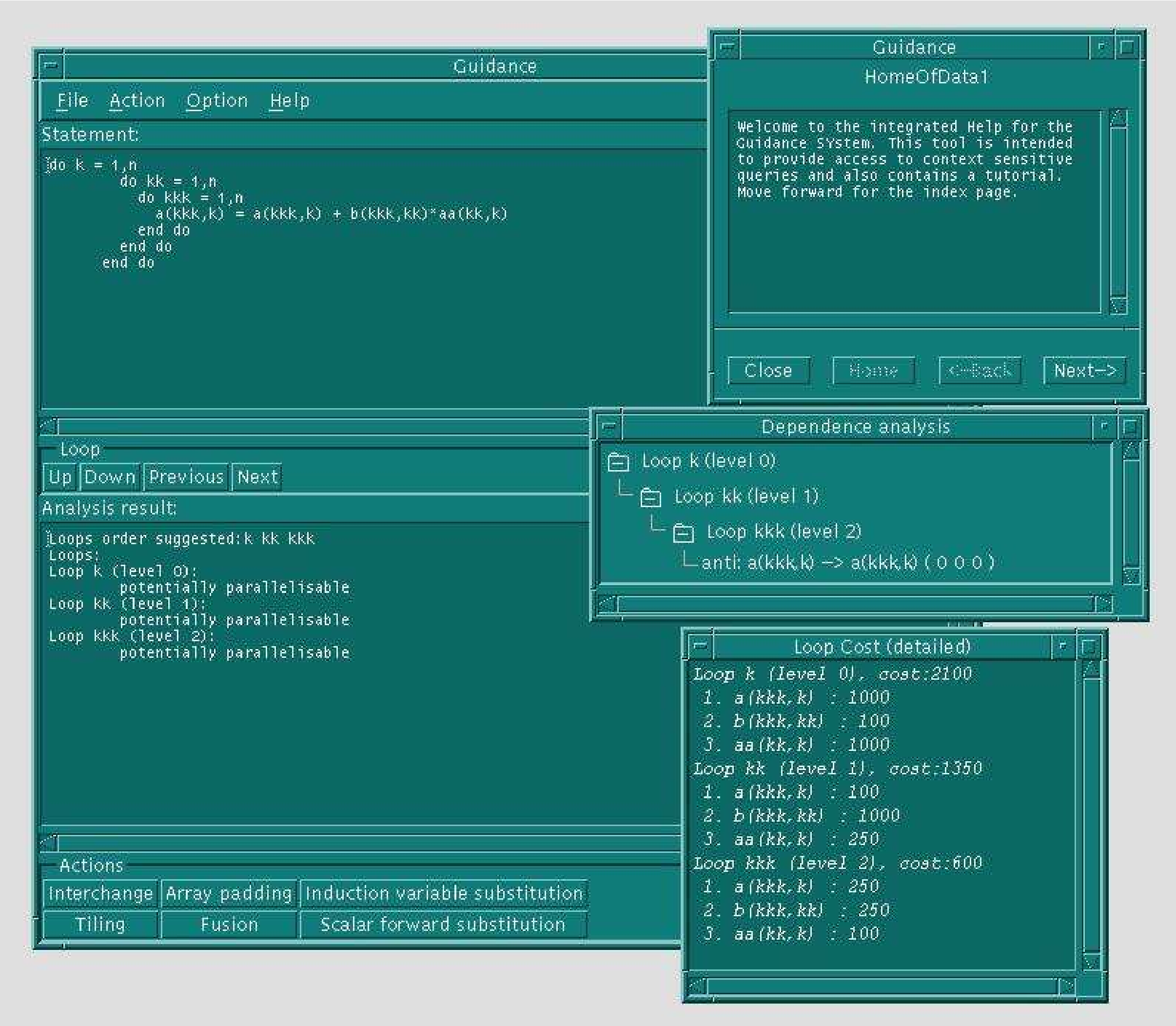

Figure 6: User Guidance Module

possible to devise TSF scripts that will assist with applying the necessary transformations, where these need to be applied repeatedly to similar loops within the code.

In order to achieve our second objective we wish to use the insight gained from the parallelisation of these particular loop nests, and recognise situations when we can apply the same techniques to other codes in a more general context. To this end we are experimenting with a Case-Based Reasoning (CBR) approach [12] to build a database of code examples which can be searched in order to provide advice on parallelisation strategies that are specifically related to the loop nest under consideration.

