Parallel Software Products Inc.

ParaWise – Widening Accessibility to Efficient and Scalable Parallel Code

White paper
WP-2004-01

Parallel Software Products Inc.
1945 South 1100 East, Suite 100, Salt Lake City, Utah 84106
website http://www.parallelsp.com

Executive Summary

There is a universal understanding that massively parallel computing has the potential to completely transform the way computation is used in many areas of science and industry. Despite this, its use is not as widespread as one would hope. Historically, parallel computing is perceived as a niche technology, where it is utilized only by those that invest significantly in both the hardware and software. The introduction of cluster systems has attempted to address the high hardware cost, but the overwhelming task of transforming existing serial codes to execute efficiently in parallel on any high performance computing (HPC) system still remains and is typically magnified on clusters.

Simply put, the task of efficiently parallelizing a user's serial code is complex. While parallelizing compilers are relatively quick to generate a parallel executable from a serial code, the performance achieved is somewhat limited. On the other hand, a manual parallelization requires a great deal of expertise and is both time-consuming and costly, but often results in impressive parallel performance. What has been missing is a tool that can produce equivalent parallel performance to that of a manual parallelization, but in a much shorter time frame and without requiring significant skill and effort from the user. The ParaWise parallelization tool from Parallel Software Products has been developed in collaboration with NASA Ames to fill this gap and address the major obstacle of effective software parallelization (for both shared and distributed memory systems).

ParaWise is capable of generating software that is recognisable to the code's author (enabling subsequent code maintenance) and also delivers parallel performance and scalability akin to a manual parallelization. It is able to achieve this through the deployment of powerful code analysis techniques and sophisticated code generation algorithms, along with information provided by the user about the code being parallelized. ParaWise is of tremendous benefit to both experienced parallelization experts and users unfamiliar with parallel programming because it has parallelization expertise embedded within it and performs all code alterations, only requiring the user to provide information about their application code. It not only automates almost all of the parallelization process but also provides detailed and accurate information about the application code during each and every stage.

With ParaWise capable of generating efficient parallel code for a wide range of users and applications, the potential of HPC systems can be realized, making the purchase of a machine more attractive, allowing many more users to take advantage of what parallel computing has to offer.

Contents

1 The Current Problems Of High Performance Computing (HPC)

Parallel processing can be defined as using a number of processors to co-operate in the execution of an application. By dividing the total problem between processors, the overall runtime for the application can be dramatically cut, ideally by a factor proportional to the number of processors employed.

The possibility of exploiting parallelism is dependent on the nature of the application code, however in reality almost all codes contain inherent parallelism. Independence between individual statements (i.e. where a program statement does not have to wait to use values computed by another statement), between iterations of loops (allowing many iterations to be performed simultaneously) and between routine calls, all provide enough parallelism to enable a selective choice of the best sources of parallelism to be made. The task of parallelization thus reduces to the problems of parallelism identification and implementation given the constraints of the parallel hardware available. In general, the computations being performed on a processor are not completely independent from other processors and so they must interact throughout the execution of the application. Computations performed by a processor will, for example, often require data calculated on others and thus cannot be performed until that data is available to this processor. Such interactions can have a significant effect on the overall performance of a parallel code with the cost of inter-processor synchronization and the associated idle time (awaiting data that is being calculated elsewhere) being significant. Access to data from another processor can also incur a time penalty due to the latency in reaching the memory area concerned, often involving the use of shared hardware mechanisms that provide the connection between processors.

Given the tremendous potential for parallelism, it was hoped that the use of large parallel systems would be commonplace by now, with all users that require significant computational power being able to harness HPC. The reality is, however, that parallel processing has only been fully exploited in large organizations that have the resources to employ dedicated parallelization experts who can convert serial application codes to efficiently use expensive parallel machines. The main cause of this is that the runtime overheads of parallel systems are very unforgiving of ineffective parallelization, particularly as processor speeds have increased so rapidly for many years. If the selected parallelism at any point in the application code requires frequent interaction between processors then speedup can be dramatically diminished and the scalability to take advantage of larger numbers of processors is significantly restricted. Any serial section of code can also limit scalability, as described by Amdahl’s law [1], and so requires attention if there is to be any point executing on large parallel systems. For automatic compilers and inexperienced code parallelizers, this is a very challenging problem that often prevents the production of effective parallel code.

The question that needs to be addressed is how can a far wider audience exploit the full power of large parallel systems?

2 History's Effect On The HPC Market

For many years, the use of software tools has been vital to the success and popularity of computing with compilers being of fundamental importance. The history of computing has had a significant effect on the perception of software tools by their potential market.

2.1 User Expectations Of Parallelizing Compilers And Tools

One of the first uses of program analysis was to identify and apply serial code optimizations as part of the compilation process. These optimizations are fully automatic and can provide significant computation speed increases that lead to the perception that optimization is successful. With the advent of machines capable of performing operations on vectors of data, the automatic compiler concept was extended to include vectorization, where this was perceived as part of the optimization process. The performance improvements again led to the view that fully automatic vectorizing compilers were successful and so the expectation of similar automatic parallelizing compilers for shared and distributed memory parallel machines was created. The reality has been that automatic parallelizing compilers have not achieved the success the market desired. The analysis capabilities and code generation techniques used are highly advanced, but the lack of additional insight and information about the application codes significantly limits performance. This perception of success and failure is, in fact, somewhat misguided as the achievements of the optimizing and vectorizing compilers are also greatly restricted by the analysis quality, but unlike parallel processing where the number of processors implies the expected performance, no clear quantifiable measure for "success" exists. This has led to a major inhibitor to the commercial success of parallelization software. Many potential users are waiting for the "successful" automatic parallelizing compiler to be produced and see any requirement of user interaction as a failure of the analysis.

Currently, there exists a fairly small market driven by dedicated parallelization experts that is the focus of much of the current parallel hardware and software development. Another class of user relies on the automatic parallelizing compilers, but the performance improvements usually achieved are more in line with the expectation from optimization rather than parallelization and they typically exploit systems that have only a few processors. The accuracy of the analysis used in such compliers often forces parallelism to be exploited for the inner loops of a nest of loops, forcing the inevitable runtime overheads to be frequently incurred and detracting from performance. This has been further magnified by the common policy of parallelizing compilers that achieve speedup where possible, but avoid slowdown at all costs. This results in loops where parallel execution is deemed not profitable in a runtime test being executed in serial, avoiding slowdown, but not providing speedup and preventing scalability (Amdahl's law).

2.2 Parallel Languages And Tools

Attempts to provide a means to ease the production of parallel code have met with, at best, limited success. New parallel languages, language extensions such as High

Performance Fortran (HPF) or OpenMP, along with many parallelization tools, found interest in the dedicated parallelization expert community, but not from other groups that are awaiting fully automatic parallelization of serial code.

The idea that parallel computing could drive all code authors to adopt new languages that specifically enable parallel execution was obviously over ambitious. With a massive archive of “dusty deck” codes and code authors who find serial programming a challenging task that is only to serve their main areas of interest, the discarding of these codes and re-training of these unwilling authors was infeasible.

HPF was supposed to be an extension to the standard Fortran language, however in practice, significant code re-authoring was often essential. It still placed the burden of distributing data and parallelism determination on the user whilst automatically performing communication generation for distributed memory systems. As communication generation to achieve efficient performance is one of the hardest tasks in parallelization and often requires interprocedural operation involving the entire application code to be successful, the effectiveness of HPF codes was typically disappointing. In effect, it took the “worst of both worlds” with a significant and apparently insurmountable burden placed on both the user and the compiler.

