Parallel Sparse Direct And
Multi-Recursive Iterative Linear Solvers

PARDISO

User Guide Version 6.0.0

(Updated January 06, 2018)

Olaf Schenk and Klaus Gärtner

Since a lot of time and effort has gone into the new PARDISO version 6.0.0, please cite the following publications if you are using PARDISO for your own research:

References

[1] Cosmin G. Petra, Olaf Schenk, and Mihai Anitescu. Real-time stochastic optimization of complex energy systems on high-performance computers, *IEEE Computing in Science & Engineering*, 16(5):32–42, 2014.

[2] Cosmin G. Petra, Olaf Schenk, Miles Lubin, and Klaus Gärtner. An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization. *SIAM Journal on Scientific Computing*, 36(2):C139–C162, 2014.

Note: This document briefly describes the interface to the shared-memory and distributed-memory multiprocessing parallel direct and multi-recursive iterative solvers in PARDISO 6.0.0[1]. A discussion of the algorithms used in PARDISO and more information on the solver can be found at `http://www.pardiso-project.org`

[1] Please note that this version is a significant extension compared to Intel's MKL version and that these improvements and features are not available in Intel's MKL 9.3 release or later releases.

Contents

1 Introduction **4**
1.1 Supported Matrix Types . . . 4
1.1.1 Symmetric Matrices . . . 4
1.1.2 Structurally Symmetric Matrices . . . 5
1.1.3 Nonsymmetric Matrices . . . 5
1.2 Computing Selected Elements of the Inverse A^{-1}: PARDISO-INV . . . 5
1.3 Computing the Schur-complement: PARDISO-SCHUR . . . 5
1.4 Parallel Distributed-Memory Factorization for Symmetric Indefinite Linear Systems . . . 6
1.5 Multi-Recursive Iterative Sparse Solver for Symmetric Indefinite Linear Systems . . . 6
1.6 Direct-Iterative Preconditioning for Nonsymmetric Linear Systems . . . 6

2 Specification **7**
2.1 Arguments of PARDISOINIT . . . 8
2.2 Arguments of PARDISO . . . 10
2.3 IPARM Control Parameters . . . 12
2.4 DPARM Control Parameters . . . 20
2.5 Summary of arguments and parameters . . . 22
2.6 Computing the Schur-complement: Example for nonsymmetric matrices . . . 23

3 Linking and Running the Software **31**
3.1 LINUX on IA64 platform . . . 31
3.1.1 Linking the libraries . . . 31
3.1.2 Setting environment . . . 32
3.2 Mac OSX on 64-bit platforms . . . 32
3.2.1 Linking the libraries . . . 32
3.2.2 Setting environment . . . 32
3.3 Windows on IA64 platforms . . . 33
3.3.1 Linking the libraries . . . 33

4 Using the MATLAB interface **33**
4.1 An example with a real matrix . . . 33
4.2 A brief note of caution . . . 34
4.3 An example with a real, symmetric matrix . . . 34
4.4 Complex matrices . . . 34

5 Installing the MATLAB interface **35**

6 Using the PARDISO Matrix Consistency Checker **35**

7 Authors **36**

8 Acknowledgments **36**

9 License **36**

10 Further comments **38**

A Examples for sparse symmetric linear systems **38**
A.1 Example results for symmetric systems . . . 38
A.2 Example pardiso_sym.f for symmetric linear systems (including selected inversion) . . . 41
A.3 Example pardiso_sym.c for symmetric linear systems (including selected inversion) . . . 45
A.4 Example pardiso_sym_schur.cpp for Schur-complement computation for symmetric linear systems . . . 50

A.5 Example iterative solver pardiso_iter_sym.f for symmetric linear systems 55

B Examples for sparse symmetric linear systems on distributed-memory architectures 58
B.1 Example pardiso_sym_mpp.c for the solution of symmetric linear systems on distributed memory architecutes 58

C Release notes 62

1 Introduction

The package PARDISO is a high-performance, robust, memory–efficient and easy to use software for solving large sparse symmetric and nonsymmetric linear systems of equations on shared–memory and distributed-memory architectures. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [15]. In order to improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update, and pipelining parallelism is exploited with a combination of left- and right-looking supernode techniques [9, 11, 12, 14]. The parallel pivoting methods allow complete supernode pivoting in order to balance numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory and distributed-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed. The approach is based on OpenMP [4] directives and MPI parallelization, and has been successfully tested on almost all shared-memory parallel systems.

PARDISO calculates the solution of a set of sparse linear equations with multiple right-hand sides,

$$AX = B,$$

using a parallel LU, LDL^T or LL^T factorization, where A and X, B are n by n and n by $nrhs$ matrices respectively. PARDISO supports, as illustrated in Figure 1, a wide range of sparse matrix types and computes the solution of real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian sparse linear systems of equations on shared-memory multiprocessing architectures.

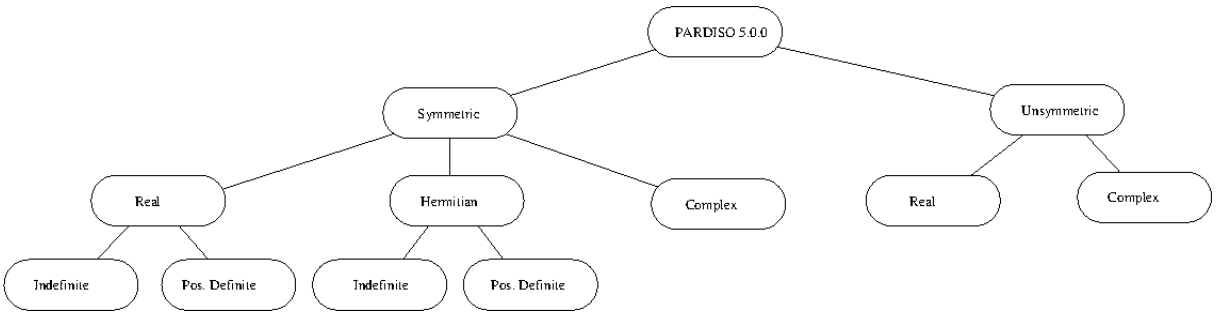

Figure 1: Sparse matrices that can be solved with PARDISO Version 6.0.0

PARDISO 6.0.0 has the unique feature among all solvers that it can compute the exact bit-identical solution on multicores and cluster of multicores (see release notes).

1.1 Supported Matrix Types

PARDISO performs the following analysis steps depending on the structure of the input matrix A.

1.1.1 Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [6] or the nested dissection algorithm from the METIS package [2], followed by the parallel left-right looking numerical Cholesky factorization [15] $PAP^T = LL^T$ or $PAP^T = LDL^T$ for symmetric, indefinite matrices. The solver uses diagonal pivoting or 1×1 and 2×2 Bunch-Kaufman pivoting for symmetric indefinite matrices and an approximation of X is found by forward and backward substitution and iterative refinement.

The coefficient matrix is perturbed whenever numerically acceptable 1×1 and 2×2 pivots cannot be found within a diagonal supernode block. One or two passes of iterative refinement may be required

to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore the accuracy of this method is for a large set of matrices from different applications areas as accurate as a direct factorization method that uses complete sparse pivoting techniques [13]. Another possibility to improve the pivoting accuracy is to use symmetric weighted matchings algorithms. These methods identify large entries in A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

1.1.2 Structurally Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and iterative refinement.

1.1.3 Nonsymmetric Matrices

The solver first computes a non-symmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim to place large entries on the diagonal, which enhances greatly the reliability of the numerical factorization process [1]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization $QLURA_2$, $A_2 = PP_{MPS}D_rAD_cP$ with supernode pivoting matrices Q and R and $P = P(P_{MPS})$ to keep sufficiently large parts of the cycles of P_{MPS} in one diagonal block. When the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [5]. The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \epsilon \cdot ||A_2||_\infty$, where ϵ is the machine precision, and $||A_2||_\infty$ is the ∞-norm of the scaled and permuted matrix A. Any tiny pivots l_{ii} encountered during elimination are set to the $\text{sign}(l_{ii}) \cdot \epsilon \cdot ||A_2||_\infty$ — this trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice it is observed that the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

1.2 Computing Selected Elements of the Inverse A^{-1}: PARDISO-INV

The parallel selected inversion method is an efficient way for computing e.g. the diagonal elements of the inverse of a sparse matrix. For a symmetric matrix A, the selected inversion algorithm first constructs the LDL^T factorization of A, where L is a block lower diagonal matrix called the Cholesky factor, and D is a block diagonal matrix. In the second step, the selected inversion algorithm only computes all the elements A^{-1}_{ij} such that $L_{ij} \neq 0$ in **exact** arithmetic. The computational scaling of the selected inversion algorithm is only proportional to the number of nonzero elements in the Cholesky factor L, even though A^{-1} might be a full matrix. The selected inversion algorithm scales as $O(N)$ for quasi-1D systems, $O(N^{1.5})$ for quasi-2D systems, $O(N^2)$ for 3D systems. The selected inversion method achieves significant improvement over the tradtional forward/backward method at all dimensions.

The parallel threaded implementation [3] of the selected inversion algorithm, called PARDISO-INV, is available for sparse symmetric or unsymmetric matrices, and it is fully integrated into the PARDISO package. In addition, it includes pivoting to maintain accuracy during both sparse inversion steps.

1.3 Computing the Schur-complement: PARDISO-SCHUR

The Schur-complement framework is a general way to solve linear systems. Much work has been done in this area; see, for example [8] and the references therein. Let $Ax = b$ be the system of interest. Suppose A has the form

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

A traditional approach to compute the exact Schur complement $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ by using off-the-shelf linear solvers such as PARDISO is (i) to factor A_{11} and (ii) to solve for each right-hand side of A_{12} (in fact for small blocks of right-hand sides). However, this approach has two important drawbacks: (1) the sparsity of the right-hand sides A_{12} and A_{21} is not exploited, and (2) the calculations limit the efficient use of multicore shared-memory environments because it is well known that the triangular solves do not scale very well with the number of cores. In [7] we developed a fast and parallel reformulation of the sparse linear algebra calculations that circumvents the issues in the Schur complement computations. The idea is to leverage the good scalability of parallel factorizations as an alternative method of building the Schur complement basecd on forward/backward substitutions. We employ an *incomplete augmented factorization* technique that solves the sparse linear systems with multiple right-hand sides at once using an incomplete sparse factorization of the matrix A.

The parallel threaded implementation of the Schur-complement algorithm, called PARDISO-SCHUR, is available for sparse symmetric or unsymmetric matrices, and it is fully integrated into the PARDISO package. In addition, it includes pivoting to maintain accuracy during the Schur complement computations.