We reproduce a third loop from the Comet code in Figure 7 below in order to illustrate this aspect of the system. The structure of this loop nest is typical for many in unstructured codes. It consists of an outer loop that iterates over cells in the mesh, and an inner loop that iterates over the neighbours of each cell. In this application, there are up to six neighbouring cells in the inner loop, but possibly hundreds of thousands of cells. Most of the exploitable parallelism is therefore to be found in the outer loop. In the form shown in the figure, `wrk` is involved in a data dependency that cannot be resolved at compile time, and the automatic parallelisation facility is only able to parallelise the inner loop. Although correct, transformation of this loop provides little exploitable parallelism and in fact, the parallelisation overheads led to a slowdown when this strategy was applied to the project code. (Note that in a CBR system, even such negative cases can be used to warn the user of inadvisable strategies.)

For this loop nest, two alternative approaches were devised to identify cells which could be calculated independently, using a wavefront re-ordering and grid-partitioning, respectively. Only the grid-partitioning approach was found to be scalable.

Note that documentation describing the manual adaptations required to apply each strategy, and their impact on performance for the example code provided, is considered to be an essential part of any stored "case", enabling it to serve as a possible model for the adaptation of similar loops in other applications. These examples are also available under the Help facility.

```
i = ...
do ip = nstartcell, nendcell
   wsp = wrk ( ip )
   do il = 1, numface ( ip )
     i = i + 1
     wsp = wsp + af ( 2, lfp ( i )) * wrk ( lf ( 1, lfp ( i ) )
   end do
   wrk ( ip ) = wsp * di ( ip )
end do
```

Figure 7: Double loop nest where only the inner loop is automatically parallelisable

We can also make use of the insights gained from parallelisation of the loop shown previously in Figure 2. With some further generalisation we are able to extend the prototype pattern and script, shown in Figures 3 and 5, so that our Guidance module will detect similar loops, and assist with a corresponding restructuring in other unstructured codes. In the first instance, the help we can give is to provide a description of how this problem was tackled in our example case. It is then up to the user to identify or implement data structures corresponding to `max_neighbours, nstart_cell, nend_cell and iface` in his or her code. If this can be done, we are then able to provide further help by prompting the user for the names of the corresponding data-structures, which are read by a TSF script and used to effect the desired restructuring. Having accomplished this first difficult phase of identifying a potential solution and providing the required data structures, the system can then be used to seek out similar instances elsewhere in the user code and apply the restructuring with little further effort from the user. Thus scripts such as the one shown previously in Figure 5 can be used as components of a more ambitious approach to parallelisation guidance which we describe next.

Case Based Reasoning

There have been some experiments in the past which have applied expert system technology, generally in the form of rule-based systems, to a range of problems in compiling. However, the expert knowledge associated with parallelisation is not well structured and does not lend itself to conventional rule-based systems. In contrast, the Case-Based Reasoning (CBR) approach [12] aims to solve a given problem by adapting the solution of a similar one already encountered. Typically the solution process requires the following steps:

- *Identification of the problem* – In our situation, the user is prompted to select the loop that he wishes to investigate but alternatively the most time consuming loops can be identified by profiling.
- *Retrieval of similar cases from past experience* - the definition of similarity is non-trivial and often involves more than just superficial similarity. In our example, pattern matching alone is sufficient. However, we may often wish to use other information. For instance, in the TEFSI code, it is necessary to look at the dependency vectors of the loop nests to help identify where loop skewing ([26, 27]) may be useful.
- *Solution re-use* – In some situations, the transformations used to effect a previous solution can be re-used automatically, in which case the process terminates here. However, there may be reasons, such as the occurrence of still unresolved dependencies, why the solution cannot be applied directly. In this case the next two steps are relevant.
- *Solution revision* – the previous solution has to be adapted to fit the current problem. This may involve consideration of further cases relevant to the outstanding difficulties.
- *Solution retention* – If a substantially new solution is derived, this should be stored back into the CBR system as a new case for comparison with future problems.

An experimental framework for CBR has been built on top of TSF by Y. Mevel [18]. We utilise this framework wherever possible for our own experimental implementation. Although one of the advantages of a CBR system is that cases can be added and refined in an incremental fashion, the design and population of a full-scale CBR system is a major undertaking. In the POST project, our initial objective is to demonstrate that a system can be built on top of FORESYS and TSF which applies the principles of CBR to the limited cases representing problems encountered during the parallelisation of the project applications. In the first instance, we wish to demonstrate that we can re-apply the cases to previously unknown codes which use similar methods (eg. other unstructured codes for our face-oriented loop example). In the longer term, validation is itself a research issue, as the process of determining how successful we are at detecting or missing similar cases can be fed back into refining the notions of similarity in our case studies.