OpenMP is another language extension aimed at simplifying the creation of efficient parallel code. Although it does not suffer from all the problems of HPF, it still requires the user to determine parallelism and, worse still, it introduces the concept of variable privatization. With real world application codes, the task of identifying parallelism and privatization in an interprocedural framework has again proven too complex for many potential users. The compiler’s role for OpenMP is, however, fairly simple and is not fundamental to the parallel performance as was the case with HPF. This, along with its neat representation of parallelism, does make OpenMP an attractive way of representing parallelism in an application code without inhibiting the user’s recognition of their code.

Other tools aim to aid the various mechanical processes of parallel code generation rather than attempting significant automation of the entire process. The level of user effort required and the performance of the resultant parallel codes has deterred much of the market from being interested in such tools.

Most of these new languages, language extensions and tools are based on the premise that the necessary tasks cannot be performed (or even significantly assisted), by an automatic compiler. As a result, the border between the tool’s role and the user’s role has been significantly skewed to place the burden (and blame when performance is poor) on the user.

This leads to the question as to whether this is a fundamental problem for parallelization or just a historical legacy?

3 Who Are Parallelization Tools Aimed At?

Although the benefits of parallel processing are obvious, the number of HPC users is of concern, and even more so is the fact that only a few of those HPC users are harnessing the power of parallel systems effectively. The term HPC user refers both to people that use HPC systems for the computation speed they provide, and also to those people that actually parallelize application codes. This type of user can be categorized into three classes:

- *expert parallelizers – with many years experience*
- *non-expert parallelizers – with little experience*
- *serial code authors and serial application users – want/need to use HPC systems*

The expert parallelizers are typically employed by large organizations that see their cost as insignificant when compared to their investment in HPC systems. As they understand the concepts of parallelization, they can easily interpret detailed information relating to parallelism in application codes as provided by tools. Their requirement from tools is to increase their productivity in producing parallel codes without having to sacrifice the performance they would be able to achieve through a fully manual parallelization.

A number of HPC users can be classed as those people that have made reasonable attempts to parallelize application codes, but have not been as successful as the experts due to their lack of parallelization experience. Their inexperience forces them to rely on judgments and experimentation when making decisions relating to parallelization. These users sometimes have access to large HPC systems, but their lack of parallelization experience means that they may require considerable time to produce a parallelization that utilizes it effectively, if such a version is ever achieved.

This leaves many other potential HPC users, including the code authors who specialize in other areas of science, still reliant on serial computation because the price of HPC machines and the expertise in parallelization are beyond their reach. Additionally, many users in this group want to exploit HPC for the speed and memory benefits, but do not want to know how it is done. As a result, many are awaiting fully automatic parallelizing compilers and may not all be amenable to parallelizing tools regardless of how simple the user's role is. As this group is the largest of those discussed here, providing tools that will be of interest to at least part of this market is seen as very important.

4 How ParaWise Solves These Problems

In attempting to efficiently parallelize a code, the ideal solution for the code parallelizer would be to use a fully automatic parallelizing compiler, however as discussed earlier, this is not viable. The other extreme of undertaking a manual parallelization does not provide a general solution due to the expertise and time required, particularly to a code author who specializes in other areas of science. The solution would therefore appear to be a hybrid approach, utilizing the best qualities of both the compiler technology and human insight. To this end, the ParaWise parallelization tool has been developed to deliver high quality parallel code using a source to source transformation.

ParaWise does not compromise parallel performance for automation and does not involve the user, except where essential. It uses techniques based on the "best manual practice" and expertise derived from the parallelization (both manually and using the tool) of numerous large real world application codes. Figure 1 illustrates where the ParaWise parallelization tool can be placed with respect to the level of automation, ease-of-use and parallel performance in relation to alternative methods ranging from automatic parallelizing compilers to fully manual MPI parallelization.

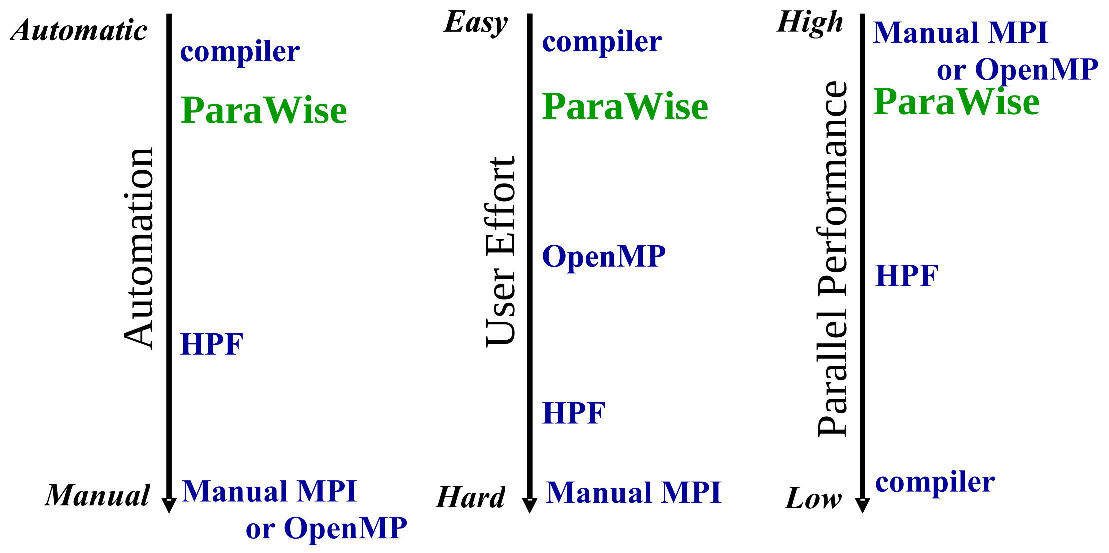


Figure 1 Placement of ParaWise in the context of level automation, the user effort required and the performance of the resultant parallel code, against a range of approaches.

To appeal to the large sector of the market that may be waiting in anticipation of fully automatic parallelizing compilers, ParaWise automates most of the parallelization process and has expertise in parallelization embedded in its algorithms. Although not fully automated like the parallelizing compiler, the level of automation is far higher than HPF, where the compiler only deals with inter-processor communication, and the fully manual MPI and OpenMP parallelizations. Similarly, the user effort required is kept low by performing a high quality analysis and only involving the user when it is unavoidable. Again, this requires more user effort than an automatic compiler, but is far easier than manual OpenMP parallelization which in turn is simpler than implementing HPF or manually using MPI. This is only of use, of course, if the resultant parallel code provides performance comparable to that produced manually. The analysis and code generation techniques used along with user interaction enable performance that is far superior to that achieved by HPF or fully automatic compilers and is comparable with manual MPI or OpenMP parallelizations.

5 The Components Of ParaWise That Enable Effective Parallelization

The ParaWise parallelization tool has been developed over the past 15 years by a team dedicated to widening and advancing the use of high performance computing. The powerful algorithms that underlie this sophisticated technology are based on experiences of manually parallelizing real-world application codes. Given a serial Fortran application code, ParaWise can be used to produce a parallel version of the code within a relatively short time frame and with little user effort, where the performance of the generated parallel code is comparable to that obtained by a manually parallelized code. The generated code can then be compiled and executed on any parallel platform using various processor topologies as specified at runtime.

5.1 Facilitating Efficient Parallelization Through Accurate Code Analysis

To be commercially acceptable, a compiler must perform a code parallelization in a fairly short space of time (typically, in the order of minutes) and this places a huge restriction on the level and detail to which, for example, the dependence analysis can be carried out. In an interactive system, however, the key restriction is that of human time, where a low quality analysis will greatly increase the required human time since the analysis will camouflage key information. With a poor analysis, the user will require considerable effort to optimize the generated parallel code, whose quality has been directly diminished by the poor analysis. ParaWise therefore adopts a strategy that performs an in-depth interprocedural, symbolic, value-based (data-flow) analysis [2] to give an accurate representation of actual data movement within the code. Numerous other techniques have also been developed based on the symbolic algebra and proving techniques in ParaWise to further enhance the quality of the resultant set of data dependencies. Although this kind of analysis can take longer to perform for large codes and for codes with complex control flow compared to a compiler level analysis, this is greatly offset by the increased accuracy provided and the subsequent reduction in user effort.

