Technical Report
RAL-TR-2003-012

CLRC

Mixed MPI - OpenMP Programming: A Study in Parallelisation of a CFD Multiblock Code

R F Fowler C Greenough

16th April 2003

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

Library and Information Services
CCLRC Rutherford Appleton Laboratory
Chilton Didcot
Oxfordshire OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44 (0)1235 446403
Email: library@rl.ac.uk

CCLRC reports are available online at:
http://www.clrc.ac.uk/Activity/ACTIVITY=Publications;SECTION=225;

ISSN 1358-6254

Mixed MPI - OpenMP Programming: A Study In Parallelisation Of A CFD Multiblock Code

R.F. Fowler and C. Greenough

Computational Science and Engineering Department
CLRC Rutherford Appleton Laboratory,
Chilton, Didcot OX11 0QX, UK

Email: r.f.fowler@rl.ac.uk or c.greenough@rl.ac.uk

This report is available from http://www.cse.clrc.ac.uk/Activity/HPCI

December 16, 2002

Abstract

Two of the most common and best standardised methods for parallel coding of scientific and engineering applications are OpenMP and MPI. OpenMP is now widely supported on shared memory systems while MPI is available on almost all distributed and shared memory machines. With the advent of more machines that combine both shared and distributed memory architectures it is becoming more common to combine both techniques in a single application. In this report we look at a simple case study of the parallelisation of a multiblock CFD code using both methods. The objective was to rapidly add some parallelism to an existing serial code to exploit a small Beowulf cluster of dual processor PC nodes. It is shown that combining both techniques can yield better performance then either method on its own, without much additional work in code and algorithm restructuring.

Keywords: Fortran 90, MPI, OpenMP

Contents

1 Introduction

Many scientific and engineering applications require use of substantial computational resources. Parallel systems are the only way to meet the most demanding computational requirements and currently there is a growth in popularity of cluster computing, such as Beowulf systems. In the past such clusters tended to consist of single processors with some fast interconnect to give the usual distributed memory architecture. Recently it has become more common to see systems where each node is actually a shared memory multiprocessor. These can range from cheap dual processor PC style board, using Athlon or Xeon CPUs, up to multi-way SMP systems from SGI, IBM, SUN, etc.

While such systems can offer vast computational power, there is the question of how best to utilise such non-uniform architectures. A thread based method, such as OpenMP, gives a relatively easy way to exploit parallelism on single SMP nodes, but does not work across cluster nodes. A message passing implementation, like one based on MPI, will work both within an SMP node and also across the cluster. However, the message passing code is usually more complex to develop.

Another possible approach is to combine both message passing between nodes with a thread based parallelism on the individual processors. For some applications this may offer advantages over either approach on its own. A technical watch report [1] has looked into some of the cases for which such mixed programming methods are useful. References therein show that such techniques are been successfully used in a wide range of applications.

In this report we look at a particular case study where combining MPI and OpenMP methods can prove useful. This is where an existing application is to be ported to a small Beowulf cluster of dual processor nodes. The code in question is a large and complex CFD application modelling 3D flow over turbine blades. Since the serial code is still under development, and currently has very limited documentation, it was desirable to make as few changes as possible to the existing structure. This would suggest an OpenMP approach, but with a target architecture of a cluster of dual processor nodes, this limits the performance gain to a factor of two. The most efficient solution in terms of scalability on this architecture would be one that involves a careful decomposition combined with algorithmic changes to minimise communication costs. This would require considerable time and effort both for the development of the message passing implementation and to verify the correctness of the changes.

It was seen as more cost effective to implement a message passing solution that would only have limited parallelism, but would preserve both the structure and algorithms of the original code. Adding OpenMP parallelism to this would enable useful speed ups to be obtained on the target architecture.

2 Genesis: A CFD application for turbine blade design

2.1 Overview

In this section we give a brief overview of the Genesis CFD solver. This package has been developed by ALSTOM for use in their design studies of turbine blades. Currently the code solves the steady state flow conditions over sets of turbines blades in either two or three dimensions. The more realistic three dimensional cases can easily take a day or longer to run on current workstations. For blade optimisation it is necessary to perform a large number of such calculations with varying geometries and physical conditions.

The serial version of the code consists of approximately 25,000 lines of code in over 100 subroutines and functions. The code base was originally Fortran 77, but has now been updated to use many Fortran 90 features. Typical of many codes that have been developed by several authors in an originally ad hoc fashion, the software has limited documentation and comments. This is being changed, with newer developments more extensively documented, but the task of understanding the algorithms and control flow is non-trivial.

2.2 Execution analysis

When presented with such an application code, the first step is to gain a better understanding of the structure and control flow within the serial code. Two types of software tools are useful in this part of the work:

- Basic execution profiling. On many machines this is built into the compiler. Profiling is usually done at the subroutine level, though some compilers also offer line level profiling.
- Call tree diagram generation and call count statistics. Tools such as ftnchek can generate basic call trees for Fortran 77 codes. Many Fortran 90 compilers are also able to provide call tree listings. These are useful to get a better understanding of the control flow within the code. Some profiling compilers also give more detailed statistics about how many times each routine was called from another in test runs.

Care has to be taken in interpreting profiling results, particular at the line level, as the overheads involved can distort the run times. Profiling on more than one test case and using different compilers and hardware is important to get an accurate understanding of the costs involved. Sometimes it is useful to include explicit timing measurements into the software to resolve certain details, such as the cost of a function call from one particular location. A good resolution system time function is needed for this. Compiler profiling tends to just give the total cost of all calls to a particular function.

An example of the profiling results obtained for the Genesis software is shown in Table 1. This function level profiling result shows the routines that are responsible for over 95% of the total run time. In this case a mesh with 237900 cells was used running on a single 1200MHz AMD Athlon processor. The Portland Group compiler, pgf90, was used with function level profiling to generate these results. A graphical display of profiing data is shown in Figure 1. This was also generated using pgf90, though on a 2D test case.

