Evaluation of MATPOWER and OpenDSS load flow calculations in power systems using parallel computing

Gerardo Guerra, Juan A. Martinez-Velasco

Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: gerardo.guerra@upc.edu

Published in *The Journal of Engineering*; Received on 27th January 2017; Accepted on 21st March 2017

Abstract: This study presents the work carried out by the authors to apply Intel MKL PARDISO (a parallel sparse matrix solver) to the load flow solution algorithms of MATPOWER and OpenDSS. The goal is to explore the potential execution time reduction obtained when working with large power systems and multi-core installations. Test systems of different sizes were solved in order to observe the time reduction as function of the system size and the number of cores used in the parallel execution. Results show that except for the full Newton–Raphson algorithm, one should not expect a reduction in execution times when using a parallel routine. The use of a parallel sparse matrix solver is only justified when the sparse matrix must be recalculated at every step of the solution process or the systems under study are much larger than those analysed here. This study also presents how a parallel computing solution can be implemented in different applications by using available high-performance parallel libraries.

1 Introduction

Parallel computing has become a cost-effective solution for large and data-intensive problems when the objective is to reduce total execution times and utilise larger memory/storage resources [1–3]. Recently multi-core processors in personal computers and workstations have become a standard feature, while easier access to multi-core equipment has facilitated the development and improvement of parallel-oriented software tools. The progress in both areas (hardware and software) has pushed forward the growth of parallel computing applications.

Parallel computing has had a significant impact on a variety of fields ranging from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing [3]. The use of multi-core computing in studies related to power systems is a natural approach given the capabilities of the software tools used in the simulation of power systems and the nature of the analysed problems. In recent years many important works have been conducted on the application of parallel computing to power systems analysis; see [4–9].

Access to powerful hardware and software applications has led to a continuous effort to analyse and solve large and realistic systems. In [10] a test system with 87,263 network nodes and 674,027 network devices was used to test an industrial-grade data translator for the interfacing of power-flow programs with EMTP-type programs. Individual executions (e.g. snapshot load flow) do not represent a major concern, even when solving systems with so many elements; it is the time-domain or time-driven execution of such systems that may result in too high execution times. Moreover, optimisation or parametric studies can require performing a large number of executions, which can lead to prohibitive simulation times.

The introduction of parallel computing into the simulation of power systems can be carried out assuming two different perspectives:

- In data-parallel execution, system solution is concurrently run on multiple cores [3], where each core works on a specific part of the problem. This approach requires partitioning the analysed system into several sub-problems according to the available number of cores; each sub-problem is solved with a different core but the required information is exchanged among working cores. One example of data-parallel execution can be found in real-time systems [11], where test systems are decoupled into two or more groups that are simultaneously solved in parallel cores; an efficient implementation has been proposed in [12], an algorithm similar to that proposed in [13].
- Task-parallel execution is an adequate approach when the same system is required to be simulated many times with only small variations in certain input parameters (e.g. optimisation or parametric studies) [3]. Under these circumstances the total number of system executions is distributed among available cores, where they are solved independently from the executions performed in other cores. The authors of [14] presented a parallel Monte–Carlo method for the optimum allocation of distributed generation.

The main objective of this paper is to apply the use of parallel sparse matrix solvers in the solution of load flow problems and to assess its impact on the reduction of execution times for large power systems when working with a multi-core environment. A second objective is aimed at proposing a practical rule that could allow users to determine the optimal number of cores (to be used in the parallel computation) based on the size of the system under study. A final objective is to show how parallel computing can be applied to engineering problems by means of high-performance parallel libraries without the need of being an expert programmer or having a deep knowledge of parallel algorithms.

Different works on parallel load flow algorithms with central processing units have been presented to date; see, for instance, [15–19]. In recent years the focus has been on using graphical processing units (GPU) [20–23]; most of these works present custom-made applications that seek to exploit parallelism in all stages of a load flow algorithm (i.e. admittance matrix formation, solution of linear system, error calculation). Although this approach may lead to more efficient algorithms, it also requires advanced programming skills and a high level of expertise in mathematical algorithms.

Many software tools are currently available for analysing power systems; although commercial packages are widely used, open-source software has become a valid option for companies and researchers given the powerful capabilities and features present in some of them (see [24]). In addition, there are also a great number of available libraries for implementing high-performance

doi: 10.1049/joe.2017.0023

applications that exploit the resources of multi-core processors (e.g. [25–28]). MATPOWER [29] and OpenDSS [30] are the tools selected in this paper for the simulation of power systems, while Intel MKL [25] is the library chosen for the parallel solution. Simulation control and data gathering was carried out using MATLAB and Visual Studio.

The paper has been organised as follows. Section 2 summarises the main features of Intel MKL PARDISO, the parallel sparse matrix solver used in this work, and how it can be applied to the solution of load flow algorithms. Section 3 details the implementation of Intel MKL PARDISO in MATPOWER and OpenDSS; the section includes a description of the test systems used with each tool, and a summary of results. A discussion about alternatives to the solution considered in this work is presented in Section 4, while the main conclusions are summarised in Section 5.

2 Application of a parallel sparse matrix solver to the load flow solution

2.1 Intel MKL and Intel MKL PARDISO

The Intel Math Kernel Library is a package of highly optimised and extensively threaded routines for high-performance applications [25]. Optimised for Intel processors, Intel MKL provides FORTRAN and C/C++ programming interfaces, as well as a DLL than can be used for software redistribution. Capabilities included in the library cover areas such as linear algebra routines, distributed processing linear algebra routines, sparse solver routines, fast Fourier transform functions, vector mathematics routines, data fitting library, and eigensolvers.

Intel MKL PARDISO is a direct sparse solver routine based on the PARDISO solver [31], a high-performance software package for solving large sparse linear systems of equations on shared-memory multiprocessors [32]. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques, while Level-3 BLAS update and pipelining parallelism are used with a combination of left- and right-looking supernode techniques for improving sequential and parallel sparse numerical factorisation performance [32]. It can cope with a variety of sparse matrices, including real and complex, symmetric and non-symmetric, positive definite and indefinite sparse linear systems of equations. For sufficiently large-size problems, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

The execution of Intel MKL PARDISO can be divided into the following phases: (i) fill-reduction analysis and symbolic factorisation; (ii) numerical factorisation; and (iii) forward and backward solve. These stages can be executed in one sequence or individually at different times.

2.2 Load flow solution using a parallel solver

Sparse matrix solvers are optimised routines that seek to exploit the sparsity of linear systems represented by the following equation

$$[\boldsymbol{b}] = [\boldsymbol{A}][\boldsymbol{x}] \tag{1}$$

where $\boldsymbol{A}$ is a sparse matrix, and $\boldsymbol{b}$ and $\boldsymbol{x}$ are vectors. Furthermore, parallel sparse matrix solvers take advantage of the resources present in multi-core computers in order to speed up calculations.