4 Related Work

In order to understand how best to create OpenMP program code, we have drawn upon a variety of experiments of our own, as well as from a number of other sources [3, 9]. We have also looked at applications using a combination of OpenMP and MPI [5, 10] and a approach to parameterizing hybrid OpenMP/MPI applications for different platforms [21].

In addition to vendor-supplied parallel program development toolsets, such as the SGI automatic parallelisation options [23], there are a range of products from individual vendors to support the creation of OpenMP code. These include the Kuck and Associates product line [13] for shared memory parallelisation. Their *Visual KAP for OpenMP* operates as a companion preprocessor for the Digital Visual Fortran compiler. Although it has many user-controlled options, it essentially functions as a batch tool on the source files. The *FORGE* parallelisation tools [1] from Applied Parallel Research enable source code editing and offer control and data flow information, in combination with an HPF compiler. *VAST* [20] from Pacific Sierra Research helps a user migrate from Fortran 77 to HPF code by automatic code cleanup and translation. We are not aware of tools which are capable of supporting development of programs that combine the OpenMP and MPI paradigms.

Transformations to support the parallelisation of loop nests have been studied for many years, and there is a rich history both of individual transformations [28], as well as of experiences in their application. Recent work has aimed at the development of strategies for the co-ordinated application of transformations to support both cache optimisation and parallelisation [24], and we follow existing efforts where they are applicable to our work. However, most such transformations have operated on loop nests with affine subscript expressions, and it is only in very recent work that researchers have attempted to overcome this strong limitation. Given that most of the applications in POST are unstructured mesh codes, a major focus of our work is to extend existing techniques to enable parallelisation of the loop nests that are typical for them.

Interactive display of analysis results and the provision of user transformations was pioneered in the *ParaScope Editor* [2], for improving shared memory programs. *CAPTools* [11] interacts with the user on the basis of a program data base, which the user may query and modify, in order to improve the automatic transformation of a sequential program to an MPI one. One of the authors has previously developed a prototype system for the interactive analysis and transformation of sequential Fortran programs in preparation for parallelisation [6]. Finally, *SUIF Explorer* [14] supports parallelisation by identifying the most important loops via profiling. It takes a different approach from POST to help the user perform this task, for example, applying program slicing to present information on dependence-causing variables.

5 Conclusions and Future Work

In this paper we have described a portable and extensible programming environment for the development of parallel Fortran applications under OpenMP. We are creating a set of tools that provide different levels of automation for the task of OpenMP code generation. FORESYS provides automatic OpenMP parallelisation of loop nests using conventional compiler analysis techniques. It also permits the user to insert directives immediately into the source text, performing syntax checks to help eliminate user errors. TSF scripts provide a more flexible aid to parallelising program constructs; TSF is particularly useful for performing repetitive, possibly application-specific, program modifications where user insight is required to specify the translation. In addition, within the User Guidance Module, we are exploring the usefulness of a combination of recent and novel techniques to provide a range of alternative approaches to program parallelisation, including a Case Based Reasoning facility. These techniques provide varying levels of automation under user control. The system attempts to provide support even for challenging parallelisation problems, such as those posed by unstructured mesh codes. The availability of a high-level interface (TSF) to the program analysis capabilities of the underlying system (FORESYS), enables us to create prototype systems of the kind described, within a reasonable time frame.

Our work has been greatly supported by interaction with a group of application developers, who are creating real-world OpenMP versions of their application codes. In addition to their feedback on the functionality and features of our environment, they have provided us with many program examples that have stimulated our ability to provide support for their transformations.

The development of a program database that will cover a reasonable fraction of existing codes is a daunting task. It is no less a challenge to current technology to determine a widely applicable strategy for identifying code fragments "similar" to specific stored cases in an application presented to such a system. The work performed here can be no more than a modest beginning to such an undertaking. However, we feel that such a system, even though limited in its scope, may provide useful support to application developers by providing expert information directly related to parallelisation problems encountered in practice. Despite significant differences in detail, there remain many broad similarities between application codes and the nature of the porting problems they pose. We hope that this work may be extended and form a more extensive exploration of this technology in future.

Acknowledgments

This work is part-funded by the ESPRIT project 29920, POST (Programming with the OpenMP STandard), whose name we have borrowed.

We thank Jerry Yan at NASA Ames and Tor Sorevik at Parallab, Norway, for discussions on a number of related topics, including the NAS benchmarks and for providing benchmark results.