1.4 Parallel Distributed-Memory Factorization for Symmetric Indefinite Linear Systems

A new and scalable MPI-based numerical factorization and parallel forward/backward substitution on distributed-memory architectures for symmetric indefinite matrices has been implemented in PARDISO 6.0.0. It has the unique feature among all solvers that it can compute the exact bit-identical solution on multicores and cluster of multicores (see release notes).

This solver is only available for real symmetric indefinite matrices.

1.5 Multi-Recursive Iterative Sparse Solver for Symmetric Indefinite Linear Systems

The solver also includes a novel preconditioning solver [10]. Our preconditioning approach for symmetric indefinite linear system is based on maximum weighted matchings and algebraic multilevel incomplete LDL^T factorizations. These techniques can be seen as a complement to the alternative idea of using more complete pivoting techniques for the highly ill-conditioned symmetric indefinite matrices. In considering how to solve the linear systems in a manner that exploits sparsity as well as symmetry, we combine a diverse range of topics that includes preprocessing the matrix with symmetric weighted matchings, solving the linear system with Krylov subspace methods, and accelerating the linear system solution with multilevel preconditioners based upon incomplete factorizations.

1.6 Direct-Iterative Preconditioning for Nonsymmetric Linear Systems

The solver also allows a combination of direct and iterative methods [16] in order to accelerate the linear solution process for transient simulation. Many applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but the same identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. PARDISO uses a numerical factorization $A = LU$ for the first system and applies these exact factors L and U for the next steps in a preconditioned Krylov-Subspace iteration. If the iteration does not converge, the solver will automatically switch back to the numerical factorization. This method can be applied for nonsymmetric matrices in PARDISO and the user can select the method using only one input parameter. For further details see the parameter description (IPARM(4), IPARM(20)).

2 Specification

The library is available on 32-bit and 64-bit architectures. The library is based on Fortran and C source code and the top level driver routines PARDISO and PARDISOINIT assume the following interface.

On 32-bit architectures:

```
SUBROUTINE PARDISOINIT(PT, MTYPE, SOLVER, IPARM, DPARM, ERROR)
INTEGER*4 PT(64), MYTPE, SOLVER, IPARM(64), ERROR
REAL*8    DPARM(64)
```

and

```
SUBROUTINE PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE, N, A, IA, JA,
1                  PERM, NRHS, IPARM, MSGLVL, B,  X, ERROR, DPARM)
INTEGER*4 PT(64)
INTEGER*4 MAXFCT, MNUM, MTYPE, PHASE, N, NRHS, ERROR
1         IA(*), JA(*), PERM(*), IPARM(*)
REAL*8    DPARM(64)
REAL*8    A(*), B(N,NRHS),  X(N,NRHS)
```

On 64-bit architectures:

```
SUBROUTINE PARDISOINIT(PT, MTYPE, SOLVER, IPARM, DPARM, ERROR)
INTEGER*8  PT(64)
INTEGER*4  PT(64), MYTPE, SOLVER, IPARM(64), ERROR
REAL*8     DPARM(64)
```

and

```
SUBROUTINE PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE, N, A, IA, JA,
1                  PERM, NRHS, IPARM, MSGLVL, B,  X, ERROR, DPARM)
INTEGER*8 PT(64)
INTEGER*4 MAXFCT, MNUM, MTYPE, PHASE, N, NRHS, ERROR
1         IA(*), JA(*), PERM(*), IPARM(*)
REAL*8    DPARM(64)
REAL*8    A(*), B(N,NRHS),  X(N,NRHS)
```

The sparse direct solvers use 8-byte integer length internally that allows storage of factors with more than 2^{32} nonzeros elements. The following are examples for calling PARDISO from a Fortran program

```
c...Check license of the solver and initialize the solver
    CALL PARDISOINIT(PT, MTYPE, SOLVER, IPARM, DPARM, ERROR)

c...Solve matrix sytem
    CALL PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE, N, A, IA, JA,
  1              PERM, NRHS, IPARM, MSGLVL, B,  X, ERROR, DPARM)
```

for calling PARDISO from a C program

```
    /* Check license of the solver and initialize the solver */
    pardisoinit(pt, &mtype, &solver, iparm, dparm, &error);

    /* Solve matrix sytem */
    pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja,
             perm, &nrhs, iparm, &msglvl, b,  x, &error, dparm)
```

	1	2	3	4	5	6	7	8
1	7.		1.			2.	7.	
2		–4.	8.		2.			
3			1.					5.
4				7.			9.	
5		–4.						
6			7.			3.		5.
7		17.					11.	
8			–3.				2.	5.

	1	2	3	4	5	6	7	8
1	7.		1.			2.	7.	
2		–4.	8.		2.			
3			1.					5.
4				7.			9.	
5					5.	–1.	5.	
6						0.		5.
7							11.	
8								5.

	Nonsymmetric Matrix			Symmetric Matrix		
K	IA(K)	JA(K)	A(K)	IA(k)	JA(K)	A(K)
1	1	1	7.	1	1	7.
2	5	3	1.	5	3	1.
3	8	6	2.	8	6	2.
4	10	7	7.	10	7	7.
5	12	2	-4.	12	2	-4.
6	13	3	8.	15	3	8.
7	16	5	2.	17	5	2.
8	18	3	1.	18	3	1.
9	21	8	5.	19	8	5.
10		4	7.		4	7.
11		7	9.		7	9.
12		2	-4.		5	5.
13		3	7.		6	-1.
14		6	3.		7	5.
15		8	5.		6	0.
16		2	17.		8	5.
17		7	11.		7	11.
18		3	-3.		8	5.
19		7	2.			
20		8	5.			

Figure 2: Illustration of the two input compressed sparse row formats of an 8×8 general sparse nonsymmetric linear system and the upper triangular part of a symmetric linear system for the direct solver PARDISO. The algorithms in PARDISO require JA to be increasingly ordered per row and the presence of the diagonal element per row for any symmetric or structurally symmetric matrix. The nonsymmetric matrices need no diagonal elements in the PARDISO solver.

2.1 Arguments of PARDISOINIT

PARDISOINIT checks the current license in the file pardiso.lic and initializes the internal timer and the address pointer PT. It sets the solver default values according to the matrix type.

PT(64) — INTEGER Input/Output

On entry: This is the solver's internal data address pointer. These addresses are passed to the solver and all related internal memory management is organized through this pointer.

On exit: Internal address pointers.

Note: PT is a 32-bit or 64-bit integer array on 32 or 64 bit operating systems, respectively. It has 64 entries.
Never change PT!

MTYPE — INTEGER Input

On entry: This scalar value defines the matrix type. PARDISO supports the following matrices

TYPE	Matrix
1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

Note: The parameter influences the pivoting method.

SOLVER — INTEGER Input

On entry: This scalar value defines the solver method that the user would like to use.

SOLVER	Method
0	sparse direct solver
1	multi-recursive iterative solver

IPARM (64) — INTEGER Input/Output

On entry: IPARM is an integer array of size 64 that is used to pass various parameters to PARDISO and to return some useful information after the execution of the solver. PARDISOINIT fills IPARM(1), IPARM(2), and IPARM(4) through IPARM(64) with default values and uses them.

See section 2.3 for a detailed description.

Note: There is no default value for IPARM(3), which reflects the number of processors and this value must always be supplied by the user. On distributed-memory architectures, the user must also supply the value for IPARM(52), which reflects the number of compute nodes. Mixed OpenMP-MPI parallelization is used for the distributed-memory solver.

DPARM (64) — REAL Input/Output

On entry: DPARM is a double-precision array of size 64 that is used to pass various parameters to PARDISO and to return some useful information after the execution of the solver. The array will only be used to initialize the multi-recursive iterative linear solver in PARDISO. PARDISOINIT fills DPARM(1) through DPARM(64) with default values and uses them.

See section 2.4 for a detailed description.

ERROR — INTEGER Output

On output: The error indicator

Error	Information
0	No error.
-10	No license file pardiso.lic found
-11	License is expired
-12	Wrong username or hostname

2.2 Arguments of PARDISO

Solving a linear system is split into four tasks: analysis and symbolic factorization, numerical factorization, forward and backward substitution including iterative refinement, and finally termination to release all internal solver memory. The calling sequence and description of the PARDISO parameters is as follows. When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a place holder for that argument.

PT(64) — INTEGER Input/Output

see PARDISOINIT.

MAXFCT — INTEGER Input

On entry: Maximal number of factors with identical nonzero sparsity structure that the user would like to keep at the same time in memory. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data management of the solver. In many applications this value is equal to 1.

Note: Matrices with different sparsity structure can be kept in memory with different memory address pointers PT.

MNUM — INTEGER Input

On entry: Actual matrix for the solution phase. With this scalar the user can define the matrix that he would like to factorize. The value must be: $1 \leq \text{MNUM} \leq \text{MAXFCT}$. In many applications this value is equal to 1.

MTYPE — INTEGER Input

see PARDISOINIT. **Note:** The parameter influences the pivoting method.

PHASE — INTEGER Input

On entry: PHASE controls the execution of the solver. It is a two-digit integer ij ($10i + j$, $1 \leq i \leq 3$, $i \leq j \leq 3$ for normal execution modes). The i digit indicates the starting phase of execution, and j indicates the ending phase. PARDISO has the following phases of execution:

1. Phase 1: Fill-reduction analysis and symbolic factorization
2. Phase 2: Numerical factorization
3. Phase 3: Forward and Backward solve including iterative refinement
4. Termination and Memory Release Phase (PHASE ≤ 0)

If a previous call to the routine has computed information from previous phases, execution may start at any phase:

PHASE	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
-22	Selected Inversion
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
0	Release internal memory for L and U matrix number MNUM
-1	Release all internal memory for all matrices

Note: The analysis phase needs the numerical values of the matrix A in case of scalings (IPARM(11)=1) and symmetric weighted matching (IPARM(13)=1 (normal matching) or IPARM(13)=2 (advanced matchings)).

N — INTEGER Input

On entry: Number of equations in the sparse linear systems of equations $A \times X = B$.

Constraint: $N > 0$.

A(∗) — REAL/COMPLEX Input

On entry: Nonzero values of the coefficient matrix A. The array A contains the values corresponding to the indices in JA. The size and order of A is the same as that of JA. The coefficients can be either real or complex. The matrix must be stored in compressed sparse row format with increasing values of JA for each row. Refer to Figure 2 for more details.

Note: For symmetric or structural symmetric matrices it is important that the diagonal elements are also available and stored in the matrix. If the matrix is symmetric or structural symmetric, then the array A is only accessed in the factorization phase, in the triangular solution with iterative refinement. Nonsymmetric matrices are accessed in all phases of the solution process.

IA(N+1) — INTEGER Input

On entry: IA is an integer array of size N+1. IA(i) points to the first column index of row i in the array JA in compressed sparse row format. Refer to Figure 2 for more details.