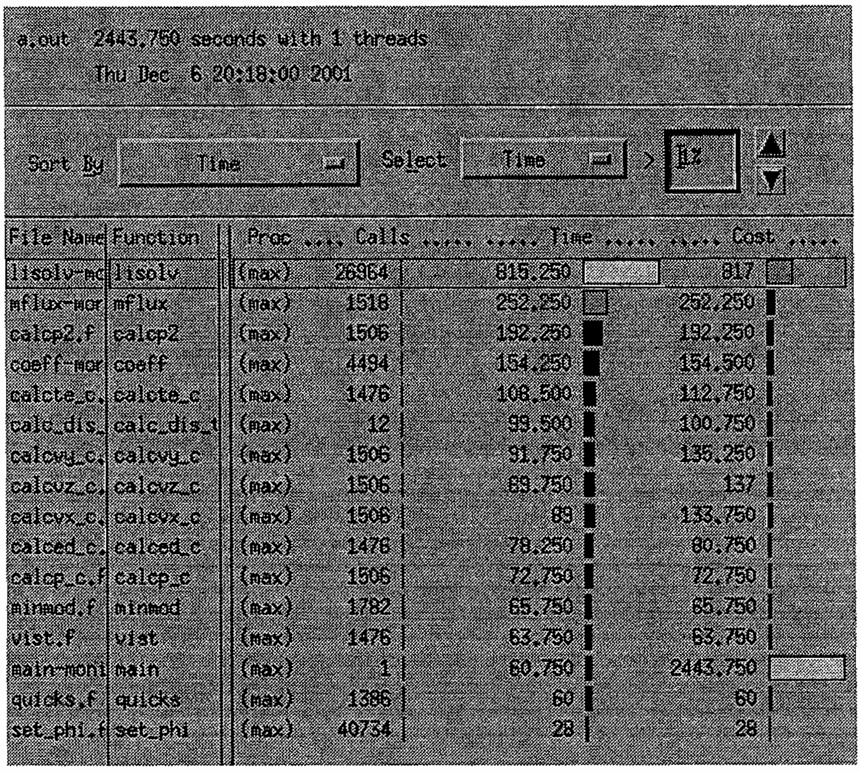
While profiling results identify which parts of the code are most important computationally, it is still necessary to understand the control flow and algorithms used to make a sensible parallel implementation. An example of the call tree output that can be obtained from SUN Fortran 90 compiler and the associated Workshop software is shown in Figure 2. This aides analysis of control flow which, along with inspection of the actual code, allows better understanding of the algorithms used by the software.

The detailed control flow within the Genesis code is quite complex and, as mentioned previously not very well documented. Using the call tree and direct code inspection the major operations of the software are as follows:

- Data input. This includes reading mesh data and run parameters.
- Initialisation. Preprocessing steps set the starting conditions.
- Solution on a coarse mesh followed by interpolation onto the fine mesh.

Function	Calls	Time(s)	Percent
lisolv	289380	7858.7	34.10
mflux	15006	2637.4	11.46
calcp2	15000	1884.7	8.19
coeff	44706	1647.1	7.16
quicks	30618	1368.7	5.95
calcte_c	14700	1098.0	4.77
calced_c	14700	862.4	3.75
calcp_c	15000	809.7	3.52
main	1	712.8	3.10
vist	14700	708.4	3.08
calcvy_c	15000	684.3	2.98
calcvz_c	15000	673.5	2.93
calcvx_c	15000	661.9	2.88
set_phi	430134	353.4	1.54
density	15006	309.3	1.34
calcho_c	15000	215.6	0.94

Table 1: Function profile results for the serial version of Genesis on a 3D test case.



a.out 2443.750 seconds with 1 threads
Thu Dec 6 20:18:00 2001

Sort By: Time
Select: Time > 1%

File Name	Function	Proc	Calls	Time	Cost
lisolv-mc	lisolv	(max)	26964	815.250	817
mflux-mor	mflux	(max)	1518	252.250	252.250
calcp2.f	calcp2	(max)	1506	192.250	192.250
coeff-mor	coeff	(max)	4494	154.250	154.500
calcte_c.	calcte_c	(max)	1476	108.500	112.750
calc_dis_	calc_dis_t	(max)	12	99.500	100.750
calcvy_c.	calcvy_c	(max)	1506	91.750	135.250
calcvz_c.	calcvz_c	(max)	1506	89.750	137
calcvx_c.	calcvx_c	(max)	1506	89	133.750
calced_c.	calced_c	(max)	1476	78.250	80.750
calcp_c.f	calcp_c	(max)	1506	72.750	72.750
minmod.f	minmod	(max)	1782	65.750	65.750
vist.f	vist	(max)	1476	63.750	63.750
main-moni	main	(max)	1	60.750	2443.750
quicks.f	quicks	(max)	1386	60	60
set_phi.f	set_phi	(max)	40734	28	28

Figure 1: Graphical display of profile data by Portland Group Fortran compiler for a 2D test case.

