

Rapid Parallelisation of the Industrial Modelling Code PZFlex

Rob Baxter[†], Paul Graham[†], Michael Bowers[†]
John Mould^{*}, Greg Wojcik^{*}, Dave Vaughan^{*}

[†]EPCC, University of Edinburgh

^{*}Weidlinger Associates

Abstract

This paper presents the results of a recent collaborative project between EPCC at the University of Edinburgh and Scottish engineering consultants Weidlinger Associates Ltd. The aim of this work was to extend the solver capabilities, and in particular the parallel solver capabilities, of an industrial modelling code, *PZFlex*. Two key solvers were compared, a preconditioned conjugate gradient (PCG) iterative solver and a state-of-the-art direct solver (MFACT). The PCG solver was parallelised using OpenMP, the new standard for compiler directive based programming of shared memory multiprocessor systems. Our results show that OpenMP delivers parallel efficiencies of up to 80% for the PCG solver. It also offers good scalability, allowing the simple PCG algorithm to beat the complex MFACT direct solver for typical simulation runs of *PZFlex*.

1 Introduction

PZFlex [1] is owned by Weidlinger Associates and is used for the finite-element modelling of piezoelectric devices. It supports a broad range of modelling capabilities in both two and three dimensions, and is used widely in industry and academia for such tasks as transducer modelling, sonar design and medical ultrasound modelling.

OpenMP, the new standard for compiler directive based programming of shared memory multiprocessors, offers a straightforward way to exploit thread-level parallelism and is particularly easy to apply to existing serial programs. This, and the increasing availability of desktop and desk-side multiprocessor systems, are reasons Weidlinger has chosen OpenMP to accelerate the performance of *PZFlex* for as broad a user-base as possible.

In common with many finite element codes, the performance of *PZFlex* is dominated by a small number of key solver kernels. For this reason it can be made particularly amenable to straightforward parallelisation using OpenMP. This allows *PZFlex* to exploit relatively low-cost hardware such as multiprocessor PC systems in an efficient way.

In this work we implement and compare two key solvers for *PZFlex*:

- a parallel preconditioned conjugate gradients (PCG) iterative solver, and
- a version of the direct multi-frontal solver MFACT.

Firstly there is a brief discussion of the background to the *PZFlex* algorithms. Then we introduce the concept of the OpenMP paradigm for shared memory programming. Then the implementation details for developing the parallel PCG solver are discussed, and then finally the two solvers are compared.

2 Background

Piezoelectric transducers convert electrical signals to mechanical signals and vice versa. They serve as transmitters and receivers in imaging systems for sonar, medical, and non-destructive evaluation applications, as well as in non imaging applications like surface acoustic wave devices in signal processing. The piezoelectric transducer market is broad and the technology, despite its relative maturity, has great potential for improvement and innovation. In order to aid the design of such devices, discrete numerical modelling methods are utilised.

The electro-mechanical finite element equations are derived from the piezoelectricity constitutive relations

and the equations of mechanical and electrical equilibrium [2]. Applying the formalism and adding or assembling the local equations for all elements in the model yield the global system of ODEs:

$$M_{uu} \frac{d^2 u}{dt^2} + C_{uu} \frac{du}{dt} + C_{u\psi} \frac{d\psi}{dt} + K_{uu} u + K_{u\phi} = F \quad (1)$$

$$K_{u\phi}^T u - K_{\phi\phi} \phi = Q \quad (2)$$

$$M_{\psi\psi} \frac{d^2 \psi}{dt^2} - C_{u\psi}^T \frac{du}{dt} + C_{\psi\psi} \frac{d\psi}{dt} + K_{\psi\psi} \psi = 0 \quad (3)$$

