A Comparison of MPI and OpenMP Implementations of a Finite Element Analysis Code

Michael Bane
Centre for Novel Computing
Department of Computer Science
University of Manchester
Oxford Road, Manchester M13 9PL
United Kingdom
m.bane@cs.man.ac.uk

Rainer Keller
C/O Michael Resch
Allmandring 30
70550
Stuttgart
Germany
Rainer.keller@studmail.uni-stuttgart.de

Michael Pettipher
Manchester Computing
University of Manchester
Oxford Road, Manchester M13 9PL
United Kingdom
m.pettipher@man.ac.uk

Ian Smith
School of Engineering
University of Manchester
Oxford Road, Manchester M13 9PL
United Kingdom
ian.smith@man.ac.uk

Abstract. In this paper we describe the steps involved, the effort required and the performance achieved in both MPI and OpenMP implementations of a Fortran 90 finite element analysis code on an SGI Origin2000 using the MIPSpro compiler. We demonstrate that a working OpenMP version is easier to write, and then explain how to overcome some restrictions of the first version of the API to obtain better performance.

1 Introduction

Finite Element Analysis is used in a wide variety of applications, including the design of aircraft and cars; civil engineering construction projects; the study of geological processes such as fault prediction and the structural development of bones in the human body. The requirement to solve such problems with increasing detail and in shorter timescales – either to reduce the overall project development time, or for example to permit a consultant to provide almost immediate feedback to a patient, has encouraged the development of parallel implementations, particularly based on iterative algorithms.

In an earlier paper Pettipher and Smith [5], it was shown that a parallel implementation using an element level approach could achieve very good results using MPI [3] on a Cray T3D. For this paper, the same code has been run on an SGI Origin2000, producing good performance, albeit on a relatively small number of processors. The objective here is to investigate how well the same code can be implemented using OpenMP [4].

2 The Codes

The codes, based on the preconditioned conjugate gradient algorithm for the solution of the linear system Ax=b, rely heavily on matrix and vector operations and consequently map very well to Fortran 90, making extensive use of array syntax and array intrinsics, particularly MATMUL and DOT_PRODUCT.

The basic form of the iterative section of the conjugate gradient algorithm is:

```
iterations: DO
        iters = iters + 1
        u = MATMUL(A,p)
        up = DOT_PRODUCT(r,r)
        alpha = up/DOT_PRODUCT(p,u)
        xnew = x + p*alpha
        r = r - u*alpha
        beta = DOT_PRODUCT(r,d)/up
        p = r + p*beta
        x = xnew            ! after testing for convergence
END DO iterations
```

where A is the coefficient matrix, *p, u, r, x, xnew* are all vectors, and *alpha* and *beta* are scalars. Further details are given in various texts, but see in particular Freeman and Phillips [2], and Smith and Griffiths[7].

Using the element-based approach gives rise to repeated (one for each of the finite elements) dense matrix-vector operations with vectors of length 60 (20 node brick elements, each with 3 degrees of freedom per node), and to operations with vectors whose length matches the number of equations. (The test case used for this paper had 8000 elements and 100840 equations). However, this requires each of the length 60 vectors to be generated by a gather from the *p* vector, and the result of the matrix vector calculation to be scattered to vector *u*, before generating the new solution vector. In constant stiffness cases, it can be arranged that the dense matrix is the same for all vectors, so the matrix-vector operations can be replaced by a single matrix-matrix computation, which permits the use of level 3 BLAS routines giving substantial performance improvements.

With the diagonal preconditioner included, and with rearrangement to reduce the three dot products to two which can then be performed concurrently the code looks like this:

```
iterations: DO
        iters = iters + 1
        pmul = p(g)
        utemp = MATMUL(km,pmul)
        u(g) = u(g) + utemp
        udiag = u*diag
        dotpu = DOT_PRODUCT(p,u)
        dotuu = DOT_PRODUCT(u,udiag)
        alpha = dotrdold/dotpu
        xnew = x + p*alpha
        r = r - u*alpha
        d = diag*r
        beta = alpha*dotuu/dotpu - 1
        dotrdold = beta*dotrdold
        p = d + p*beta
        x = xnew            ! after testing for convergence
END DO iterations
```

The key difference is the replacement of the statement:

```
u = MATMUL(A,p)
```

where A is a large sparse matrix, with

```
pmul = p(g)
utemp = MATMUL(km,pmul)
u(g) = u(g) + utemp
```

where km is a small dense matrix such that the sum of all km is equivalent to A, and pmul and utemp are matrices containing details for all elements. (It is possible to put these 3 statements within a loop over the elements, with pmul and utemp then being vectors, but the matrix version is used not only to permit matrix-matrix computation when km is the same for all elements, but also to reduce the amount of communication in the MPI implementation – see the MPI section.)