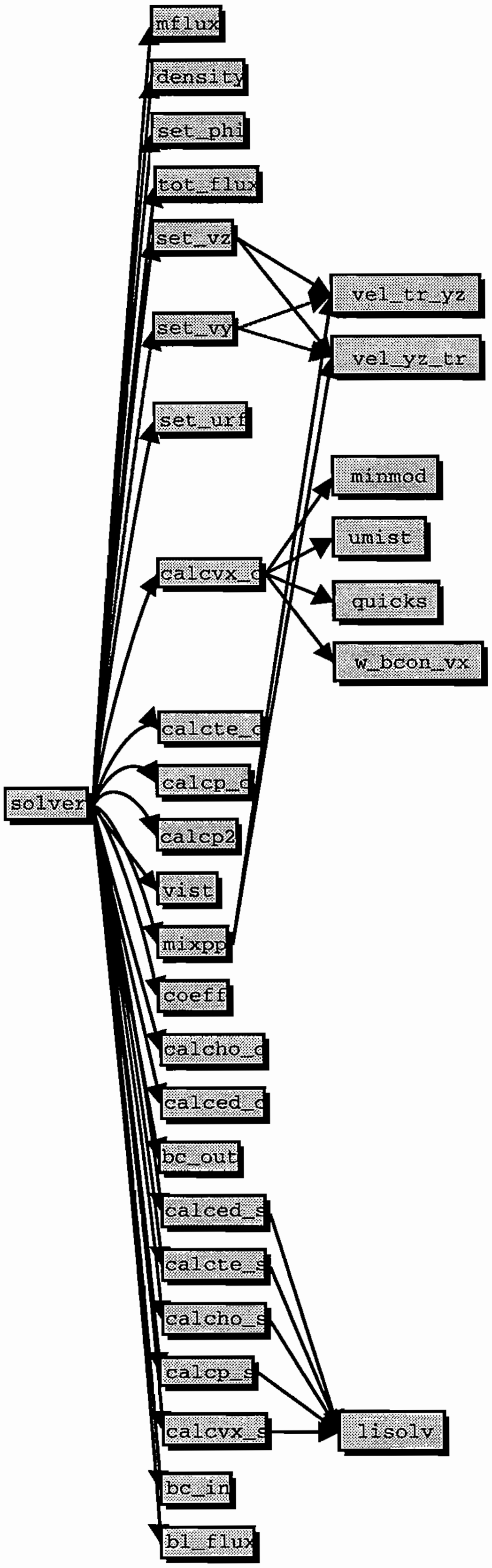


Figure 2: Call tree output from the SUN Workshop suite for the Genesis "`solver`" subroutine.

- Solution on the fine mesh.
- Output of final data to file.

Such an approach is typical of many steady state mesh based simulations.

The multiblock nature of the code is most evident in the solution process that is actually used on the coarse and fine meshes. In general the blocks are treated as independent systems. Discrete forms of the appropriate equations are formed independently on each block and partially solved by a few iterations of an iterative solver. Coupling between blocks is then achieved by having a mesh overlap at block boundaries. After each block has been partially solved there is an interchange of boundary data in a "halo" region surrounding each of them. This ensures a global convergence.

3 Parallel Methods for Mesh Based Computations

There are a range of methods which can be used to parallelise existing applications such as Genesis. Some of the most common methods are as follows:

Automatic compiler parallelisation. This is obviously the easiest method to use and depends of the compiler being able to generate parallel code from the serial version. It is supported by many vendors on shared memory systems. However the results can be disappointing in some cases. The Genesis code is a case where automatic compiler parallelisation produces worse results than the serial case. The reasons for this are:

- The most important computational loops contain a recurrence, which prevented the compiler from making them parallel.
- Many loops are long and complex, with indirect addressing of some arrays. This inhibits parallelisation as the complier cannot be sure that no recurrence exists.
- High level loops, such as those over mesh blocks, are not made parallel because they contain subroutine calls.
- The support of both 2D and 3D cases means that outer loops often only have a range of one for 2D problems. Automatic parallelisation usually selects the outer most loops, and will be ineffective on 2D problems.

For example, using the SGI Fortran 90 with its auto-parallelisation option, only 243 of 1314 loops within Genesis are deemed safe to make parallel, and these represent only a small fraction of the total computational work. The most time consuming loops are generally the most complex and it is these that the compiler fails to make parallel, even though it is safe to do so in most cases. A speed up of the order of 1.2 on 4 processors was seen for a 3D test case for compiler parallelisation, using SGI's f90 compiler.

OpenMP parallelisation. This is method for shared memory machines is now supported by most SMP machines. Comment style directives allow the user to define which loops should be made parallel and how local variables are to be treated. This means that serial code base can be retained in most cases, making maintenance of serial and parallel version of the software easier. However, in some cases it may be necessary to alter the actual code. Examples of this might be:

- To reorder a loop computation to make it parallel, perhaps avoiding a recurrence.
- To explicitly control the number of execution threads used by parts of the program. This can be done using special OpenMP function calls.

By concentrating only on the most time consuming loops the user can often find those which are safe to make parallel more efficiently then the compiler can. OpenMP can also be applied to parallelism at higher level, for example executing several subroutines in parallel.

To get optimum performance on large numbers of processors some SMP manufactures (e.g. SGI, Compaq) provide additional directives to control the placement of data arrays [3]. These directives are not yet standardised and hence are not portable.

Message Passing Parallelisation. Message passing parallelisation requires much more work to implement than a loop based OpenMP solution, since the user has to manage all data movements explicitly. It is of course applicable to both shared and distributed memory architectures. The standard MPI interface library is now available on almost all parallel machines, with both public domain and proprietary implementations available. Other message passing are available, such as PVM and BSP, but these are less widely implemented.

High Performance Fortran. This is another standard for parallelism through use of directives. Unlike OpenMP, HPF is aimed at distributed memory architectures and hence includes commands to control the data placement, as well as which loops should be made parallel. It is not as widely supported as OpenMP and MPI, though several vendors do now provide compilers. Benchmarks show that good scalability can be achieved for some cases [2].

As the target architecture for the Genesis program was to include distributed memory machines, it was clear that a solution would have to include either explicit message passing or HPF approaches. The message passing approach was selected on the grounds that the MPI standard is more widely implemented. While several vendors offer HPF compilers, MPI is still thought to be a more fully portable option.

While OpenMP is limited to shared memory machines, and therefore could not offer a complete solution, it may still be useful to employ this on individual nodes. This combination of message passing and OpenMP has been successfully exploited in previous parallelisation work [1]. In this case we wish to evaluate the effectiveness of the method for rapid parallelisation on a small Beowulf system, of a type which is becoming affordable to small research groups.

4 OpenMP Parallelisation

4.1 Parallel options

The structure of the Genesis code is such that there are two obvious levels at which OpenMP parallelisation could be used:

- Loop level, applying parallel directives to the main computational loops.
- Block level. The multiblock code performs computations over all the available mesh blocks

The are three main advantages in applying OpenMP at a high level, over the operations that are performed on mesh blocks. The first of these is the relative simplicity of implementation: the changes would only have to be made within one subroutine which performs the most important loops over blocks. The second advantage is that there is a lot of work associated with each block operation so that the any start up overhead associated the parallel loop should be negligible.