governing elastic (1), electric (2), and acoustic (3) fields. Note that the mechanical Equations, (1) and (3), are dynamic while the electric Equation, (2), is quasi-static [2]. The global unknowns, u , ϕ and ψ are, respectively, the elastic displacement vector, the electric potential vector, and the velocity potential vector, while F is the applied force vector and Q is the charge vector. These vectors are defined by field values at all nodes in the model. Coefficients M , C , and K denote the various uncoupled and coupled mass, damping, and stiffness matrices, respectively. To solve these ODEs it is necessary to make assumptions about the temporal behaviour of the electro-mechanical phenomena. Frequency-domain solutions assume time-harmonic behaviour, effectively removing time as an independent variable. Time-domain solutions assume general temporal evolution of the system, requiring step-by-step integration of the equations. Integration can be done using either an *implicit* or an *explicit* method [3].

For frequency-domain analysis of Equations (1)-(3) the unknowns become $u = \hat{u}e^{i\omega t}$, $\psi = \hat{\psi}e^{i\omega t}$, and $\phi = \hat{\phi}e^{i\omega t}$, which yields three inhomogeneous systems of implicit equations for, \hat{u} , $\hat{\psi}$, and $\hat{\phi}$. Direct solution by Gaussian elimination is only practical in 2D because 3D typically leads to prohibitively large system bandwidth and memory needs. The alternative is an iterative solution. If the system is symmetric and positive definite (positive eigenvalues) then the conjugate gradient (CG) method is appropriate. In practice, material attenuation and radiation boundary conditions make the system of equations complex, non-hermitian, and typically indefinite, requiring a more general iterative solver, like GMRES [4] or the new, more robust QMR algorithm [5].

When transient signals are of principal interest, the most direct solution method is step-by-step integration in time. There are many ways to evaluate the current solution from known results at previous time

steps. Implicit methods couple the current solution vector, hence, the global system of equations must be solved at each time step. Their advantage is unconditional stability with respect to time step. By contrast, explicit methods decouple the current solution vector and eliminate the global system solve, but they are only conditionally stable, that is, there is a time step limit (CFL condition, [6]) above which the method is unstable. The caveat for implicit integration of wave phenomena is that solution accuracy requires a time step smaller than one-tenth the period of the highest frequency to be resolved. This is close to the CFL stability limit for explicit methods and effectively removes the principal advantage of implicit integration.

Explicit integration of (1) and (3) involves diagonalising the uncoupled mass and damping matrices, M_{uu} , $M_{\psi\psi}$, C_{uu} , $C_{\psi\psi}$, using nodal lumping, replacing the time derivatives with finite differences, and integrating using a central difference scheme (2nd order accurate). For stability the time step Δt must be smaller than the shortest wave transit time across any element (CFL condition). This follows from the hyperbolic (wave) nature of the original PDEs, that is, during a time step the field at a point is only influenced by the field at neighbouring points within a sphere of radius $\Delta x = c_p \Delta t$ where c_p is the fastest local wave speed. Therefore, for $\Delta t < \Delta x / c_p$ nodal fields are decoupled during a single time step and can be integrated independently.

The point is that it is possible to eliminate the manipulation and solution of large systems of electro-mechanical equations by integrating Equations (1) and (3) *explicitly* in time for u and ψ using ϕ from the previous time step, and then solving (2) *implicitly* for the new ψ from u using a preconditioned CG iteration (diagonal scaling). Thus, the algorithm operates on an element-by-element basis, where elemental contributions are accumulated in intermediate global vectors, for example, the nodal force or charge vectors, and the algorithm processes these vectors only. This is equivalent to matrix-vector arithmetic without actually forming the matrix, which facilitates vectorisation and parallelisation. The aspect of this algorithm which this paper focuses on is the *implicit* solve on the electric equation, the so-called ‘‘Electric Solve’’ routines.

2.1 The PZFlex code

In the Electric Solve routines, the electrostatic linear system comes from a structured finite element grid that is logically rectangular in two dimensions, or a cuboid in three. Physically, the elements can be skewed and non-equally spaced, so the coefficients

vary. The matrix is symmetric, positive definite and has nine non-zeros per row in two dimensions or 27 non-zeros per row in three. Because of this behaviour, the electrostatic system is amenable to solution by an iterative method such as conjugate gradients, or by a direct approach.