The form in (1) can also be recognised in the iterative solution of the full Newton–Raphson (NR) [33], fast decoupled NR [34], and default OpenDSS [30] load flow algorithms; therefore, the application of the Intel MKL PARDISO to the solution of the aforementioned algorithms is straightforward.

One important aspect of the fast decoupled NR and default OpenDSS algorithms is the use of constant matrices for the iterative solution; in such cases constant matrices must be calculated and triangulated only once. Thanks to Intel MKL PARDISO's capability of dividing the solution process in different stages (analysis, numerical factorisation, and solve), the iterative scheme remains unchanged and the solution process can be carried out using the triangulated matrix.

Intel MKL PARDISO requires that the sparse matrix $\boldsymbol{A}$ be defined using the compressed sparse row (CSR) format [32]. Additional operations are necessary for obtaining the three-vector representation of the admittance matrix; however, the overhead produced by these operations does not represent a significant loss of performance.

3 Implementation and testing of the parallel solver

3.1 Application with MATPOWER

MATPOWER is an open-source software package that allows users to perform load flow (AC and DC) and optimal power flow calculations for power systems [29]. It has been implemented in MATLAB and consists of a group of *m*-files, which grants a great flexibility and allows the user to modify and improve the existing code. For AC load flow calculations, the user can choose among several solution algorithms (full NR, fast decoupled NR, and Gauss-Seidel). System information is provided in form of matrices that contain the system parameters (i.e. parameters of branches and generators, as well as buses) [35].

The codes implemented in this work to run MATPOWER *m*-files when using the Intel MKL PARDISO routine in the full NR and fast decoupled NR solution algorithms are shown in the Appendix.

The power system selected in this work for testing the performance of MATPOWER is the system referred to as *9241Pegase*, which is included among the cases provided with this tool. The network accurately represents the size and complexity of the European high-voltage (HV) transmission network; it contains 9241 buses, 1445 generators, and 16,049 branches. The operating voltages of this system are 750, 400, 380, 330, 220, 154, 150, 120, and 110 kV [36].

Although the *9241Pegase* system provides a valid test case for assessing the potential time reduction achieved with the introduction of parallel sparse matrix solvers, additional tests with systems of larger sizes are also required. However, finding or developing large size test systems (i.e. with several thousand buses) is not an easy task; therefore, it was decided to create a set of new test systems by simply replicating the *9241Pegase* case as many times as it was desired. As a result, the sizes of the new test systems will be 2, 10, and 20 times that of the *9241Pegase* case.

Tables 1–4 show the results obtained for the different test systems; these tables compare the execution times with different number of cores (threads) and also present the solution times found when using MATLAB capabilities to solve the load flow algorithms. Note that the results focus on execution times and not solution values; however, the number of required iterations for problem solution have been included to show that the use of Intel MKL PARDISO does not affect algorithm convergence. Execution times were measured by comparing the processor's clock before and after evaluating a function.

MATPOWER test systems were solved for nominal load (i.e. snapshot load flow). The results shown in Tables 1 and 2 were obtained using a laptop computer with an Intel Core i7-3630QM processor (4 Cores, Clock frequency =2.4–3.4 GHz), 8 GM RAM, and Windows 8 OS, whereas results shown in Tables 3 and 4 were obtained using a high-performance computing (HPC) server with 2 Intel Xeon E5-2660 processors (8 Cores, Clock frequency =2.2–3.0 GHz), 128 GB RAM, and Ubuntu OS (running a Windows 7 Professional Virtual Machine).

Figs. 1 and 2 depict the time reduction with different number of threads when comparing the performance of Intel MKL PARDISO with respect to MATLAB capabilities. Note that a negative *Time Reduction* can result and it indicates an increment in execution time.

Table 1 Execution times for full NR's method – Laptop computer

	Intel MKL PARDISO			MATLAB
		Case 9241Pegase		
iterations	6	6	6	6
no. of threads	1	2	4	—
total time	0.5109	0.4040	0.3660	0.4949
CSR format	0.0040	0.0040	0.0040	—
iteration solver	0.0689	0.0500	0.0430	0.0700
		Case 9241Pegase × 2		
iterations	6	6	6	6
no. of threads	1	2	4	—
total time	1.0470	0.8290	0.7210	1.0089
CSR format	0.0100	0.0100	0.0100	—
iteration solver	0.1400	0.1030	0.0840	0.1430
		Case 9241Pegase × 10		
iterations	6	6	6	6
no. of threads	1	2	4	—
total time	5.9640	4.7350	4.0370	5.6780
CSR format	0.0510	0.0510	0.0510	—
iteration solver	0.7930	0.5890	0.4680	0.7819
		Case 9241Pegase × 20		
iterations	6	6	6	6
no. of threads	1	2	4	—
total time	12.750	10.090	8.5310	11.343
CSR format	0.1010	0.1010	0.1010	—
iteration solver	1.6900	1.2439	0.9820	1.5509

Table 2 Execution times for fast decoupled NR's method – Laptop computer

	Intel MKL PARDISO			MATLAB
		Case 9241Pegase		
P iterations	14	14	14	14
Q iterations	13	13	13	13
no. of threads	1	2	4	—
total time	0.1150	0.0940	0.0860	0.0890
factorisation	0.0599	0.0419	0.0330	0.0419
CSR format	0.0030	0.0030	0.0030	—
		Case 9241Pegase × 2		
P iterations	14	14	14	14
Q iterations	13	13	13	13
no. of threads	1	2	4	—
total time	0.2220	0.1769	0.1689	0.1759
factorisation	0.1250	0.0849	0.0629	0.0869
CSR format	0.0060	0.0060	0.0060	—
		Case 9241Pegase × 10		
P iterations	14	14	14	14
Q iterations	13	13	13	13
no. of threads	1	2	4	—
total time	1.1459	0.9329	0.8020	0.7849
factorisation	0.7189	0.4629	0.3330	0.4280
CSR format	0.0260	0.0260	0.0260	—
		Case 9241Pegase × 20		
P iterations	14	14	14	14
Q iterations	13	13	13	13
no. of threads	1	2	4	—
total time	2.5580	2.1200	1.7960	1.8699
factorisation	1.5320	0.9980	0.7179	0.8990
CSR format	0.0540	0.0540	0.0540	—

From the results found with the fast decoupled NR method it can be seen that matrix factorisation (i.e. analysis and numerical factorisation stages) is the most computationally expensive operation; depending on the number of threads this time may represent between 25 and 60% of the total execution time. One can observe that individual execution times for other operations are very short (note that the iterative scheme performs 14 P-iterations and 13 Q-iterations) and that the time spent performing P and Q iterations can increase with the number of threads used in the parallelisation.

A better performance is achieved for the full NR method given that Intel MKL PARDISO uses parallel routines for both matrix factorisation and the solution of linear equation systems. Moreover, since the sparse Jacobian matrix used in this method is considerably larger than the constant matrices used in the fast decoupled method, multithreaded routines have a greater effect on larger matrices.