Finally, with the block based parallelism, the linear solver does not have to be modified from the serial version, since a separate copy acts on each mesh block.

There are also some serious disadvantages to using OpenMP at this high level for the Genesis application. Most significant of these is that real problems tend to have only a small number of mesh blocks in total, typically between 5 and 20, and these vary greatly in size. While this would not be too bad when running the whole computation on one dual processor node, it can be a problem when trying to combine this with MPI based parallelism. The number of blocks assigned to each processor node will fall and load balance will soon be a problem even with only dual CPUs on each node.

Even though the requirements of this project suggest the a loop level OpenMP parallelisation would be most appropriate, we have chosen to implement both methods. The block level OpenMP parallelisation is relatively simple to implement and it can be compared with the efficiency of the loop level version.

4.2 Block level parallelisation

The block level implementation of OpenMP mainly involved changes to the high level subroutine `solver`. This is called repeatedly to perform a single set of iterations over all the physical variables and over all mesh blocks.

One problem in using OpenMP at a high level is that it is necessary to ensure that all subroutine calls made from the parallel loops in the controlling routine are thread safe. An example of the OpenMP parallel version of one loop within the main computational routine is shown in Figure 3.

```
C$OMP PARALLEL DO PRIVATE(N_BL) SCHEDULE(DYNAMIC)
       DO N\_BL1=1, M_BL
          N_BL = ORDER(N_BL1)
          CALL calced\_c(N_BL)
       END DO
C$OMP END PARALLEL DO
```

Figure 3: OpenMP constructs to parallelise one of the loops over mesh blocks. The called subroutine and all its descendants must be thread safe.

Inspection of the Genesis software shows that, though the vast majority of the code called from the loops over blocks is thread safe, there is one area where module arrays are shared between separate block iterations. If two or more threads try reading and writing to this area at the same time the results will be ill defined. This can be avoided by replacing the module arrays with workspace that is dynamically allocated as required. This makes the parallelisation task rather more complex than it would be other wise. For debugging such problems it is helpful if the parallel solution should be identical to the serial case, i.e. no changes have to be made to the algorithms used. In this case the only source for differences between the serial and parallel solutions is the fact that certain sums may be performed in a different order in the latter case. These effects can be minimised by using extended precision for the summations.

As there are only a small number of mesh blocks available, between 6 and 14 in the test cases, the parallelism is quite limited. Load balancing will be a problem because the size of the mesh blocks can vary significantly. To minimise the effect of this, the mesh blocks have been ordered in terms of block size, so that the largest ones are processed first. This is combined with explicit requests for dynamic scheduling of the parallel loops.

Within the solver routine there are 22 loops similar to that shown in Figure 3 which can be executed in parallel. From typical profiling data, such as that described previously, it is known that 95% of the serial CPU time is consumed by the solver, and the subroutines that it calls. Hence it sufficient to focus only on this routine when using small numbers of processors.

In addition to the OpenMP directives added to the solver, changes had to be made to six other routines in the solver to ensure that they were thread safe. This required allocating temporary work space arrays in each thread, rather than using a shared data area within a module. OpenMP version 2 allows module data to be declared thread private, in a similar way to which common blocks can be made thread private in the current version. However, not all compilers currently support version 2, and it was more convenient to change Genesis to use local work arrays in this case.

4.3 Loop level parallelisation

The loop level parallel version of Genesis requires rather more changes to the code than the block level version. The number of changes is of course dependent on how many loops are thought worthwhile of parallelisation. From the previous profiling data the 15 most time consuming subroutines were selected for parallelisation. These represented over 95% of the total CPU time.

Within these routines 35 separate loops were identified as been significant enough justify using OpenMP directives on. In many cases these loops were particularly long and complicated. One difficulty with using OpenMP on such large loops is that it is necessary to identify and declare the type of local variables. In general most arrays were left as shared objects (the default in OpenMP) while the majority of scalars were made private to each thread. An example OpenMP header from one such loop is shown in Figure 4.

```
C$OMP PARALLEL DO PRIVATE( apu_e, apu_l, apu_n, apu_s, apu_u,
C$OMP& apu_w, apv_e, apv_l, apv_n, apv_s, apv_u, apv_w, apw_e,
C$OMP& apw_l, apw_n, apw_s, apw_u, apw_w, areae_jx, areae_jy,
C$OMP& areae_jz, areal_kx, areal_ky, areal_kz,
C$OMP& arean_ix, arean_iy, arean_iz,
C$OMP& areas_ix, areas_iy, areas_iz, areau_kx,
C$OMP& areau_ky, areau_kz, areaw_jx,
C$OMP& areaw_jy, areaw_jz, coefin, coefis, coefje, coefjw,
C$OMP& coefkl, coefku, dene, denl, denn, dens, denu, denw,
C$OMP& im1, ip1, jm1, jp1, km1, kp1)
C$OMP& REDUCTION(max:resmax)
        DO K = 3 , KMM1
           DO J = 3 , JMM1
              DO I = 3 , IMM1
                 ....
```

Figure 4: OpenMP constructs to parallelise a large loop with many scalars that much be declared private. A reduction variable, resmax, must also be explicitly described.

As well declaration of local scalars as being thread private, it is also necessary to identify values such as resmax which is used as a reduction variable. In this loop it is used to determine a maximum value. A shell script can be used to extract most of the variables which may need to be declared, but it is necessary to inspect each value manually to determine its true nature.

Some compilers are more helpful than others in this process. For example, the SGI f90 compiler will provide warning messages for certain local scalars in parallel loops that it thinks may have been mistakenly left as shared when they ought to be thread private.