Our aims in this work were to implement a parallel version of an iterative solver, to introduce an effective direct solver tailored for symmetric positive-definite matrices, and to compare the performance of both. We chose to implement (1) a parallel preconditioned conjugate gradients (PCG) iterative solver, and (2) a version of the direct multi-frontal solver MFACT.

PCG is a standard iterative solver. We chose an implementation designed for maximum parallel efficiency in the key matrix-vector product routines. We also investigated a range of preconditioners, both in terms of improved solution performance and amenability to efficient parallelisation. Our findings on preconditioners are discussed in Section 4.2.

Multi-frontal direct solvers have undergone a renaissance in research effort in recent years, with a large focus being on ordering heuristics within the symbolic and numeric factorisation stages. Some current examples include UMFPACK [7], SuperLU [8] and MA38 from AEA Technology plc's HSL library [9]. These solvers are generally complex pieces of software with many years of development effort invested in them.

MFACT is another example for sparse, symmetric, positive definite systems, written by Raghavan and Ng [10]. It uses an ordering heuristic called modified multiple minimum degree (MMMD) [11] which offers significant improvements in factorisation speed. We used the public domain serial version of this library as a base and re-engineered it for *PZFlex*.

Section 5 offers a performance comparison of these two solver approaches for a range of piezoelectric models.

3 A rapid parallelisation approach: OpenMP

The parallelisation approach we took was to make use of the OpenMP paradigm for shared memory programming. OpenMP is based on a combination of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. Symmetric multiprocessor (SMP) computing resources are becoming increasingly popular, and OpenMP offers

a straightforward way to exploit these resources with minimum effort.

The main technique used to parallelise code in OpenMP are the compiler directives. The directives are added to the source code as an indicator to the compiler of the presence of a region to be executed in parallel, along with some instruction on how that region is to be parallelised. These directives facilitate one of the more important features of OpenMP, that is, *incremental* parallelism. This allows the developer to target the areas of a code which are the most time-consuming without having to worry about the parallelisation of the less compute-intensive sections. This is especially useful for the speedy parallelisation of existing serial programs, for if a message-passing paradigm such as MPI were implemented a major rewrite of the code would be required, whereas with OpenMP a couple of directives intelligently placed can often demonstrate significant parallel speedup. This also maintains the portability of the code in that if, say, it was required to run on a serial machine then the OpenMP directives are simply ignored.

Figure 1 shows a typical OpenMP directive which allows the `do` loop to be executed in parallel. During execution, when the `PARALLEL` directive is encountered a team of threads are created, the number of threads in the team being set by the user. The `DO` directive tells the compiler that the iterations of the loop are to be distributed amongst this team so that they can be executed in parallel. The declarations `PRIVATE` and `SHARED` refer to the scope of the named variables in the parallel region. In this example the loop index `i` is declared as `PRIVATE` which means that each thread executing the parallel region has its own private copy of this variable which cannot be accessed by the other threads. The remaining variables are declared as `SHARED` which means that each thread has access to the same storage area for those variables. At the `END PARALLEL DO` directive the team of threads is destroyed and the code continues on executing sequentially. As shown the compiler directives begin with a comment mark so for a non-OpenMP compliant compiler they are ignored as such and the code executes sequentially.

```
C$OMP PARALLEL
C$OMP DO PRIVATE(i) SHARED(x,y,n)
  do i = 2, n
    x(i) = (y(i)-y(i-1))
  enddo
C$OMP END DO
C$OMP END PARALLEL
```

Figure 1: Simple parallel do loop OpenMP example.

This may seem to be quite a long-winded way of parallelisation but due to the defaults and shortcuts of OpenMP this can be written as one line, as in Figure 2.

```
C$OMP PARALLEL DO
  do i = 2, n
    x(i) = (y(i)-y(i-1))
  enddo
```

Figure 2: Concise OpenMP example.