3 Reproducible Timings

The SGI Origin2000 system on which the work was performed is a heavily used production system, is designed to maximise use of resources by sharing processors and memory wherever possible. This means in particular that an individual process might:

- share processors with processes from other users on a timesharing basis;
- share memory on a node with other processes;

and

- code might migrate to other processors;
- data might migrate to other processors.

A consequence of this is that it is difficult to ensure that an individual job uses the same resources in the same way in different runs, and consequently timings between different runs can vary considerably. Exclusive access or possibly partitioning with cpu sets would be ideal, but both require system administrator control, and neither are practical options in this case.

A mechanism for 'nailing' jobs to reserve processors was devised by Graham Riley and Mark Bull while in the Centre for Novel Computing at the University of Manchester (Mark Bull has now moved to EPCC). This guaranteed exclusive access to processors, *but not to memory*, and thus alleviated some but not all of the problems. Subsequently SGI now provide the miser utility which provides similar functionality (and other features). As this is supported by SGI and does not require special system privilege.,it seems the best solution for timing codes on the Origin2000. Even so, there can be variation between runs – typically of up to 10% (but nearly 40% has been observed. The variation is largest on a heavily loaded (in terms of memory and I/O as well as CPU usage) system, particularly for the short runs carried out here.. However, in light of the results obtained so far, such variation does not alter the general interpretation with respect to a comparison between the MPI and the OpenMP results. No attempt has been to average results – figures quoted are typically from a single run, although reasonable confidence is obtained from similar run.. For the testing reported on here, miser is currently restricted to 16 processors, so no figures are given for more than 16 processors.

4 The MPI Implementation

As described in Pettipher and Smith [5] a simple strategy is adopted for the data distribution. Consider again the three statements:

pmul = p(g)
utemp = MATMUL(km,pmul)
u(g) = u(g) + utemp

km is small (60x60), so is replicated on all processors;
p and u (and the other vectors) are distributed with one dimensional blocking;
pmul and utemp (60x8000) are distributed in blocked column format

Thus the MATMUL involves no communication itself, but there may be substantial communication for the gather and scatter. However, as the data for all elements (the columns of matrices pmul and utemp) is stored, there is only one gather and one scatter for each PCG iteration – special gather and scatter routines were written to handle this. The disadvantage of using matrices for pmul and utemp (and elsewhere within the whole code), instead of temporary vectors, reused for each element, is the additional memory required, but this has not yet been a problem.

On the SGI Origin2000, the following results were obtained for a problem with 8000 elements and 100840 equations, and the number of PCG iterations limited to 30 to reduce the total time. In a more realistic run, the iteration count would be much higher resulting in the PCG time being a higher proportion of the total time.

Processors	1	16
Total time	17.8	1.7
PCG time	14.6	1.0
PCG speedup	1.0	14.6
PCG performance - %peak	31%	27%

The PCG performance figure is based on an estimate of the number of floating point operations performed in this section and the peak performance of 16 processors..

These results seem very encouraging, and are as expected in light of earlier work on distributed memory systems [5].

5 OpenMP Implementation

One of the attractions of the shared memory parallelisation approach using OpenMP is the opportunity to perform the work incrementally, without the major rewriting associated with message passing codes. It is also expected that the final code will be much closer to the original code and therefore easier to understand, maintain and adapt for different problems. Thus the development of the OpenMP implementation started with the original Fortran 90 code. However, knowledge gained during the message passing implementation is relevant for any other parallel implementation – the key areas to be parallelised are still the same and issues such as the use of BLAS routines instead of Fortran 90 intrinsics should be investigated. What was not so clear at the start of this work is the extent to which the data distribution should be managed by the programmer - the Origin2000 is a distributed memory machine and remote data access is more expensive than local. How well does the compiler cope with this and how much must the programmer control?