Note that the outer most loop over K has been made parallel in this case. This is the loop over the "third" mesh dimension. Using the outer loop ensures that there will be as much work as possible in each iteration. This is important since there is usually a fixed overhead in starting up a parallel section of code. The only drawback of this is that the parallelism will not work for 2D cases, where this dimension is only one cell deep. As 3D cases are much more expensive than 2D ones, this is a reasonable approach to follow.

One complication in the loop based parallelisation is within the routine `lisolv`. This is the single most computationally expensive routine taking a third of the run time in the serial test case. It is a simple iterative linear solver using line SOR. each iteration is intrinsically sequential as it is written, since each new point value is based on the most recently updated neighbouring values. The use of the most recent values is known to improve the convergence of the method.

To make the loops within `lisolv` parallel, a simple red-black reordering has been used in the K dimension. This is achieved by first iterating over the K values and then over the odds one in a second loop. Within each set then any two K iterations can safely be performed in parallel.

This reordering does have an adverse effect on convergence, but it appears to be quite small in practise. This is probably because in most cases of interest the coupling in the K dimension is relatively weak.

4.4 Comparison of block and loop parallelisation

The above two parallelisation methods were compared using a short 3D test case on a 4 processor shared memory SGI Origin. Full optimisation was used in all cases.

Figure 5 compares one measure of the convergence of the two implementations. The block method gives results that are identical with the serial case using 1 to 4 processors. The changes to the linear solver mean that the loop based method has slightly worse convergence than the serial case, though it is independent of the number of processors used.

Figure 5: A comparison of the residuals in the two different parallelisation cases. The block based method is identical with the serial case.

For the target architecture we are mainly interested in the speed up that can be obtained using the dual processor nodes of a Beowulf cluster. However, using the SGI machine allows OpenMP results with up to 4 processors to be compared. Table reftabspeedup compares the timings and speed up seen in the two cases. Figure 6 compares the speed up with the linear ideal.

Figure 6: Comparison of speed up on the SGI Origin using 1 to 4 processors.

In both cases the speed up seen is substantially less than linear with efficiency in the range 58 to 86%. The loop based parallelisation method is seen to give the best results. No allowance

Number of	Loop		Block	
processors	Time(s)	Speed up	Time(s)	Speed up
1	2500	1.000	2541	1.000
2	1450	1.724	1551	1.638
3	1066	2.344	1204	2.110
4	883	2.832	1089	2.334

Table 2: Comparison of speed up seen for a 3D test case on an SGI Origin. The loop based OpenMP directives are seen to be slightly more effective than the simpler block based implementation. This may be due to the poor load balance possible with only 6 mesh blocks.

has been made for the fact that the loop method gives a slightly worse convergence rate, but the effect is quite small in the test cases examined.

Factors that limit the parallel efficiency of OpenMP in these two cases include:

- Not all the code has been made parallel in either case. In particular all the data input and output performed on a single processor. It is possible to code parallel I/O using the C version of OpenMP, but this would require substantial work on data handling interface, and is not justified for the very small possible gains.
- For the loop based method, the more significant computational parts of the preprocessing steps have been made parallel, but this has not been done for the block method.
- The simpler method, based on parallelisation of the blocks, will be limited by the fact that it is not possible to achieve very good load balance due to the small number of mesh blocks and their range of sizes. A simple comparison of block sizes suggests that with 4 processors a best case partitioning would be limited by load imbalance to a speed up of around 3.6. Allowing for this would make the parallel efficiency of the two methods fairly close to each other.
- No significant attempt has been made to optimise either OpenMP implementation, due to the small scale nature of the project. Clearly this would be required to get acceptable performance beyond 4 processors.

Optimisation might include the memory layout, trying to ensure that data is on the processor that uses it most [3]. This is particularly important for ccNUMA machines, when using more than a small number of processors. The block based approach should particularly amenable to this by ensuring that same processor always operates on the same mesh data. This would be more difficult to achieve in the loop based method. In fact the partitioning of work between processors at the block level is the first step towards making an OpenMP SPMD (Single Program, Multiple Data) program.

An SPMD approach can be particularly efficient on OpenMP systems as the data is partitioned between processors at the start and all local variables are thread private. The only synchronisation points then required would be when boundary data exchanges occur and for global sums. The current block code does not make an explicit allocation of blocks to processors, instead allowing them to be allocated by a dynamic scheduling within each loop. This means blocks can migrate between processors at each loop, which is not good for non-uniform memory architectures.

For good efficiency in an SPMD approach it would be necessary to move mesh block boundaries to allow better load balancing. This is certainly possible within the simple mesh structure of the supplied 3D example cases, though this was not investigated. The preprocessing steps should also be included fully in the SPMD model otherwise this serial section will limit efficiency, and may be prevent data from been associated with a single processor.

Within the loop based code there are also a number of ways in which the OpenMP performance might be improved. One way is to reduce the number of barrier points that occur in the code, since these can be very expensive for large numbers of processors. A barrier occurs every time a parallel do loop is ended by `C$OMP END PARALLEL DO`. By adding the extra clause `NO WAIT` to the above statement, the barrier is avoided and each thread can proceed to the next parallel loop and start processing that. Clearly there are some points where a synchronisation will be needed, such as between assembly and solution phases, but these are far less than are used at present. In many cases it should be possible to add nowait clauses to do loops and provide explicit waits between important phases.

Various web sites provide more details of possible OpenMP optimisations. See for example [4].

5 Message Passing Parallelisation

5.1 MPI and other message passing libraries

The Message Passing Interface (MPI) has become the most widely used method for running tightly coupled applications on distributed memory systems. It has standardised interfaces defined for Fortran, C and, more recently, C++. Several public domain and commercial implementations now exist.

Other message passing libraries are available, the most notable being PVM (Parallel Virtual Machine), along with others such as BSP. Since the target architecture, Beowulf systems with high performance communication hardware such as Myrinet and Scali SCI, only provide MPI libraries, the use of PVM was not an option. To make the code easier to port to other message passing systems in the future a simple set of wrapper routines where written to encapsulate any calls to MPI functions. Adaption to another system such as PVM would then just require writing a new interface layer, rather than changes to numerous files. Only a very small set of MPI routines have been used in the initial message passing implementation of Genesis.