Note: The array is accessed in all phases of the solution process. Note that the row and column numbers start from 1.

JA(∗) — INTEGER Input

On entry: The integer array JA contains the column indices of the sparse matrix A stored in compress sparse row format. The indices in each row must be sorted in increasing order.

Note: The array is accessed in all phases of the solution process. For symmetric and structurally symmetric matrices it is assumed that zero diagonal elements are also stored in the list of nonzeros in A and JA. For symmetric matrices the solver needs only the upper triangular part of the system as shown in Figure 2.

PERM (N) — INTEGER Input

On entry: The user can supply his own fill-in reducing ordering to the solver. This permutation vector is only accessed if IPARM(5) = 1.

Note: The permutation PERM must be a vector of size N. Let A be the original matrix and $B = PAP^T$ be the permuted matrix. The array PERM is defined as follows. Row (column) of A is the PERM(i) row (column) of B. The numbering of the array must start by 1 and it must describe a permutation.

NRHS — INTEGER Input

On entry: NRHS is the number of right-hand sides that need to be solved for.

IPARM (64) — INTEGER Input/Output

On entry: IPARM is an integer array of size 64 that is used to pass various parameters to PARDISO and to return some useful information after the execution of the solver. If IPARM(1) is 0, then PARDISOINIT fills IPARM(2) and IPARM(4) through IPARM(64) with default values.

See section 2.3 for a detailed description.

On output. Some IPARM values will contain useful information, e.g. numbers of nonzeros in the factors, etc. .

Note: There is no default value for IPARM(3) and this value must always be supplied by the user, regardless of whether IPARM(1) is 0 or 1. On distributed-memory architectures, the user must also supply the value for IPARM(52), which reflects the number of compute nodes. Mixed OpenMP-MPI parallelization is used for the distributed-memory solver.

MSGLVL — INTEGER Input

On entry: Message level information. If MSGLVL = 0 then PARDISO generates no output. If MSGLVL = 1 the direct solver prints statistical information to the screen. The multi-recursive iterative solver prints statistical information in the file pardiso-ml.out.

B (N,NRHS) — REAL/COMPLEX Input/Output

On entry: Right-hand side vector/matrix.

On output: The array is replaced by the solution if IPARM(6) = 1.

Note: B is only accessed in the solution phase.

X (N,NRHS) — REAL/COMPLEX Output

On output: Solution if IPARM(6)= 0.

Note: X is only accessed in the solution phase.

ERROR — INTEGER Output

On output: The error indicator

Error	Information
0	No error.
-1	Input inconsistent.
-2	Not enough memory.
-3	Reordering problem.
-4	Zero pivot, numerical fact. or iterative refinement problem.
-5	Unclassified (internal) error.
-6	Preordering failed (matrix types 11, 13 only).
-7	Diagonal matrix problem.
-8	32-bit integer overflow problem.
-10	No license file pardiso.lic found.
-11	License is expired.
-12	Wrong username or hostname.
-100	Reached maximum number of Krylov-subspace iteration in iterative solver.
-101	No sufficient convergence in Krylov-subspace iteration within 25 iterations.
-102	Error in Krylov-subspace iteration.
-103	Break-Down in Krylov-subspace iteration.

DPARM (64) — REAL Input/Output

On entry: DPARM is a double-precision array of size 64 that is used to pass various parameters to PARDISO and to return some useful information after the execution of the iterative solver. The array will be used in the multi-recursive iterative linear solver in PARDISO. This iterative solver is used in case of IPARM(32) = 1. In addition, DPARM returns double return values from the direct solver such as the determinant for symmetric indefinite matrices or unsymmetric matrices.

See section 2.4 for a detailed description.

2.3 IPARM Control Parameters

All parameters are integer values.

IPARM (1) — **Use default values.** Input

On entry: If IPARM(1) = 0, then IPARM(2) and IPARM(4) through IPARM(64) are filled with default values, otherwise the user has to supply all values in IPARM.

Note: If IPARM(1) $\neq 0$ on entry, the user has to supply all values form IPARM(2) to IPARM(64).

IPARM (2) — **Fill-In reduction reordering.** Input

On entry: IPARM(2) controls the fill-in reducing ordering for the input matrix. If IPARM(2) is 0 then the minimum degree algorithm is applied [6], if IPARM(2) is 2, the solver uses the nested dissection algorithm from the METIS package version 4.1 [2]. If IPARM(2) is 3, the solver uses the nested dissection algorithm from the METIS package version 5.1.

The default value of IPARM(2) is 2.

IPARM (3) — **Number of processors.** Input

On entry: IPARM(3) must contain the number of processors that are available for parallel execution. The number must be equal to the OpenMP environment variable `OMP_NUM_THREADS`.

Note: If the user has not explicitly set `OMP_NUM_THREADS`, then this value can be set by the operating system to the maximal numbers of processors on the system. It is therefore always recommended to control the parallel execution of the solver by explicitly setting `OMP_NUM_THREADS`. If fewer processors are available than specified, the execution may slow down instead of speeding up.

There is no default value for IPARM(3).

IPARM (4) — **Preconditioned CGS.** Input

On entry: This parameter controls preconditioned CGS [16] for nonsymmetric or structural symmetric matrices and Conjugate-Gradients for symmetric matrices. IPARM(4) has the form

$$\text{IPARM(4)} = 10 * L + K \geq 0.$$

The values K and L have the following meaning:

Value K:

K	Description
	K can be chosen independently of MTYPE.
	LU was computed at least once before call with IPARM (4) $\neq 0$.
1	LU preconditioned CGS iteration for $A \times X = B$ is executed.
2	LU preconditioned CG iteration for $A \times X = B$ is executed.

Value L: The value L controls the stopping criterion of the Krylov-Subspace iteration:
$\epsilon_{CGS} = 10^{-L}$ is used in the stopping criterion $||dx_i||/||dx_1|| < \epsilon_{CGS}$ with $||dx_i|| = ||(LU)^{-1} r_i||$ and r_i is the residual at iteration i of the preconditioned Krylov-Subspace iteration.

Strategy: A maximum number of 150 iterations is set on the expectation that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and the norm of the residual are checked and can terminate the iteration process. The basic assumption is: LU was computed at least once — hence it can be recomputed if the iteration process converges slowly.
If PHASE=23: the factorization for the supplied A is automatically recomputed in cases where the Krylov-Subspace iteration fails and the direct solution is returned. Otherwise the solution from the preconditioned Krylov Subspace iteration is returned.
If PHASE=33: the iteration will either result in success or an error (ERROR=4), but never in a recomputation of LU. A simple consequence is: (A,IA,JA) may differ from the (A,IA,JA)' supplied to compute LU (MTYPE, N have to be identical). More information on the failure can be obtained from IPARM(20).

Note: The default is IPARM(4)=0 and other values are only recommended for advanced users.

Examples:

IPARM(4)	Examples
31	LU-preconditioned CGS iteration with a stopping tolerance of 10^{-3} for nonsymmetric matrices
61	LU-preconditioned CGS iteration with a stopping tolerance of 10^{-6} for nonsymmetric matrices
62	LU-preconditioned CG iteration with a stopping tolerance of 10^{-6} for symmetric matrices

IPARM (5) — **User permutation.** Input

On entry: IPARM(5) controls whether a user-supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithm.

Note: This option may be useful for testing reordering algorithms or adapting the code to special application problems (e.g. to move zero diagonal elements to the end of PAP^T). The permutation must be a vector of size N. Let A be the original matrix and $B = PAP^T$ be the permuted matrix. The array PERM is defined as follows. Row (column) of A is the PERM(i) row (column) of B. The numbering of the array must start at 1.

The default value of IPARM(5) is 0.

IPARM (6) — **Write solution on X.** Input

On entry: If IPARM(6) is 0, which is the default, then the array X contains the solution and the value of B is not changed. If IPARM(6) is 1 then the solver will store the solution on the right-hand side B.

Note: The array X is always changed.

The default value of IPARM(6) is 0.

IPARM (7) — **Number of performed iterative refinement steps.** Output

On output: The number of iterative refinement steps performed during the solve step are reported in IPARM(7).

IPARM (8) — **Iterative refinement steps.** Input

On entry: On entry to the solve and iterative refinement step, IPARM(8) should be set to the maximum number of iterative refinement steps that the solver will perform. Iterative refinement will stop if a satisfactory level of accuracy of the solution in terms of backward error has been achieved.
The solver will automatically perform two steps of iterative refinement when perturbed pivots have occured during the numerical factorization and IPARM(8) was equal to zero.
The number of iterative refinement steps is reported in IPARM(7).

Note: If IPARM(8) < 0 the accumulation of the residual is using complex*32 or real*16 in case that the compiler supports it (GNU gfortran does not). Perturbed pivots result in iterative refinement (independent of IPARM(8)=0) and the iteration number executed is reported in IPARM(7).

The default value is 0.

IPARM (9) — **Unused.** Input

This value is reserved for future use. Value must be set to 0.

IPARM (10) — **Pivot perturbation.** Input

On entry: IPARM(10) instructs PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices (MTYPE=11, MTYPE=13) and symmetric matrices (MTYPE=-2, MTYPE=-4, or MTYPE=6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factorize the supernodes with

this pivoting strategy, it uses a pivoting perturbation strategy similar to [5, 13], compare **??**. The magnitude of the potential pivot is tested against a constant threshold of $\alpha = \epsilon \cdot ||A_2||_\infty$, where $\epsilon = 10^{-IPARM(10)}$ and $||A_2||_\infty$ is the ∞-norm of the scaled and permuted matrix A. Any tiny pivots l_{ii} encountered during elimination are set to the $\text{sign}(l_{ii}) \cdot \epsilon \cdot ||A_2||_\infty$.

The default value of IPARM(10) is 13 and therefore $\epsilon = 10^{-13}$ for nonsymmetric matrices. The default value of IPARM(10) is 8 and $\epsilon = 10^{-8}$ for symmetric indefinite matrices (MTYPE=-2, MTYPE=-4, or MTYPE=6)

IPARM (11) — **Scaling vectors.** Input

On entry: PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal and to scale the matrix so that the largest elements are equal to 1 and the absolute value of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices (MTYPE=11 or MTYPE=13). The scaling can also be used for symmetric indefinite matrices (MTYPE=-2, MTYPE=-4, or MTYPE=6) if symmetric weighted matchings is applied (IPARM (13)=1 or IPARM (13)=3).

Note: It is recommended to use IPARM(11)=1 (scaling) and IPARM(13)=1 (matchings) for highly indefinite symmetric matrices e.g. from interior point optimizations or saddle point problems. It is also very important to note that the user must provide in the analysis phase (PHASE=11) the numerical values of the matrix A if IPARM(11)=1 (scaling) or PARM(13)=1 or 2 (matchings).