The curves presented in Fig. 1 show a saturation effect: they exhibit a rapid initial growth but there is no significant increment in time reduction when Intel MKL PARDISO uses more than 8 threads. Furthermore, it can be seen that the solution process suffers a decrease in performance for the smallest system (*9241Pegase*). The theoretical speed-up that can be achieved through parallel computing is defined by the Amdahl's law [37–39], which is a function of the code's fraction that can be parallelised and the number of threads used in the parallelisation. Thus for a given algorithm, there exists a number of threads that produces a maximum speed-up. Moreover, according to Fig. 1 the real speed-up is also dependent on the size of the solved problem. The saturation effect, which is also a consequence of Amdahl's law, will allow determining the optimal number of threads that must be used in the parallelisation of the load flow algorithms; therefore, the optimal number of threads will be equal to the number of threads where curve saturation is produced.

The results shown in Fig. 2 present important differences between the Laptop computer and the HPC server; however, if the results from the HPC server are taken as a reference, it can be concluded that the maximum time reduction will be achieved with 8 threads.

3.2 *Application with OpenDSS*

OpenDSS is an open-source simulator for analysis of electric utility distribution systems [30], implemented as both a stand-alone executable program and a COM DLL that can be driven from some software platforms. The executable version adds a basic user interface to the solution engine to assist users in developing scripts and viewing solutions. Modelling and calculation capabilities of this tool allow users to represent the most important distribution components and perform studies considering both deterministic and probabilistic calculations. Built-in solution capabilities include snapshot and time-mode power flow, harmonics, fault current study, dynamics, parametric and probabilistic studies [40]. OpenDSS can be used for planning and analysis of multi-phase distribution systems, analysis of distributed generation interconnection, annual simulations, storage modelling and analysis, and other studies.

OpenDSS uses the single-core routine KLUSolve for the solution of the resulting sparse system [41]. The built-in KLUSolve functions only return the Compressed Sparse Column (CSC) format [32]; this situation does not pose a problem if the correct parameters are used for matrix factorisation.

The code added to generate the CSC format of the admittance matrix and perform matrix factorisation in OpenDSS is shown in the Appendix; additionally, it presents the changes introduced to the source code when using the Intel MKL PARDISO routine.

Table 5 presents the main characteristics of the three test systems developed for testing the parallelised OpenDSS. They are 50 Hz overhead systems and consist of a simplified representation of the HV system, a Medium-voltage (MV) network, and low-voltage (LV) networks served from the secondary side of distribution MV/LV transformers. The LV networks are based on the low voltage test feeder shown in Fig. 3 and provided with OpenDSS. The configuration of the three test systems is shown in Fig. 4. LV loads

Table 3 Execution times for full NR's method – HPC server

	Intel MKL PARDISO					MATLAB
			Case 9241Pegase			
iterations	6	6	6	6	6	6
no. of threads	1	4	8	12	16	—
total time	0.9050	0.6710	0.5609	0.5780	0.6400	0.9049
CSR format	0.0150	0.0150	0.0150	0.0150	0.0150	—
iteration solver	0.1100	0.0630	0.0619	0.0620	0.0630	0.1410
			Case 9241Pegase × 2			
iterations	6	6	6	6	6	6
no. of threads	1	4	8	12	16	—
total time	1.7309	1.3729	1.1540	1.1700	1.1700	1.9660
CSR format	0.0310	0.0310	0.0310	0.0310	0.0310	—
iteration solver	0.2190	0.1560	0.1239	0.1250	0.1250	0.2650
			Case 9241Pegase × 10			
iterations	6	6	6	6	6	6
no. of threads	1	4	8	12	16	—
total time	9.7190	7.1609	6.3820	6.3800	6.2870	9.9369
CSR format	0.1410	0.1410	0.1410	0.1410	0.1410	—
iteration solver	1.1860	0.7650	0.6710	0.6699	0.6550	1.3730
			Case 9241Pegase × 20			
iterations	6	6	6	6	6	6
no. of threads	1	4	8	12	16	—
total time	20.945	14.741	13.978	13.820	13.587	20.732
CSR format	0.2809	0.2809	0.2809	0.2809	0.2809	—
iteration solver	2.6360	1.6379	1.4819	1.4510	1.4190	2.6209

Table 4 Execution times for fast decoupled NR's method – HPC server

	Intel MKL PARDISO					MATLAB
			Case 9241Pegase			
P iterations	14	14	14	14	14	14
Q iterations	13	13	13	13	13	13
no. of threads	1	4	8	12	16	—
total time	0.2029	0.1409	0.1400	0.1400	0.1560	0.1560
factorisation	0.1089	0.0470	0.0459	0.0470	0.0620	0.0629
CSR format	0.0045	0.0045	0.0045	0.0045	0.0045	—
			Case 9241Pegase × 2			
P iterations	14	14	14	14	14	14
Q iterations	13	13	13	13	13	13
no. of threads	1	4	8	12	16	—
total time	0.3589	0.2810	0.2339	0.2650	0.2800	0.2660
factorisation	0.2029	0.1250	0.0929	0.1250	0.1250	0.1410
CSR format	0.0160	0.0160	0.0160	0.0160	0.0160	—
			Case 9241Pegase × 10			
P iterations	14	14	14	14	14	14
Q iterations	13	13	13	13	13	13
no. of threads	1	4	8	12	16	—
total time	1.8719	1.4510	1.3580	1.4349	1.8870	1.3099
factorisation	1.1240	0.5920	0.4839	0.5150	0.4840	0.7019
CSR format	0.0620	0.0620	0.0620	0.0620	0.0620	—
			Case 9241Pegase × 20			
P iterations	14	14	14	14	14	14
Q iterations	13	13	13	13	13	13
no. of threads	1	4	8	12	16	—
total time	4.0709	3.1040	2.9640	3.1820	3.4949	2.9799
factorisation	2.4340	1.2329	0.9979	1.0609	0.9360	1.5129
CSR format	0.1240	0.1240	0.1240	0.1240	0.1240	—

are represented by a ZIP model; each part of the load model has been assigned a weighting factor equal to 1/3 for both active and reactive powers and use curve shapes derived with a procedure presented in [42]. It is important to mention that the test system 3 is composed by two systems such as the one shown in Fig. 4*c*.

OpenDSS test systems were simulated for one year using a 1-hour time step; only information related to the number of iterations performed at each step and solution convergence was collected during the simulation in order to avoid unnecessary overhead times. The yearly simulation allows for a better performance assessment of the Intel MKL PARDISO routine, since it can average out the variations that result with individual executions.

