Journal of Applied Fluid Mechanics, Vol. 5, No. 2, pp. 123-132, 2012.
Available online at www.jafmonline.net, ISSN 1735-3572, EISSN 1735-3645.
DOI: 10.36884/jafm.5.02.12174

High Performance Computing of Three-Dimensional Finite Element Codes on a 64-bit Machine

M.P. Raju[1†] and S.K. Khaitan[2]

[1] Department of Mechanical Engg., Case Western Reserve University, Cleveland, OH, 44106, USA
[2] Department of Electrical Engg., Iowa State University, Ames, IA, 50014, USA

†Corresponding Author Email: raju192@gmail.com

(Received March 8, 2009; accepted March 28, 2009)

ABSTRACT

Three dimensional Navier-Stokes finite element formulations require huge computational power in terms of memory and CPU time. Recent developments in sparse direct solvers have significantly reduced the memory and computational time of direct solution methods. The objective of this study is twofold. First is to evaluate the performance of various state-of-the-art sequential sparse direct solvers in the context of finite element formulation of fluid flow problems. Second is to examine the merit in upgrading from 32 bit machine to a 64 bit machine with larger RAM capacity in terms of its capacity to solve larger problems. The choice of a direct solver is dependent on its computational time and its in-core memory requirements. Here four different solvers, UMFPACK, MUMPS, HSL_MA78 and PARDISO are compared. The performances of these solvers with respect to the computational time and memory requirements on a 64-bit windows server machine with 16GB RAM is evaluated.

Keywords: *Multifrontal, UMFPACK, MUMPS, HSL MA78, PARDISO, 64-bit.*

1. INTRODUCTION

Finite element discretization of Navier-Stokes equations involves a large set of non-linear equations, which can be converted into a set of linear algebraic equations using Picard's or Newton's iterative method. The resulting set of weak form (algebraic) equations in such problems may be solved either using a direct solver or an iterative solver. The direct solvers are known for their generality and robustness. However, the use of direct solution methods like the traditional frontal algorithms (Irons 1970) is limited by its huge memory requirements. The advent of multifrontal (Davis and Duff 1999) solvers and the efficient ordering techniques have increased the efficiency of direct solvers, in terms of memory and computational speed, for sparse linear systems. They make full use of the high computer architecture by invoking level 3 Basic Linear Algebra Subprograms (BLAS) library. Thus the memory requirement is greatly reduced and the computing speed is greatly enhanced. Multifrontal solvers have been successfully used for two-dimensional simulations both in the context of finite volume problems (Raju and T'ien, 2008a,b,c), in finite element problems (Gupta and Paglthivarthi 2007) and in power simulation systems (Khaitan et al. 2008, 2010). Raju and T'ien (2008a) implemented a finite volume formulation for solving gas combustion in an axi-symmetric candle flame. UMFPACK solver was used as a sparse direct solver. It has been demonstrated that the use of multifrontal solvers can significantly reduce the computational time when compared to the traditional iterative methods like SIMPLE algorithm used in finite volume formulations. Gupta and Pagalthivarthi (2007) implemented a finite element formulation for solving multi-size particulate flow inside a rotating channel and UMFPACK solver was used as a sparse direct solver. The advantage of using multifrontal solver over the traditional frontal solver in terms of both computational time and memory is demonstrated. However, the disadvantage of using sparse direct solvers (even multifrontal and other similar direct solvers) is that the memory size increases much more rapidly than the problem size itself (Gupta and Pagalthivarthi 2007) and the use of 64 bit machine with larger RAM has been recommended. On a 32 bit machine, the in-core memory is limited to 4 GB (3 GB on a windows machine). Hence as the problem size increase, memory becomes a limitation for direct solvers. To circumvent this problem, out-of-core solvers (Reid and Scott 2009; Raju and Khaitan 2009a) have been developed which has the capability of storing the factors on the disk during factorization but has increased computational burden. Another alternative is to use a larger RAM on a 64 bit machine to solve using in-core memory. Out-of-core solvers increase the computational burden due to the I/O operations to and from the hard disk. By using a 64 bit machine with a larger RAM, the factorization can be done in-core to reduce the computational time.

Computational times of sparse direct solvers can be significantly reduced by parallel implementation on

multiple processors. Parallel implementation can be either shared memory implementation on a multiple core machines (Raju and Khaitan 2010) or distributed computing on multiple machines using MPI programming (Raju 2009, Raju and Khaitan 2009b).

Iterative methods are known to be memory efficient. As the size of the system increases, the iterative methods outperform the performance of direct solvers and hence are preferred for large three dimensional problems. However, the matrices generated from the fully coupled solution of Navier-Stokes formulations are saddle point type matrices. These matrices are different from those normally encountered in applied mathematics are not easily amenable to the traditional iterative methods (Habashi and Hafez 1995). The application of traditional methods like SIMPLE iterations, pseudo time stepping or multigrid methods based on block gauss-seidal smoothers either fail or exhibit very slow convergence for the solution of finite element Navier-Stokes problems (Habashi and Hafez 1995). The choice of a good preconditioner is extremely challenging. Elman et al. (2005) discusses the development of preconditioners for such matrices. The implementation of these preconditioners is not straightforward and is very application specific. Recent efforts (Rehman et al. 2008) have demonstrated the successful implementation of classical ILU preconditioners with suitable reordering techniques referred as SILU. However the convergence of such preconditioners varies with the grid size and Reynolds number. On the other hand, direct solvers are preferred for their robustness. However, direct solvers are limited by its memory requirements. So it is up to the user to choose a direct solver or an iterative solver based on their needs. A comparison of the relative performance of direct solvers and iterative solvers for such saddle point matrices generated from three dimensional grids is not yet available in the literature. Commercial finite element based CFD package like FIDAP uses segregated approach for solving large 3D problems. However, the rate of convergence of segregated approach is much slower compared to the fully coupled Newton's approach. In addition, the application of modified Newton (or modified Picard (Raju and T'ien 2008a) can significantly reduce the CPU time of direct solvers. This is demonstrated in the later section of the paper.