The default value of IPARM(11) is 1 for nonsymmetric matrices (MTYPE=11 or MTYPE=13) and IPARM(11) is 0 for symmetric matrices (MTYPE=-2, MTYPE=-4, or MTYPE=6).

IPARM (12) — **Solving with transpose matrix.** Input

On entry: Solve a system $A^T x = b$ by using the factorization of A.

The default value of IPARM(12) is 0. IPARM(12)=1 indicates that you would like to solve transposed system $A^T x = b$.

IPARM (13) — **Improved accuracy using (non-)symmetric weighted matchings.** Input

On entry: PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to our factorization methods and can be seen as a complement to the alternative idea of using more extensive pivoting techniques during the numerical factorization.

Note: It is recommended to use IPARM(11) = 1 (scalings) and IPARM(13)=1 (normal matchings) and IPARM(13)=2 (advanced matchings, higher accuracy) for highly indefinite symmetric matrices e.g. from interior point optimizations or saddle point problems. Note that the user must provide in the analysis phase (PHASE=11) the numerical values of the matrix A in both of these cases.

The default value of IPARM(13) is 1 for nonsymmetric matrices (MTYPE=11 or MTYPE=13) and 0 for symmetric matrices (MTYPE=-2, MTYPE=-4, or MTYPE=6).

IPARM (14) — **Number of perturbed pivots.** Output

On output: After factorization, IPARM(14) contains the number of perturbed pivots during the elimination process for MTYPE=-2, -4, 6, 11, or 13.

IPARM (15) — **Peak memory symbolic factorization.** Output

On output: IPARM(15) provides the user with the total peak memory in KBytes that the solver needed during the analysis and symbolic factorization phase. This value is only computed in phase 1.

IPARM (16) — **Permanent memory symbolic factorization.** Output

On output: IPARM(16) provides the user with the permanently needed memory — hence allocated

at the end of phase 1 — in KBytes to execute the factorization and solution phases. This value is only computed in phase 1 and IPARM (15) > IPARM (16) holds.

IPARM (17) — **Memory numerical factorization and solution.** Output

On output: IPARM(17) provides the user with the total double precision memory consumption (KBytes) of the solver for the factorization and solution phases. This value is only computed in phase 2.

Note: Total peak memory consumption is

$$\text{peakmem} = \max(\text{IPARM}(15),\ \text{IPARM}(16)+\text{IPARM}(17)).$$

IPARM (18) — **Number nonzeros in factors.** Input/Output

On entry: if IPARM(18) < 0 the solver will report the number of nonzeros in the factors.

On output: The numbers of nonzeros in the factors are returned to the user.

The default value of IPARM(18) is -1.

IPARM (19) — **GFlops of factorization.** Input/Output

On entry: if IPARM(19) < 0 the solver will compute the number of GFlops (10^9) to factor the matrix A. This will increase the reordering time.

On output: Number of GFlops (10^9 operations) needed to factor the matrix A.

The default value of IPARM(19) is 0.

IPARM (20) — **CG/CGS diagnostics.** Output

On output: The value is used to give CG/CGS diagnostics (e.g. the number of iterations and cause of failure):

- IPARM(20) > 0: CGS succeeded and the number of iterations is reported in IPARM(20).
- IPARM(20) < 0: Iterations executed, but CG/CGS failed. The error details in IPARM(20) are of the form:

$$\text{IPARM}(20) = \text{- it\_cgs*10 - cgs\_error.}$$

If PHASE was 23, then the factors L, U have been recomputed for the matrix A and the error flag ERROR should be zero in case of a successful factorization. If PHASE was 33, then ERROR = -4 will signal the failure.

Description of cgs_error:

cgs_error	Description
1	Too large fluctuations of the residual;
2	$\|\|dx_{max\_it\_cgs/2}\|\|$ too large (slow convergence);
3	Stopping criterion not reached at max_it_cgs;
4	Perturbed pivots caused iterative refinement;
5	Factorization is too fast for this matrix. It is better to use the factorization method with IPARM(4) = 0.

IPARM (21) — **Pivoting for symmetric indefinite matrices.** Input

On entry: IPARM (21) controls the pivoting method for sparse symmetric indefinite matrices. If IPARM(21) is 0, then 1×1 diagonal pivoting is used. If IPARM(21) is 1, then 1×1 and 2×2 Bunch-Kaufman pivoting will be used within the factorization process.

Note: It is also recommended to use IPARM(11) = 1 (scaling) and IPARM(13) = 1 or 2 (matchings) for highy indefinite symmetric matrices e.g. from interior point optimizations or saddle point problems.

Bunch-Kaufman pivoting is available for matrices MTYPE=-2, -4, or 6.

The default value of IPARM(21) is 1.

IPARM (22) — **Inertia: Number of positive eigenvalues.** Output

On output:
The number of positive eigenvalues for symmetric indefinite matrices is reported in IPARM(22).

IPARM (23) — **Inertia: Number of negative eigenvalues.** Output

On output:
The number of negative eigenvalues for symmetric indefinite matrices is reported in IPARM(23).

IPARM (24) — **Parallel Numerical Factorization.** Input

On entry: This parameter selects the scheduling method for the parallel numerical factorization. If IPARM(24) = 0 then the solver will use a parallel factorization that has been used in the solver during the last years. IPARM(24) = 1 selects a two-level scheduling algorithms which should result in a better parallel efficiency.

The default value of IPARM(24) is 1.

IPARM (25) — **Parallel Forward/Backward Solve.** Input

On entry: The parameter controls the parallelization for the forward and backward solve. IPARM(25) = 0 indicates that a sequential forward and backward solve is used, whereas IPARM(25) = 1 selects a parallel solve.

The default value of IPARM(25) is 1.

IPARM (26) — **Splitting of Forward/Backward Solve.** Input

On entry: The user can apply a partial forward and backward substitution with this parameter. IPARM(26) = 0 indicates that a normal solve step is performed with both the factors L and U. With unsymmetric matrix IPARM(26) = 1 indicates that a forward solve step is performed with the factors L or U^T. IPARM(26) = 2 indicates that a backward solve step is performed with the factors U or L^T — compare IPARM(12). With symmetric matrix IPARM(26) = 1 or -1 indicates that a forward solve step is performed with the factors L or U^T. IPARM(26) = -2 indicates that a backward solve step is performed with the factor D. IPARM(26) = -3 indicates that a backward solve step is performed with the factors U or L^T. IPARM(26) = 2 indicates that a backward solve step is performed with the factors D and U or L^T — compare IPARM(12).

The default value of IPARM(26) is 0.

IPARM (27) — **Unused.** Input

This value is reserved for future use. Value must be set to 0.

IPARM (28) — **Parallel Reordering for METIS.** Input

On entry: The parameter controls the parallel execution of the METIS reordering. IPARM(28) = 1 indicates that a parallel reordering is performed, IPARM(28) = 0 indicates a sequential reordering.

The default value of IPARM(28) is 0.

IPARM (29) — **Switch between 32-bit and 64-bit factorization.** Input

On entry: The parameter controls the IEEE accuracy of the solver IPARM(29) = 0 indicates that a 64-bit factorization is performed and IPARM(29) = 1 indicates a 32-bit factorization.

The default value of IPARM(29) is 0. The option is only available for symmetric indefinite matrices (MTYPE=-2) and for real unsymmetric matrices (MTYPE=11).

IPARM (30) — **Control the size of the supernodes.** Input

On entry: The parameter controls the column size of the supernodes, e.g., IPARM(30) = 40 indicates that 40 columns are used for each supernode.

The default value of IPARM(30) is 80 for symmetric matrices.

IPARM (31) — **Partial solve for sparse right-hand sides and sparse solution.** Input

On entry: The parameter controls the solution method in case that the user is interested in only a few components of the solution vector X. The user has to define the vector components in the input vector PERM. An entry PERM(i)=1 means that either the right-hand side is nonzero at this component, and it also defines that the solution i-th component should be computed in the vector X. The vector PERM must be present in all phases of PARDISO.

The default value of IPARM(31) is 0. An entry equal IPARM(31)=1 selects the sparse right-hand sides solution method.

IPARM (32) — **Use the multi-recursive iterative linear solver.** Input

On entry: The default value of IPARM(32) is 0 (sparse direct solver) and IPARM(32)=1 will use the iterative method in PARDISO.

IPARM (33) — **Determinant of a matrix.** Input

On entry: IPARM(33)=1 will compute the determinant of matrices and will return in DPARM(33) the real part of the determinant and, if necessary, in DPARM(32) the complex part of the determinant.
On output: The parameter returns the natural logarithm of the determinant of a sparse matrix A.

The default value of IPARM(33) is 0.

IPARM (34) — **Identical solution independent on the number of processors.** Input

On entry: IPARM(34)=1 will always compute the identical solution independent on the number of processors that PARDISO is using during the parallel factorization and solution phase. This option is only working with the METIS reordering (IPARM(2)=2), and you also have to select a parallel two-level factorization (IPARM(24)=1) and a parallel two-level solution phase (IPARM(25)=1). In addition, the user has define the upper limit of processors in IPARM(3) for which he would like to have the same results.

Example IPARM(3)=8, IPARM(2)=2, and IPARM(34)=IPARM(24)=IPARM(25)=1: This option means that the user would like to have identical results for up to eight processors. The number of cores that the user is actually using in PARDISO can be set by the OpenMP environment variable `OMP_NUM_THREADS`. In our example, the solution is always the same for `OMP_NUM_THREADS`=1 up to `OMP_NUM_THREADS`=IPARM(3)=8.

IPARM (35) — **Unused.** Output

This value is reserved for future use. Value MUST be set to 0.

IPARM (36) — **Selected inversion for A^{-1}_{ij}.** Input

On entry: IPARM(36) will control the selected inversion process based on the internal L and U factors. If IPARM(36) = 0 and PHASE = -22, PARDISO will overwrite these factors with the selected inverse elements of A^{-1}. If IPARM(36) = 1 and PHASE = -22, PARDISO will

not overwrite these factors. It will instead allocate additional memory of size of the numbers of elements in L and U to store these inverse elements.

The default value of IPARM(36) is 0.

The following code fragment shows how to use the selected inversion process in PARDISO from a Fortran program

```
c...Reorder and factor a matrix sytem
PHASE=12
CALL PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE,N, A, IA, JA,
PERM, NRHS, IPARM, MSGLVL, B, X, ERROR, DPARM)

c...Compute selected inversion process and return inverse elements
c...in CSR structure (a,ia,ja)
PHASE=-22
IPARM(36)=0
CALL PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE,N, A, IA, JA,
PERM, NRHS, IPARM, MSGLVL, B, X, ERROR, DPARM)
```

IPARM (37) — **Selected inversion for A_{ij}^{-1}.** Input

On entry: IPARM(37) will control the selected inversion process for symmetric matrices. If IPARM(37) = 0, PARDISO will return the selected inverse elements A_{ij}^{-1} in upper symmetric triangular CSR format. If IPARM(37) = 1, PARDISO will return the selected inverse elements A_{ij}^{-1} in full symmetric triangular CSR format.