Fig. 5 shows the time reduction in 'total solve time' for the three test systems when using different number of threads and the default

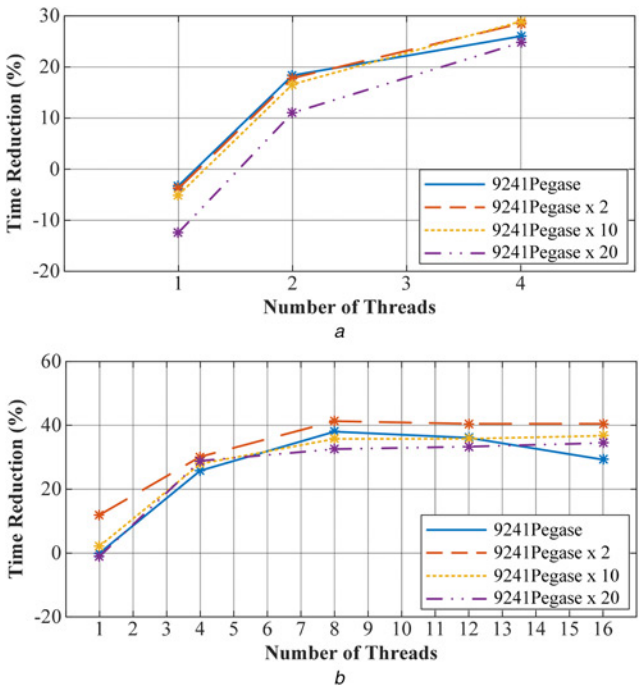

Fig. 1 *Total time reduction – full NR*
a Laptop computer
b HPC server

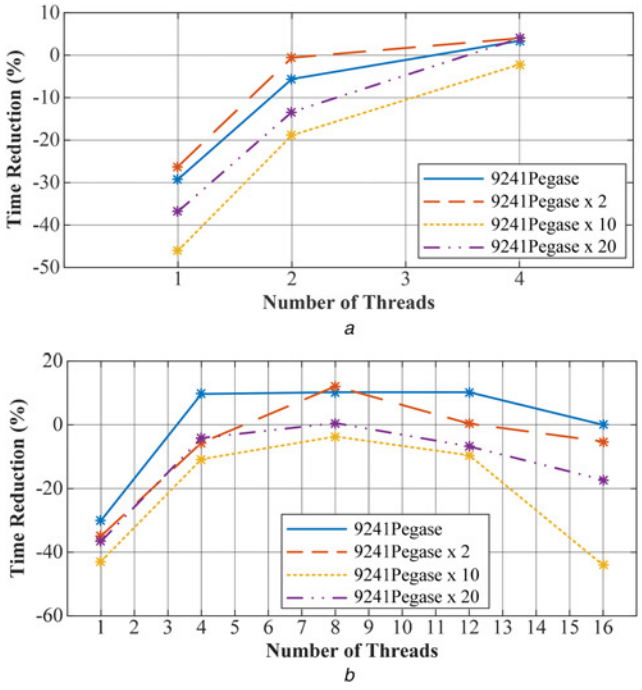

Fig. 2 *Total time reduction – fast decoupled NR*
a Laptop computer
b HPC server

OpenDSS solution algorithm. Although the results show significant differences between the laptop computer and the HPC server, it is clear that the use of the Intel MKL PARDISO has caused an increment in execution times. Moreover, the general trend is that increasing the number of threads leads to a greater performance loss.

It can be noted from these results that for test system 1 the number of threads has very little impact on the performance of the Intel MKL PARDISO routine, whereas for the other two test

Table 5 OpenDSS test systems characteristics

	System 1	System 2	System 3
high-voltage rating	230 kV	230 kV	230 kV
medium-voltage rating	11 kV	11 kV	11 kV
low-voltage rating	0.48 kV	0.48 kV	0.48 kV
rated power substation transformer	1000 kVA	5000 kVA	5000 kVA[a]
number of distribution transformers	8	60	100
total medium-voltage overhead feeder length	9.5 km	35 km	55 km
total low-voltage network line length	11.4 km	85.8 km	143 km
total rated load active power	440 kW	3300 kW	5500 kW
total number of LV loads	440	3300	5500
number of voltage nodes	21,777	163,296	272,139

[a]Test system 3 has two substation transformers of equal rated power.

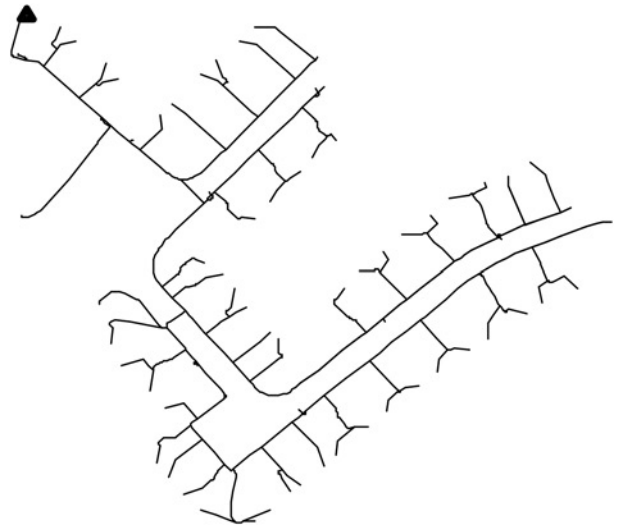

Fig. 3 *Schematic diagram of the LV networks*

systems it depends on the platform (i.e. laptop computer or HPC server). The time reduction curve in Fig. 5*a* shows a clear negative tendency as a function of the number of threads for test systems 2 and 3; a similar behaviour is present in Fig. 5*b* for a number of threads equal or greater than 8. This behaviour is an indication that the algorithm has passed the optimal number of cores and increasing the number of threads will only further decrease the routine's performance.

With a smaller number of threads, it can be assumed that the performance of the Intel MKL PARDISO is almost constant, since the variations in execution times are not significant. A small overhead is caused by the generation of the CSC format of the admittance matrix and performing matrix factorisation; this cannot be avoided since the OpenDSS code relies on KLUSolve to generate and factor the admittance matrix. Although negligible when compared to total simulation times, that overhead is much larger than the 'average individual solve time' (see Tables 6 and 7).

This behaviour indicates that the solve stage of the Intel MKL PARDISO routine is the least expensive in computational terms. As result, in order to obtain any performance gain at the solve stage it will be necessary to work with much larger systems than those presented here.

The differences between the laptop computer and the HPC server could be caused by the behaviour of the virtual machine used in the HPC server and to the fact that Intel's hyper-threading could not be switched off in the laptop computer.