This paper specifically addresses the memory requirement issues of direct solvers by upgrading from a 32 bit machine to a 64 bit machine with 16 GB RAM. Correlations are developed for the in-core memory requirement as a function of the problem size. This will give a fair idea of the size of the RAM required for solving a given problem on a 64 bit machine. Based on this, one could make an appropriate choice of whether to go for a 64 bit machine or to use an out-of-core solver. To the best of author's knowledge, this kind of work has not been reported in the literature.

In this paper the in-core implementation of the solvers is examined for the 3D finite element Navier-Stokes equations on a 64 bit machine. The performances of four state-of-art sparse direct solvers - UMFPACK, MUMPS, HSL, and PARDISO are evaluated in this paper. Highly optimized Intel® Math Kernel Library BLAS is used to improve the computational efficiency of the solvers. Flow through a three dimensional rectangular channel is taken as a benchmark problem. First the mathematical formulation for the primitive variables u,v,w is presented. This is followed by the description of the Newton's and modified Newton's algorithm. A brief overview of the sparse direct solvers used in this study is presented. This is followed by the presentation of results and discussion.

2. MATHEMATICAL FORMULATION

The governing equations for laminar flow through a three-dimensional rectangular duct are presented below in the non-dimensional form. In three-dimensional calculations, instead of the primitive u,v,w formulation, penalty approach is used to reduce the memory requirements.

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}=0, \tag{1}$$

$$\frac{\partial}{\partial x}\left(u^2\right)+\frac{\partial}{\partial y}(uv)+\frac{\partial}{\partial z}(uw)=\lambda\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\right)+\frac{\partial}{\partial x}\left(\frac{2}{Re}\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\frac{1}{Re}\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right)+\frac{\partial}{\partial z}\left(\frac{1}{Re}\left(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}\right)\right), \tag{2}$$

$$\frac{\partial}{\partial x}(uv)+\frac{\partial}{\partial y}\left(v^2\right)+\frac{\partial}{\partial z}(vw)=\lambda\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\right)+\frac{\partial}{\partial x}\left(\frac{1}{Re}\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right)+\frac{\partial}{\partial y}\left(\frac{2}{Re}\frac{\partial v}{\partial y}\right)+\frac{\partial}{\partial z}\left(\frac{1}{Re}\left(\frac{\partial v}{\partial z}+\frac{\partial w}{\partial y}\right)\right), \tag{3}$$

and

$$\frac{\partial}{\partial x}(uw)+\frac{\partial}{\partial y}(vw)+\frac{\partial}{\partial z}\left(w^2\right)=\lambda\frac{\partial}{\partial z}\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\right)+\frac{\partial}{\partial x}\left(\frac{1}{Re}\left(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}\right)\right)+\frac{\partial}{\partial y}\left(\frac{1}{Re}\left(\frac{\partial v}{\partial z}+\frac{\partial w}{\partial y}\right)\right)+\frac{\partial}{\partial z}\left(\frac{2}{Re}\frac{\partial w}{\partial z}\right). \tag{4}$$

where u, v, w are the x, y and z components of velocity. The bulk flow Reynolds number, $Re=\rho U_o L/\mu$, U_o being the inlet velocity, ρ the density, L the channel length, μ is the dynamic viscosity and λ is the penalty parameter. Velocities are non-dimensionalized with respect to U_o.

The boundary conditions are prescribed as follows:

(1) Along the channel inlet:

$$u=1;\ v=0;\ w=0. \tag{5}$$

(2) Along the channel exit :

$$\frac{\partial u}{\partial x}=0;\ \frac{\partial v}{\partial x}=0;\ \frac{\partial w}{\partial x}=0. \tag{6}$$

(3) Along the walls:

$$u=0;\ \ v=0;\ \ w=0. \tag{7}$$

3. NUMERICAL FORMULATION

Galerkin finite element method (GFEM) is used for the discretization of the above penalty based Navier Stokes equations. Three dimensional brick elements are used. The nonlinear system of equations obtained from GFEM is solved by Newton's method. Let $\underset{\sim}{X}^{(n)}$ be the available vector of field unknowns for the n^{th} iteration. Then the update for the $(n+1)^{st}$ iteration is obtained as

$$\underset{\sim}{X}^{(n+1)} = \underset{\sim}{X}^{(n)} + \alpha\, \delta \underset{\sim}{X}^{(n)}, \quad (8)$$

where α is an under-relaxation factor, and $\delta \underset{\sim}{X}^{(n)}$ is the correction vector obtained by solving the linearized system

$$[J]\left\{\delta \underset{\sim}{X}^{(n)}\right\} = -\left\{\underset{\sim}{R}_X\right\}^{(n)}. \quad (9)$$

Here, [J] is the Jacobian matrix,

$$[J] = \frac{\partial \underset{\sim}{R}_X^{(n)}}{\partial \underset{\sim}{X}^{(n)}}. \quad (10)$$

and $\left\{\underset{\sim}{R}_X\right\}^{(n)}$ is the residual vector. Newton's iteration is continued till the infinity norm of the correction vector $\delta \underset{\sim}{X}^{(n)}$ converges to a prescribed tolerance of 10^{-6}. In modified Newton's algorithm, the Jacobian is calculated only during the first iteration and is not updated during the subsequent iterations.