Autoparallelisation

The MIPSpro compiler (version 7.3.1.1m was used) provides autoparallelisation by the use of the -apo compiler flag. This attempts to parallelise DO loops within the source code by use of the OpenMP directives. Currently, OpenMP version 1 is implemented by the compiler.

It will also consider unrolling Fortran 90 array syntax and then parallelising the resulting code (for example for the initialisation of array A=0.0). However, the -apo flag will not convert Fortran90 intrinsics into parallelisable DO loops - thus MATMUL and DOT_PRODUCT are not parallelised.

(It is noted in passing that the use of these intrinsics in a OpenMP parallel region means replication of work in that each thread will perform the complete operation. There is on-going discussions about whether this will change with OpenMP-2)

The -apo flag will also honour the user's OpenMP directives. Alternatively, the user may use the -mp compiler flag to have only their own directives used.

(It is also noted that just using –apo gave no measurable performance increase, whether using array intrinsics or BLAS routine.)

Using BLAS Routines

As noted above, the F90 intrinsic MATMUL is not auto-parallelised. The following options were therefore considered:

- replacing the intrinsic with explicit OpenMP DO loops
- the use of the Basic Linear Algebra Subprograms (BLAS), particularly the level-3 BLAS dgemm.

To use the BLAS, the user has to link with the relevant library. The obvious choice is of -lblas and -lblas_mp but alternatives could be -lscs_blas, -lscs and -lnag for the former or -lscs_blas and -lscs_mp for the latter.

The -lblas links the object codes with a serial implementation whereas the -lblas_mp links to a parallel version of the required BLAS routine. It is not stated in the SGI documentation how the parallelisation of the latter is controlled. It is noted that under miser, -lblas_mp will use (up to) the number of processors given in the miser command line.

Note also that the use of -lblas_mp for calls to BLAS which are already in a parallel region leads to a significant *decrease* in performance, as compared to the use of the serial implementation of the BLAS in that parallel region. It is believed that this is due to each thread in the parallel region spawning additional threads which are then timesliced resulting in an excessive number of threads and decreased performance.

Code Implementation

As indicated in previous sections, the major computational part of the code is the matrix multiplication part of the PCG iterations. Consider the following DO loop format:

```
DO iel = 1, nels
        pmul(:,iel) = p(g(:,iel))                    ! Gather
END DO
        utemp = MATMUL(km,pmul)
DO iel = 1, nels
        u(g(:,iel)) = u(g(:,iel)) + utemp(:,iel)    ! Scatter
END DO
```

The question is how best to parallelise with OpenMP?

As MATMUL will not parallelise, this should be replaced, either using the parallel BLAS routine, dgemm, or an explicit DO loop with OpenMP directives. Both were tried:

	dgemm_mp		Explicit	
Procs	Time	Speedup	Time	Speedup
1	6.3	1.0	9.9	1.0
2	3.3	1.9	4.7	2.1
4	1.9	3.4	2.5	4.0
8	1.0	6.1	1.2	8.0
16	0.6	10.2	0.6	15.3

Thus the time on 16 processors is the same whichever is used, but dgemm is better cache optimised on a single processor (no attempt was made to do any cache optimisation in the explicit version). It appears therefore that the dgemm routine is preferable, but there are implications which should be born in mind.

First, linking with the parallel BLAS library as used here, might limit the use of serial BLAS routines as mentioned above.

Secondly, the performance of any code on the Origin2000 depends on where data resides. (This is true of any NUMA architecture.) Although it is tempting to incrementally add OpenMP statements to parallelise time consuming loops, this may not be the optimal solution. It is important to consider how the operating system determines the placement of data. On the Origin2000, the default is a "first touch" policy - the thread that first touches a data item (that is not already placed) will force the operating system to place the page containing that data item on the memory associated with that thread (which may or may not be the physically closer memory). Furthermore, by default, data will not migrate between memories as a result of the program execution. (However, it may change due to the operating system paging out the program completely and then paging it back in but on different threads and memories.)

