Parallel Programming the Finite Element Method

Atique Ahmad

Department of Mechanical Engineering, PIEAS, Islamabad.

ABSTRACT Nowadays engineers and researchers are faced with solving very large scale problems in finite element analysis. Solution of such problems necessitates the use of parallel computers. The obvious option is to buy a cluster or supercomputer with specialized software (such as ANSYS Distributed Solver). This option is of course very expensive. The other low cost option is to build a Beowulf class PC cluster and then code the required programs using such free and open-source tools as MPI and OpenMP. This paper discusses the second approach of low cost supercomputing. Fortran has been used to program two-dimensional finite element analysis and the same code is then modified for use in a parallel computing environment. The performance of the serial and parallelized code is then studied on computers of varying specifications.

INTRODUCTION

The finite element method has become a powerful tool for the numerical simulation of a wide range of engineering problems. Applications range from deformation and stress analysis of automotive, aircraft, building and bridge structures to field analysis of heat flux, fluid flow, magnetic flux to such advanced applications such as micro-electromechanical systems and nanotechnology.

With the recent advances in computer technology and CAD systems, it has now become much easier to model large complex problems. Thus several computer models can be simulated before developing a prototype. This not only improves the quality of the engineering product but also reduces the product development time.

All this suggests that it is necessary to keep pace with any new research and development by understanding the basic theory, modeling techniques and computational aspects of the finite element method.

The finite element method involves discretization of the continuum into a mesh consisting simple geometric shapes. It is vitally important that the mesh be appropriate enough to ensure solution accuracy. However, in order to improve the quality of mesh, it may be necessary to make elements smaller. This in turns increases the total number of elements in the problem, thus increasing the requirements of computer memory and speed. Although, computer technology has
seen great advances in past few decades, so has increased the size, scale and complexity of the problems faced by scientists and engineers. This necessitates the use of parallel computing. Parallel computing involves programming of multiple connected computers (Beowulf cluster) or computers with multiple processors (dual-core, quad-core, supercomputers, etc). However, just running a sequential code on a parallel machine is not the solution. It is necessary to modify the codes in order to utilize the full capacity of parallelization. This modification may be done using such tools such as MPI (Message Passing Interface) and OpenMP (Open specifications of Multi Processing).

This work mainly discusses OpenMP parallelization of finite element code with its advantages and shortcomings.

**Beowulf class PC cluster**

Beowulf is a design for high-performance parallel computing clusters on inexpensive personal computer hardware. Originally developed at NASA, Beowulf systems are now deployed worldwide, chiefly in support of scientific computing.

A Beowulf cluster is a group of usually identical PC computers running Unix-like operating system, such as Linux. They are networked into a small TCP/IP LAN, and have libraries and programs installed which allow processing to be shared among them. Thus the Beowulf cluster forms a parallel, virtual supercomputer.

Beowulf cluster at Computational Mechanics Lab. of Department of Mechanical Engineering, PIEAS presently consists of four Pentium 4 (2.2 GHz) computers, each with 128 MB RAM and 40 GB hard disk. The operating system used is Fedora Core 5. The client nodes are connected to the server via its eth1 forming a cluster local network. The server PC is connected to the main PIEAS LAN (Fig. 1) via eth0. Thus only the server is visible to anyone connected to the main network, and the clients are hidden. All the client nodes have identical configuration. Only the server has some differences as required. The programmer may log on to the server through secure shell (ssh).

To enable parallel computing the cluster is equipped with MPI and OpenMP as well. For compiling the programs used in this work, Intel® Fortran Compiler 9.1 for Linux has been installed. This compiler is available as free download for non-commercial use only. It can compile programs with OpenMP directives, and is optimized for Pentium class processors.

All this makes the Beowulf cluster a simple parallel computer; a very inexpensive tool for learning and teaching parallel computing techniques. Even if a supercomputer is available, it may be beneficial to test the algorithms on the Beowulf first and then use the supercomputer for actual simulations.
PROGAMMING THE FINITE ELEMENT METHOD

Fortran 95 has been used for programming of finite element method for analysis of two-dimensional linear elastic problems. Fortran 95 is the latest dialect of FORTRAN. It has very radical improvements compared with the previous (and the most famous) standard, FORTRAN 77. FORTRAN 77 still remains, overwhelmingly, the most widely used programming language used for scientific and engineering programs. Fortran 95 includes almost all the features of FORTRAN 77. The radical improvements include whole array manipulation, better conditional statements, pointers, operator overloading, etc. By far the most important feature is the modular programming approach. This allows the programmer to create a library of subprograms (functions and subroutine) which may be precompiled and the main programs be linked to the parts of the library that are needed [1, 2]. This approach has been followed in this work.