We observe that the discretizations of the governing partial differential Eqs. (7)-(10) by the GFEM scheme results in a set of nonlinear equations. The core of the resulting nonlinear equations is the solution of a sparse linear system, which is the most computationally intensive part of the solver. This is exploited in the work described here, via implementation of the state-of-the-art algorithms for ordering, preprocessing, and LU factorization. Highly optimized Intel® Math Kernel Library BLAS is used during the factorization and solve phase. Here four different solvers, UMFPACK, MUMPS, HSL-MA78 and PARDISO implemented and are compared.

The numerical algorithm can be broadly classified into three categories namely 1) numerical formulation with finite element discretization, 2) solution strategy for solving the resultant nonlinear equations and 3) solution of linear equations. The solution of the linear system of equations is the bottleneck both in term of computational time and memory requirements. Good non-linear solution strategy like modified Newton can help reduce the number of times the system of linear equations being solved. The latter section of the paper deals with the reduction in CPU time offered by modified Newton's method.

4. LINEAR SOLVERS

The core of any iterative solver like Newton iteration is the solution of a system of equations represented by Eq. (9). The Jacobian matrix is highly sparse and the fill-in is very low. We use this fact to gain computational efficiency by employing direct sparse linear solvers which use multifrontal or supernodal methods.

In general, the algorithms for sparse matrices are more complicated than for dense matrices. The complexity is mainly attributed to the need to efficiently handle fill-in in the factor matrices. A typical sparse solver consists of four distinct phases as opposed to two in the dense case:

1. The ordering step minimizes the fill-in and exploits special structures such as block triangular form.
2. An analysis step or symbolic factorization determines the nonzero structures of the factors and creates suitable data structures for the factors.
3. Numerical factorization computes the factor matrices.
4. The solve step performs forward and/or backward substitutions.

The section below presents a brief overview of the sparse direct solvers that are being used in this study.

4.1 Overview of the sparse direct solvers

UMFPACK

In the present study, UMFPACK v5.3.0 (Davis et al. 2004) is used. UMFPACK consists of a set of ANSI/ISO C routines for solving unsymmetric sparse linear systems using the unsymmetric multifrontal method. It requires the unsymmetric, sparse matrix to be input in a sparse triplet format. Multifrontal methods are a generalization of the frontal methods developed primarily for finite element problems (Amestoy and Duff 1989) for symmetric positive definite systems which were later extended to unsymmetric systems (Davis and Duff 1999).

UMFPACK first performs a column pre-ordering to reduce the fill-in. It automatically selects different strategies for pre-ordering the rows and columns depending on the symmetric nature of the matrix. The solver has different built in fill reducing schemes-COLAMD (Davis et al. 2004) and AMD (approximate minimum degree) (Davis 2004). During the factorization stage, a sequence of dense rectangular frontal matrices is generated for factorization. A supernodal column elimination tree is generated in which each node in the tree represents a frontal matrix. The chain of frontal matrices is factorized in a single working array.

MUMPS

MUMPS 4.8.3 ("Multifrontal Massively Parallel Solver") written in Fortran 90, is a package, based on multifrontal algorithms (Amestoy et al. 2000, 2002, 2006; Duff and Reid 1983, 1984; Guermouche and L'Excellent 2006; Guermouche et al. 2003), for solving systems of linear equations of the form in Eq. (9), where A is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. It performs a direct factorization $A = LU$ or $A = LDL^T$ depending on the symmetry of the matrix. MUMPS is primarily a parallel solver designed for computational efficiency and exploits both parallelism arising from sparsity in the matrix A and from dense factorizations kernels. The parallel version of MUMPS requires MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries. The sequential version only relies on BLAS. MUMPS has several built-in ordering algorithms, and provides a tight interface to external ordering packages such as PORD (Schulze 2001), METIS (Karypis and Kumar 1998) and also a possibility for the user to input a given ordering. In this paper, only the sequential version of the solver is evaluated.

HSL_MA78

HSL_MA78 solves one or more sets of sparse linear equations, Ax = b or A^Tx = b, by the multifrontal method, optionally using direct access files for the matrix factors. The code has low in-core memory requirements. HSL_MA78 is written using FORTRAN 95. The code implements multifrontal algorithm and takes advantage of the dense linear algebra kernels. These kernels are available as a separate package HSL_MA74, which uses high level BLAS to perform the partial factorization of frontal matrices. Ordering has to be supplied by hooking external ordering packages like METIS etc. HSL package HSL_MC68 offers efficient implementation of minimum degree algorithm (Tinney and walker 1967) and approximate minimum degree algorithm (Amestoy et al. 1996, 2004). The code is primarily designed as an efficient out-of-core solver, although it is capable of solving it in-core. While solving in-core, if the memory is not sufficient, the code automatically shifts to out-of-core solution. In this paper, only the in-core option of the solver is evaluated.