This demonstrates the conciseness and ease of use of OpenMP for generating parallel versions of serial code at `do` loop level. As well as the `DO` directive there are several others which enable the user to parallelise non-loop code quite straightforwardly. OpenMP offers a portable, scalable, efficient and high-level parallel programming solution for shared memory platforms.

SMP-based desktop and desk-side systems continue to increase in power, giving the modelling engineer an increasingly powerful and easy to use simulation tool, right there in the office. OpenMP allows the engineer to make use of these systems efficiently, porting existing simulation codes in a quick and simple way. This rise in “high performance computing on the desktop” is an area that EPCC, as a technology transfer organisation, is especially keen to promote.

4 Implementation of PCG in *PZFlex*

4.1 The *PZFlex* serial iterative solver

Currently, *PZFlex* users can only specify one iterative method to solve the Electric Solve linear matrix problem. This is the conjugate gradient (CG) method with the choice of either Direct Scaling (DSCG) or Incomplete Cholesky factorisation (ICCG) as the coefficient matrix preconditioner. The specific solver routine used is derived from the Sparse Linear Algebra Package (SLAP) mathematical software library [13] which is written in Fortran 77. This requires *PZFlex* to convert the coefficient matrix, A , from a general sparse matrix format (i.e. SLAP Triad) to the SLAP Column format. It is assumed that the matrix is always symmetric and therefore only the lower triangle is stored to reduce memory requirements.

The preconditioned CG (PCG) method is mainly composed of a set of matrix-vector and vector-vector multiplications, plus the preconditioner algorithm.

For a typical *PZFlex* example problem, the matrix-vector multiplication accounts for over 90% of the execution time for each CG iteration.

A good OpenMP parallel implementation of the PCG therefore requires a good parallel matrix-vector multiplication implementation and a good parallel preconditioner implementation. This depends ultimately on the interaction between array storage schemes used and the particular algorithms used.

4.2 Investigation of iterative preconditioners for *PZFlex*

There are many other matrix preconditioners other than direct scaling or incomplete Cholesky factorisation. In order to investigate as wide a selection of preconditioners as possible, we used the Nonsymmetric Preconditioned Conjugate Gradient (NSPCG) V1.0 software package [12].

The NSPCG Fortran 77 package provides an ideal modular structure for research on iterative methods. It provides a large range of preconditioners, accelerators and matrix storage schemes. It can be used to evaluate iterative methods for a specific problem and can provide information on the following solution parameters:

- the convergence or nonconvergence of an iterative method;
- the number of iterations required for convergence;
- the existence of a preconditioner (e.g. incomplete factorisations).

Table 1 summarises experiments carried out using the example problem ‘ab11’, a three-dimensional quartz resonator model with 201,465 mesh points and a matrix A of size $2,630,069 \times 2,630,069$. The matrix coefficients are stored using the symmetric compressed diagonal storage scheme as described in Section 4.3. NSPCG was not integrated with *PZFlex* but the relevant data was passed via a file to a driver routine for NSPCG.

NSPCG can be called using a single call statement with parameters such as preconditioner name, accelerator name etc. being passed through the argument list.

Both line and block versions of the above preconditioners were also tested but gave significantly worse results than their corresponding point equivalents in the above table.

Preconditioner	Time (s)	Its.
JAC (no scaling, no adapting)	38.01	24
JAC (scaling, no adapting)	38.01	23
JAC (scaling, adapting EMAX)	36.56	23
JAC (scaling, adapting EMIN)	52.97	35
SSOR (no scaling, adapting ω)	48.68	15
SSOR (scaling, adapting ω)	50.45	15
IC (scaling)	45.79	12
IC (no scaling)	43.49	12
MIC (scaling)	52.95	14
MIC (no scaling)	43.86	11
LSP (scaling, degree 1)	52.76	17
LSP (scaling, degree 2)	63.69	14
LSP (scaling, degree 4)	75.97	10
LSP (scaling, degree 8)	86.75	6
NEU (no scaling, degree 1)	44.42	15
NEU (no scaling, degree 2)	56.26	13
NEU (no scaling, degree 4)	71.84	10

Table 1: NSPCG preconditioner results for example problem ‘ab11’.