The default value of IPARM(37) is 0.

IPARM (38) — **Schur-complement computation.** Input

On entry: IPARM(38) will control the Schur-complement computation and it will compute the partial Schur complement $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ for the matrix $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$. IPARM(38) indicates the numbers of rows/columns size of the matrix A_{22}.

The default value of IPARM(38) is 0.

The following code fragment shows how to use the Schur-complement method in PARDISO from a C++ program to compute a Schur-complement of size 5.

```
// Reorder and factor a matrix sytem
int phase = 12;
int nrows = 5;
iparm[38-1] = nrows;
phase = 12;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, &idum, &nb, &iparm[0],
&msglvl, &ddum, &ddum, &error, dparm);
// allocate memory for the Schur-complement and copy it there.
int nonzeros = iparm[39-1];
int* iS = new int[nrows+1];
int* jS = new int[nonzeros];
double* S = new double[nonzeros];
pardiso_get_schur(pt, &maxfct, &mnum, &mtype, S, iS, jS);
```

IPARM (39) — **Nonzeros in Schur-complement $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$.** Output

On Output: The numbers of nonzeros in the Schur-complement $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$.

IPARM (40) to IPARM(49) — **Unused.** Output

These values are reserved for future use. Value MUST be set to 0.

IPARM (50) — **Use of out-of-core solver option.** Input

On Input: IPARM(50)=0, then PARDISO will always use the in-core version. IPARM(50) defines the number of submatrices that are written to files with names 'pardiso-factor-diag.mat', 'small-ooc-domain', 'pardiso-ooc-domain', and 'pardiso-sep-domain'. The number of files is used as $2^{\text{iparam}(50)}$ in the solver. This option is only available for real symmetric indefinite matrices (MTYPE=-2) and complex symmetric matrices (MTYPE=6).

The default value of IPARM(50) is 0.

IPARM (51) — **Use parallel distributed-memory solver.** Input

On Input: IPARM(51)=1 will always use the distributed MPI-based parallel numerical factorization and parallel forward-backward substitution.

The default value of IPARM(51) is 0.

IPARM (52) — **Number of compute nodes for the distributed-memory parallel solver.** Input

On Input: IPARM(52) defines the number of compute nodes (e.g. an eight core Intel Nehalem node) that are used with the distributed MPI-based parallel numerical factorization and parallel forward-backward substitution.

The default value of IPARM(52) is 1.

Example IPARM(52)=2 (number of compute nodes), IPARM(3)=8 (number of cores on each node): This option means that the user would like to use 16 cores in total for the distributed-memory parallel factorization.

IPARM (53) — **Block size.** Input

On Input: This parameter sets the block size of block matrices.

- IPARM(53)= 0: Normal CSR matrix format is expected on input, block compression is not used
- IPARM(53)$\geq$ 1: Block CSR matrix format is expected on input, IPARM(53) is the block size, block compression is used

The default value of IPARM(53) is 0.

IPARM (54) — **Dense columns.** Input

On Input: This parameter sets the number of dense columns at the end of the matrix.

- IPARM(54) = 0: Dense column compression is not used
- IPARM(54) > 1: IPARM(54) dense columns are expected at the end of the matrix, dense column compression is used

The default value of IPARM(54) is 0.

IPARM (55) to IPARM(65) — **Unused.** Output

These values are reserved for future use. Value MUST be set to 0.

2.4 DPARM Control Parameters

All parameters are 64-bit double values. These parameters are only important for the multirecursive iterative solver (except DPARM(33) which returns the determinant of symmetric real indefinite matrices). A log-file with detailed information of the convergence process is written in pardiso-ml.out. The initial guess for the solution has to be provided in the solution vector x. A vector x initialized with zeros is recommended.

DPARM (1) — **Maximum number of Krylov-Subspace Iteration.** Input

On entry: This values sets the maximum number of SQMR iteration for the iterative solver.

The default value of DPARM(1) is 300.

DPARM (2) — **Relative Residual Reduction.** Input

On entry: This value sets the residual reduction for the SQMR Krylov-Subspace Iteration.

The default value of DPARM(2) is 1e-6.

DPARM (3) — **Coarse Grid Matrix Dimension .** Input

On entry: This value sets the matrix size at which the multirecursive method switches to the sparse direct method.

The default value of DPARM(3) is 5000.

DPARM (4) — **Maximum Number of Grid Levels .** Input

On entry: This value sets the maximum number of algebraic grid levels for the multirecursive iterative solver,

The default value of DPARM(4) is 10.

DPARM (5) — **Dropping value for the incomplete factor .** Input

On entry: This value sets the dropping value for the incomplete factorization method.

The default value of DPARM(5) is 1e-2.

DPARM (6) — **Dropping value for the schurcomplement .** Input

On entry: This value sets the dropping value for the schurcomplement.

The default value of DPARM(4) is 5e-3.

DPARM (7) — **Maximum number of Fill-in in each column in the factor.** Input

On entry: This value sets the maximum number of fill-in in each column in the factor compared to the original matrix.

The default value of DPARM(7) is 10.

DPARM (8) — **Bound for the inverse of the incomplete factor L.** Input

On entry: This value sets a numerical bound κ for the inverse of the incomplete factor.

The default value of DPARM(8) is 500.

DPARM (9) — **Maximum number of stagnation steps in Krylov-Subspace method.** Input

On entry: This value sets the maximum number of non-improvement steps in Krylov-Subspace method.

The default value of DPARM(9) is 25.

DPARM (10) to DPARM(32) — **Unused.** Output

These values are reserved for future use. Value MUST be set to 0.

DPARM (32) — **Determinant for complex unsymmetric matrices.** Output

On output: This value returns the complex value of a determinant for unsymmetric matrices. The is only computed in case of IPARM (33)=1

DPARM (33) — **Determinant for real matrices.** Output

On output: This value returns the determinant for real symmetric indefinite matrices. The is only computed in case of IPARM (33)=1

DPARM (34) — **Relative residual after Krylov-Subspace convergence.** Output

On output: This value returns the relative residual after Krylov-Subspace convergence. The is only computed in case of IPARM (32)=1

DPARM (35) — **Number of Krylov-Subspace iterations.** Output

On output: This value returns the number of Krylov-Subspace iterations. The is only computed in case of IPARM (32)=1

DPARM (36) to DPARM(64) — **Unused.** Output

These values are reserved for future use. Value MUST be set to 0.

2.5 Summary of arguments and parameters

```
    CALL PARDISOINIT(PT, MTYPE, SOLVER, IPARM, DPARM, ERROR)
    CALL PARDISO(PT, MAXFCT, MNUM, MTYPE, PHASE, N, A, IA, JA,
  1              PERM, NRHS, IPARM, MSGLVL, B,  X, ERROR, DPARM)
    CALL PARDISO_GET_SCHUR(PT, MAXFCT, MNUM, MTYPE, S, IS, JS)
    CALL PARDISO_RESIDUAL(MTYPE, N, A, IA, JA, B, X, Y, NORM_B, NORM_RES)

    pardisoinit (void *pt[64], int *mtype, int *solver, int iparm[64],
                 double dparm[64], int *error);
    pardiso (void *pt[64], int *maxfct, int *mnum, int *mtype,
             int *phase, int *n, double a[], int ia[], int ja[],
             int perm[], int *nrhs, int iparm[64], int *msglvl,
             double b[], double x[], int *error, double dparm[64] );
    pardiso_get_schur(pt, &maxfct, &mnum, &mtype, S, iS, jS);
    pardiso_residual (&mtype, &n, a, ia, ja, b, x, y, &norm_b, &norm_res);
```

2.6 Computing the Schur-complement: Example for nonsymmetric matrices

Suppose the system of linear equations has the form

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

The Schur complement matrix $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ can be returned to the user (see IPARM (39) and IPARM (39)). PARDISO provides a partial factorization of the complete matrix and returns the Schur matrix in user memory. The Schur matrix is considered as a sparse matrix and PARDISO will return matrix in sparse CSR format to the user. The partial factorization that builds the Schur matrix can also be used to solve linear systems $A_{11}x_1 = b_1$ associated with the A_{11} block. Entries in the right-hand side corresponding to indices from the Schur matrix need not be set on entry and they are not modified on output. The following code fragment shows how to use the Schur-complement method in PARDISO from a C++ program to compute a Schur-complement.

```
// Reorder and partial factorization of the matrix sytem
int phase    = 12;
iparm[38-1]  = xxx; // size of the A_22 block
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja,
        &idum, &nb, &iparm[0], &msglvl, &ddum, &ddum, &error,  dparm);

// Schur-complement matrix S=(S, is, jS) in CSR format.
int nonzeros = iparm[39-1];
int* iS        = new int[nrows+1];
int* jS        = new int[nonzeros];
double* S      = new double[nonzeros];
pardiso_get_schur(pt, &maxfct, &mnum, &mtype, S, iS, jS);

// solve with A_11x1= b1;
int phase    = 33;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja,
        &idum, &nb, &iparm[0], &msglvl, &b1, &x1, &error,  dparm);
```

Table 1: Overview of subroutine arguments.

Name	Type	Description	Input/Output
PT (64)	INT	Internal memory address pointer.	I/O
MAXFCT	INT	Number of numerical factorizations in memory.	I
MNUM	INT	Actual matrix to factorize.	I
MTYPE	INT	Matrix type.	I
	1	real and structurally symmetric, supernode pivoting	
	2	real and symmetric positive definite	
	-2	real and symmetric indefinite, diagonal or Bunch-Kaufman pivoting	
	11	real and nonsymmetric, complete supernode pivoting	
	3	complex and structurally symmetric, supernode pivoting	
	4	complex and hermitian positive definite	
	-4	complex and hermitian indefinite, diagonal or Bunch-Kaufman pivoting	
	6	complex and symmetric	
	13	complex and nonsymmetric, supernode pivoting	
PHASE	INT	Solver Execution Phase.	I
	11	Analysis	
	12	Analysis, Numerical Factorization	
	13	Analysis, Numerical Factorization, Solve, Iterative Refinement	
	22	Numerical Factorization	
	-22	Selected Inversion	
	23	Numerical Factorization, Solve, Iterative Refinement	
	33	Solve, Iterative Refinement	
	-1	Release all internal memory for all matrices	
	0	Release memory for matrix number MNUM	
N	INT	Number of equations.	I
A (∗)	R/C	Matrix values.	I
IA (N+1)	INT	Beginning of each row.	I
JA (∗)	INT	Column indices.	I
PERM (N)	INT	User permutation.	I
NRHS	INT	Number of right-hand sides.	I
IPARM (64)	INT	Control parameters.	I/O
MSGLVL	INT	Message level.	I
	0	No output.	
	1	Output statistical information	
B (N, NRHS)	R/C	Right-hand sides.	I/O
X (N, NRHS)	R/C	Solution vectors (see IPARM(6)).	O
ERROR	INT	Error indicator.	O
DPARM (64)	REAL	Control parameters for iterative solver	I/O

Table 2: Overview of subroutine arguments.