5.2 Decomposition for distributed memory systems

For distributed memory systems the most natural mapping of a mesh based calculation, such as that in the Genesis software, is to assign separate parts of the physical domain to each processor. As much computation as possible should be done locally since communication tends to be very expensive on such systems. In fact the SPMD approach discussed in relation to OpenMP parallelisation is the obvious choice for this case. However there are two problems when trying to map the existing mesh block structure onto a parallel system:

1. Load balance. The 3D example case only has 6 mesh blocks and the size of these varies greatly. While moderately good balance can be achieved up to 4 processors it is difficult to go beyond this without splitting up existing mesh blocks.
2. Communication costs. The way that mesh blocks are distributed to processors controls the size of the interfaces between neighbouring domains. This in turn determines how much data needs to be exchanged at the boundaries.

The load balancing problem will be lessened when looking at problems with more mesh blocks, for example when simulating multiple blade rows of the turbo machinery. It is also the case that this initial parallelisation is aimed at Beowulf clusters of small size, so a highly scalable solution is not required. The mesh generation process may also be altered so as to generate block sizes that allow better load balance. This may make mesh generation harder, which hinders the multi block approach, but may not be too great an inconvenience.

Controlling the communication through the size of the interface between domains is an additional constraint on the mapping problem. Much work has been done on finding good heuristic solutions to this problem for large unstructured meshes [5]. A simple example of the effect of different ways of partitioning the data is shown in Figure 7. With the constraint of having very few mesh blocks to distribute it was felt more important to partition for best load balance and ignore the communication costs in the first instance. It is also possible to specify the partition manually for such small problems.

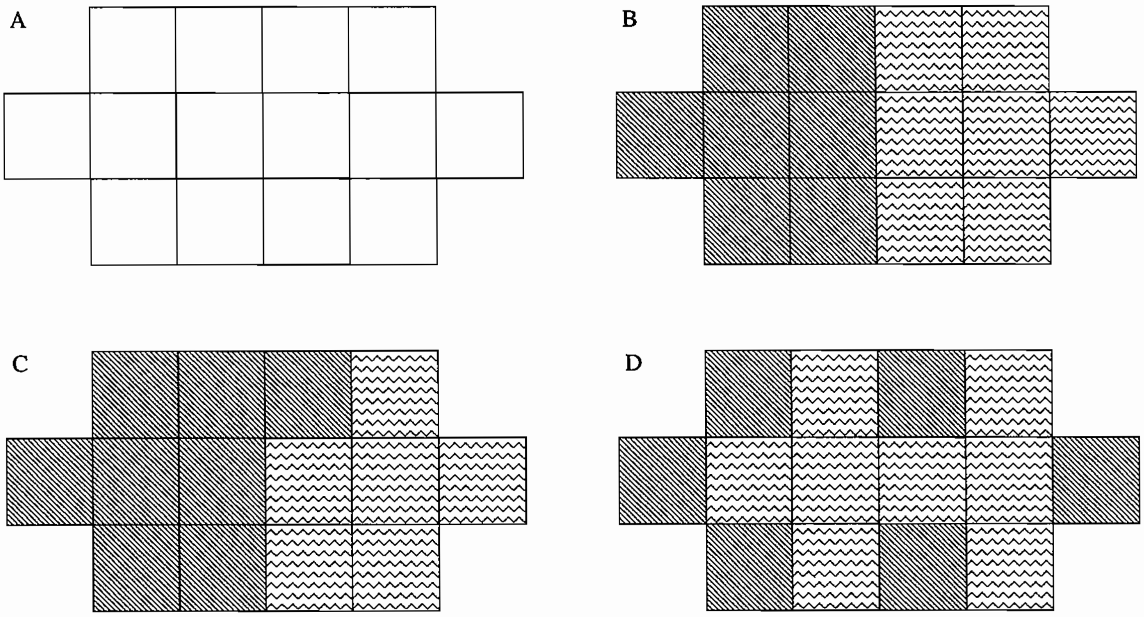


Figure 7: Examples of partitioning mesh blocks. (A) shows the topological connectivity of a 14 block mesh - the physical mesh is mapped to the blade surfaces. The number of cells within each block can be vary greatly. (B) shows a partitioning of the blocks between two processors to give a small interface, while (C) has a larger interface, but may have better load balance. (D) show a case where load balance may be optimised, but the interface is very large.

A simple partition code has been developed to automatically partition the mesh blocks. This is driven only by load balance requirements and in practise with such small numbers of blocks it is easier to specify the mapping manually. When simulating multiple blade rows it may be reasonable to assign one blade to each processor.

5.3 Message passing implementation

Within Genesis the coupling between solutions on different mesh blocks is dealt with by frequent exchange of block boundary values. This is inexpensive in the serial code, but may become more of a bottleneck in the parallel version.

The algorithm used can be approximately described as:

- Data input
    - Read run parameters
    - Read mesh block data
- Initialisation
    - Set initial constants, boundary conditions
    - Generate coarse mesh for initial solution
    - Calculate cell-wall distances
- Calculate initial pressure field
    - Loop over:
        - Blockwise assembly of terms associated with the variable
        - Blockwise iterative solution of the resultant linear system
        - Exchange block boundary data
    - Set initial values for other field variables
- Main solution loop - repeat until converged:
    - For each field variable:
        - Repeat until converged
            - Blockwise assembly of pressure terms
            - Blockwise solution of pressure terms
            - Exchange block boundary data
    - Write convergence data
    - If required, interpolate solution onto a finer mesh
- Write final solution data

This gives an outline of the solution method, though a great number of details have been omitted. The solution is first calculated on a coarse version of the grid and then interpolated onto successively finer grids until the final version is obtained.

With a given mapping of blocks onto processors the parallel version of the code has to address the following problems:

1. Loading and distribution of the mesh and run data.
2. Initialisation calculations.
3. Mapping between local and global numbering systems.
4. Exchange of data between block boundaries on different processors.
5. Summing residuals across all blocks.
6. Output to file of intermediate data and final solution.