The best preconditioner is the Jacobi preconditioning with EMAX adapting which produced a solution time of 36.6 seconds. The incomplete Cholesky preconditioner reduced the number of iterations to convergence by almost 50% but with an increase in overall solution time by about 25%. Again, SSOR, MIC, LSP and NEU preconditioning reduced the number of iterations required for convergence but overall time also increased.

With regards to parallel preconditioners, Jacobi is trivial to parallelise whereas incomplete Cholesky is not straightforward. An alternative to a parallel incomplete Cholesky may be a parallel Neumann polynomial. A Neumann polynomial converts the matrix inversion to series of matrix-matrix multiplications. From Table 1 it seems that their serial performance is very comparable.

4.3 Parallel implementation details

As noted above, the key to an efficient parallel conjugate gradient solver is an efficient matrix-vector multiplication operation since this forms the bulk of the computational requirement for the algorithm. There are two loop nests within the matrix-vector product which, for maximum efficiency, must both be fully parallelised using OpenMP directives. Unfortunately, neither the SLAP Triad nor SLAP Column matrix storage schemes employed in *PZFlex* allow this to be done.

Thus, to obtain better matrix-vector multiplication

parallel efficiency it is necessary to use a different co-efficient storage scheme. The storage scheme chosen should remove the need for indirect addressing by having some sort of predictive structure but also it should minimise any extra memory requirements.

Compressed diagonal storage (CDS) fulfils the above two criteria and is particularly useful for a matrix that arises from Finite Element (FE) or Finite Difference (FD) discretisations. Typically, the matrix A is banded with a bandwidth that does not vary greatly from row to row. This scheme stores sub-diagonals of the matrix in consecutive columns.

The banded matrix A will have nonnegative constants p, q called the left and right *halfbandwidth*, such that $A_{i,j} \neq 0$ only if $i - p \leq j \leq i + q$. Thus, we can allocate for matrix A a new storage array, $ANEW(1:N, -P:Q)$, where N is the number of rows in A . Note, that if A is symmetric (i.e. $p = q$) then the new storage array becomes $ANEW(1:N, 0:P)$.

Consider the symmetric matrix A defined below.

$$A = \begin{pmatrix} 11 & 21 & 31 & 41 & 51 \\ 21 & 22 & 32 & 42 & 52 \\ 31 & 32 & 33 & 43 & 53 \\ 41 & 42 & 43 & 44 & 54 \\ 51 & 52 & 53 & 54 & 55 \end{pmatrix}$$

This matrix can be stored in the symmetric CDS format in an array with the dimensions $ANEW(5, 0:4)$ by using the mapping $ANEW(I, J) = A_{i,i+j}$ where $j \geq 0$ as shown in Table 2.

Row	Columns 0:4				
1	11	21	31	41	51
2	22	32	42	52	0
3	33	43	53	0	0
4	44	54	0	0	0
5	55	0	0	0	0

Table 2: Symmetric storage array for matrix A .

Note that several zeros have to be stored which is typical of banded storage even for the dense example matrix A . For matrices formed by Finite Element discretisations it is possible that within the half-bandwidths that there will be diagonals with no nonzero values. In order to reduce the need to store such diagonals, the symmetric CDS can be augmented by a one-dimensional stencil array, $STEN(1:MXNRE)$, that stores the offset from the main diagonal. $MXNRE$ is the number of non-main diagonals in the original matrix A which have at least one nonzero array element. Thus, for the dense example:

$$\text{STEN} = (1 \ 2 \ 3 \ 4)$$

This augmented symmetric CDS method has been implemented in the parallel version of the iterative CG solver.