Dependence analysis must be conservative to ensure correct parallel code is produced. This forces a dependence between two statements to be set unless it is proven that no such relation exists. Each dependence forces an execution order on the related statements where one must be executed before the other. This can, for example, force a loop to execute in serial when a dependence from an assignment in an iteration leads to a usage in a later iteration of that same loop.

As dependence information is crucial to all phases in the subsequent parallelization, much of the user’s effort can directly or indirectly be as a result of assumed dependencies. The analysis used in ParaWise creates a very accurate graph of dependencies, however, there are a number of cases where assumed dependencies will still be set. A common case is when information about the values of variables that are read into an application at runtime is not available to a static analysis of the source code. The information can be as simple as knowing a variable will always have a positive value at runtime, however, the analysis must take the worst case assumption that the variable can have a positive or negative value and assume dependencies accordingly. This is not a deficiency of the analysis and only the user can provide the required constraints. ParaWise therefore allows the user to volunteer such knowledge and also asks the user questions that, if they can be answered positively, provide the key information required.

A number of browsers are provided to allow users to inspect the results of the code analysis. This allows the structure of the application code and dependencies to be examined by the user which can greatly assist in a manual parallelization. The main purpose of the analysis is, however, to provide a foundation for the parallel code generation algorithms of ParaWise.

5.2 Efficient And Scalable OpenMP Code Generation

The OpenMP directive code generation module in ParaWise (CAPO) [3, 4] has been developed at the NASA Ames Research Center and is based on the expertise and experiences of manually parallelizing codes for use on shared memory systems. It automatically inserts the relevant directives into the existing application code based on the accurate dependence information provided by ParaWise. Some of the techniques that ultimately generate efficient and scalable parallel code include:

- Automatic scoping of variables, for example determination of variables that need to be PRIVATE, FIRSTPRIVATE etc.
- Interprocedural operation without the need to inline code sections. This enables the parallelization of loops containing calls whilst preserving code appearance.
- Merging of parallel regions to increase uninterrupted parallel execution, reducing runtime overheads
- The discriminating use of the NOWAIT clause to avoid unnecessary synchronizations between loops in a parallel region
- Automatic routine cloning for varying levels of parallelism for different calls
- Automatic use of the THREADPRIVATE clause for common blocks, but applying a local transformation to a common block variable when this is illegal
- Use of OpenMP function calls to enforce software pipelines when inherently serial recurrence relations exist in a nest of loops

An example illustrating the power of the OpenMP code generation using a code fragment from a real application code is shown in Figure 2. The figure displays both the original serial code fragment and the generated parallel code.

SERIAL

```
do j=2,jmax
  call r2r(j,kmax,w)
enddo
call r2r(1,kmax,w)




subroutine r2r(j,kmax,w)
do k=1,kmax
  . . .
enddo
```

CAPO GENERATED OpenMP CODE

```
!$OMP PARALLEL DEFAULT(SHARED),PRIVATE(J)
!$OMP DO
      do j=2,jmax
        call cap_r2r(j,kmax,w)
      enddo
!$OMP ENDDO NOWAIT
      call r2r(1,kmax,w)
!$OMP END PARALLEL

      subroutine r2r(j,kmax,w)
!$OMP DO
      do k=1,kmax
        . . .
      enddo
!$OMP ENDDO

      subroutine cap_r2r(j,kmax,w)
      do k=1,kmax
        . . .
      enddo
```

Figure 2 Example of the code generation techniques used in OpenMP code generation

The production of this efficient parallel code resulted from the use of a number of powerful techniques. It uses interprocedural parallelism detection, automatic variable privatization, automatic routine cloning, interprocedural parallel region merger and avoidance of unnecessary synchronization within a parallel region. The interprocedural nature of this parallelization is further demonstrated in Figure 3 where the ParaWise Call Graph browser shows the highlighted call tree called from inside routine *r2r* (from Figure 2). Efficient parallel execution of this entire call tree within a distributed loop in routine *rhs* relied on dependencies determined interprocedurally for all data accesses in all the routines in the tree.

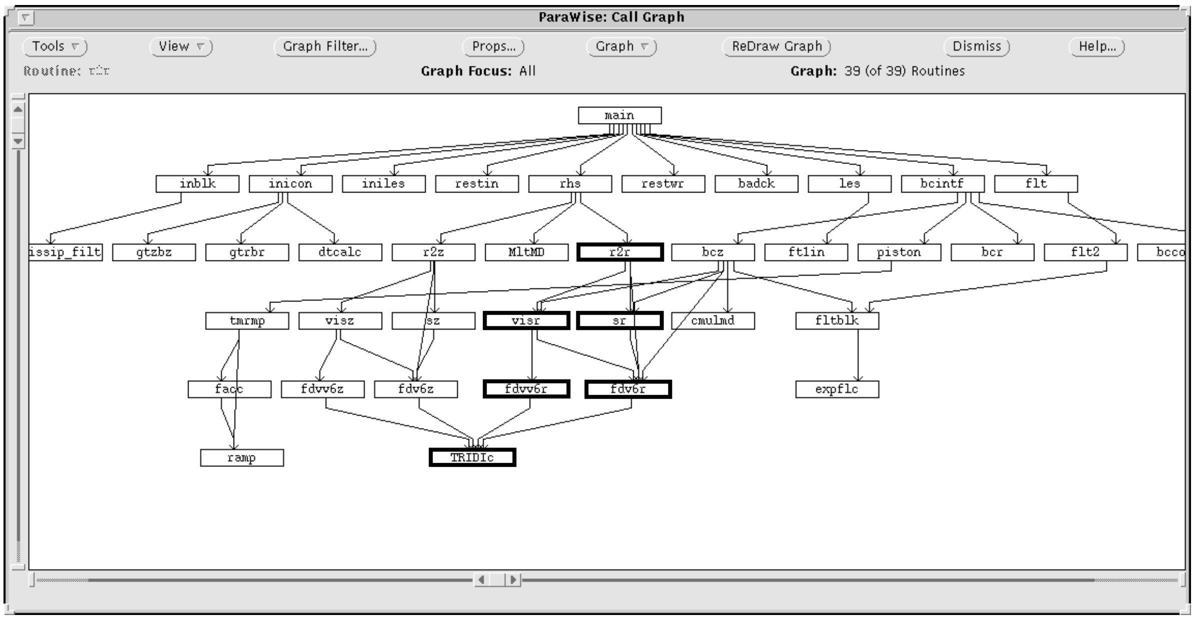
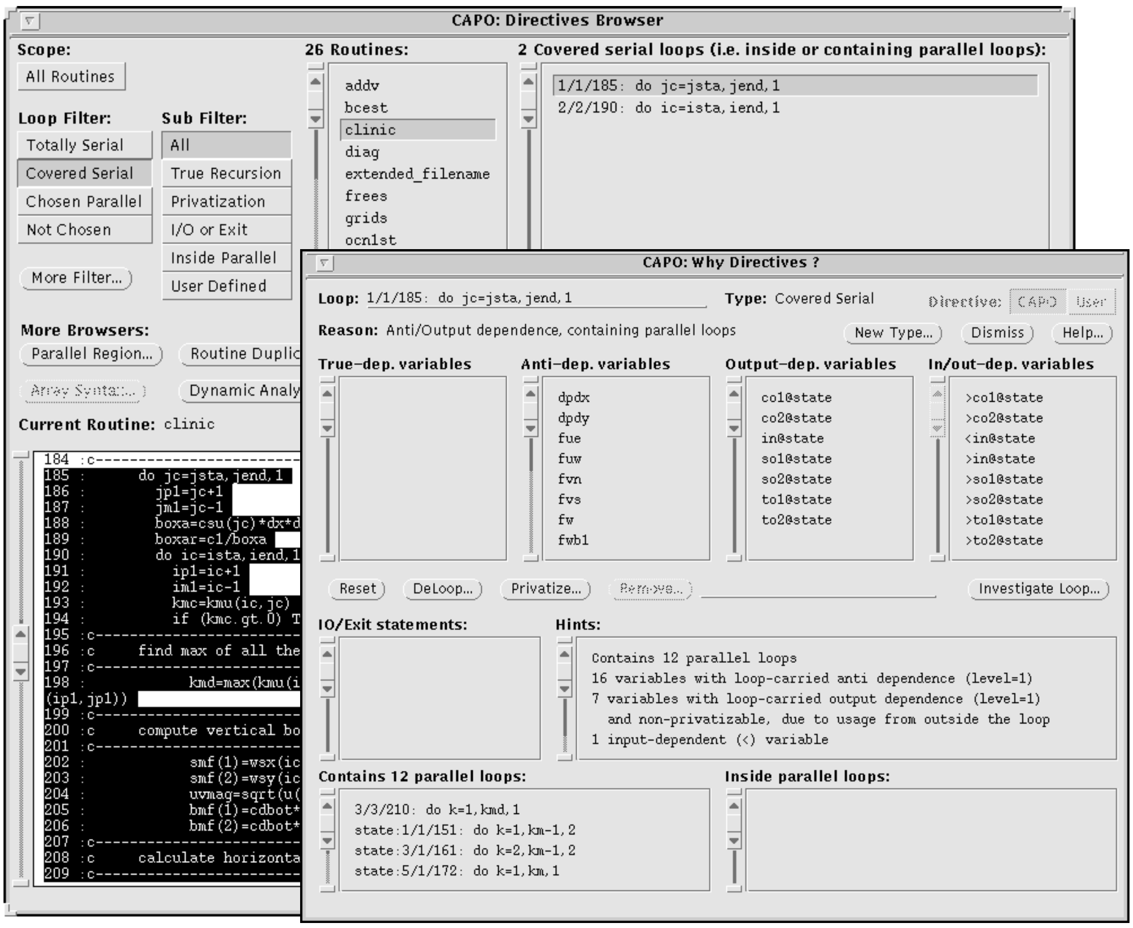