Since the time taken to read and write data is very small fraction of the total run time, this is all handled by a single master processor. The master also performs some of the initialisation calculations since these are also a very small part of the overall time. Slave processors then wait for the master to send them the mesh blocks allocated to them, along with the initial values that have been determined.

The boundary data exchange is performed by packing the required values and sending them to the remote processor. The true values of residuals are obtained using the MPI function MPI_Allreduce. All I/O to files, including writing the final solution, is done by the master processor which collects all the partial contributions from the slaves.

5.4 Performance of the MPI version

A selection of performance figure are show in Table 3 for the run time and actual speed up of the MPI version of Genesis. The code has been run on a range of machines to test both portability as well as performance.

Machine	Speed up		
	NP=2	NP=3	NP=4
Beowulf (AMD 800MHz/Myrinet)	1.745	2.483	2.130
Beowulf (AMD 1.2GHz/Scali)	1.825	2.566	2.490
SGI Origin	1.855	2.691	3.486
DEC Alpha cluster	1.733	1.783	2.350

Table 3: Comparison of speed up seen for a 3D test case on several machines using the MPI implementation of Genesis.

The message passing version of Genesis was developed in a short period of time as a solution to extracting parallelism on small Beowulf clusters. Hence it has not been designed for high scalability, or indeed greatly optimised. In addition the particular 3D test case used is only capable of limited parallelism due to the obtainable load balance of the mesh blocks sizes. This is reflected in the relatively limited speed up figures seen on most machines.

The best results are seen on the SGI Origin with a speed up of nearly 3.5 on 4 processors. This is quite good considering the load balance limitations imposed by the underlining mesh block sizes. The other systems show more limited speed up with 4 processors, often less than that seen with 3 nodes. This is thought to be due to the fact that the parallel version follows the serial version in using very frequent exchanges of boundary information. This ensures identical convergence in the two cases but results in a large amount of communication. The high speed of message passing on the shared memory SGI Origin allows this to be done more efficiently than on the other machines. Optimisation of the parallel method, with fewer data exchanges should give better performance on distributed memory machines. The convergence should not be greatly degraded by such changes.

The speed up results seen for the SGI Origin are better for the MPI version than for the OpenMP versions of the code (see Table 2). The two likely causes of the are:

- The MPI version has fewer synchronisation points than the OpenMP loop based code.
- The data space in the MPI version is clearly partitioned between processors. This is not the case for either OpenMP version. This can be significant on machines with Non-Uniform Memory Architecture.

For the Beowulf cluster, we can only compare the two processor results using MPI and OpenMP. In this case the results are quite similar, with the MPI speed up only slightly better than the OpenMP one, 1.825 against 1.795. This may be due to the differences in memory architecture or compiler.

6 Combining OpenMP and Message Passing methods

6.1 Controlling OpenMP threads under MPI

Since the parallel implementation using MPI can only achieve limited parallel efficiency on its own, and the target Beowulf system consists of dual processor nodes, it seems worthwhile trying to combine the two approaches. This just required merging the MPI version with the loop based OpenMP code. The latter was chosen because it gives superior results and fits better with the limited number of mesh blocks available on each processor when the problem is partitioned for message passing parallelism.

The only change made was to add explicit calls to the OpenMP routine `omp_set_num_threads`. This sets the number of threads to use within the code. Normally this may be controlled via the environment variable `OMP_NUM_THREADS`, but this is difficult to set in a portable way for the MPI slaves. The program can also use this to control the number of threads used in the initialisation phase separately from those used in the main part of the run. This may be useful on machines such as the SGI where there are many processors available which can be used in either MPI or OpenMP modes. The initialisation calculation, which has not been distributed in the MPI implementation, can then make use of all available threads under OpenMP. When this is completed, the distributed MPI code can be used for the main solution process, as this gives better speed up results than OpenMP.

To illustrate this point consider running the combined Genesis code on an 8 processor shared memory machine. To utilise as many processors as possible 8 threads could be assigned to the initialisation computation. After this there are several options for the main solution process:

- A single MPI process with 8 threads
- 2 MPI processes with 4 threads each
- 4 MPI processes with 2 threads each

The limited number of mesh blocks in the test case rules out an 8 way MPI partitioning. Of the viable options, the previous speed up data suggests that the last option would be best, with 4 MPI processes each with 2 OpenMP threads.

6.2 Performance results for the OpenMP/MPI version of Genesis

The combined MPI/OpenMP code has been run on a dual processor Beowulf cluster and on a 4 processor SGI Origin. The standard 3D test example was run using various combinations of MPI processors and OpenMP threads.

For the SGI machine we have already seen that a pure MPI solution always gives better speed up than a pure OpenMP version using up to 4 processors. For example the MPI speed up with 4 processors is 3.49 while the loop based OpenMP gives only 2.83. Because the initialisation step only takes a small fraction of the total run time, the addition of OpenMP to this part does not alter overall speed up significantly for realistic run lengths. Mixing MPI and OpenMP is this cases gives intermediate results, for example with 2 MPI processes, each using two OpenMP threads a speed up of 3.26 is obtained.

For an 8 processor SGI machine, it would be necessary to combine OpenMP and MPI methods, since load balance prevents the latter been used beyond 4 processes in the test example. An overall speed up of ~6 might be expected with 2 threads per MPI process, though this may not be achieved because of memory bandwidth limitations.

For the Beowulf cluster with dual processor nodes it is reasonable to use two OpenMP threads on each node. The initialisation calculations on the master can also make use of two threads. Combining the two methods allows up to 8 processors to be used on the 3D test case. Some results are shown in Table 4 and compared with the SGI performance in Figure 8. For the SGI only pure MPI and Pure OpenMP data is shown, the combined code gives results intermediate between these two cases.

