Finite Element Analysis on the Cray MTA-2

Jon Gibson and **Mike Pettipher**, *CSAR, University of Manchester*

ABSTRACT: *This paper explains how the Cray MTA-2 might be a good platform on which to run an existing suite of finite element codes. We discuss the issues involved and compare the performance on the MTA with that on an SGI Origin 3800.*

1. Introduction

The Cray MTA-2

The Cray MTA-2 (Multi-Threaded Architecture) uses multiple lightweight threads on each processor as a novel way to bypass the increasing problem of memory latency on high performance architectures. It can have up to 128 of these threads on each of 16 to 256 processors (noting that the largest machine currently in existence is the 40 processor one at NRL). By switching between the active threads at each clock cycle, the processor is kept busy and the time taken to access memory is not wasted. This avoids the need for caches and the multi-level memory hierarchy normally associated with high performance architectures. In fact, the machine provides a scalable uniform access to a global shared memory, at a bandwidth of 2.4GB/s and with an impressive 4GB of memory per processor. Another desirable feature of the machine is that it is “easy to program”. The parallelism is mainly loop-based and is programmed using directives (in a similar way to OpenMP) or, where possible, implemented automatically by the compiler. Additionally, the MTA's uniform memory access simplifies the task of the programmer, since issues such as data locality and optimal cache usage are not relevant.

The Finite Element Codes

Over a number of years, Professor Ian Smith of the University of Manchester has developed a suite of template finite element codes. They are serial Fortran 90 codes written so as to be easily adaptable by a user to suit their individual requirements. They use an element by element (EBE) method, employing iterative solvers. The codes are widely used by engineers worldwide and an earlier version of the codes was used as the basis of the NAG finite element library. They cover all the usual fields where FEA is employed, e.g. material behaviour (elastic-plastic); fluid and heat flow; eigenvalue problems; forced vibrations; and coupled physical processes such as magnetohydrodynamics.

More recently, a lot of work has been done to provide an equivalent set of templates which can be used for parallel FEA, using MPI. The idea is to maintain as much of the structure and syntax as possible from the serial code and to hide away all the parallel MPI directives in library routines. This is to allow users with minimal knowledge of parallel programming to adapt the parallel code templates, in the same way as the serial ones, to their own particular problems.

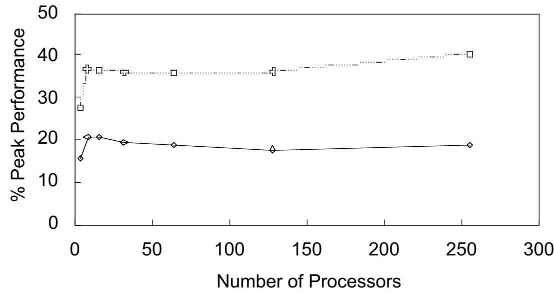
The element by element approach used in all of these codes leads to a lot of inherent parallelism. All of the codes are completely dominated by a number of loops over the elements, all of which can be parallelized. For the purposes of this paper, we will consider one routine in particular, p60pcs, which calculates the 3D strain of an elastic-plastic solid. The time in this program is dominated by the pre-conditioned conjugate gradient (PCG) solver, which is itself based around a matrix-vector computation. This consists of a matrix-matrix multiplication, preceded by a gather and followed by a scatter, where data is distributed as required. These gather and scatter sections require a significant amount of communication. We wrote our own routines which packed and sent the data once for all gathers and once for all scatters, giving a small number of large messages. It took a substantial amount of time to develop this code. Table 1 shows the timings we obtained for this MPI implementation on a Cray T3E.

Processors	8	512
PCG elapsed time	104.9	3.1
PCG speedup	1.0	33.8 (out of 64)
PCG % peak	22%	12%

Table 1: Timings of Elastic-Plastic Analysis Code on the Cray T3E.

As a second example, we will quickly consider a magnetohydrodynamics code. This has been discretised using the finite element (EBE) method into 20-node quadrilateral bricks. It contains a Navier-Stokes solver and

reaches an iterative solution using a BiCGStab(l) algorithm. This is necessary to deal with the unsymmetrical stiffness matrix. The stiffness matrix and is different for each element, resulting in poor cache re-use and hence, poor performance. By finding the common structure in the stiffness matrices and recoding, it is possible to improve the use of cache and so the overall performance. The figure below shows the percentage of peak performance obtained with 256 processors on an SGI Origin 3800.



There are therefore two major factors affecting the performance of the MPI implementations of these codes. Firstly, the time for the gather and scatter is significant, although this does not affect the scaling. Secondly, special coding is required to avoid performance problems due to cache re-use in matrix-vector operations. In other words, good performance has been achieved after a significant amount of work was spent on various parts of the code. The question we were asking was could the Cray MTA-2 do any better?