Figure 3 Call graph displayed in the ParaWise Call Graph browser for code fragment shown in Figure 2 highlighting the call tree called from within routine *r2r* that will execute in parallel

The generated directives are detailed and explained in the Directives browser as shown in Figure 4. The browser can be used to view loops that will be executed in parallel, along with the associated SHARED, PRIVATE, FIRSTPRIVATE, LASTPRIVATE and reduction variables that are displayed in the Why Directives browser. More importantly, it is also possible to use the browser to view loops that cannot execute in parallel in the Directives browser, along with an explanation of why parallelism could not be exploited, as demonstrated in Figure 4.



26 Routines:

```
addv
bcest
clinic
diag
extended_filename
frees
grids
ocn1st
```

2 Covered serial loops (i.e. inside or containing parallel loops):

```
1/1/185: do jc=jsta,jend,1
2/2/190: do ic=ista,iend,1
```

Current Routine: clinic

```
184 :c------------------------
185 :       do jc=jsta,jend,1
186 :         jp1=jc+1
187 :         jm1=jc-1
188 :         boxa=csu(jc)*dx*d
189 :         boxar=c1/boxa
190 :         do ic=ista,iend,1
191 :           ip1=ic+1
192 :           im1=ic-1
193 :           kmc=kmu(ic,jc)
194 :           if (kmc.gt.0) T
195 :c------------------------
196 :c      find max of all the
197 :c------------------------
198 :               kmd=max(kmu(i
(ip1,jp1))
199 :c------------------------
200 :c      compute vertical bo
201 :c------------------------
202 :               smf(1)=wsx(ic
203 :               smf(2)=wsy(ic
204 :               uvmag=sqrt(u(
205 :               bmf(1)=cdbot*
206 :               bmf(2)=cdbot*
207 :c------------------------
208 :c      calculate horizonta
209 :c------------------------
```

Loop: 1/1/185: do jc=jsta,jend,1   Type: Covered Serial

Reason: Anti/Output dependence, containing parallel loops

True-dep. variables	Anti-dep. variables	Output-dep. variables	In/out-dep. variables
	dpdx	co1@state	>co1@state
	dpdy	co2@state	>co2@state
	fue	in@state	<in@state
	fuw	so1@state	>in@state
	fvn	so2@state	>so1@state
	fvs	to1@state	>so2@state
	fw	to2@state	>to1@state
	fwb1		>to2@state

IO/Exit statements:

Hints:

```
Contains 12 parallel loops
16 variables with loop-carried anti dependence (level=1)
7 variables with loop-carried output dependence (level=1)
  and non-privatizable, due to usage from outside the loop
1 input-dependent (<) variable
```

Contains 12 parallel loops:

```
3/3/210: do k=1,kmd,1
state:1/1/151: do k=1,km-1,2
state:3/1/161: do k=2,km-1,2
state:5/1/172: do k=1,km,1
```

Inside parallel loops:

Figure 4 The Directive browser showing a serial loop with the inhibitors to parallel execution and contained parallel loops also displayed

Based on the information provided by the Directive Browser, the user can exploit their experience of OpenMP parallelization to enable parallel execution. This can, for example, involve selecting variables preventing parallelization because they could not be privatized and then using the "*Privatize*" button to force them to be private. Alternatively, the related data dependencies for those variables can be examined where, for example, knowledge required by ParaWise could be identified and added allowing this loop, and possibly many others, to use that information to prove that parallel execution is legal. To make this process accessible to non-expert parallelizers and code authors (as well as greatly assisting the experts), an investigation of the generated parallel code can be performed automatically using the ParaWise Expert Assistant as discussed in section 7.

5.3 Efficient and Scalable Message-Passing Code Generation

The ParaWise message-passing code generation algorithms use the best manual practice of the domain decomposition method that has proven successful in many manual parallelizations. The generated code follows the single program multiple data (SPMD) model where each processor executes a copy of the same executable but operates over a specific subset of the computational domain in the application code. This domain takes the form of program arrays that are distributed between the processors. It is essential that the entire application code is then parallelized so that any assignment to an array element where that array has been distributed is only performed on the processor that has been allocated the array element. The option of leaving a loop that processes distributed data to execute on a single processor is not possible as this would require full copies of the distributed arrays to be held on a that processor where the size of these full arrays may exceed the available local memory.

There are a number of stages involved in the creation of a message-passing version of an application code [5, 6]. Based on the dependence analysis, arrays in the application code can be automatically partitioned by ParaWise using an initial user prescription. The user's input to this process is as simple as selecting a routine, an array and an index of that array. This information is then inherited by all related arrays throughout the code using the ParaWise partitioning algorithms. ParaWise provides a number of data decomposition strategies including block, cyclic, block/cyclic and unstructured (based on a graph partition calculated by the Jostle [7] or Metis [8] tools). Figure 5 shows an example of a two dimensional block decomposition mapped onto a 3x3 grid of processors. The *cap_l*, *cap_h*, *cap2_l* and *cap2_h* variables are generated in the parallel code by ParaWise and calculated at runtime based on the number of processors requested. They are used to enforce the partition on each processor indicating the locally "owned" set of data. The selected strategy is dependent on the nature of the application code.