4.4 PCG Performance Results

In testing the performance of the new CDS-based parallel PCG routines, we performed a series of timings on two test cases: ‘array2d’, a small two-dimensional transducer model running one timestep and ‘combo3dc’, an approximation of a three-dimensional transducer array.

Our timing tests were performed on a Sun HPC 3500 system, an SMP system with eight 400MHz UltraSPARC-II processors and 8 Gbyte of shared memory. For compilation we used Kuck and Associates Guide f90 OpenMP system, with high levels of optimisation (the Fortran 90 compiler options `-fast -xchip=ultra2 -xarch=v8plusa`). In both cases we timed only the parallel conjugate gradient routine.

Table 3 shows the times for the parallel CG algorithm for both test problems. In the ‘array2d’ case, the times are the averages for ten calls to the PCG solver; in the ‘combo3dc’ case the averages are over five calls. In this table, N_p is the number of processors used and we define simple speedup $S_p = T_1/T_p$ and simple parallel efficiency $E_p = S_p/N_p$, where T_1 is the single-processor time and T_p the time on N_p processors.

The timing data for these two cases are plotted in Figures 3 and 4.

‘array2d’ times			
N_p	time (s)	S_p	E_p
1	4.58 ± 0.04	–	100%
2	3.37 ± 0.04	1.4	70.0%
4	2.52 ± 0.05	1.8	45.0%
6	2.33 ± 0.10	2.0	33.3%
8	2.28 ± 0.11	2.0	25.0%
‘combo3dc’ times			
N_p	time (s)	S_p	E_p
1	267.5 ± 0.5	–	100%
2	166.9 ± 2.1	1.6	80.0%
4	106.4 ± 0.3	2.5	62.5%
6	88.3 ± 0.5	3.0	50.0%
8	82.1 ± 0.3	3.3	41.2%

Table 3: Parallel PCG times for *PZFlex* running test cases ‘array2d’ and ‘combo3dc’.

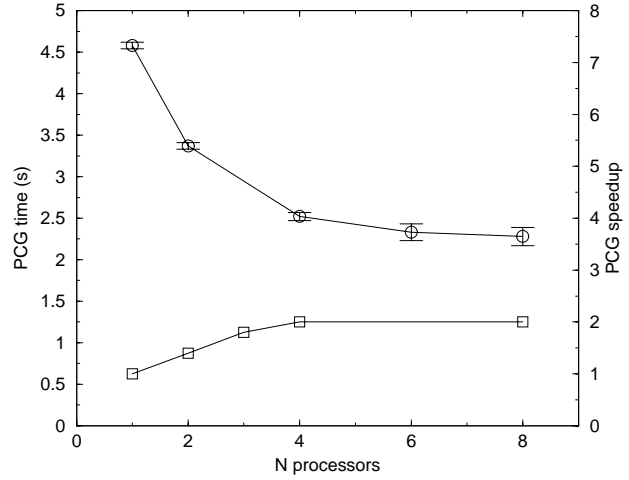


Figure 3: PCG timings for ‘array2d’ on a Sun E3500.

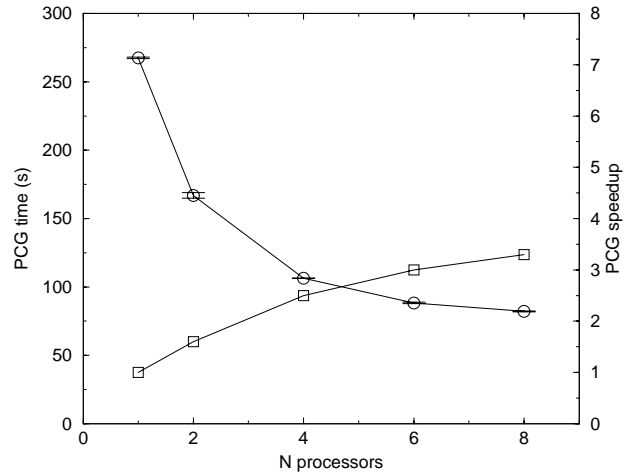


Figure 4: PCG timings for ‘combo3dc’ on a Sun E3500.