PARDISO

PARDISO solver is a part of the INTEL MKL Library. The PARDISO package is high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques (Schenk et al. 2000). To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining is used with a combination of left- and right-looking supernode techniques (Schenk 2000, Schenk and Gartner 2001, 2002, 2004). Unsymmetric permutation of rows is used to place large matrix entries on the diagonal. Complete block diagonal supernode pivoting allows dynamical interchanges of columns and rows during the factorization process. The level-3 BLAS efficiency is retained and an advanced two-level left–right looking scheduling scheme is used to achieve higher efficiency. The goal is to preprocess the coefficient matrix A so as to obtain an equivalent system with a matrix that is better scaled and more diagonally dominant. This preprocessing reduces the need for partial pivoting, thereby speeding up the factorization process. PARDISO also supports out-of-core solution. In this paper PARDISO is evaluated for the in-core sequential solver on a single processor.

5. COMPUTATIONAL CHALLENGES

It is to be noted that for three dimensional grids, the matrices generated are less sparse compared to the matrices generated from two-dimensional grid. Typically an interior node in a three-dimensional finite element grid is connected to 27 nodes including it. Since there are 3 dof's at each node, a typical row consists of 81 non-zero entries. In a two-dimensional grid, a typical row consists of 27 non-zero entries. This would increase the frontal size considerably. Hence solving three-dimensional finite element problems using direct solvers is quite challenging both in terms of computational time and memory requirements. Large problems cannot be solved on a 32-bit machine using in-core techniques. Hence there is a need for alternative strategies for handling large three-dimensional problems. There are 3 different ways of handling this problem.

(a) Using a 64 bit machine with larger RAM

(b) Using out-of-core solvers

(c) Using parallel solvers

This paper studies the performance of sequential in-core direct solvers on a 64 bit machine with 16GB RAM. All the computations are run a windows machine with Intel Xeon processor.

6. IMPLEMENTIONAL CHALLENGES

The solvers used in this paper are either public domain solvers or commercial solvers. The solvers are hooked to the finite element code to solve the linear system of equations. Although the codes are readily available, the integration of the solvers with the finite element code on a 64 bit windows machine is not straightforward. UMFPACK, MUMPS are public domain solvers. PARDISO is a commercial solver available within the Intel MKL package. HSL_MA78 is part of the HSL 2007 package. An evaluation version of HSL_MA78 is being used in this paper. Except for PARDISO, the source code is available for the remaining solvers. The following points will serve as guideline for future researchers who would like to implement these solvers on a 64 bit windows machine.

(1) MUMPS has both C and Fortran routines. Separate libraries are built for the C and Fortran routines using Intel Visual Fortran and Microsoft Visual Studio 2005. Makefiles are supplied within the MUMPS package only for Linux environments. The project workspaces are appropriately built to include the preprocessing directives to build libraries. UMFPACK is based on C code. Library is also built for the UMFPACK solver on the 64 bit machine. The main finite element code is based on Fortran 90. Hence appropriate interfacing routines are written to call routines from a C library. METIS library is not available for 64 bit windows machine. Hence a 64 bit library is built using the METIS source code. While compiling the solvers on a 64 bit machine and integrating it to the finite element code, precaution has to be taken to prevent mixing of short integers and long integers. This could lead to garbage values.

(2) The finite element code generates element entries for each element. Except HSL_MA78 and MUMPS, all the other solvers requires global assembly matrix as the input. UMFPACK provides a matrix manipulation routine (umfpack_triplet_to_col) which converts matrix entries in coordinate format to compressed column format. This routine automatically sums up the duplicate entries arising from finite element matrix entries. Hence it is a useful routine for handling finite element entries. UMFPACK requires the assembled matrix in compressed column format. PARDISO requires the assembled matrix in compressed row format. By suitably modifying the umfpack_triplet_to_col routine, finite element entries can be assembled to a compressed row format.

(3) METIS has to be externally linked to HSL_MA78 to generate the reduced fill ordering. For this the finite element entries have to be assembled without the numerical values to generate the adjacency matrix. This can be provided as an input to METIS_NODEND routine to generate the reduced fill ordering.

7. RESULTS AND DISCUSSION

Before comparing the various solvers for their relative performances, each individual solver is tuned for its optimal performance, specifically the choice of the ordering package. Each solver has inbuilt ordering packages, whose choice can affect the performance of the solver.

Mumps has different inbuilt ordering packages (AMD, QAMD, AMF) and a strong coupling interface with external orderings like METIS and PORD. Memory relaxation is taken as 100%. It is commonly observed that if we use values much lower than 100%, the solver crashes for some problems due to insufficient memory. The actual memory used by the solver does depend on the choice of the memory relaxation parameter. MUMPS in-core solver is used for all the cases. Table 1 shows the performance comparison of the various ordering methods. All the cases are run for 30x30x30 mesh. The term "30x30x30" represents a grid with 30 elements in the x direction, 30 elements in the y direction and 30 elements in the z directions. The CPU time and memory for each of the solver are compared. The CPU time reported is the CPU time for a single Newton iteration. It includes analysis phase, factorization phase and solve phase. Table 1 shows that AMD and QAMD perform very poorly for the given system of equations resulting from three dimensional finite element simulations. Of all the ordering packages, METIS gives best results. Compared to AMD, METIS results in almost one-third of the floating point operations. The computational time and memory requirements are lower for the METIS ordering. Based on this result, METIS ordering is used for all subsequent runs using MUMPS solver.