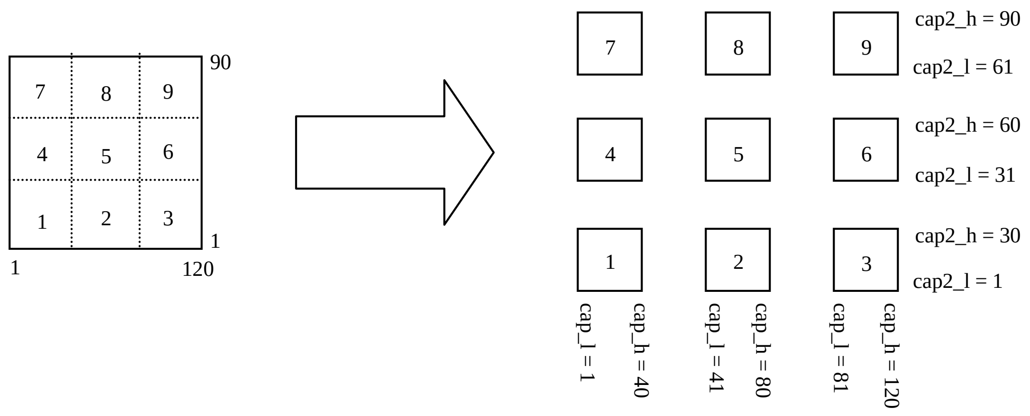


Figure 5 Example of a block data decomposition provided by ParaWise for arrays of a 120x90 mesh in an application code mapped onto a 3x3 grid of processors

The computation is then distributed by automatically adding execution control masks to selective statements. These are determined from the partitioned data and using a set of rules including the "owner computes" rule. Masks are set to control as much code as is possible, often controlling every computation in a loop allowing the loop iterations to be distributed across processors, even when the loop contains many calls. Ultimately the masks appear in the generated code either as loop limit changes or IF statements limiting statement execution to a particular subset of processors.

Communication requests are then determined for data used on one processor but computed by another. These requests are migrated up the code as far as is legally possible, out of loops and inter-procedurally into caller routines, reducing the frequency of communication and the associated runtime overheads in the parallel code. They are then merged into as small a number of requests as possible, avoiding repeated communication of the same data and also further reducing communication frequency.

The user may then either choose to perform a multi-dimensional parallelization [6] by repeating the process of partitioning, mask and communication generation for another dimension of the application code, or save the parallel code ready for compilation and execution.

An example of the code generated by ParaWise for a two dimensional block partition is shown in Figure 6. The interprocedural capabilities of ParaWise are again highlighted here as the computations in routine *r2r* and the call tree contained within it are all distributed in the same loop and so operate in parallel (where the call tree is shown in Figure 3). Both the *j* and *k* loops have been distributed automatically by altering their loop limits based on the variables used on each processor to enforce the data partition. The communication shown has migrated from many requests (usages) within this call tree, including the usage shown in routine *r2r*.

SERIAL

```
      do j=2,jmax
S1      call r2r(j,kmax,w)
      enddo
S2    call r2r(1,kmax,w)


      subroutine r2r(j,kmax,w)
      do k=1,kmax
        . . .
        work(j)=w(l,k,j+1)
        . . .
      enddo
```

ParaWise GENERATED MESSAGE-PASSING CODE

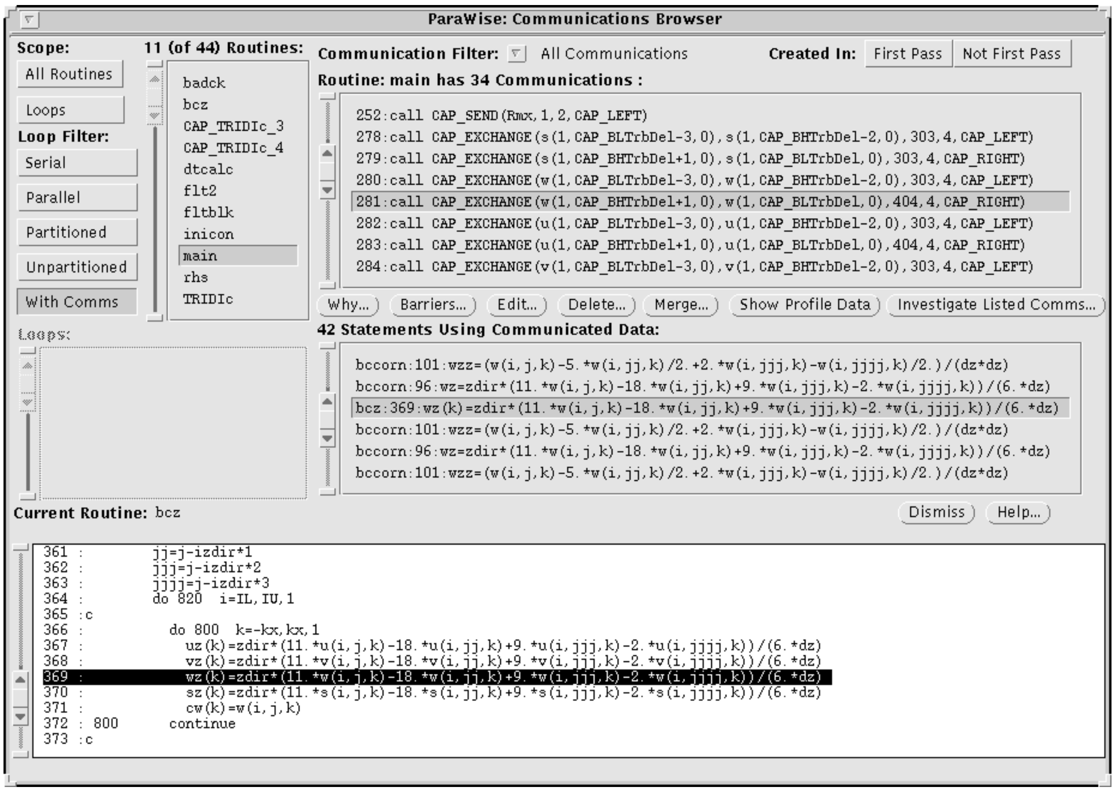
```
 call cap_bexchange(w(1,cap2_l,cap_h+1),
+                   w(1,cap2_l,cap_l),ni,siz1,
+                   cap2_h-cap2_l+1,3,cap_right)

 do j=max(2,cap_l),min(jmax,cap_h)
   call r2r(j,kmax,w)
 enddo
 if (1.ge.cap_l.and.1.le.cap_h) then
   call r2r(1,kmax,w)
 endif


 subroutine r2r(j,kmax,w)
 do k=max(1,cap2_l),min(kmax,cap2_h)
   . . .
   work(j)=w(l,k,j+1)
   . . .
 enddo
```

Figure 6 Example fragment of ParaWise message-passing code generation

Many other techniques have been devised to enhance the quality of the generated parallel code. For example, recurrences in distributed loops are handled by generating communications to form pipelines where parallelism can be exploited from iterations of outer loops. Also, a wide range of reduction style operations are detected and efficient code generated, for example, where the location and value of the maximum element in an array is required. A range of techniques have also been developed to encourage migration of communication requests out of loops by, for example, splitting a communication request when the subsequent requests are loop invariant and can therefore migrate out of more loops than the original (e.g. when an "upwind" scheme is used in a fluid dynamics code or when a code uses either conditional assignments or a function to implement periodic boundary updates).

Unstructured mesh codes use the well known inspector/executor technique [9, 10] where the inspector is used to interface the data structure used in the application code with the data structure required in the Jostle graph partitioner and the ParaWise library calls. These library calls are used to set up communication sets of the data needed from other processors to update the halo data on a processor and also to perform the communications using those sets. Techniques that build inspector loops from contributing statements in many routines and use an induction based proof to merge the inspectors have been implemented to promote migration out of loops and the re-use of communication sets. Many other techniques have been developed based on experience of real world unstructured mesh application codes (including finite element codes) and the best manual techniques for handling the cases encountered have been embedded in ParaWise.