Name	Description	
IPARM(1)	Use default options.	
	0	Set all entries to their default values *except* IPARM(3).
IPARM(2)	Use METIS reordering.	
	0	Do not use METIS.
	2*	Use METIS nested dissection reordering.
	3	Use METIS51 nested dissection reordering.
IPARM(3)	Number of processors.	
	p	Number of OpenMP threads. This *must* be identical or slightly larger to the environment variable `OMP_NUM_THREADS`.
IPARM(4)	Do preconditioned CGS iterations (see description). Default is 0.	
IPARM(5)	Use user permutation.	
	0*	Do not use user perm.
	1	Use the permutation provided in argument PERM.
IPARM(6)	Solution on X / B	
	0*	Write solution to X
	1	Write solution to B
IPARM(8)	Max. numbers of iterative refinement steps.	
	k=0*	Do at most k steps of iterative refinement for all matrices.
IPARM(10)	eps pivot (perturbation 10^{-k}).	
	13*	Default for nonsymmetric matrices
	8*	Default for symmetric indefinite matrices
IPARM(11)	Use (non-) symmetric scaling vectors.	
	0	Do not use .
	1*	Use (nonsymmetric matrices).
	0*	Do not use (symmetric matrices).
IPARM(12)	Solve transposed matrix.	
	0*	Do normal solve.
	1	Do solve with transposed matrix.
IPARM(13)	Improved accuracy using (non-)symmetric matchings	
	0	Do not use.
	1*	Use (nonsymmetric matrices).
	2	Use a very robust method for symmetric indefinite matrices.
	0*	Do not use (symmetric matrices).
IPARM(18)	Number of nonzeros in LU.	
	0	Do not determine.
	−1 *	Will only be determined if −1 on entry.
IPARM(19)	Gflops for LU factorization.	
	0 *	Do not determine.
	−1	Will only be determined if −1 on entry. Increases ordering time.

Table 3: Overview of input IPARM control parameters. An asterisk (*) indicates the default value.

Name	Description	
IPARM(21)	Pivoting for symmetric indefinite matrices. Default is 1.	
	0	1×1 Diagonal Pivoting
	1*	1×1 and 2×2 Bunch-Kaufman pivoting.
IPARM(24)	Parallel Numerical Factorization	
	0	Do one-level parallel scheduling.
	1*	Do two-level parallel scheduling.
IPARM(25)	Parallel Forward/Backward Solve	
	0	Do sequential solve.
	1*	Do parallel solve.
IPARM(26)	Partial Forward/Backward Solve	
	0*	Do forward/backward solve with L and U.
	1	Do forward solve with L or U^T.
	2	Do backward solve with U or L^T.
IPARM(28)	Parallel METIS reordering	
	0*	Do sequential METIS reordering.
	1	Do parallel METIS reordering.
IPARM(29)	32-bit/64-bit IEEE accuracy	
	0*	Use 64-bit IEEE accuracy.
	1	Use 32-bit IEEE accuracy.
IPARM(30)	Control size of supernodes	
	0*	Use default configuration.
	1	Use use configuration.
IPARM(31)	Partial solve for sparse right-hand sides and sparse solutions	
	0*	Compute all components in the solution vector.
	1	Compute only a few selected in the solution vector.
IPARM(32)	Use the multi-recursive iterative linear solver	
	0*	Use sparse direct solver.
	1	Use multi-recursive iterative solver.
IPARM(33)	Determinant for a matrices	
	0*	Do not compute determinant.
	1	Compute determinant.
IPARM(34)	Identical solution independent on the number of processors	
	0*	No identical parallel results.
	1	Identical parallel results

Table 4: Overview of input IPARM control parameters. An asterisk (*) indicates the default value.

Name	Description	
IPARM(36)	Selected inversion for A_{ij}^{-1}	
	0*	Overwrite internal factor with inverse elements.
	1	Do not overwrite internal factor with inverse elements.
IPARM(37)	Selected inversion for A_{ij}^{-1} for symmetric matrices	
	0*	Return inverse elements in upper triangular symmetric compressed CSR format (1-index based).
	1	Return inverse elements in full symmetric compressed CSR format (1-index based).
IPARM(38)	Schur-complement computation	
	0*	Indicates the numbers of rows/columns in S.
	k	Schur-complement matrix S is a $k\times$ matrix.
IPARM(50)	Use out-of-core solver	
	0*	Use in-core solver
	> 0	Use out-of-core solver
IPARM(51)	Use parallel distributed-memory solver	
	0*	Use OpenMP-threaded solver
	1	Use Mixed OpenMP-MPI solver
IPARM(52)	Number of nodes for distributed-memory solver	
	1*	For OpenMP-threaded solver
	p	Use p compute nodes
IPARM(53)	Block size of Compression	
	0*	Normal CSR matrix format is expected on input, block compression is not used
	p	Use p are the block size, block compression is used
IPARM(54)	This parameter sets the number of dense columns for compression	
	0*	Dense column compression is not used
	p	Use p dense columns at the end of the matrix, dense column compression is used

Table 5: Overview of input IPARM control parameters. An asterisk (*) indicates the default value.

Name	Description	
IPARM(7)	Number of iterative refinement steps.	
IPARM(14)	Number of perturbed pivots.	
IPARM(15)	Peak Memory in KBytes during analysis.	
IPARM(16)	Permanent Memory in KBytes from analysis that is used in phases 2 and 3.	
IPARM(17)	Peak Double Precision Memory in KBytes including one LU Factor.	
IPARM(18)	Number of nonzeros in LU.	
	0	Do not determine.
	-1 *	Will only be determined if -1 on entry.
IPARM(19)	Gflops for LU factorization.	
	0 *	Do not determine.
	-1	Will only be determined if -1 on entry. Increases ordering time.
IPARM(20)	Numbers of CG Iterations.	
IPARM(22)	Number of positive eigenvalues for symmetric indefinite systems.	
IPARM(23)	Number of negative eigenvalues for symmetric indefinite systems.	
IPARM(39)	Number of nonzeros in Schur-complement matrix S	
DPARM(32)	Determinant (complex value) for an unsymmetric matrices.	
DPARM(33)	Determinant for real matrices.	
DPARM(34)	Relative residual after Krylov-Subspace convergence.	
DPARM(35)	Number of Krylov-Subspace iterations.	

Table 6: Overview of output IPARM/DPARM control parameters. An asterisk (*) indicates the default value.

Name	Description
IPARM(1)	Use default options.
IPARM(2)	Fill-in reduction reordering (METIS).
IPARM(3)	Number of processors.
IPARM(4)	Preconditioned CGS / CG iterations.
IPARM(5)	Use user permutation.
IPARM(6)	Solution on X / B.
IPARM(7)	Executed number of iterative refinement steps.
IPARM(8)	Max. numbers of iterative refinement steps.
IPARM(9)	For future use.
IPARM(10)	Size of Pivot perturbation
IPARM(11)	Use (non-) symmetric scaling vectors.
IPARM(12)	Solve transposed matrix.
IPARM(13)	Use (non-)symmetric matchings
IPARM(14)	Number of perturbed pivots.
IPARM(15)	Peak memory [kB] phase 1.
IPARM(16)	Permanent integer memory [kb].
IPARM(17)	Peak real memory [kB] including one LU.
IPARM(18)	Number of nonzeros in LU.
IPARM(19)	Gflops for LU factorization.
IPARM(20)	Numbers of CG Iterations.
IPARM(21)	Pivoting for symmetric indefinite matrices.
IPARM(22)	Number of positive eigenvalues for symmetric indefinite systems
IPARM(23)	Number of negative eigenvalues for symmetric indefinite systems
IPARM(24)	Scheduling type for numerical factorization.
IPARM(25)	Parallel forward/backward solve.
IPARM(26)	Partial forward/backward solve.
IPARM(27)	For future use.
IPARM(28)	Parallel METIS reordering.
IPARM(29)	32-bit/64-bit IEEE accuracy.
IPARM(30)	Size of a panel in supernodes.
IPARM(31)	Partial solve for sparse right-hand sides and sparse solutions.
IPARM(32)	Use the multi-recursive iterative linear solver.
IPARM(33)	Compute determinant of a real symmetric indefinite systems.
IPARM(34)	Identical solution independent on the number of processors.
IPARM(35)	Internal use.
IPARM(36)	Selected inversion for A_{ij}^{-1}.
IPARM(37)	Selected inversion for A_{ij}^{-1} for symmetric matrices.
IPARM(38)	Schur-complement computation.
IPARM(39)	Number of nonzeros in Schur-complement matrix S.
IPARM(50)	Use out-of-core solver.
IPARM(51)	Use parallel distributed-memory solver.
IPARM(52)	Number of nodes for distributed-memory solver.
IPARM(53)	Block size of compression.
IPARM(54)	Number of dense columns for compression.

Table 7: Overview IPARM control parameters.

Name	Description	
DPARM(1)	Maximum number of Krylov-Subspace Iteration.	
	300*	Default. $1 \leq DPARM(1) \leq 500$
DPARM(2)	Relative Residual Reduction.	
	$1e-6$*	Default. $1e-1 \geq DPARM(2) \geq 1e-12$
DPARM(3)	Coarse Grid Matrix Dimension.	
	$1e-6$*	Default. $1e-1 \geq DPARM(3) \leq ge-12$
DPARM(4)	Maximum Number of Grid Levels.	
	10*	Default. $1 \leq DPARM(4) \leq 1000$
DPARM(5)	Dropping value for the incomplete factor.	
	1e-2*	Default. $1e-16 \leq DPARM(5) \leq 1$
DPARM(6)	Dropping value for the schurcomplement.	
	5e-5*	Default. $1e-16 \leq DPARM(6) \leq 1$
DPARM(7)	Maximum number of fill-in in each column in the factor.	
	10*	Default. $1 \leq DPARM(7) \leq 1000000000$
DPARM(8)	Bound for the inverse of the incomplete factor L.	
	500*	Default. $1 \leq DPARM(9) \leq 1000000000$
DPARM(9)	Maximum number of non-improvement steps in Krylov-Subspace method	
	25*	Default. $1 \leq DPARM(9) \leq 1000000000$
DPARM(32)	Returns the determinant (complex value) for complex unsymmetric matrix.	
DPARM(33)	Returns the determinant fo a real matrix.	
DPARM(34)	Relative residual after Krylov-Subspace convergence.	
DPARM(35)	Number of Krylov-Subspace iterations.	