One important aspect to consider is that the presented results are only a reference; actual solution times may vary depending on the

doi: 10.1049/joe.2017.0023

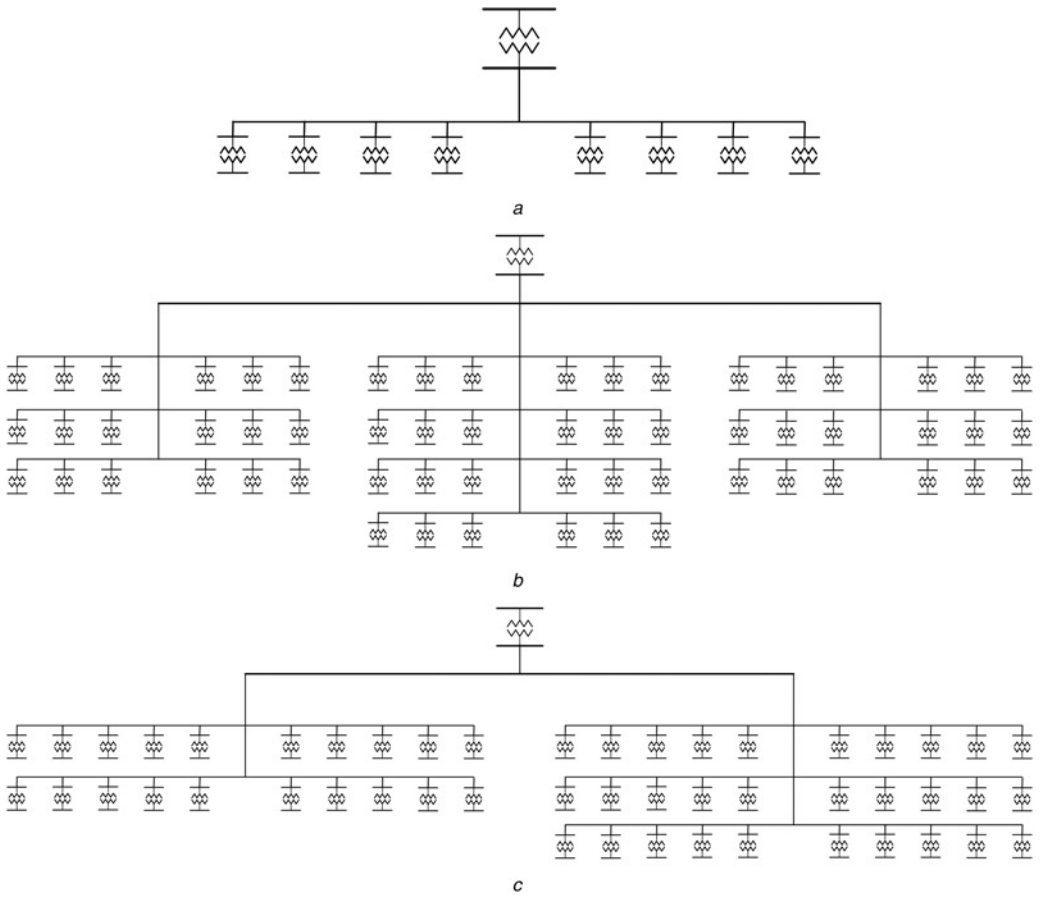

Fig. 4 *Configuration of the systems tested with OpenDSS*
a Configuration of test system 1
b Configuration of test system 2
c Configuration of test system 3

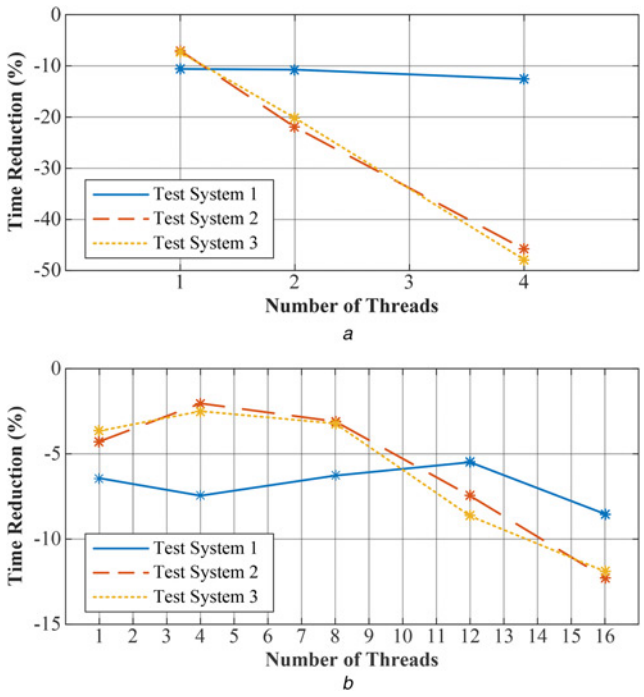

Fig. 5 *Time reduction in 'total solve time' – default OpenDSS*
a Laptop computer
b HPC server

Table 6 Execution times default OpenDSS – Laptop computer

	Intel MKL PARDISO			KLUSolve
		Test system 1		
no. of threads	1	2	4	—
compilation time, s	0.578	0.563	0.582	0.546
total solve time, s	156.258	156.445	159.037	141.273
total number of iterations	20,505	20,505	20,505	20,505
CSC format and factorisation time, s	0.032	0.017	0.036	—
average individual solve time, s[a]	0.00762	0.0076	0.0077	0.0068
		Test system 2		
no. of threads	1	2	4	—
compilation time, s	2.500	2.360	2.539	2.109
total solve time, s	1184.09	1349.28	1613.08	1106.31
total number of iterations	20,447	20,447	20,447	20,447
CSC format and factorisation time, s	0.391	0.251	0.430	—
average individual solve time, s[a]	0.0579	0.0659	0.0788	0.0541
		Test system 3		
no. of threads	1	2	4	—
compilation time, s	4.172	4.043	4.089	3.625
total solve time, s	1990.76	2228.76	2741.68	1854.53
total number of iterations	20,524	20,524	20,524	20,524
CSC format and factorisation time, s	0.547	0.418	0.464	—
average individual solve time, s[a]	0.0969	0.1085	0.1335	0.0903

[a]It includes time required to collect compensation currents [30].

Table 7 Execution times default OpenDSS – HPC server

	Intel MKL PARDISO					KLUSolve
			Test system 1			
no. of threads	1	4	8	12	16	—
compilation time, s	0.812	0.796	0.827	0.843	0.811	0.764
total solve time, s	226.855	229.039	226.528	224.858	231.332	213.159
total number of iterations	20,505	20,505	20,505	20,505	20,505	20,505
CSC format and factorisation time, s	0.048	0.032	0.063	0.079	0.047	—
average individual solve time, s[a]	0.0110	0.0111	0.0110	0.0109	0.0112	0.0103
			Test system 2			
no. of threads	1	4	8	12	16	—
compilation time, s	3.370	3.525	3.557	3.650	3.916	3.104
total solve time, s	1871.13	1831.11	1849.96	1928.33	2014.30	1794.48
total number of iterations	20,447	20,447	20,447	20,447	20,447	20,447
CSC format and factorisation time, s	0.266	0.421	0.453	0.546	0.812	—
average individual solve time, s[a]	0.0915	0.0895	0.0904	0.0943	0.0985	0.0877
			Test system 3			
no. of threads	1	4	8	12	16	—
compilation time, s	6.038	5.897	5.819	6.302	6.084	5.445
total solve time, s	3137.66	3102.56	3124.23	3288.48	3386.90	3026.98
total number of iterations	20,524	20,524	20,524	20,524	20,524	20,524
CSC format and factorisation time, s	0.593	0.452	0.374	0.857	0.639	—
average individual solve time, s[a]	0.1528	0.1511	0.1522	0.1602	0.1650	0.1474

[a]It includes time required to collect compensation currents [30].