As might be expected, the speedup for the smaller test problem is not particularly impressive, especially with call times of the order of a few seconds. The ‘array2d’ case has a matrix size of $34,136 \times 34,136$ with 169,521 nonzero elements. The ‘combo3dc’ case is around an order of magnitude larger in terms of physical mesh points, yielding a matrix of $381,024 \times 381,024$ with 5,146,012 nonzero elements; this increase in computational workload allows the multi-threading parallelisation to demonstrate greater efficiency and a reasonable speedup of 2.5 over 4 processors.

5 Performance Comparison: PCG vs. MFACT

A key aim of this work was a direct comparison between the simple OpenMP version of PCG and the more complex MFACT. The timing tests for the MFACT solver were performed on the same system, using the same compilers and options, as the previous PCG tests. Two test cases were chosen: ‘array2d’ again and ‘combo3da’, a smaller version of ‘combo3dc’ with a matrix of size $63,504 \times 63,504$ and 824,632 non-zeros. In both cases we ran MFACT against the parallel CG solver for a direct versus iterative comparison.

Table 4 shows the times for serial runs of the MFACT solver against PCG for the two cases. For ‘array2d’ the total MFACT time beats the PCG algorithm on both one and two processors, albeit by a small margin, being pipped by the four processor run. For ‘combo3da’ the situation is heavily reversed, with a large symbolic/numeric factorisation time slowing MFACT right down. However, the factorisation time is a one-off cost for MFACT; once it is done and the information cached, subsequent “solve” calls avoid this overhead. In the case of *PZFlex* the electrostatic matrix A is quasi-static, so a typical run will involve many calls to the solver for one factorisation call.

We show a rough model of this in Figure 5. Here we plot a linear projection of total time against number of iterations for the ‘combo3da’ results for MFACT and PCG (one, two and four processors). In this plot, MFACT very quickly overhauls the one-processor PCG case at around 200 iterations, but only catches the two processor case at around 2,000. The four-processor PCG run ultimately scales better than MFACT.

Problem	MFACT times (s)			PCG times (s)		
	Factor	Solve	Total	$N_p=1$	$N_p=2$	$N_p=4$
‘array2d’	3.0	0.21	3.21	4.58	3.37	2.52
‘combo3da’	240.4	1.56	242.0	2.69	1.67	1.15

Table 4: Timings for MFACT and PCG for the ‘array2d’ and ‘combo3da’ test cases.