Various modules have been created. The most important of these is the module FEAmodule, which contains subprograms related to finite element computations, such as, calculation of element stiffness matrix \([k^{(e)}]\), calculation of material constitutive matrix \([D]\), calculation of strain-displacement matrix \([B]\), etc. The code for 2D analysis has been thoroughly verified using different case studies. Efforts are underway to extend this work to three-dimensional analysis, since it is the 3D analysis that involves “large” problems and really benefits from parallelization.

Parallel Programming

Parallel programming environments involve several distinct processors (few expensive ones or many cheaper ones). Programs and/or data can reside on different processors, which have to communicate with each other. Two methods are commonly used in this regard: MPI and OpenMP. Here the programmer takes control of the communication process between various processors. It is the de facto industry standard for parallel computing [3, 4]. MPI standard defines a core set of library routines that are useful in message-passing programs. There are over 125 routines in MPI, however it is possible to write fully functional programs using as low as 6
routines only. The difficulty with MPI is that the programmer must explicitly control the parallelization of program and data. This may require a thorough redesign of the entire code. The other approach is to use OpenMP. OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs [6]. These are inserted at specific locations in an existing serial code and implemented by the compiler to parallelize those particular sections of the code. Usually these directives are used to parallelize time consuming DO-loops [1]. It is claimed that this approach is much easier to implement as compared to using MPI [3, 4].

OpenMP employs fork-join model of parallel execution. All OpenMP programs begin as a single thread called master thread. The master thread executes serially until a parallel region construct (!$OMP PARALLEL) is encountered. The master thread then creates a team of parallel threads. At the end of the parallel region (specified by !$OMP END PARALLEL) all threads join and terminate leaving only the master thread [6].

The work in this paper mainly concerns converting an existing serial code using OpenMP. After verification of the serial code, an attempt has been made to parallelize it. The objective was to investigate the ease of OpenMP parallelization.

The Serial Code for Assembly Process

In the serial code, there are many subprograms which all work on a single element at a time. The main program contains a DO-loop that spans the whole mesh (element by element), calculates element stiffness matrices \([k(e)]\) and assembles them into a large global stiffness matrix \([K]\). Afterwards, the system of equations is solved using pre-compiled BLAS and LAPACK routines. This work concerns the parallelization of the element assembly process.

As mentioned earlier, Fortran 95 has many attractive array manipulation features. Whole array operations are permissible with straightforward syntax. Arrays may have vector subscripts allowing for easier gather-scatter operations. This particular feature allows that a large program may be compacted into a small and easier to understand code of much fewer lines. Many DO-loops necessary with FORTRAN 77 are removed and the code looks quite compact:

```
DO e = 1, nElems
  xy = NodeXY( ElemConn(e, :, :) ! nodal coordinates of e\textsuperscript{th} element )
  g( (/ 1,3,5,7, 9,11,13,15 /) ) = ( (/ 2*ElemConn(e,:) -1/) )
  g( (/ 2,4,6,8,10,12,14,16 /) ) = ( (/ 2*ElemConn(e,:)   /) )
  ke = K2D(D, xy) ! calculates stiffness matrix of e\textsuperscript{th} element
  K(g,g) = K(g,g) + ke ! the assembly process
ENDDO
```

The work in this paper mainly concerns converting an existing serial code using OpenMP. After verification of the serial code, an attempt has been made to parallelize it. The objective was to investigate the ease of OpenMP parallelization.

The Serial Code for Assembly Process

In the serial code, there are many subprograms which all work on a single element at a time. The main program contains a DO-loop that spans the whole mesh (element by element), calculates element stiffness matrices \([k(e)]\) and assembles them into a large global stiffness matrix \([K]\). Afterwards, the system of equations is solved using pre-compiled BLAS and LAPACK routines. This work concerns the parallelization of the element assembly process.

As mentioned earlier, Fortran 95 has many attractive array manipulation features. Whole array operations are permissible with straightforward syntax. Arrays may have vector subscripts allowing for easier gather-scatter operations. This particular feature allows that a large program may be compacted into a small and easier to understand code of much fewer lines. Many DO-loops necessary with FORTRAN 77 are removed and the code looks quite compact:

This code segment also signifies the use of whole array operations and vector subscripts. This makes the programmer to focus more on algorithms. However, the performance of such codes may be implementation dependent.
OpenMP Implementation

The main attraction of using OpenMP for parallelization is that the programmer can work incrementally without major rewriting of the code. It is also expected that the final code would be as close to the original one and therefore easier to understand and maintain. The serial code above was parallelized by using OpenMP directives as:

```c
!$OMP PARALLEL DO NUM_THREADS(n) &
!$OMP& SHARED(NodeXY, ElemConn, K) &
!$OMP& PRIVATE(e, xy, g, ke), FIRSTPRIVATE(nElems, D) &
... (same code as above) ...
!$OMP END PARALLEL DO
```