Table 8: Overview of DPARM control parameters. An asterisk (*) indicates the default value.

3 Linking and Running the Software

A general remark for Unix operating systems is that the environment variable `LD_LIBRARY_PATH` must be pointing to the PARDISO library. The environment variable `OMP_NUM_THREADS` must be set in all cases of using the multi-threaded library version.

The software can be downloaded from the PARDISO page at `http://www.pardiso-project.org`. A license file *pardiso.lic* is required for the execution of the solver and it will be generated at the download area of the web page. The user must supply the user identification (Unix: "`whoami`", Windows: username exactly as Windows has it). This license file must always be present either in the home directory of the user or in the directory from which the user runs a program linked with the PARDISO library. Alternatively, this file can be placed in a fixed location and the environment variable `PARDISO_LIC_PATH` must be set to the path of its location.

3.1 LINUX on IA64 platform

The current version of PARDISO requires that either the following GNU or Intel compilers are installed on your system:

1. GNU Fortran *gfortran* compiler version 4.6.3 or higher for 64-bit mode
2. GNU C compiler *gcc* version 4.6.3 or higher for 64-bit mode

or

1. INTEL Fortran compiler *ifort* version 13.0.1 for 64-bit mode
2. INTEL C compiler *icc* version 13.0.1 for 64-bit mode

3.1.1 Linking the libraries

The libraries contain C and Fortran routines and can be linked with GNU *gfortran or gcc*, or Intel *ifort or icc*. Currently, the following libraries are available for LINUX IA32 and IA64 platforms:

- **libpardiso500-GNU463-X86-64.so** for linking with GNU compilers version 4.6.3 on IA64 in 64-bit mode.
- **libpardiso500-INTEL1301-X86-64.so** for linking with Intel compilers version 13.0.1 on IA64 in 64-bit mode.

Here are some examples of linking user code with the PARDISO solver on 64-bit LINUX architectures. The user may need to add additional flags and options as required by the user's code.

1. GNU version 4.6.1, 64-bit mode:

```
gfortran <source/objects files> -o <executable>
         -L <Path to directory of PARDISO> -lpardiso500-GNU461-X86-64
         -L <Path to directory of LAPACK/BLAS>
         -l <fast LAPACK and BLAS libraries> -fopenmp -lpthread -lm

gcc <source/objects files> -o <executable>
    -L <Path to directory of PARDISO> -lpardiso500-GNU461-X86-64
    -L <Path to directory of LAPACK/BLAS>
    -l <Fast LAPACK and BLAS libraries> -lgfortran -fopenmp -lpthread -lm
```

2. INTEL version 13.0.1, 64-bit mode:

```
ifort <source/objects files> -o <executable>
      -L <Path to directory of PARDISO> -lpardiso500-INTEL1301-X86-64
      -L <Path to directory of LAPACK/BLAS>
      -l <fast LAPACK and BLAS libraries>
      -L/opt/intel/mkl/lib/intel64 -lmkl_gf_lp64 -lmkl_sequential
      -lmkl_core -lm -lgfortran -fopenmp

icc <source/objects files> -o <executable>
    -L <Path to directory of PARDISO> -lpardiso500-INTEL1301-X86-64
    -L <Path to directory of LAPACK/BLAS>
    -l <Fast LAPACK and BLAS libraries>
    -L <Path to INTEL System libraries>
    -L/opt/intel/mkl/lib/intel64 -lmkl_gf_lp64 -lmkl_sequential
    -lmkl_core -lm -lgfortran -fopenmp
```

3.1.2 Setting environment

The user can control the parallel execution of the PARDISO solver with the OpenMP environment variable `OMP_NUM_THREADS`. It is very important for parallel execution that the input parameter IPARM(3) be set to the corresponding correct value of `OMP_NUM_THREADS`. The environment variable `LD_LIBRARY_PATH` must point to the PARDISO library. It is also recommended to set the Intel OpenMP environment variable to `MKL_SERIAL=YES`.

3.2 Mac OSX on 64-bit platforms

The current version of PARDISO requires that the following GNU compilers be installed on your system:

1. GNU C compiler *gcc* version 4.7.1 or higher for 64-bit mode

3.2.1 Linking the libraries

The libraries contain C and Fortran routines and can be linked with the GNU *gfortran or gcc* compilers. Currently, the following library is available:

- Parallel **libpardiso500-MACOS-X86-64.dylib** for linking with GNU compilers in 64-bit mode.

Here are some examples of linking user code with the PARDISO solver on Mac OSX. The user may need to add additional flags and options as required by the user's code.

1. GNU 64-bit mode, sequential:

```
gfortran <source/objects files> -o <executable>
         -L <Path to directory of PARDISO> -lpardiso500-MACOS-X86-64.dylib
         -L <Path to directory of LAPACK/BLAS>
         -l <fast LAPACK and BLAS libraries> -liomp5.dylib

gcc <source/objects files> -o <executable>
    -L <Path to directory of PARDISO> -lpardiso500-MACOS-X86-64.dylib
    -L <Path to directory of LAPACK/BLAS>
    -l <Fast LAPACK and BLAS libraries> -liomp5.dylib
```

3.2.2 Setting environment

The user can control parallel execution of the PARDISO solver with the OpenMP environment variable `OMP_NUM_THREADS`. It is very important for the parallel execution that the input parameter IPARM(3) be set to the corresponding correct value of `OMP_NUM_THREADS`. The environment variable `LD_LIBRARY_PATH` must point to the PARDISO library.

3.3 Windows on IA64 platforms

The current version of PARDISO requires the Intel compilers and Microsoft Visual Studio 2012 compilers are installed on your system. This might require that the machine PARDISO runs on has VS2K8 installed (or the Windows SDK for Windows Server 2012).

3.3.1 Linking the libraries

The Windows libraries comes with a library and associated import library (DLL, LIB), contain C and Fortran routines and the following libraries are available for Windows IA64 platforms:

- Parallel **libpardiso500-WIN-X86-64.[dll, lib]** for linking with Microsoft cl compilers on Intel64 in 64-bit mode.

Here are some examples of linking user code with the PARDISO solver on 64-bit Windows architectures. The user may need to add additional flags and options as required by the user's code.

1. This 64-bit Windows version comes internally with optimized BLAS/LAPACK objects from the Intel MKL library Version 10.2

```
cl <source/objects files> -o <executable>
   -L <Path to directory of PARDISO> -lpardiso500-WIN-X86-64.lib
```

4 Using the MATLAB interface

This section illustrates how to use the MATLAB interface.[2] Its usage very much follows the conventions described in previous sections, so I assume the reader has read these sections prior to using the MATLAB interface. For instructions on how to install this package, see the section below.

4.1 An example with a real matrix

Suppose you have a real, sparse, matrix `A` in MATLAB, as well as a real, dense matrix `B`, and you would like to use PARDISO to find the solutions `X` to the systems of linear equations

```
A * X = B
```

The first step is to initialize PARDISO by running the following command in the MATLAB prompt:

```
info = pardisoinit(11,0);
```

This tells PARDISO that we would like to solve systems involving real, sparse non-symmetric matrices, and using the sparse direct solver. The general call looks like

```
info = pardisoinit(mtype,solver);
```

where `mtype` and `solver` correspond to the arguments to PARDISOINIT defined in Sec. 2.1. You will notice that the variable `info` contains the following fields: the matrix type (`mtype`), the arrays `iparm` and `dparm` storing PARDISO control parameters (see Sections 2.3 and 2.4), and a field which keeps track the location in memory where the PARDISO data structures are stored. Do not modify this value!

The second step is to create a symbolic factorization of the matrix via

```
info = pardisoreorder(A,info,verbose);
```

where `verbose` is either `true` or `false`. Then we can ask PARDISO to factorize the matrix:

```
info = pardisofactor(A,info,verbose);
```

[2]This section and the next section were contributed by Peter Carbonetto, a Ph.D. student at the Dept. of Computer Science, University of British Columbia.

Once we've factorized the sparse matrix A, the last step is to compute the solution with the following command:

```
[X, info] = pardisosolve(A,B,info,verbose);
```

The return value `X` should now contain the solutions to the systems of equations `A * X = B`. You can now use the same factorization to compute the solution to linear systems with different right-hand sides. Once you do not need the factorization any longer, you should deallocate the PARDISO data structures in memory with the commands

```
pardisofree(info);
clear info
```

4.2 A brief note of caution

In a sparse matrix, MATLAB only stores the values that are not zero. However, in PARDISO it is important that you also provide it with the diagonal entries of a matrix, *even if those entries are exactly zero*. This is an issue that commonly arises when using PARDISO to solve the linear systems that arise in primal-dual interior-point methods. The solution is then to add a very small number to the diagonal entries of the matrix like so:

```
A = A + eps * speye(size(A));
```

4.3 An example with a real, symmetric matrix

Now suppose that your sparse matrix `A` is symmetric, so that `A == A'`. You must now initialize PARDISO with a different matrix type like so:

```
info = pardisoinit(-2,0);
```

Let's further suppose that you think you can obtain a factorization more efficiently with your own customized reordering of the rows and columns of of the $n \times n$ matrix `A`. This reordering must be stored in a vector `p` of length n, and this vector must be a permutation of the numbers 1 through n. To complete the numeric factorization, you would then issue the following commands in the MATLAB prompt:

```
info = pardisoreorder(tril(A),info,verbose,p);
info = pardisofactor(tril(A),info,verbose);
```

Notice that you must only supply the *lower triangular* portion of the matrix. (Type `help tril` in MATLAB for more information on this command.) If you supply the entire matrix, MATLAB will report an error.

4.4 Complex matrices

The MATLAB interface also works with complex matrices (either symmetric, Hermitian or non-symmetric). In all these cases, it is important that you initialize PARDISO to the correct `mtype`. Another thing you must do is make sure that both the sparse matrix `A` and the dense matrix `B` are complex. You can check this with the `iscomplex` command. If all the entries of `B` are real, you can can convert `B` to a complex matrix by typing

```
B = complex(B);
```

5 Installing the MATLAB interface

The MATLAB interface for PARDISO was created using the MATLAB external ("MEX") interface. For more information on the MEX interface, consult the MathWorks website. In order to be able to use this interface in MATLAB, you will first have to compile the MEX files on your system.