Total Processors	MPI Processes	OpenMP threads	Time (secs)	Speedup
1	1	1	5566.4	1.000
2	1	2	3100.6	1.795
4	2	2	1697.6	3.279
6	3	2	1217.2	4.573
8	4	2	1238.3	4.495
2	2	1	3119.0	1.785
4	4	1	2250.5	2.473

Table 4: Comparison of speed up seen for a 3D test case on a Beowulf cluster, using dual processor AMD 1.2GHz nodes and Scali interconnect.

On the Beowulf system, combining OpenMP and MPI gives reasonable speed up until 4 MPI processes (8 CPUs) are used, when performance falls in real terms as extra nodes are added. Again, this may be overcome by better load balancing in the mesh generation and in changes to the algorithm to reduce communication. For the cases where the total number of processors are the same, it can seen that using two OpenMP threads and half the number of MPI processes gives the best performance. This is in contrast to the SGI case, where single threaded MPI processes were to be preferred. This may be a function of the different memory architectures and compiler technology.

Figure 8: Speed up observed on the Beowulf cluster for pure MPI and the combined OpenMP-MPI implementations. Also shown for comparison are the SGI pure MPI and pure OpenMP results.

7 Conclusions

Parallel computing is becoming more common place with cheap Beowulf systems and dual processor PC machines available. Larger systems from manufactures such as IBM and SGI are also based on tightly interconnected clusters of SMPs. The MPI and OpenMP standards offer interfaces to exploit these resources in a portable fashion. While OpenMP often offers the simplest and fastest way to exploit many small SMP systems, it may take a lot more work to efficiently exploit larger shared memory systems. Such systems, like the SGI Origin, tend to be more expensive and to have a complex memory architecture. More careful OpenMP programming, possibly with the use on non-standard extensions to control memory placement, may be needed to scale beyond the order of ten processors.

MPI parallelism is often based on an explicit decomposition of the computational domain, and hence is good both at reducing the amount of communication required in a distributed memory machine and in maximising the data locality for the shared memory case. It is more complex to develop an MPI code, especially when trying to do so based on an existing serial code.

The simple minded MPI implementation used in this case is based on using the existing mesh blocks of a multiblock code. While this allowed the rapid development of a parallel version of the software it has significant limitations in terms of scalability due to the load imbalance and the large amount of communication required by the current algorithm. These problems can be overcome with more time and effort, but this is not always practical.

OpenMP provides an easy way to exploit shared memory systems, but these tend to be expensive when moving beyond small numbers of processors. It can also be the case that use of OpenMP without regard to memory movement between processors can give worse results than a MPI implementation. This is the case for the Genesis OpenMP code on the SGI system, where the efficiency with 4 processors was 87% with MPI, and only 71% with OpenMP. Again it is likely that optimisation of the OpenMP code could improve the performance, but at the cost of more time and effort.

The combination of OpenMP and MPI on the Beowulf system allowed the current test case to run with reasonable efficiency with 6 processors. When moving to the larger multiple blade row simulations the scalability of the code should increase further. As the target cluster will only have 16 processors in total, and be used as a shared resource, this level of parallelism is sufficient for the current needs of the design engineers.

In the parallelisation of a quantum Monte-Carlo code [1], it is shown that MPI and OpenMP both show very good scalability using up to 32 processors of an SGI Origin 2000. Increasing the number of processors up to 96, the MPI implementation performs better. The combination of OpenMP and MPI performs better than pure OpenMP, but not better than the pure MPI version. Hence there is not a clear case for the combined approach for this code on such a machine.

For parallel Genesis running on a merely 4 processor SGI we see a similar result, in that the present MPI implementation is always better the OpenMP version. Because of the limited partitioning possibilities of the test example, it would only be profitable to use more than 4 SGI processors in the combined code. While more development work would enable the MPI code to scale better, the combined code provides a easier route to exploit parallelism up to 8 or so processors.

Another example where combined MPI and OpenMP is superior to either on its own is that of a photon transport code [6]. In this case the code is highly parallel but is run on an IBM SP system with clusters of 8 way shared memory nodes. This means that the OpenMP code cannot scale beyond 8 processors. With 128 processors the speed up of the highly parallel core of the code is 113 (88% efficient) using only MPI but 121 (95% efficient) using 16 MPI processes each with 8 OpenMP threads.

Thus we agree with the conclusions of [1] that the combination of the two methods can be very useful in some situations. In the test case discussed in this paper, an existing CFD application has been adapted for small scale parallelism. Both MPI and OpenMP offer ways in which limited parallelism can be introduced with relatively small amounts of work. To make either method highly scalable would require a more substantial reorganisation of the existing code. By combining both methods together it has been possible gain reasonable performance benefits on the small Beowulf clusters and SGI machines that are in use in the typical design office.

References

[1] L.A. Smith, *Mixed Mode MPI /OpenMP Programming* UKHEC Technology Watch Report, Edinburgh Parallel Computing Centre, `http://www.ukhec.ac.uk/publications/tw/mixed.pdf`.

[2] Y.C.Hu,G.Gin, S.L.Johnsson, D.Kehagias and N.Shalaby, *HPFBench: A High Performance Fortran Benchmark Suite* ACM Transactions on Mathematical Software, Vol. 26, No. 1, pages 99-149, March 2000.

[3] B.Chapman, O.Hernandez, A.Patil and A.Prabhakar, *Program Development Environment for OpenMP Programs on ccNUMA Architectures*, Proc. Large Scale Scientific Computations 2001, Sozopol, 2001.

[4] See for example: `http://www.cs.utk.edu/ london/MPPopt/ptools99-opt/sld106.html` and `dynamo.ecn.purdue.edu/ eigenman/EE563/Handouts/BWsc00introOMP.pdf`

[5] G.Karypis and V.Kumar, *Multilevel k-way Partitioning Scheme for Irregular Graphs*, SIAM Review, 41, No. 2, p278 (1999).

[6] A.Majumdar, *Parallel performance study of Monte Carlo photon transport code on shared, distributed, and distributed-shared-memory architectures* Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc, Los Alamitos, p.842 (2000).