UMFPACK by default chooses the CHOLAMD ordering method for unsymmetric matrices. METIS ordering is not provided within the solver. So the default ordering CHOLMOD is retained for the subsequent calculations for UMFPACK solver. HSL_MA78 solver can be externally hooked to another HSL routine which generates the ordering (HSL_MC68) or it can be hooked to METIS ordering. HSL_MC68 uses minimum degree algorithms to generate the ordering of matrices generated from finite element assembly. Table 2 shows that METIS ordering gives very good performance compared to the HSL_MC68 ordering. PARDISO has two ordering methods within the solver itself - minimum degree (MD) ordering and METIS ordering. Table 3 shows that METIS ordering significantly improves the efficiency of the solver. Table 4 shows the comparison of computational time and memory requirement of different solvers. The computational time is split into 4 stages. (a) time for matrix generation (b) time for analysis phase or symbolic factorization, (c) time for numeric factorization, (d) time for the solve phase.

***Table 1** Performance of different orderings for MUMPS solver*

Ordering	#dof's	Cpu time (sec)	Memory (GB)
AMD	89373	142.8	4.06
QAMD	89373	142.75	4.04
AMF	89373	105.7	3.48
PORD	89373	86.6	3.18
METIS	89373	59.01	3.02

***Table 2** Performance of different orderings for HSL solver*

Ordering	#dof's	Cpu time (sec)	Memory (GB)
HSL_MC68	89373	524	6.88
METIS	89373	318	4.63

***Table 3** Performance of different orderings for PARDISO solver*

Ordering	#dof's	Cpu time (sec)	Memory (GB)
MD	89373	162	2.78
METIS	89373	60	1.42

This table will give an idea of how much time the solver is spending on each of the stages. The table shows that factorization is the most time consuming step. Around 85-99% of the time is spent in numerical factorization step. Note that for HSL solver, the time for numerical factorization and solve phase are reported together as the solver performs these both functions together. HSL-MA78 takes the longest time for solving the linear system of equations. It is around two times slower than UMFPACK and around six times slower than MUMPS or PARDISO solver. The memory requirement for HSL is similar to that of MUMP solver. MUMPS and PARDISO perform equally well in terms of computational time. In terms of memory requirement, UMFPACK requires largest memory and PARDISO requires the least memory. UMFPACK requires around 4 times the memory required by PARDISO. In summary, MUMPS and PARDISO perform equally well in terms of computational time and PARDISO requires the least memory compared to all the other solvers.

Tables 5-8 show the comparison of different solvers for different mesh sizes. The general observation is that as the number of degrees of freedom (dof's) increase, the computational time and memory requirements increase, which is expected. However, the increase is not linearly proportional to the number of the dof's. For example, if we compare the mesh sizes 100x20x10 and 50x20x20, the number of dof's are the same for both meshes. But the computational time and memory requirement are quite different. This observation is valid for all the solvers.

Table 4 *Computational time and memory requirements for 30x30x30 grid*

	Computational time (Seconds)					***Memory (MB)***
Solver	*Matrix assembly*	*Analysis phase*	*Numeric factorization*	*Solve phase*	*Total time*	
UMFPACK	*4*	*0.84*	*154.8*	*1.1*	*160.74*	*5520*
MUMPS	*4*	*1.51*	*53.3*	*0.58*	*59.39*	*3020*
PARDISO	*4*	*2.19*	*52.6*	*0.47*	*59.26*	*1420*
HSL_MA78	*4*	*0.64*	*313.14*		*317.78*	*4720*

Table 5 *Performance of UMFPACK solver for different mesh sizes*

				UMFPACK		
nex	***ney***	***nez***	***#dof's***	***Cpu time (sec)***	***Memory (MB)***	***Gflops***
50	*10*	*10*	*18513*	*1.75*	*200*	*5.10E+09*
100	*10*	*10*	*36663*	*4.29*	*470*	*1.40E+10*
200	*10*	*10*	*72963*	*7.7*	*840*	*2.70E+10*
50	*20*	*10*	*35343*	*9.3*	*630*	*3.90E+10*
100	*20*	*10*	*69993*	*22.94*	*1220*	*1.00E+11*
100	*20*	*20*	*133623*	*188*	*7170*	*2.56E+12*
100	*50*	*20*	*324513*	*1505.6*	*15830*	*9.30E+12*
100	*50*	*50*	*788103*	*-*	*insufficient*	*-*
50	*20*	*20*	*67473*	*64.92*	*3580*	*3.50E+11*
50	*50*	*10*	*85833*	*65.28*	*1960*	*3.60E+11*
50	*50*	*20*	*163863*	*522*	*9170*	*3.10E+12*
50	*50*	*50*	*397953*	*-*	*insufficient*	*-*

Table 6 *Performance of MUMPS solver for different mesh sizes*

				MUMPS		
nex	***ney***	***nez***	***#dof's***	***Cpu time (sec)***	***Memory (MB)***	***Gflops***
50	*10*	*10*	*18513*	*2.79*	*230*	*6.70E+09*
100	*10*	*10*	*36663*	*5.32*	*520*	*1.52E+10*
200	*10*	*10*	*72963*	*10.6*	*1100*	*3.00E+10*
50	*20*	*10*	*35343*	*8.4*	*650*	*2.73E+10*
100	*20*	*10*	*69993*	*16.7*	*1400*	*6.50E+10*
100	*20*	*20*	*133623*	*67.6*	*3870*	*3.34E+11*
100	*50*	*20*	*324513*	*384*	*134200*	*2.32E+12*
100	*50*	*50*	*788103*	*-*	*insufficient*	*-*
50	*20*	*20*	*67473*	*27*	*1810*	*1.30E+11*
50	*50*	*10*	*85833*	*29.7*	*2110*	*1.40E+11*
50	*50*	*20*	*163863*	*133.7*	*5930*	*7.60E+11*
50	*50*	*50*	*397953*	*-*	*insufficient*	*-*