ParaWise provides browsers for all the stages in message-passing code generation to display and explain to the user what has been done and why. Figure 7 shows an example of the ParaWise Communications browser showing a selected communication (highlighted) along with a list of the 42 statements from 5 different routines that requested and will use the communicated data. These requests have been automatically migrated out of surrounding loops and inter-procedurally into the *Main* program and then merged to form a single communication. This produces an efficient parallel code where the communication overheads are low enough to achieve high efficiency and scalability to larger numbers of processors.



11 (of 44) Routines:

```
badck
bcz
CAP_TRIDIc_3
CAP_TRIDIc_4
dtcalc
flt2
fltblk
inicon
main
rhs
TRIDIc
```

Routine: main has 34 Communications :

```
252:call CAP_SEND(Rmx,1,2,CAP_LEFT)
278:call CAP_EXCHANGE(s(1,CAP_BLTrbDel-3,0),s(1,CAP_BHTrbDel-2,0),303,4,CAP_LEFT)
279:call CAP_EXCHANGE(s(1,CAP_BHTrbDel+1,0),s(1,CAP_BLTrbDel,0),303,4,CAP_RIGHT)
280:call CAP_EXCHANGE(w(1,CAP_BLTrbDel-3,0),w(1,CAP_BHTrbDel-2,0),303,4,CAP_LEFT)
281:call CAP_EXCHANGE(w(1,CAP_BHTrbDel+1,0),w(1,CAP_BLTrbDel,0),404,4,CAP_RIGHT)
282:call CAP_EXCHANGE(u(1,CAP_BLTrbDel-3,0),u(1,CAP_BHTrbDel-2,0),303,4,CAP_LEFT)
283:call CAP_EXCHANGE(u(1,CAP_BHTrbDel+1,0),u(1,CAP_BLTrbDel,0),404,4,CAP_RIGHT)
284:call CAP_EXCHANGE(v(1,CAP_BLTrbDel-3,0),v(1,CAP_BHTrbDel-2,0),303,4,CAP_LEFT)
```

42 Statements Using Communicated Data:

```
bccorn:101:wzz=(w(i,j,k)-5.*w(i,jj,k)/2.+2.*w(i,jjj,k)-w(i,jjjj,k)/2.)/(dz*dz)
bccorn:96:wz=zdir*(11.*w(i,j,k)-18.*w(i,jj,k)+9.*w(i,jjj,k)-2.*w(i,jjjj,k))/(6.*dz)
bcz:369:wz(k)=zdir*(11.*w(i,j,k)-18.*w(i,jj,k)+9.*w(i,jjj,k)-2.*w(i,jjjj,k))/(6.*dz)
bccorn:101:wzz=(w(i,j,k)-5.*w(i,jj,k)/2.+2.*w(i,jjj,k)-w(i,jjjj,k)/2.)/(dz*dz)
bccorn:96:wz=zdir*(11.*w(i,j,k)-18.*w(i,jj,k)+9.*w(i,jjj,k)-2.*w(i,jjjj,k))/(6.*dz)
bccorn:101:wzz=(w(i,j,k)-5.*w(i,jj,k)/2.+2.*w(i,jjj,k)-w(i,jjjj,k)/2.)/(dz*dz)
```

Current Routine: bcz

```
361 :        jj=j-izdir*1
362 :        jjj=j-izdir*2
363 :        jjjj=j-izdir*3
364 :        do 820  i=IL,IU,1
365 :c
366 :          do 800  k=-kx,kx,1
367 :            uz(k)=zdir*(11.*u(i,j,k)-18.*u(i,jj,k)+9.*u(i,jjj,k)-2.*u(i,jjjj,k))/(6.*dz)
368 :            vz(k)=zdir*(11.*v(i,j,k)-18.*v(i,jj,k)+9.*v(i,jjj,k)-2.*v(i,jjjj,k))/(6.*dz)
369 :            wz(k)=zdir*(11.*w(i,j,k)-18.*w(i,jj,k)+9.*w(i,jjj,k)-2.*w(i,jjjj,k))/(6.*dz)
370 :            sz(k)=zdir*(11.*s(i,j,k)-18.*s(i,jj,k)+9.*s(i,jjj,k)-2.*s(i,jjjj,k))/(6.*dz)
371 :            cw(k)=w(i,j,k)
372 :  800     continue
373 :c
```

Figure 7 ParaWise Communications browser showing the generated communication statements

An examination of a communication in the Communication Browser reveals the precise reasons why it was required. This explanation includes which statement is using the data and on which processor, and where that data is owned. From this, the user may decide to try to improve the parallelization by adding knowledge about the value of variables, examining and possibly deleting dependencies, altering the details of the array partition or altering a statement's execution control mask. As with the OpenMP parallelization discussed in the previous section, the ParaWise Expert Assistant automates this investigation process allowing any user to add information about the application code to improve the quality of the generated parallel code (as described in section 7).

ParaWise generates message-passing calls to the ParaWise Communication Library (CAPLib) [11]. CAPLib is a thin layer library over MPI, PVM, Cray SHMEM and other propriety API's. It provides all the functionality required by the ParaWise code generation model. Using CAPLib allows the generated code to be portable across many systems whilst maintaining the flexibility of choice in which underlying message-passing library to use for maximum performance. CAPLib has been ported to several faster machine-specific API's and delivered superior performance than more general libraries such as MPI. The CAPLib library is freely available in binary (and source code on request) from the ParaWise web site (http://www.parallelsp.com).

CAPLib also provides commands to simplify the compilation and parallel execution of the generated parallel code which can be important for many users of parallel codes. For users of multiple platforms, parallel libraries and environments these commands offer the advantage of a uniform interface to parallel compilation and execution.

The CAPLib command *capf90* is used to compile and link ParaWise generated parallel code. The *capf90* script takes the set of user source files specified on the command line and then compiles and links them with the CAPLib libraries using an appropriate compiler and the required flags and options. Command options (that override the default environment variables) allow for multiple variants of parallel library and compiler to be supported simultaneously. Any specific compiler options are just passed through. The *capf90* command has a similar look and feel as a standard Linux/Unix f90 compiler. Some examples of usage are:

`capf90 fabpar.F –o fabpar`	Compiles and link with a default CAPLib library to create a parallel executable
`capf90 –p mpich-1.2.5 fabpar.F –o fabpar`	Compiles and link *fabpar.F* with a CAPLib library compiled to use mpich-1.2.5 to create a parallel executable
`capf90 –p shmem –m t3e fabpar.F –o fabpar`	Compiles the file *fabpar.F* with the Cray T3E SHMEM version of CAPLib

The *caprun* script is an optional command supplied with CAPLib that allows the user to run a previously generated executable under a particular parallel machine environment. In some environments such as MPI the user would normally just use the existing *mpirun* command, however for other environments (such as PVM) there is often a need for a simple command that takes care of any complex parallel execution setup. Here are some examples of using *caprun*:

`caprun –top grid8x8 fabpar`	Execute *fabpar* using a grid topology of 64 processors
`caprun –p mpich-1.2.5 –top pipe4 –hosts n1,n2,n3 fabpar`	Execute *fabpar* on current host and hosts n1,n2,n3

`caprun -p pvm3 -hostfile=hosts -nolocal -top pipe16 fabpar`	Execute *fabpar* on a pipeline of 16 processors using machines listed in *hostfile*, but do not include the local host.

The number of processors to be used does not need to be specified until runtime, with the chosen topology reflecting the partitioning type used during the partitioning stage of message-passing code generation.