OS and computer hardware. In addition, they can also be affected by other processes: resources not required for load flow computations (e.g. Web browser, text processing software, anti-virus software, Wi-Fi Card, etc.) should be closed or turned off in order to prevent any unnecessary actions by the OS.

4 Discussion

Multi-core processors are not the only option for implementing parallel computing applications; in recent years the use of GPUs has been widely extended in high-performance engineering and scientific applications. Different works have been conducted on GPU-based parallel load flow algorithms (e.g. [20–23]); although results are mostly positive, they should be regarded with care. As mentioned in [21], most works solve and compare dense matrices and such results cannot be extrapolated to those obtained with sparse matrix solvers. The work presented in [22, 23] provide great insight into the potential performance of GPUs when working with large sparse matrices in load flow algorithms. The presented results show important algorithm speed-ups although they only evaluate individual executions; therefore, it is still necessary to assess the impact that GPUs may have on the consecutive execution of the load flow algorithms when the sparse matrix remains constant. Future work could be aimed at linking GPU parallel libraries (e.g. CUDA Toolkit [43]) with MATPOWER and OpenDSS in order to explore the potential improvement in execution times. That work should also focus on determining how large the test systems must be in order to obtain a positive impact.

5 Conclusion

This study has presented how the parallel sparse matrix solver Intel MKL PARDISO can be introduced into the iterative solution scheme of MATPOWER and OpenDSS. The use of this solver did not require extensive alterations of the respective source codes (see Appendices 1–3). From a programming point of view, the hardest task was identifying the code sections that needed to be modified in order to include the parallel solver. Other tasks such as linking the redistribution DLL, accessing the parallel routines from within the DLL, and manipulating the input/output data can be readily completed with intermediate programming skills. In a broad sense, the Intel MKL PARDISO could be considered as 'plug-and-play', since the effort to link it to other software applications cannot be compared to that required for implementing a custom-made parallel application that includes a parallel sparse matrix solver.

The use of a parallel sparse matrix solver has proven to be a valid choice when the sparse matrix $\boldsymbol{A}$ in (1) must be updated at every step of the iterative solution: results found with the full NR method show that significant reductions in total execution times can be achieved. However, it seems clear that the use of parallel sparse matrix solvers is not adequate for the consecutive execution of iterative algorithms that rely on constant matrices (e.g. OpenDSS time-driven simulations), since time increments in individual solutions (caused by problem partitioning in the solve stage) can lead to much larger differences in total execution times. According to the obtained results, the Intel MKL PARDISO can produce time reductions of up to 40% for the full NR method; however, its use did not lead to significant performance improvements for the fast decoupled NR method. Results also show that the number of threads used in the parallelisation of the Intel MKL PARDISO routine has an important impact on the time reduction that can be achieved. Based on the curves presented in Figs. 1 and 2, it can be observed that, for systems with sizes similar to those used in this study, eight threads are enough to achieve an important reduction of execution times: using more than eight threads with the full NR method does not produce significant improvements in performance, whereas for the fast decoupled NR method it can lead to an increment in execution times.

The results obtained with OpenDSS show that the Intel MKL PARDISO routine can lead to time increments of up to 47%, causing an important loss in performance when compared to KLUSolve. Fig. 5 demonstrates that increasing the number of threads used in the parallelisation of the Intel MKL PARDISO routine causes an increment in solution times. This behaviour is typical of small systems where the overhead caused by partitioning the problem (according to the number of threads) outweighs the time gain of the parallel solution.

On the basis of the saturation effect shown in Fig. 1*b*, it can be determined that eight is the optimal number of threads for the parallelisation of the full NR algorithm (when solving test systems of sizes similar to those presented here). The fast decoupled NR method only shows very limited improvement and the default OpenDSS algorithm presents a clear diminishment in its

performance. This behaviour is an indication that the best option is to rely on highly-efficient single-core routines.

Matrix factorisation is the most computationally expensive operation in a parallel sparse matrix solver and for iterative algorithms that rely on constant matrices (e.g. default OpenDSS algorithm and fast decoupled NR) the parallel sparse routine has a negative effect when solving equation systems with sizes similar to those presented here. The solution of the Intel MKL PARDISO can be divided into three stages: analysis, numerical factorisation, and solve. The analysis and numerical factorisation stages are always performed together and in the case of constant matrices they must be executed only once, therefore the time spent in these stages is accounted as one. For the default OpenDSS algorithm it is clear that the 'average individual solve time' is much shorter than the time spent for the generation of the CSC format and matrix factorisation (i.e. analysis and numerical factorisation stages); take into account that this time includes the time required to gather the compensation currents vector. For the fast decoupled NR algorithm an average solve time has not been calculated; however, the time spent to perform the P and Q iterations can be calculated by subtracting the 'factorisation' and 'CSR format' times from 'total time'. Although these times vary according to the number of threads used in the parallel routine, it is clear that the time needed to execute the solve stage is much shorter than the 'factorisation' time.

Although other parallel high-performance libraries may exhibit similar or slightly superior performance than the Intel MKL PARDISO (see [19]), it is not expected that this or other libraries will produce a significant performance improvement with respect to highly-efficient single-core routines, unless the equation systems are much larger than those analysed here. In [44, 45] a system with over one million buses was used to test single-core Newton–Krylov solvers for very large power systems. The ongoing efforts to test systems of such size using single-core routines can be seen as an indication that in order to obtain a significant performance gain using parallel routines it is necessary to study systems several times larger than that used in [44, 45].

It is important to emphasise that linking a high-performance library (e.g. Intel MKL PARDISO) with pre-existing software tools (e.g. MATPOWER and OpenDSS) can be a suitable option for implementing parallel solutions. Although the development of a custom-made parallel routine is a difficult task and requires deep some knowledge of parallel algorithms and programming, this paper has shown that some available libraries can be deployed with interfaces that are readily used to connect them to other applications. Furthermore, most routines can be initialised with a set of default parameters, which facilitates its use for non-expert users. However, a clear understanding of sparse matrices and sparse solvers will always provide a great aid to maximise the performance of such routines.