Table 7 *Performance of HSL MA78 solver for different mesh sizes*

				HSL_MA78		
nex	***ney***	***nez***	***#dof's***	***Cpu time (sec)***	***Memory MB)***	***Gflops***
50	*10*	*10*	*18513*	*11*	*240*	*4.40E+10*
100	*10*	*10*	*36663*	*20.9*	*500*	*8.40E+10*
200	*10*	*10*	*72963*	*35.4*	*790*	*1.48E+11*
50	*20*	*10*	*35343*	*46.7*	*650*	*1.95E+11*
100	*20*	*10*	*69993*	*63.4*	*1530*	*2.89E+11*
100	*20*	*20*	*133623*	*290*	*3910*	*1.3E+12*
100	*50*	*20*	*324513*	*1732*	*13940*	*7.9E+12*
100	*50*	*50*	*788103*	*-*	*Insufficient*	*-*
50	*20*	*20*	*67473*	*145*	*1980*	*6.28E+11*
50	*50*	*10*	*85833*	*159*	*2380*	*6.81E+11*
50	*50*	*20*	*163863*	*715*	*7120*	*3.1E+12*
50	*50*	*50*	*397953*	*-*	*Insufficient*	*-*

Table 8 *Performance of PARDISO solver for different mesh sizes*

				PARDISO		
nex	***ney***	***nez***	***#dof's***	***cpu time (sec)***	***Memory (MB)***	***Gflops***
50	*10*	*10*	*18513*	*2.3*	*80*	*6.71E+09*
100	*10*	*10*	*36663*	*4.36*	*250*	*1.60E+10*
200	*10*	*10*	*72963*	*9.4*	*570*	*3.50E+10*
50	*20*	*10*	*35343*	*6.3*	*290*	*2.60E+10*
100	*20*	*10*	*69993*	*14.6*	*690*	*6.70E+10*
100	*20*	*20*	*133623*	*64.4*	*1920*	*3.48E+11*
100	*50*	*20*	*324513*	*391*	*6620*	*2.30E+12*
100	*50*	*50*	*788103*	*-*	*insufficient*	*-*
200	*50*	*20*	*645813*	*901*	*17200*	*6.00E+12*
50	*20*	*20*	*67473*	*25.9*	*850*	*1.35E+11*
50	*50*	*10*	*85833*	*27.4*	*1020*	*1.38E+11*
50	*50*	*20*	*163863*	*134*	*2860*	*7.63E+11*
50	*50*	*50*	*397953*	*1059*	*10670*	*6.30E+12*

This kind of behavior is to be expected when using direct solvers using frontal type methods. The size of the frontal matrix or frontal width depends on the grid structure and hence the performance will be dependent on the grid distribution.

Both MUMPS and PARDISO perform well in terms of computational time, especially for finer meshes. For very coarse meshes, UMFPACK seems to perform slightly better. However as the number of dof's increase, MUMPS and PARDISO outperform UMFPACK. Of the two, PARDISO perform slightly better than MUMPS in most of the cases. HSL-MA78 is the slowest solver amongst all of them. The numbers of floating point operations (FLOPS) follow similar trends as the computational time. In terms of memory requirement, PARDISO outperforms all the other solvers. UMFPACK seems to require largest memory requirement for most of the cases. HSL and MUMPS require similar amounts of memory.

Table 8 shows that on a 64 bit machine with 16 GB memory, PARDISO was able to solve up to 200x50x20 grid size, which corresponds to approximately 0.65 million dof's. This shows that even with the best solver and using 64 bit machine and 16 GB RAM, the size of the problems being handled is still moderate. For fair comparison of the maximum capacity for 32 bit and 64 bit machines, grids of aspect ratio 1 are considered. On a 32 bit machine, a maximum of 36x36x36 grid (151959 dof's) can be solved. This grid requires around 3 GB of RAM, which is the maximum capacity that windows 32 bit machine can handle. On a 64 bit machine with 16 GB RAM, a maximum of 54x54x54 grid (499125 dof's)

can be solved, which requires 15.8 GB. An increase in capacity of around 3.3 times (in terms of dof's) by upgrading from 3GB RAM to 16 GB RAM (an increase by 5.33 times). By upgrading from 32 bit, 3GB machine to 64 bit, 16 GB machine, moderately larger problems can be solved. This paper can serve as a guideline to decide if one would like to go a 64 bit machine to enhance the computational capacity or to go for an out-of-core solver, with a compromise in the computational time. In essence, using a 64 bit machine with larger RAM can help solve moderately larger problems than with using a 32 bit machine but will eventually run out of memory.

Correlations are generated for the computational time and memory requirement of the four solvers as a function of the number of grid nodes and grid aspect ratios. Correlations will give an idea of how the solver performs in terms of computational time and memory as the size of problem increases. In addition to the results shown in the Tables 3-6, the correlations are generated from a set of 50 different grid sizes (not presented here). The correlations are presented below. In the Eqs. 11-18, T represents the computational time in seconds, n represents the number of degrees of freedom, ar_1 and ar_2 are the grid aspect ratio's nex/ney and nex/nez respectively and M is the memory requirement in Mega Bytes. The variables nex, ney and nez represent the number of grid elements in the x,y and z respectively. The exponent's of ar_1 and ar_2 are purposely chosen to be identical in the correlation.