6 Examples Of Speedup And Scalability Of ParaWise Generated Codes

The performance measures used here to determine the effectiveness of ParaWise are primarily based on speedup and the time to complete the parallelization. Speedup is defined to be the improvement in the performance of the parallelized application executed on a parallel system when compared to the execution of the original serial application on a single processor of the same parallel system. Ideally, one would like to see the performance double when the number of processors used is also doubled (referred to as a linear or ideal speedup). For real world applications this does not usually occur, but the performance is instead sub-linear as runtime overheads in the parallel execution have an effect. The scalability of the application is an indication of how beneficial it is to the overall parallel execution time when larger numbers of processors are used.

6.1 Speedup And Scalability Of Generated OpenMP Codes

It is widely accepted that the “best” parallel performance is achieved by manually parallelizing an application, however, this approach is usually expensive, error prone and time-consuming. Therefore, as well as measuring and contrasting speedup and scalability it is also worthwhile comparing the time taken to complete the parallelization using the respective approaches (in cases where a manual parallelization has been performed). Table 1 shows such a comparison for the OpenMP code generated by the tool for a number of applications. These range from benchmarks (BT,SP,LU), weather, ocean and cloud modelling applications (CTM, SEA, GCEM3D) to an aerospace application (OVERFLOW). ParaWise generates parallel code that is almost as efficient and as scalable as a manual parallelization but which is produced in a time frame that is at least an order of magnitude faster. Additionally, in many of these cases the input data used was related to a problem size small enough to run in a reasonable time using the original serial version. As a result, the parallel runtimes on larger numbers of processors are very low, reducing the speedup achieved. If input data for larger problem sizes were used (to provide greater accuracy in the computational models for example) then the parallel performance would typically improve (i.e. the speedup would increase) as the amount of computation would have risen making the parallel runtime overheads a smaller proportion of the total execution time.

Application	Approximate Code Size	Approximate ParaWise Parallelization Time	Approximate Manual Parallelization Time	Parallel Performance of Tools verses Manual	Sample Speedup
BT,SP,LU	3K lines benchmark	few hours	several weeks	within 5-10% each	BT: 30 on 32 threads on SGI O3K
CTM	16K lines, 105 routines	couple of days	several months	better by up to 30%	3.5 on 4 threads on Compaq SC45
SEA	7.3K lines, 26 routines	couple of hours	-	-	17 on 24 processors on SUN E2900
GCEM3D	18K lines, 100 routines	few days	few weeks*	better by a factor of 8*	24 on 32 threads on SGI O3K
OVERFLOW	100K lines, 851 routines	few days	many months	slightly better	16 on 32 threads on SGI O2K

Table 1 Comparison of performance and effectiveness of ParaWise against a manual parallelization approach using OpenMP directives (*manual parallelization of GCEM3D abandoned after using ParaWise/CAPO)

6.2 Speedup And Scalability Of Generated Message-Passing Codes

Table 2 shows the parallelization time and performance results on a range of platforms for the message-passing based approach to parallelizing a number of applications using ParaWise. These range from benchmarks (BT,SP,LU,ARC3D), an ocean modelling application (SEA), to aerospace (FPBNDRY7) and an oil reservoir modelling application (BOAST-PDL). Once again, ParaWise generates efficient and scalable parallel code. With the exception of the benchmark applications, the task of manually parallelizing the real world applications was not undertaken as this was assessed to be too expensive and time consuming. As with the OpenMP results, most of the test cases here involved fairly small problem sizes that completed quickly on larger numbers of processors. With larger problem sizes, the extra computation will typically improve the speedup obtained.

Application	Approximate Code Size	Approximate ParaWise Parallelization Time	Sample Speedup
BT, SP, LU	3K lines benchmark	few hours	BT: 215 on 16x16 processors on Cray T3E
ARC3D	3.6K lines, 25 routines	couple of hours	31 on 8x8 processors on Cray T3D
SEA	7.3K lines, 26 routines	couple of hours	53 on 64 processors on SGI O2K
FPBNDRY7	9.9K lines, 69 routines	few hours	24 on 32 processors on IBM SP
BOAST-PDL	8K lines, 48 routines	few hours	23 on 32 processors on Cray T3E

Table 2 Performance results of ParaWise generated parallel code using message-passing on different parallel platforms.

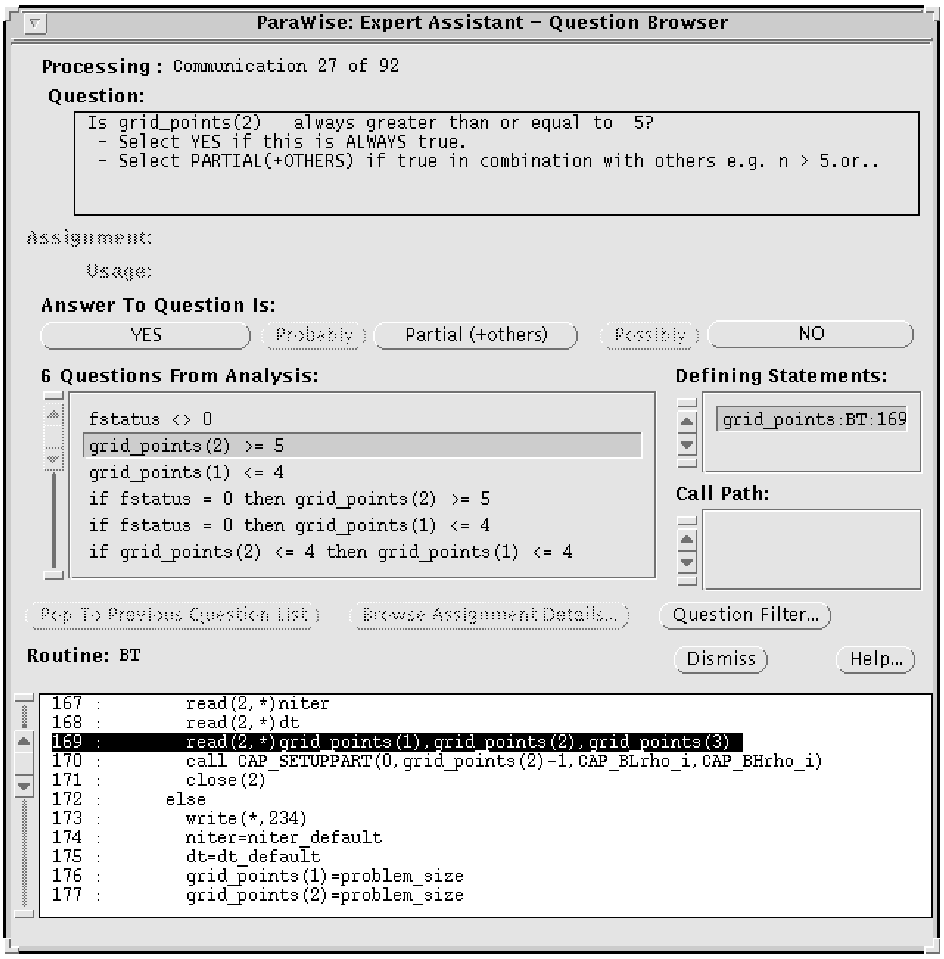
7 Widening Accessibility To Efficient ParaWise Generated Parallel Code

The analysis and code generation techniques of ParaWise produce efficient parallel code, but some user interaction is usually essential. To achieve the goal of making efficient parallelization accessible to code authors and others with no knowledge or interest in the details of parallelization, an Expert Assistant has been developed within ParaWise [12]. Its function is to examine the generated OpenMP or message passing code and determine what information the user could provide in order to improve parallel performance. Internally, the Expert Assistant examines serializing dependencies, non-privatizable variables etc. for loops in OpenMP codes and it examines communications, the associated execution control masks, array partitions and dependencies for message passing parallelizations.