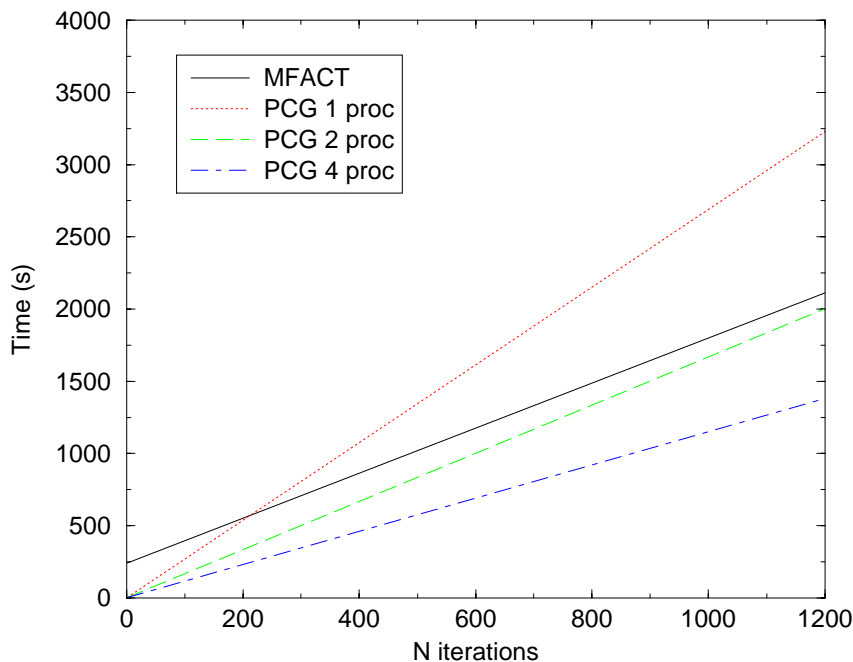


Figure 5: Projected scaling of solver times against number of calls to solver routines.

6 Conclusions

This work has performed a fairly comprehensive review and implementation of options for the acceleration of the performance of the electric-window solver routines within the *PZFlex* piezo-electric modelling code. The key conclusions can be summarised as follows:

- an OpenMP implementation of the parallel conjugate gradient (PCG) algorithm with simple diagonal scaling offers good parallel scalability provided the necessary steps are taken to optimise the matrix storage scheme;
- the simple parallel PCG on small numbers of processors compares very well with the more complex MFACT direct solver;
- parallel efficiencies of up to 80% are achievable under OpenMP for a real industrial application on low-cost hardware.

References

- [1] Wojcik, G., Vaughan, D., Abboud, N., Mould, J., “Electromechanical modelling using explicit time-domain finite elements”, *Proc. IEEE Ultrasonics Symposium*, **2**, 1107–1112, 1993.
- [2] Auld, B.A., “Acoustic Fields and Waves in Solids”, Vol 1, 2nd edition, Krieger, 1990.
- [3] Richtmeyer, R.D., Morton, K.W., “Difference Methods for Initial-Value Problems”, Interscience Publishers, 1957.
- [4] Burkhart, R.H., Young, D.P., “GMRES acceleration and optimization codes”, ETA-TR-88, Boeing Computer Services, May 1988.
- [5] Freund, R.W., “Conjugate gradient type methods for linear systems with complex symmetric coefficient matrices”, *SIAM J. Sci. Stat. Computing*, Vol 13, No.1, 1992.
- [6] Courant, R., Friedrichs, K.O., Lewy, H., “Über die partiellen differenzgleichungen der mathematischen physik”, *Math. Ann.*, 100, 32, 1928.
- [7] Davis, T.A., Duff, I.S., “A combined unifrontal/multifrontal method for unsymmetric sparse matrices”, *ACM Transactions on Mathematical Software*, vol. 25, no. 1, pp 1-19, 1999.
- [8] Demmel, J., Eisenstat, S., Gilbert, J., Li, X., and Liu, J., “A Supernodal Approach to Sparse Partial Pivoting”, to appear in *SIAM J. Mat. Anal. Appl.*
- [9] Duff, I.S., *et al*, “Harwell Subroutine Library”, <http://www.cse.clrc.ac.uk/Activity/HSL/>.
- [10] Raghavan, P., “MFACT: A Multifrontal Sparse Direct Solver”, 1997, <http://www.cs.utk.edu/~padma/mfact.html>.
- [11] Ng, E.G. and Raghavan, P., “Performance of Greedy Ordering Heuristics for Sparse Cholesky Factorization”, *SIAM Journal of Matrix Analysis and Applications*, Vol 20, No 4, pp 902–914, 1999.
- [12] Oppe, T.C., Joubert, W.D., Kincaid, D.R., “NSPCG: A Package for Solving Large Sparse Linear Systems by Various Iterative Methods”, Center for Numerical Analysis, The University of Texas at Austin. Available from the Netlib repository, <http://www.netlib.org/itpack/>.
- [13] Greenbaum, A., Seager, M.K., *et al*, “The Sparse Linear Algebra Package”, Netlib repository, <http://www.netlib.org/slap/>.