UMFPACK:

$$T = 8.54\times10^{-9}\, n^{2.1} ar_1^{-0.356} ar_2^{-0.356};\quad R^2 = 0.967 \qquad (11)$$

$$M = 6.2\times10^{-5}\, n^{1.595} ar_1^{-0.21} ar_2^{-0.21};\quad R^2 = 0.85 \qquad (12)$$

MUMPS:

$$T = 1.47\times10^{-6}\, n^{1.54} ar_1^{-0.173} ar_2^{-0.173};\quad R^2 = 0.934 \qquad (13)$$

$$M = 4.76\times10^{-4}\, n^{1.37} ar_1^{-0.114} ar_2^{-0.114};\quad R^2 = 0.99 \qquad (14)$$

PARDISO:

$$T = 4.42\times10^{-7}\, n^{1.656} ar_1^{-0.21} ar_2^{-0.21};\quad R^2 = 0.912 \qquad (15)$$

$$M = 7.57\times10^{-5}\, n^{1.46} ar_1^{-0.086} ar_2^{-0.086};\quad R^2 = 0.99 \qquad (16)$$

HSL MA78:

$$T = 1.22\times10^{-6}\, n^{1.71} ar_1^{-0.218} ar_2^{-0.218};\quad R^2 = 0.853 \qquad (17)$$

$$M = 7.4\times10^{-4}\, n^{1.366} ar_1^{-0.222} ar_2^{-0.222};\quad R^2 = 0.87 \qquad (18)$$

The above correlations indicate that the computational time and memory requirement for the solvers is not only dependent on the dof's but is also dependent on the grid aspect ratio's. The above correlations indicate that the absolute values exponents of n and grid aspect ratio is higher for UMPACK. This implies that as the dof's increase, the solver becomes inefficient. For MUMPS and PARDISO, the exponent for n is around 1.5-1.7. Hence the shoot up of computational time with the number of dof's is not quite high. Hence for solving large problems, MUMPS or PARDISO solvers will be a good option. It is also observed that the memory requirement for MUMPS and PARDISO is almost independent on the grid aspect ratio's. The correlation coefficients indicate that both MUMPS and PARDISO behave almost similarly with respect to its sensitivity towards changes in grid size and grid aspect ratio's.

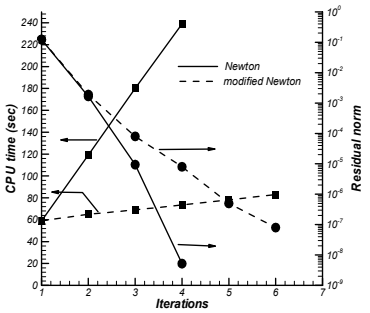

Fig. 1. *Comparison of CPU time and convergence rates of Newton and modified Newton algorithms.*

The next section deals with the performance of Newton and modified Newton algorithms. Figure 1 shows the rate of convergence of Newton's algorithm for a 30x30x30 grid using PARDISO solver. For this simple problem of flow through a rectangular channel, it is observed that Newton gives full quadratic convergence and the residual norm fall below 10^{-8} in 4 iterations. For more complex problems, initial guess may not be close enough for Newton iterations to converge.

In these cases, the first few iterations may have to be performed with Picard iteration and then Newton iterations could be applied subsequently to harness the advantage of quadratic convergence. Modified Newton takes 6 iterations to converge to a level of 10^{-7}. The rate of convergence is observed to be superlinear instead of quadratic. However, it is observed that modified Newton iterations (83 seconds) significantly much less CPU time compared to the Newton iterations (240 seconds).The CPU time for first iteration is the same in both the methods but in the subsequent iterations, the CPU time for modified Newton is significantly lower. This is because, the Jacobian is no longer updated and consequently the LU factorization step is eliminated. The LU factors from the first iteration are reused repeatedly during the solve phase. By avoiding LU factorization, a significant savings in computational time is observed. The loss in quadratic convergence is more than compensated by the significant savings in CPU time. The memory requirement is exactly identical for both the methods. This is one of the major advantages of using a direct solver. Newton or Picard can be modified in such a way that it can repeatedly reuse the LU factors and thus obtain a significant savings in the computational time.

The application of modified Picard (Raju and T'ien 2008a) has been successfully demonstrated in the context of finite volume combustion application. The major drawback of direct solvers is the memory requirement. This paper demonstrates that moderately larger problems can be solved using a 64 bit machine with a larger RAM. On a 64 bit machine with 16 GB RAM, three to four times larger problems can be solved as compared to a 32 bit machine with 3 GB RAM.

8. CONCLUSION

The performance of sequential direct sparse solvers in the context of three dimensional finite element formulations for rectangular channel is evaluated on a 64 bit windows machine with 16 GB RAM. Four sparse solvers UMFPACK, MUMPS, PARDISO, HSL-MA78 are evaluated in this paper. Based on the results, the following conclusions are derived. Of all the ordering methods, METIS gives good fill reduced ordering for three dimensional problems. In terms of computational time both MUMPS and PARDISO perform well. The memory requirement PARDISO solver is the least and hence larger problems can be solved using PARDISO. Hence PARDISO (with METIS ordering) is a better choice for selecting an in-core sparse direct solver for three dimensional problems. By upgrading from 32 bit, 3 GB machine to a 64 bit, 16 GB machine, the size of the problem could be roughly increased by a factor of 3.3 using PARDISO solver. While using a sparse direct solver, advantage can be taken from modified Newton's method to gain significant computational savings.

REFERENCES

Amestoy, P.R., A. Guermouche, J.Y. L'Excellent and S. Pralet (2006). Hybrid scheduling for the parallel solution of linear systems. Parallel Computing 32(2), 136–156.