Given that the cost of accessing data from distant memory is significantly higher than accessing data from a processor's local L2 cache, the initial data placement can have a significant effect on program performance. For a code that has parallel regions it is therefore important to ensure that the data is distributed satisfactorily so that each thread is accessing data as locally as possible. This is achieved by parallelising the initialisation part of a code as well as the time consuming parts.

For example, in this code, the main time was spent in the matrix multiplication of km by pmul to form utemp. Merely converting the MATMUL to three DO loops and parallelising them will not lead to maximum performance. As the first time the variables km, pmul and utemp are touched is another matrix multiplication, this can be parallelised in a similar fashion to ensure maximum performance for the key matrix multiplication section. Given it is not known how dgemm will force the data to be distributed, if an explicit OpenMP implemementation is used for the main matrix multiplication section it should also be used it in a similar way for the initialisation section too.

This point indicates the importance of the data management which is potentially under the control of the programmer. (It is possible to take this further still by using SGI directives to place data on particular memory and code on particular threads, but this option was not considered due to issues of portability.)

The gather loop above can be easily parallelised with OpenMP directives:

```
!$OMP PARALLEL DO default(none) &
!$OMP PRIVATE (iel) SHARED (nels, pmul, g, p)
        DO iel = 1, nels
                pmul(:,iel) = p(g(:,iel))
        END DO
!$OMP END PARALLEL
```

The scatter, however:

```
DO iel = 1, nels
        u(g(:,iel)) = u(g(:,iel)) + utemp(:,iel)    ! Scatter
END DO
```

cannot easily be parallelised because of the potential (and actual) dependency in assigning elements of vector u.

The answer to this should be to use the !$OMP ATOMIC directive, which allows the update to be performed in a controlled and parallel manner. However no performance improvement has been obtained using this directive, in comparison with a serial implementation of this loop. It is thought that the synchronisation costs negate any increase in performance due to parallelisation.

In order to circumvent this situation PRIVATE temporary arrays can be used for u on each thread which are then brought together in a REDUCTION-type operation. (Note that OpenMP version 1 allows only scalars in the REDUCTION clause). This gives (noting that it is already in a PARALLEL region):

```
my_id = omp_get_thread_num()
!$OMP DO
     do j = 1, neq
       do i = 0, omp_get_num_threads()-1
            dist_u(i, j) = u(j)
          end do
     end do
!$OMP END DO

!$OMP DO
     do iel = 1 , nels
          do i = 1 , ndof
             dist_u(my_id,g_g(i,iel)) = dist_u(my_id,g_g(i,iel)) +g_utemp(i,iel)
          end do
     end do
!$OMP END DO

!$OMP DO
     do j = 1, neq
       do i = 0, omp_get_num_threads()-1
            u(j) = u(j) + dist_u(i, j)
          end do
     end do
!$OMP END DO
```

Although this code removes the synchronisation cost of each update to the elements of u, it does so at the cost of additional memory requirements (for dist_u) and copying arrays. (There are also 3 barriers, one at the end of each OMP DO construct.)

Once all the time consuming loops (gather, multiply, scatter) have been put into OpenMP constructs, it is necessary to consider simple transformations of the code in order to minimise synchronisation costs. Examples include, creating the largest possible PARALLEL region, using as many NOWAIT clauses with DO loops as possible, and moving all sequential parts of the code into one SINGLE clause (or replicating work, where possible). Thus the final OpenMP code does not map simply onto the original code with just additions of OMP directives.

Whereas the compiler is good at basic code manipulation for optimisation on a single node (eg prefetching) there is no support for rearranging OpenMP constructs. Furthermore, there are often several alternative methods of using OMP directives to obtain the same result. The choice of ATOMIC, CRITICAL and SINGLE for the updating of u, for example.

Other parts of the code

In general, it is relatively easy to parallelise the rest of the code, but the use of Fortran 90 array syntax and intrinsics does seem to require particular care. For example, the initialisation statements:

```
diag_precon = 0.0
oldis = 0.0
tensor = 0.0
loads = 0.0
```

where tensor is a 3D array and all the rest are vectors. All of these statements were parallelised, but as separate loops (no loop fusion). If a DO loop is used instead, a single parallel loop is generated, reducing parallelisation overheads.