6 References

[1] Cai X., Acklam E., Langtangen H.P., *ET AL.*: 'Parallel computing', in Langtangen H.P., Tveito A. (Eds.): 'Advanced topics in computational partial differential equations' (Springer, Heidelberg, Germany, 2003)

[2] Skuhersky M.: 'Introduction to Parallel Computing'. Available at http://web.mit.edu/vex/www/Parallel.pdf

[3] Grama A., Gupta A., Karypis G., *ET AL.*: 'Introduction to parallel computing' (Addison Wesley, Essex, UK, 2003)

[4] Falcao D.M., Borges C.L.T., Taranto G.N.: 'High performance computing in electrical energy systems applications', in Kumar Khaitan S., Gupta A. (Eds.): 'High performance computing in power and energy systems' (Springer, 2013)

[5] Zhou M.: 'Distributed parallel power system simulation', in Kumar Khaitan S., Gupta A. (Eds.): 'High performance computing in power and energy systems' (Springer, Heidelberg, Germany, 2013)

[6] Aristidou P., Hug G.: 'Accelerating the computation of critical eigenvalues with parallel computing techniques'. Power Systems Computation Conf. (PSCC), Genoa, Italy, 2016

[7] Tomim M., Marti J., Passos Filho J.A.: 'Parallel transient stability simulation based on multi-area Thévenin equivalents', Accepted for publication in *IEEE Trans. Smart Grid*, 2017, **8**, (3), pp. 1366–1377

[8] Lu N., Taylor Z.T., Chassin D.P., *ET AL.*: 'Parallel computing environments and methods for power distribution system simulation'. IEEE PES General Meeting, San Francisco, CA, USA, June 2005

[9] Mosaddegh A., Canizares C.A., Bhattacharya K., *ET AL.*: 'Distributed computing architecture for optimal control of distribution feeders with smart loads', Accepted for publication in *IEEE Trans. Smart Grid*, 2017, **8**, (3), pp. 1469–1478

[10] de León F., Czarkowski D., Spitsan V., *ET AL.*: 'Development of data translators for interfacing power-flow programs with EMTP-type programs: challenges and lessons learned', *IEEE Trans. Power Deliv.*, 2013, **28**, (2), pp. 1192–1201

[11] Dufour C., Jalili-Marandi V., Bélanger J., *ET AL.*: 'Power system simulation algorithms for parallel computer architectures'. IEEE PES General Meeting, San Diego, CA, USA, 2012

[12] Dufour C., Mahseredjian J., Belange J.: 'A combined state-space nodal method for the simulation of power system transients', *IEEE Trans. Power Deliv.*, 2010, **26**, (2), pp. 928–935

[13] Strunz K., Carlson E.: 'Nested fast and simultaneous solution for time-domain simulation of integrative power-electric and electronic systems', *IEEE Trans. Power Deliv.*, 2007, **22**, (1), pp. 277–287

[14] Martinez J.A., Guerra G.: 'A parallel Monte Carlo method for optimum allocation of distributed generation', *IEEE Trans. Power Syst.*, 2014, **29**, (6), pp. 2926–2933

[15] Koester D.P., Ranka S., Fox G.C.: 'A parallel Gauss-Seidel algorithm for sparse power system matrices'. Proc. of Supercomputing '94, Washington, DC, USA, 1994

[16] Wu J.Q., Bose A.: 'Parallel solution of large sparse matrix equations and parallel power flow', *IEEE Trans. Power Syst.*, 1995, **10**, (3), pp. 1343–1349

[17] Tu F., Flueck A.J.: 'A message-passing distributed-memory Newton-GMRES parallel power flow algorithm'. IEEE PES Summer Meeting, Chicago, IL, USA, 2002

[18] Wang X., Ziavras S.G., Nwankpa C., *ET AL.*: 'Parallel solution of Newton's power flow equations on configurable chips', *Int. J. Electr. Power Energy Syst.*, 2007, **29**, (5), pp. 422–431

[19] Zadeh K., Zeynal H., Nor K.M., *ET AL.*: 'Performance evaluation of SuperLU and PARDISO in power system load flow calculations', *Electr. Rev. (Przeglad Elektrotechniczny)*, 2011, **87**, (11), pp. 290–294

[20] Guo C., Jiang B., Yuan H., *ET AL.*: 'Performance comparisons of parallel power flow solvers on GPU system'. 18th Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA), Seoul, Korea, 2012

[21] Roberge V., Tarbouchi M., Okou F.: 'Parallel power flow on graphics processing units for concurrent evaluation of many networks', *IEEE Trans. Smart Grid*, 2015, **PP**, (99), pp. 1–10, DOI: 10.1109/TSG.2015.2496298

[22] Li X., Li F.: 'GPU-based two-step preconditioning for conjugate gradient method in power flow'. IEEE PES General Meeting, Denver, CO, USA, 2015

[23] Li X., Li F., Yuan H., *ET AL.*: 'GPU-based fast decoupled power flow with preconditioned iterative solver and inexact newton method', *IEEE Trans. Power Syst.*, 2016, **PP**, (99), p. 1, DOI: 10.1109/TPWRS.2016.2618889

[24] Available at http://www.openelectrical.org/wiki/index.php?title=Power_Systems_Analysis_Software

[25] Intel Math Kernel Library for Windows OS (Developer Guide), Available at https://software.intel.com/en-us/intel-mkl-support/documentation

[26] Amestoy P.R., Duff I.S., Koster J., *ET AL.*: 'A fully asynchronous multifrontal solver using distributed dynamic scheduling', *SIAM J. Matrix Anal. Appl.*, 2001, **23**, (1), pp. 15–41

[27] Hénon P., Ramet P., Roman J.: 'PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems', *Parallel Comput.*, 2002, **28**, (2), pp. 301–321

[28] Li X.S.: 'An overview of SuperLU: Algorithms, implementation, and user interface', *ACM Trans. Math. Softw.*, 2005, **31**, (3), pp. 302–325

[29] Zimmerman R.D., Murillo-Sanchez C.E., Thomas R.J.: 'Matpower: Steady-state operations, planning and analysis tools for power systems research and education', *IEEE Trans. Power Syst.*, 2011, **26**, (1), pp. 12–19

[30] Dugan R., McDermott T.E.: 'An open source platform for collaborating on smart grid research'. IEEE PES General Meeting, Detroit, MN, USA, 2011