References

[1] Applied Parallel Research. *APR's FORGE 90 parallelisation tools for High Performance Fortran*. APR, June 1993

[2] V. Balasunderam, K. Kennedy, U. Kremer, K. McKinley and J. Subhlok. *The ParaScope editor: an interactive parallel programming tool*, Proc. Supercomputing 89, Reno, 1989

[3] R. Blikberg and T. Sørevik. *Early experiences with OpenMP on the Origin 2000.* Proc. European Cray MPP meeting, Munich, Sept 1998

[4] F. Bodin, Y. Mevel and R. Quiniou. *A user level program transformation tool*, Proc. Int. Conference on Supercomputing, 1998

[5] S.W. Bova, C.P. Breshears, C. Cuicchi, Z. Demirilek and H.A. Gabb. *Dual Level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP.* Proc. Supercomputing '98, Orlando, 1998

[6] B. Chapman and M. Egg. *ANALYST: Tool Support for the Migration of Fortran Applications to Parallel Systems*, Proc. PDPTA' 97, Las Vegas, 1997

[7] B. Chapman and P. Mehrotra. *OpenMP and HPF: Integrating Two Paradigms.* Proc. 4th Int. Euro-Par Conference (Euro-Par '98), LNCS 1470, Springer Verlag, 1998

[8] B. Chapman, F. Bodin, L. Hill, J. Merlin, G. Viland and F. Wollenweber. *FITS—A Light-Weight Integrated Programming Environment*. To appear in Proc. Europar '99, Toulouse, 1999

[9] T. Faulkner. *Performance Implications of Process and Memory Placement using a Multi-Level Parallel Programming Model on the Cray Origin 2000.* Available at URL `www.nas.nasa.gov/~faulkner`

[10] L.A. Frøyland, F. Manne and N. Skjei. *2D seismic modelling on the Cray Origin 2000.* Internal report 1998-02-13 for Norsk Hydro. In Norwegian

[11] C.S. Ierotheou, S.P. Johnson, M. Cross and P.F. Leggett. *Computer aided parallelisation tools (CAPTools)—Conceptual overview and performance on the parallelisation of structured mesh codes*, Parallel Computing, 22(2), 163–195, 1996

[12] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, 1993

[13] Kuck and Associates. *KAP/Pro Toolset for OpenMP.* See `www.kai.com/kpts/`

[14] S.-W. Liao, A. Diwan, R.B. Bosh Jr., A. Ghuloum and M.S. Lam. *SUIF Explorer: An interactive and interprocedural parallelizer.* Proc. 7th ACM Symposium on POPL, 37-48, 1999

[15] H. Lu, C. Hu and W. Zwaenepol. *OpenMP on Networks of Workstations.* Proc. Supercomputing '99, Orlando FL, 1998

[16] K. S. McKinley. *A Compiler Optimization Algorithm for Shared-Memory Multiprocessors.* IEEE Transactions on Parallel and Distributed Systems, 9(8), 1998

[17] D.J. Mavriplis. *Parallel Performance Investigations of an Unstructured Mesh Navier-Stokes Solver.* NASA CR-2000-210088, ICASE, NASA Langley Research Center, 2000

[18] Y. Mevel *Environnement pour le portage de code oriente performance sur machines paralleles et monoprocesseurs.* Ph.D. Thesis, University of Rennes, France, 1999

[19] OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October, 1997

[20] C. Rodden and B. Brode. *VAST/Parallel: Automatic Parallelisation for SMP systems.* Pacific Sierra Research Corp., 1998

[21] A. Sawdey. SC-MICOM. Software and documentation available from `ftp://ftp-mount.ee.umn.edu/pub/ocean/`

[22] Simulog SA, *FORESYS, FORtran Engineering SYStem, Reference Manual V1.5*, Simulog, Guyancourt, France, 1996

[23] Silicon Graphics, Inc.: MIPSpro Fortran 77 Programmer's Guide, 1996

[24] S.K. Singhai and K.S. McKinley. *A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality.* Computer Journal, 40(6), 340–355, 1997

[25] H.L. Stone. *Iterative solution of implicit approximations of multidimensional partial differential equations.* SIAM J. Numerical Analysis, Vol 5(3), Sept. 1968.

[26] M. Wolfe. *Loop Skewing: The Wavefront Method Revisited.* Intl. Jnl. Parallel Programming 15,4, pp279-294,August 1986.

[27] M. Wolfe. *Optimising Supercompilers for Supercomputers.* Pitman Publishing, London, ISBN 0-262-73082-0, 1989.

[28] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.