Results for OpenMP implementation:

As indicated above, the scatter operation is crucial to the good performance of this code, and as yet this has not been fully dealt with. The following results were obtained using the PRIVATE temporary arrays as mentioned above:

Processors	1	16
Total time	17.4	5.2
PCG time	15.2	1.9
PCG speedup	1.0	8.0
PCG performance - %peak	30%	15%

Thus the single processor times roughly match those for the MPI version, but for 16 processors, the performance in the PCG section is roughly half as good.

6 Comparison Between MPI and OpenMP Implementations

The key issue in the comparison of the results is that on 16 processors, the OpenMP version is about twice as slow as the MPI version in the PCG section. (The OpenMP version is also slower for the other parts of the code, but this will be less important for more realistic runs when most of the time will be spent performing the PCG.)

However it should be noted that many months were spent on developing and improving the MPI version of the code, while substantially less has been spent on the OpenMP version. If the OpenMP version had been found to perform as well as the MPI version, then the reduced development time would be a clear benefit, but just because it does not, it is unfair at this stage to say that OpenMP cannot perform as well. In fact the authors are optimistic that further improvement can be obtained after more detailed analysis. For example Bull [1] noted that the !$OMP REDUCTION clause on an Origin2000 is expensive. The current implementation could possibly benefit from rewriting in a way which reduces the number of reductions.

With the above comments in mind, it is still possible to make some meaningful comparisons for this particular code.

For MPI

- Known to give good performance for large numbers of processors and for larger problems and on a variety of distributed memory systems.
- Portable across most (if not all) parallel systems.
- Easier to understand how the data is distributed and where the time is spent.
- Many tools available to assist the programmer.

Against MPI

- Coding complex to read, write and maintain

For OpenMP

- OpenMP code is initially much easier to write, modify and maintain
- Once one code is satisfactory, it should be easy to parallelise other codes in a similar way

Against OpenMP

- Too easy to overlook necessary (or un-necessary) synchronisations
- Lack of tools inhibits quick program development
- Portability limited to subset of parallel systems with (virtual) shared memory.

7 Conclusions

The main conclusion is that expecting to be able to produce a good, parallel implementation using OpenMP, *with very little effort*, is, in general, unrealistic. The programmer must be aware of the data management issues, and know when and how to control these explicitly.

The other concern with this code is that the use of Fortran 90, particularly the array intrinsics, which simplifies the coding considerably in comparison with the Fortran 77 equivalent, seems to hinder implementation with OpenMP. However as noted earlier, the next release of OpenMP will include various enhancements, including (it is believed) the parallelisation of the Fortran 90 intrinsics.

It is recognised that there is more work to do in this particular implementation, but the authors are optimistic that good performance is achievable with not too much more effort.

8 References

1. Bull, M..J. 1999, Measuring Synchronisation and Scheduling Overheads in OpenMP, First European Workshop on OpenMP, http://www.it.lth.se/ewomp99/
2. Freeman, T.L. and Phillips, C. 1992, Parallel Numerical Algorithms, Prentice Hall
3. MPI: A Message Passing Interface Standard, Message Passing Interface Forum, 1995, University of Tennessee
4. OpenMP Specification 1999, from http://www.openmp.org/
5. Pettipher, M.A. and Smith, I.M., 1997, The Development of an MPP Implementation of a Suite of Finite Element Codes, in proceedings of High Performance Computing and Networking 1997, Eds: Hertzberger, B and Sloot, P
6. Smith, I.M. 2000, General Purpose Parallel Finite Element Programming, Engineering Computations, 17, 1, 75-91
7. Smith, I.M., and Griffiths, D.V. 1997 Programming the Finite Element Method 3rd (Fortran 90) edition, Wiley

Appendix - Systems and Software used

All of the results for this paper were produced on a 40 processor (195 MHz, R10000 processors, each with 4Mb L2 cache) SGI Origin2000 system at the University of Manchester. The timings were obtained under miser, using MIPSpro 7.3.1.1m and Irix 6.5.6m. The common compiler flags used for both MPI and OpenMP versions were:
f90 –64 –mips4 –O3 –r10000 –r8.