2. Finite Element Codes on the MTA-2

There were a number of reasons why we expected the MTA-2 to show an improved performance over other HPC machines. Its flat shared memory should remove the cost of a distributed memory gather and significantly reduce that of the scatter. Since memory latency is hidden by the use of multiple threads on the MTA, efficiency of cache usage is not an issue (in fact, it doesn't even have one!). Perhaps the main advantage of the MTA here is the ease of programming. Very few changes are required to the serial codes, only a few OpenMP style directives, and a fair amount of the code can be automatically parallelized by the compiler. This simplicity is ideal for our situation, where there's a whole suite of codes to be edited for individual purposes by a large number (for the serial codes anyway) of engineers/scientists with very little parallel coding knowledge.

We will compare the timings of the aforementioned p60pcs elastic-plastic code (running 64,000 elements) on the MTA-2 with those of MPI and OpenMP codes on an SGI Origin 3800 (located here at CSAR, Manchester). Our results are only for one, two and four processor runs due to the limited sizes of the MTA's available to us. Table 2 shows the results with the Origin and Table 3, those of the MTA. All timings are in seconds.

Processors	1	2	4
Conj Grad loop	787.1	393.5	204.7
Matrix mult	378.6	192.2	98.0
Gather/scatter	254.2	132.2	72.0

Table 2: p60pcs, MPI version on the Origin 3800

Processors	1	2	4
Conj Grad loop	841.3	429.8	234.1
Matrix mult	664.5	334.7	167.7
Gather/scatter	76.7	43.7	33.8

Table 3: p60pcs on the MTA-2

It is important to note that the clock speed on the Origin is 400MHz, double that of the MTA. This explains its better performance on the matrix multiply. However, despite this, the MTA gives a significant performance improvement with the gather and scatter, leading to comparable times for the overall conjugate gradient loop. Another interesting comparison to make is the coding time required, given that on the MTA it took hours rather than months for the MPI version. This should not be the case for an OpenMP version though, where the coding style is very similar. Table 4 gives the timings for the OpenMP version of the same code on the Origin 3800.

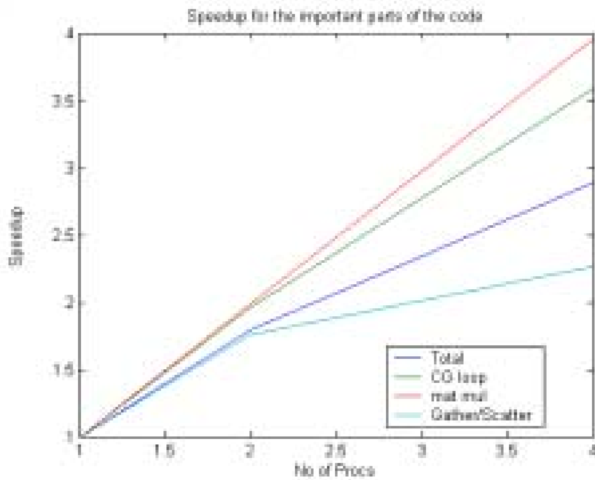
Processors	1	2	4
Conj Grad loop	678.4	473.2	359.0
Matrix mult	381.6	200.2	103.4
Gather/scatter	142.2	110.5	100.4

Table 4: p60pcs, OpenMP version on the Origin 3800

The gather and scatter performs badly enough for the MTA to be running the faster of the two by the time the jobs reach four processors in size.

The scaling of the code on the MTA is clearly not perfect, as illustrated in the figure below. It is the gather/scatter part that is scaling so poorly. Looking in a bit more detail, the gather can be seen as insignificant, taking about 0.02 seconds in all cases. A little investigation showed that the scatter involves unnecessary multiple updates to specific locations and the resulting memory contention leads

to the poor scaling. This can be fixed with a simple test, although at the expense of an additional overhead. A better solution without the overhead is being developed. The timings of this “improved” version are shown in Table 5.



Processors	1	2	4
Conj Grad loop	881.2	442.7	227.9
Matrix mult	677.5	338.7	171.9
Gather/scatter	102.4	51.4	26.2

Table 5: p60pcs on the MTA-2 with scaling scatter

Clearly, the gather/scatter is now scaling well.

Summary

We have seen that the performance of these codes on the MTA is comparable to MPI and better than OpenMP on a machine with double the clock speed. This can be done with much less coding time, especially when compared to the MPI code. Since the code used is part of a suite of similar FEA codes, widely used throughout the world, their potential applicability is enormous. One of our objectives is to encourage engineers who are not currently using HPC systems to do so. The MTA offers a simple route to HPC for engineers using this suite of FEA codes.

We intend to compare performances for a number of these codes on the MTA and to report more fully on our findings at a later date.

Acknowledgements

The author would like to thank Simon Kahan and Jim Maltby of Cray Inc, Seattle for their valuable help on the project and to Cray Inc itself for giving us access to their MTA’s.

About the Authors

Jon Gibson is a high performance computing consultant at CSAR, Manchester Computing, University of Manchester, U.K. E-mail: jon.gibson@man.ac.uk

Mike Pettipher is the manager of the HPC services group at Manchester Computing, University of Manchester, U.K. E-mail: m.pettipher@man.ac.uk