At first, this seemed the logical way to implement OpenMP parallelization. Here the DO-loop has been parallelized into \( n \) threads. The large mesh related arrays NodeXY (nodal coordinates), ElemConn (element connectivity matrix) and K (global stiffness matrix) are SHARED by all threads. The arrays related to individual elements, such as xy (element nodal coordinates), g (index or steering vector), ke (element stiffness matrix) and other variables are private to each thread. All private variables have a distinct copy local to each thread. Thus a copy of a variable \( A \) local to thread \( i \), is untouched by thread \( j \) and so on. FIRSTPRIVATE directive is similar to PRIVATE, but also initializes the variable to its value in the serial region. Note that there is some flexibility in these declarations. For performance comparison purposes, the number of threads was changed for different runs.

RESULTS AND DISCUSSION

When the parallelized code was run, and the performance was compared with that of the original serial code, some interesting results were obtained.

Table 1 shows that for a typical mesh size, parallelization caused a decrease in performance. Furthermore, it was noted that for larger mesh, the performance decreased even more. At first sight, this result seems unexpected, however, there are reasons.

The assembly into global stiffness matrix is performed by a scatter operation. In this operation, values in the element stiffness matrix \( [k^{(e)}] \) are added to specific locations, determined by the element connectivities, into the global stiffness matrix \([K]\).

Table 1

<table>
<thead>
<tr>
<th>Processes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (serial)</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>0.13</td>
</tr>
<tr>
<td>3</td>
<td>0.32</td>
</tr>
<tr>
<td>4</td>
<td>0.36</td>
</tr>
</tbody>
</table>
Parallelization of this particular step is not easy, because of the dependencies in the assignment of 2D array $[K]$. It is possible that one particular element of $[K]$ is to be updated by different threads at the same time. Since $[K]$ is shared by all threads, this is a potential problem requiring controlled synchronization. The answer to this situation might be to use synchronization constructs such as: !$OMP ATOMIC, !$OMP CRITICAL, !$OMP BARRIER, etc. All these have the !$OMP FLUSH directive implied, which provides a consistent view of memory, i.e., all thread-visible data are written to memory at this point. These directives work slightly differently, but the main purpose is to update shared variables by different threads in a synchronized manner. Thus a single thread executes the command at a time, while other threads in the team wait. This waiting is the main cause of the decrease in performance in comparison with serial implementation.

In order to solve this problem, a solution might be to use PRIVATE and REDUCTION clause. However, PRIVATE would create multiple copies of $[K]$ for every thread drastically increasing the memory requirements, and the REDUCTION clause, at present, works only for scalars. The latest OpenMP specification (version 2.5) does not allow allocatable arrays with REDUCTION clause. Compilers do not conform to version 2.5 of OpenMP at present. Most compilers (e.g., Intel's ifort) follow version 1.1 and/or most of version 2.0.

To remove the synchronization costs of each update Bane, et al, [5] suggested a method which they implemented for one dimensional arrays. Their method also involves additional memory requirements and additional DO-loops.

Other parts of the Program

Other parts of the serial finite element program were relatively easily parallelized. Mainly OpenMP directives were applied to DO-loops. Compiler options such as auto-parallelization and optimization were good enough for many tasks as well. The only problem is that these options work slightly differently with different compilers causing portability problems.

General Recommendations for parallelizing DO-loops

OpenMP allows different ways to parallelize DO-loops. It allows for nested directives. This was tried to parallelize matrix multiplication. Fortran's MATMUL routine has mostly been used. It was noted that it is no good to have all the three DO-loops parallelized using OpenMP directives before each loop. Although allowed [3, 4], it causes inefficient thread generation on every loop increment.

It seems that Fortran 90/95 array syntax and intrinsic functions are very much implementation dependent. Thus, although coding in Fortran 90/95 may be far easier than in FORTRAN 77, the parallelization may not be so. This may improve with newer versions of OpenMP and Fortran compilers.

CONCLUSIONS

While starting this work it was expected that OpenMP would provide an incremental approach to
parallelization of existing serial code. However, it has been found out that this is not necessarily the case. OpenMP is much easier than MPI only in those cases where data dependency is not an issue. It may cause the programmer to overlook necessary synchronization. In such cases the tools provided by OpenMP are (at present) lacking in quick program development. With further improvements in OpenMP and compilers it is hoped that many of these limitations may be removed. At present though, it is recognized that for some applications the serial programs require major rewriting to take care of all the data management issues.

Different compilers behave differently, especially with autoparallelization and this causes portability problems.

REFERENCES