[31] Schenk O., Gärtner K., Fichtner W.: 'Efficient sparse LU factorization with left–right looking strategy on shared memory multiprocessors', *BIT*, 2000, **40**, (1), pp. 158–176
[32] Intel Math Kernel Library (Developer Reference), Available at https://software.intel.com/en-us/intel-mkl-support/documentation
[33] Tinney W.F., Hart C.E.: 'Power flow solution by Newton's method', *IEEE Trans. Power Appar. Syst.*, 1967, **86**, (11), pp. 1449–1460
[34] Stott B., Alsaç O.: 'Fast decoupled load flow', *IEEE Trans. Power Appar. Syst.*, 1974, **93**, (3), pp. 859–869
[35] Zimmerman R.D., Murillo-Sánchez C.: 'MATPOWER User's Manual', Available at http://www.pserc.cornell.edu/matpower
[36] Fliscounakis S., Panciatici P., Capitanescu F., *ET AL.*: 'Contingency ranking with respect to overloads in very large power systems taking into account uncertainty, preventive and corrective actions', *IEEE Trans. Power Syst.*, 2013, **28**, (4), pp. 4909–4917
[37] Amdahl G.M.: 'Validity of the single processor approach to achieving large scale computing capabilities', *IEEE Solid-State Circuits Soc. Newsl.*, 2007, **12**, (3), pp. 19–20
[38] Amdahl G.M.: 'Validity of the single processor approach to achieving large scale computing capabilities'. AFIPS Conf. Proc., Atlantic City, NJ, USA, 1967, vol. **30**, pp. 483–485
[39] Hill M.D., Marty M.R.: 'Amdahl's law in the multicore era', *Computer*, 2008, **41**, (7), pp. 33–38
[40] Dugan R.C.: 'Open DSS reference guide' (EPRI, 2013)
[41] Available at https://sourceforge.net/projects/klusolve
[42] Martínez-Velasco J.A., Guerra G.: 'Analysis of large distribution networks with distributed energy resources', *Ingeniare*, 2015, **23**, (4), pp. 594–608
[43] Available at https://developer.nvidia.com/cuda-toolkit
[44] Idema R., Lahaye D.J.P., Vuik C., *ET AL.*: 'Scalable Newton-Krylov solver for very large power flow problems', *IEEE Trans. Power Syst.*, 2012, **27**, (1), pp. 390–396
[45] Idema R., Papaefthymiou G., Lahaye D., *ET AL.*: 'Towards faster solution of large power flow problems', *IEEE Trans. Power Syst.*, 2013, **28**, (4), pp. 4918–4925

7 Appendix

7.1 *Appendix 1: Code for the solution of the full NR method*

Fig. 6 presents the MATLAB code introduced into the *m*-files in order to solve the voltage corrections in the iterative scheme. The *phase* variable is set to perform all stages of the Intel MKL PARDISO solution process and the *calllib* function is used to call the routine from the MATLAB environment. The *csrmatrix_full* function was developed by the authors.

```
%Full Newton-Raphson - MATLAB
%dx = -(J \ F); %Voltage Corrections

%Full Newton-Raphson - Intel MKL PARDISO
[ia,ja,ai]=csrmatrix_full(J); %CSR Format Conversion
phase=13; %Intel MKL PARDISO Solution Steps
calllib(libname,'PARDISO',pt, maxfct, mnum, mtype, phase,
        n_pardiso, ai, ia, ja, idum, nrhs, iparm, msglvl,
        b_pardiso, x_pardiso, error_pardiso)
dx=-x_pardiso.value; %Voltage Corrections
```

Fig. 6 *Solution of voltage corrections in Full Newton-Raphson's method (Intel MKL PARDISO)*

```
%Fast Decoupled Newton-Raphson - MATLAB
%[Lp,  Up,  pp,  qp ] = lu(Bp,  'vector'); %B' Matrix Factorization
%[Lpp, Upp, ppp, qpp] = lu(Bpp, 'vector'); %B'' Matrix Factorization
%dVa = -( Up \  (Lp \ P(pp)) ); %Voltage Angle Corrections
%dVm = -( Upp \ (Lpp \ Q(ppp)) ); %Voltage Magnitude Corrections

%Fast Decoupled Newton-Raphson - Intel MKL Pardiso
[iBp,jBp,aBp]=csrmatrix_symm(Bp); %CSR Format Conversion
[iBpp,jBpp,aBpp]=csrmatrix_symm(Bpp); %CSR Format Conversion
%B' and B'' Matrix Factorization
phase=12;
calllib(libname,'PARDISO',pt_1, maxfct, mnum, mtype, phase, n_p,
        aBp, iBp, jBp, idum, nrhs, iparm, msglvl, ddum, ddum,
        error_pardiso_1)
calllib(libname,'PARDISO',pt_2, maxfct, mnum, mtype, phase, n_pp,
        aBpp, iBpp, jBpp, idum, nrhs, iparm2, msglvl, ddum, ddum,
        error_pardiso_2)
%Voltage Angle Corrections
phase=33;
calllib(libname,'PARDISO',pt_1, maxfct, mnum, mtype, phase, n_p,
        aBp, iBp, jBp, idum, nrhs, iparm, msglvl, b_p, x_p,
        error_pardiso_1)
dVa=-x_p.Value;
%Voltage Magnitude Corrections
calllib(libname,'PARDISO',pt_2, maxfct, mnum, mtype, phase, n_pp,
        aBpp, iBpp, jBpp, idum, nrhs, iparm2, msglvl, b_pp, x_pp,
        error_pardiso_2)
dVm=-x_pp.Value;
```

Fig. 7 *Solution of voltage corrections in Fast Decoupled Newton-Raphson's method (Intel MKL PARDISO)*

```
// CSC format and Matrix factorization
InitAndGetYparams(gY, nBus, nNZ);
SetLength (MKLColPtr, nBus + 1);
SetLength (MKLRowIdx, nNZ);
n_mkl:=Integer(nBus);
SetLength (MKLcVals, nNZ);
GetCompressedMatrix (gY, nBus + 1, nNZ, @MKLColPtr[0],
                     @MKLRowIdx[0], @MKLcVals[0]);
phase:=12;
PARDISO(pt[0], maxfct, mnum, mtype, phase, n_mkl, MKLcVals[0],
        MKLColPtr[0], MKLRowIdx[0], idum, nrhs, iparm[0], msglvl,
        ddum, ddum, error_mkl);

// KLU Solve for present InjCurr
//RetCode := SolveSparseSet(hY, @V^[1], @Currents^[1]);

// Solve using Intel MKL PARDISO and adapt error value
phase:=33;
PARDISO(pt[0], maxfct, mnum, mtype, phase, n_mkl, MKLcVals[0],
        MKLColPtr[0], MKLRowIdx[0], idum, nrhs, iparm[0], msglvl,
        Currents[1], V[1], error_mkl);
if error_mkl=0 then RetCode:=1 else RetCode:=0;
if error_mkl=-7 then RetCode:=2;
```

Fig. 8 *Admittance matrix CSC format, matrix factorisation, and solution of updated node voltages (Intel MKL PARDISO)*

7.2 Appendix 2: Code for the solution of the fast decoupled NR method

In the present MATLAB code the Intel MKL PARDISO solution process is divided in different stages, see Fig. 7. First the *phase* variable is set to perform only the analysis and numerical factorisation stages of the $\boldsymbol{B'}$ and $\boldsymbol{B''}$ admittance matrices [25]. Next the code required to solve the voltage angle and magnitude corrections is shown; note that the *phase* variable is set so only the solve stage is performed. The *csrmatrix_symm* function was developed by the authors.

7.3 Appendix 3: Code for the solution of the default OpenDSS solution

The Delphi code in Fig. 8 in this annex shows the lines used to retrieve the admittance matrix using the CSC format and perform the analysis and factorisation stages of the Intel MKL PARDISO. In addition, it presents the code inserted into the iteration process to calculate the updated node voltages using the factorised admittance matrix and the compensation currents vector. The *phase* variable is set accordingly and it is assigned the same values as in the code used for the fast decoupled NR.