The idea of an Expert Assistant within ParaWise will imply a different meaning depending on the skill of the user. To the expert in code parallelization it is perceived that they are the expert and that the ParaWise Expert Assistant is there to assist them. To the code author, the ParaWise Expert Assistant is the expert (with the knowledge of an expert embedded within it) that is assisting them. It may be perceived that the Expert Assistant is effectively using the user as an extra source of information about their application code to improve the parallelization. Using the Expert Assistant greatly simplifies the thought process required to effectively use ParaWise to generate efficient and scalable parallel code.

Typically, the main questions presented by the Expert Assistant to the user relate to constraints on the value of variables that are read into an application code at runtime (e.g. is a variable always positive?), dataflow in an implemented algorithm, and the need for I/O statements that have impeded efficient parallelization. For example, in Figure 8 the user may know that the value of `grid_points(2)` is always greater than or equal to 5 when read in at runtime, and so this information can be submitted to the Expert Assistant by selecting that question and pressing the *YES* button.

To answer the questions posed by the Expert Assistant, only information about the application code is required. This allows a code author who is familiar with the meaning of variables and the nature of the algorithms implemented in the application code to answer the questions and assist the tool to enable effective parallel code to be generated. For the parallelization expert, the questions are often the same as those they would have to ask of the code author to successfully perform a manual parallelization. In both cases, the complex, interprocedural investigation of the application code is performed by the Expert Assistant. It also exploits the full power of the ParaWise analysis and code generation algorithms, asking questions about the value of variables read into the application code, rather than locally computed values whose meaning may not be so clear. It also enables many questions to be answered automatically by the Expert Assistant, avoiding the need to involve the user in such cases.



Processing : Communication 27 of 92

Question:

```
Is grid_points(2)   always greater than or equal to  5?
 - Select YES if this is ALWAYS true.
 - Select PARTIAL(+OTHERS) if true in combination with others e.g. n > 5.or..
```

6 Questions From Analysis:

```
fstatus <> 0
grid_points(2) >= 5
grid_points(1) <= 4
if fstatus = 0 then grid_points(2) >= 5
if fstatus = 0 then grid_points(1) <= 4
if grid_points(2) <= 4 then grid_points(1) <= 4
```

Defining Statements: grid_points:BT:169

Routine: BT

```
167 :        read(2,*)niter
168 :        read(2,*)dt
169 :        read(2,*)grid_points(1),grid_points(2),grid_points(3)
170 :        call CAP_SETUPPART(0,grid_points(2)-1,CAP_BLrho_i,CAP_BHrho_i)
171 :        close(2)
172 :      else
173 :        write(*,234)
174 :        niter=niter_default
175 :        dt=dt_default
176 :        grid_points(1)=problem_size
177 :        grid_points(2)=problem_size
```

Figure 8 Illustration of how the user is asked a question via the Expert Assistant window

8 Conclusion

The combination of analysis techniques, code generation algorithms and the Expert Assistant can greatly improve the productivity of expert parallelizers and also enable others to produce efficient scalable parallel code. Experience has shown that user involvement is essential in order to achieve speedup and scalability for all the real world application codes parallelized. This interaction can be achieved using the Expert Assistant and only involves questions about the application code.

The success of the tool in the HPC market is dependent on whether users believe they have a need for the power that the tool provides. There is some reticence from parallelization experts to adopt ParaWise as they perceive a threat to their specialist work. We believe that the benefits of using ParaWise are clear, and that parallelization experts should view this tool as an aid in the task of code parallelization, where they can interpret and take advantage of all the information provided by the ParaWise browsers. To further counter this reticence, code authors who are only interested in a fully automatic compiler will still require expert parallelizers to port their application codes for them. So using ParaWise, the expert can improve their productivity and become more cost effective, which in turn may make their services more attractive to new customers interested in HPC.

From our experience, the most receptive market for the tool appears to be the non-expert parallelizers who will typically be relieved of both the mundane and complex aspects of code parallelization without significant fears for job security.

The remaining group (the code authors) may be more difficult to convince that they will be able to use ParaWise to produce effective parallel code, since historical problems for the acceptance of tools in this market are still significant today. Some may be deterred as user involvement is required, but it is hoped that many others will recognize that ParaWise can make code parallelization accessible to them in a short time frame and without a steep learning curve. Obviously, success stories from code authors and other groups should help to encourage further interest in the use of ParaWise and other parallelizations tools.

Finally, although ParaWise goes a long way to provide users with the fundamental parallelization tool that they evidently need, there is also a further requirement for other tools (such as profiling tools and debuggers) that would aid productivity in conjunction with using ParaWise. When manually parallelizing a code, for example, a profiling tool is often used to ascertain the values of particular variables in the application code that can assist in the decision making throughout the parallelization process, where this information is also crucial during the optimization stage. Knowing which information is vital to the performance of the parallelized code is a skill that only comes with experience, and in general is a difficult task when presented with an overwhelming amount of information from the profiling tool (which still has to be interpreted by the user). Irrespective of whether the parallel code has been generated using ParaWise, the task of recognizing and interpreting the relevant information from the profile output is particularly difficult for code authors and non-expert parallelizers, which is why work is already underway to incorporate the processing and interpretation of included profile information into the ParaWise parallelization process [13]. Also, a debugger that automatically identifies the first significant discrepancy in the values produced by the user's original serial code and parallel code generated by ParaWise is under development [14], which could be used to inform the user about any incorrect information that was submitted to ParaWise during the parallelization process. When completed, this work will provide the user with a comprehensive environment that will make code parallelization a commonplace activity in science and industry.

References

1 G. Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities”, AFIPS Conference Proceedings, (30), pp. 483-485, 1967.

2 S.P.Johnson, M.Cross and M.Everett “Exploitation of Symbolic Information In Interprocedural Dependence Analysis”, Parallel Computing, 22, 197-226, (1996).

3 H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez and X. Martorell. “Automatic Multilevel Parallelization Using OpenMP”, Proceedings of EWOMP2001, (2001).

4 H. Jin and G. Jost. “Experience on the Parallelization of a Cloud Modeling Code Using Computer-Aided Tools”. NAS Technical Report NAS-03-006, (2003).

5 S.P. Johnson, C.S. Ierotheou, and M. Cross. “Automatic Parallel Code Generation For Message Passing On Distributed Memory Systems”, Parallel Computing, 22(2):227--258, (1996).

6 E.W. Evans, S.P. Johnson, P.F. Leggett and M. Cross. “Automatic And Effective Multi-Dimensional Parallelisation Of Structured Mesh Based Codes”, Parallel Computing 26, pp 677-703, (2000).

7 http://www.gre.ac.uk/~c.walshaw/jostle

8 http://www-users.cs.umn.edu/~karypis/metis

9 K. McManus, S. P. Johnson, M. Cross “Converting best manual practice into generic automatable strategies for unstructured mesh parallelisation.”, Concurrency - Practice and Experience 11(11): 593-614, (1999).

10 S.P. Johnson, C. Ierotheou and M. Cross. “Computer Aided Parallelisation Of Unstructured Mesh Codes”, Proceedings for PDPTA 1997, Volume 1, pages 344--353, (1997).

11 P.F.Leggett, S.P.Johnson and M.Cross "CAPLib – A ‘Thin Layer' Message Passing Library to support computational mechanics codes on distributed memory parallel systems", Advances in Engineering Software 32(1): 61-83, (2001).

12 S. Johnson, H. Jin and C. Ierotheou. “The ParaWise Expert Assistant – Widening Accessibility to Efficient, Scalable Tool Generated OpenMP Code”. Proceedings of WOMPAT 2004, (2004).

13 G. Jost, R. Chun, H. Jin, J. Labarta and J. Gimenez. “An Expert Assistant for Computer Aided Parallelization”. To appear in proceedings for PARA04, (2004).

14 G. Matthews, R. Hood, S. Johnson, P. F. Leggett “Backtracking and Re-Execution in the Automatic Debugging of Parallelized Programs”, 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11, p150-,(2002).