Amestoy, P.R., T.A. Davis and I.S. Duff (2004). Algorithm 837: AMD, an approximate minimum degree ordering algorithm. ACM Trans. Mathematical Software 30(3), 381-388.

Amestoy, P.R., I.S. Duff, J. Koster and J.Y. L'Excellent (2002). A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications 23(1), 15–41.

Amestoy, P.R., I.S. Duff, J. Koster and J.Y. L'Excellent (2000). Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. 184, 501–520.

Amestoy, P.R., T.A. Davis and I.S. Duff (1996). An approximate minimum degree ordering algorithm. SIAM: Matrix Analysis and Applications 17, 886-905.

Amestoy, P.R. and I.S. Duff (1989). Vectorization of a multiprocessor multifrontal code. Int. J. Supercomputer Appl. 3(3), 41–59.

Davis, T., (2004). A column pre-ordering strategy for the unsymmetric-pattern multi-frontal method. ACM Trans. Math. Software 30(2), 165-195.

Davis, T., J. Gilbert and E. Larimore (2004). Algorithm 836: COLAMD, an approximate column minimum degree ordering algorithm. ACM Trans. Math. Software 30(3), 377-380.

Davis, T.A. and I.S. Duff (1999). A combined unifrontal/multifrontal method for unsymmetric sparse matrices. ACM Trans. Math. Soft. 25(1), 1–20.

Duff, I.S. and J.K. Reid (1984). The multifrontal solution of unsymmetric sets of linear systems. SIAM Journal on Scientific and Statistical Computing 5, 633–641.

Duff, I.S. and J.K. Reid (1983). The multifrontal solution of indefinite sparse symmetric linear systems. ACM Transactions on Mathematical Software 9, 302–325.

Elman, H.C., D.J. Silvester and A.J. Wathen (2005). Finite elements and fast iterative solvers: with applications in Incompressible Fluid dynamics. Oxford:Oxford University Press.

Guermouche, A. and J.Y. L'Excellent (2006). Constructing memory-minimizing schedules for multifrontal methods. ACM Transactions on Mathematical Software 32(1), 17–32.

Guermouche, A., J.Y. L'Excellent and G. Utard (2003). Impact of reordering on the memory of a multifrontal solver. Parallel Computing 29(9), 1191–1218.

Gupta, P.K. and K.V. Pagalthivarthi (2007). Application of Multifrontal and GMRES Solvers for Multisize Particulate Flow in Rotating Channels. Prog. Comput. Fluid Dynam. 7, 323–336.

Habashi, W.G. and M.M. Hafez (1995). Computational Fluid Dynamic Techniques. Amsterdam:Gordon and Breach Science Publishers.

Irons, B.M. (1970). A frontal solution scheme for finite element analysis. Numer. Meth. Engg. 2, 5-32.

Karypis, G., and V. Kumar (1998). METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0. University of Minnesota.

Khaitan, S., J. McCalley and M.P. Raju (2010). Numerical methods for on-line power system load flow analysis, Energy systems 1(2), 273-288.

Khaitan, S., J. McCalley and Q. Chen (2008).

Multifrontal solver for online power system time-domain simulation, Power Systems, IEEE Transactions 23 (4), 1727–1737.

Raju, M.P. and S. Khaitan (2010). Implementation of Shared Memory Sparse Direct Solvers for Three Dimensional Finite Element Codes. Journal of Computing, accepted for publication.

Raju, M.P. (2009). Parallel Computation of Finite Element Navier-Stokes codes using MUMPS Solver. International Journal of Computer Science Issues 4(2), 20-24.

Raju, M.P. and S. Khaitan (2009). High Performance Computing Using Out-of-Core Sparse Direct Solvers. International Journal of Mathematical, Physical and Engineering Sciences 3(2) 96-102.

Raju, M.P. and S. Khaitan (2009). Domain Decomposition Based High Performance Parallel Computing. International Journal of Computer Science Issues 5, 27-32.

Raju, M.P. and J.S. T'ien (2008). Development of Direct Multifrontal Solvers for Combustion Problems. Numerical Heat Transfer, Part B 53, 1-17.

Raju, M.P. and J.S. T'ien (2008). Modelling of Candle Wick Burning with a Self-trimmed Wick. Comb. Theory Modell. 12(2), 367-388.

Raju, M.P. and J.S. T'ien (2008). Two-phase flow inside an externally heated axisymmetric porous wick. J. Porous Media 11(8), 701-718.

Rehman, M., C. Vuik and G. Segal (2008). A comparison of preconditioners for incompressible Navier-Stokes solvers. Int. J. Num. Met. Fluid 57, 1731-1751.

Reid, J. and J. Scott (2009). An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. International Journal for Numerical Methods in Engineering 77(7), 901-921.

Schulze, J. (2001). Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. BIT 41(4), 800–841.

Schenk, O. and K. Gartner (2004). Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. Journal of Future Generation Computer Systems 20(3), 475-487.

Schenk, O. and K. Gartner (2002). Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems. Parallel Computing 28, 187-197.

Schenk, O. and K. Gartner (2001). Sparse Factorization with Two-Level Scheduling in PARDISO. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14.

Schenk, O., K. Gartner and W. Fichtner (2000). Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors. BIT 40(1), 158-176.

Schenk, O. (2000). Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors. Thesis (PhD). ETH Zurich.

Tinney, W.F. and J.W. Walker (1967). Direct solutions of sparse network equations by optimally ordered triangular factorization. Proc. IEEE 55, 1801-1809.