For those who have the program `make`, the simplest thing to do is edit the Makefile and type `make all` in the command prompt. (Note that the Makefile was written for those who are running Linux. For other operating systems, you may have to modify the Makefile to suit your setup.) For those of you who do not have `make`, you will have to call the `mex` command directly to compile the MEX object files.

At the top of the Makefile, there are four variables you will need to change to suit your system setup. The variable `MEXSUFFIX` must be the suffix appended to the MEX files on your system. Again, see the MathWorks website for details. The variable `MEX` must point to the `mex` executable located somewhere within your MATLAB installation directory. The variable `CXX` must point to your C++ compiler. And the variable `PARDISOHOME` must point to the directory where the PARDISO library is stored.

Note that it is crucial that your C++ compiler be compatible with your MATLAB installation. In particular, the compiler must also be of the correct version. The MathWorks website keeps a detailed list of MATLAB software package versions and the compilers that are compatible with them.

If you are not using the Makefile provided, to compile, say, the MEX File for the command `pardisoinit`, you would need to make a call to the `mex` program that looks something like this:

```
mex -cxx CXX=g++ CC=g++ LD=g++ -L\home\lib\pardiso -lpardiso
    -lmwlapack -lmwblas -lgfortran -lpthread -lm -output pardisoinit
    common.cpp matlabmatrix.cpp sparsematrix.cpp pardisoinfo.cpp
    pardisoinit.cpp
```

The flag `-lgfortran` links to a special library needed to handle the Fortran code for those that use the `gfortran` compiler. For those that use a different compiler, you may need to link to a different library. The option `-lpthread` may or may not be necessary. And, of course, depending on your system setup you may need to include other flags such as `-largeArrayDims`.

6 Using the PARDISO Matrix Consistency Checker

The PARDISO 6.0.0 library comes with new routines that can be used to check the input data for consistency. Given the input matrix A and the right-hand-side vecor or matrix B, the following tests are carried out:

- the arrays (`IA`, `JA`), that describe the nonzero structure of the matrix A, is a valid CSR format with index base 1
- The numerical values of A and B, given in `A` and `B`, are finite non-NaN values

All routines come in two flavors, one for real valued matrices and another one for complex valued matrices, which carry the suffix `_Z`. All arguments are exactly the same as for the main routine `PARDISO`, see section 2 for detailed explanation. The only output argument for all routines is `ERROR`. If it carries a non-zero value after calling any of the following routines, an error was found and an error message is printed to `STDERR`.

For testing the input matrix A as described above, use the function `PARDISO_CHKMATRIX`:

```
SUBROUTINE PARDISO_CHKMATRIX(MTYPE, N, A, IA, JA, ERROR)
SUBROUTINE PARDISO_CHKMATRIX_Z(MTYPE, N, A, IA, JA, ERROR)
```

For testing the RHS matrix B for infinite and NaN values, use the routine `PARDISO_CHKVEC`:

```
SUBROUTINE PARDISO_CHKVEC(N, NRHS, B, ERROR)
SUBROUTINE PARDISO_CHKVEC_Z(N, NRHS, B, ERROR)
```

A routine that automatically invokes `PARDISO_CHKMATRIX` and `PARDISO_CHKVEC` is the following:

```
SUBROUTINE PARDISO_PRINTSTATS(MTYPE, N, A, IA, JA, NRHS, B, ERROR)
SUBROUTINE PARDISO_PRINTSTATS_Z(MTYPE, N, A, IA, JA, NRHS, B, ERROR)
```

In addition to performing the above mentioned tests, this routines prints the range of absolute coefficient values to `STDOUT`. In many applications certain expectations for these ranges exist, so it may be useful to compare them with the printed values.

The derived `C` API prototypes are as follows. To represent complex valued matrices in `C`, please refer to the example `pardiso_unsym_complex.c`. In the following, we assume a structure

```
typedef struct {double re; double im;} doublecomplex;

void pardiso_chkmatrix   (int *mtype, int *n, double *a,
int *ia, int *ja, int *error);
void pardiso_chkmatrix_z (int *mtype, int *n, doublecomplex *a,
int *ia, int *ja, int *error);
void pardiso_chkvec   (int *n, int *nrhs, double *b,        int *error);
void pardiso_chkvec_z (int *n, int *nrhs, doublecomplex *b, int *error);
void pardiso_printstats (int *mtype, int *n, double *a,          int *ia,
int *ja, int *nrhs, double *b,        int *error);
void pardiso_printstats_z (int *mtype, int *n, doublecomplex *a, int *ia,
int *ja, int *nrhs, doublecomplex *b, int *error);
```

We recommend to execute the checking functionality prior to every call to `PARDISO` with new or modified input data. Since only a few passes through the input data structures are needed in order to carry out all the tests, the induced overhead will be negligible for most applications.

7 Authors

The author Olaf Schenk (USI Lugano) has a list of future features that he would like to integrate into the solver. In case of errors, wishes, big surprises — it may be sometimes possible to help.

8 Acknowledgments

Functionalities related to nonsymmetric maximal matching algorithms have been implemented by Stefan Röllin (Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zurich, Switzerland). We are also grateful to Esmond Ng (Lawrence Berkeley National Laboratory, Berkeley, USA) for several discussions on efficient sequential BLAS-3 supernodal factorization algorithms. We are also grateful to George Karypis (Department of Computer Science, University of Minnesota, Minnesota, USA) for his fill-in reducing METIS reordering package. Arno Liegmann (Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zurich, Switzerland) provided useful help at the start of the project.

We wish to thank Stefan Röllin (ETHZ), Michael Saunders (Stanford) for a providing very useful contributions to this project. The Matlab interface to PARDISO has been developed by Peter Carbonetto (UBC Vancouver). The PARDISO matrix consistency checking interface has been developed by Robert Luce (TU Berlin). Thank you all for these contributions.

9 License

You shall acknowledge using references [12] and [15] the contribution of this package in any publication of material dependent upon the use of the package. You shall use reasonable endeavors to notify the authors of the package of this publication. More information can be obtained from `http://www.pardiso-project.org`

The software can be downloaded from the PARDISO page[3]. A license file *pardiso.lic* is required for the execution of the solver and it will be generated at the download area of the web page. The user must supply information such as hostname (Unix: "`hostname`") and the user identification (Unix: "`whoami`"). This license file can be in the home directory of the user, or in the directory from which the user runs a program linked with the PARDISO library. Alternatively, this file can be placed in a fixed location and the environment variable `PARDISO_LIC_PATH` set to the path of its location.

References

[1] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Analysis and Applications*, 20(4):889–901, 1999.

[2] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[3] Andrey Kuzmin, Mathieu Luisier, and Olaf Schenk. Fast methods for computing selected elements of the green's function in massively parallel nanoelectronic device simulations. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 533–544. Springer Berlin Heidelberg, 2013.

[4] R. Menon L. Dagnum. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 1:46–55, 1998. http://www.openmp.org.

[5] X.S. Li and J.W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceeding of the 9th SIAM conference on Parallel Processing for Scientic Computing*, San Antonio, Texas, March 22-34,1999.

[6] J.W.H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.

[7] Cosmin G. Petra, Olaf Schenk, Miles Lubin, and Klaus Gärtner. An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization. Technical report, Preprint ANL/MCS-P4030-0113, Argonne National Laboratory, 2013.

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAMm 2nd edition, 2003.

[9] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zürich, 2000.

[10] O. Schenk, M. Bollhöfer, and R. A. Römer. On large scale diagonalization techniques for the Anderson model of localization. *SIAM Review*, 50(1):91–112, 2008. SIGEST Paper.

[11] O. Schenk and K. Gärtner. Two-level scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems. *Parallel Computing*, 28:187–197, 2002.

[12] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[13] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal*, 23:158–179, 2006.

[14] O. Schenk and K. Gärtner. Sparse factorization with two-level scheduling in PARDISO. In *Proceeding of the 10th SIAM conference on Parallel Processing for Scientic Computing*, Portsmouth, Virginia, March 12-14, 2001.

[15] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[3] The solver is available at `http://www.pardiso-project.org`

[16] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10:36–52, 1989.

10 Further comments

Note 1: PARDISO does support since version 4.1 (January 2011) the solution of sparse symmetric indefinite systems in a message-passing parallel environment. In addition, it is a thread-safe library and the calling program can execute several parallel factorization simultaneously.

Note 2: As of version 6.0.0, PARDISO has out-of-core capabilities.

Note 2: PARDISO uses a modified versions of METIS-4.01 and METIS 5.1 [2] and these versions are included in the libraries.

Note 3: Note that PARDISO needs the BLAS and LAPACK routines for the computational kernels. On IBM platforms these routines are part of the IBM Engineering and Scientific Subroutine Library (ESSL) and it is highly recommended that the user link with this thread-safe library. If this library is not available on your machine, then the ATLAS BLAS (`http://math-atlas.sourceforge.net`) or the GOTO BLAS (`http://www.cs.utexas.edu/users/flame/goto`) can be used. However, it is always important that the serial and thread-safe BLAS and LAPACK library be linked because PARDISO performs all the parallelization.

Note 4: Some technical papers related to the software and the example programs in the appendices can be obtained from the site `http://www.pardiso-project.org`.

Note 5: Some remarks on the memory management of the solver. All memory is allocated in a few steps based on the true demand and at the time of the first demand. The data that is used in the symbolic analysis phase only is released at the termination of the symbolic analysis phase. The data controlling the numeric phases is reallocated. Any memory, potentially used to solve a similar linear system again, will be released only on explicit user demand by PHASE=(0, or -1). Different pointer arrays PT can store data related to different structures. A small fraction of the memory is growing proportional to IPARM(3).

PARDISO memory is competing with later – with respect to the call tree – newly allocated memory in total memory size used. For large linear systems memory is likely to be the limiting issue, hence the user code should follow the strategy 'allocate as late and free as soon as possible', too.

A Examples for sparse symmetric linear systems

Appendices A.2 and A.3 give two examples (Fortran, C) for solving symmetric linear systems with PARDISO. They solve the system of equations $Ax = b$, where

$$A = \begin{pmatrix} 7.0 & 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 7.0 & 0.0 \\ 0.0 & -4.0 & 8.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 8.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 5.0 \\ 0.0 & 0.0 & 0.0 & 7.0 & 0.0 & 0.0 & 9.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 5.0 & -1.0 & 5.0 & 0.0 \\ 2.0 & 0.0 & 0.0 & 0.0 & -1.0 & 0.0 & 0.0 & 5.0 \\ 7.0 & 0.0 & 0.0 & 9.0 & 5.0 & 0.0 & 11.0 & 0.0 \\ 0.0 & 0.0 & 5.0 & 0.0 & 0.0 & 5.0 & 0.0 & 5.0 \end{pmatrix} \text{ and } b = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\ 5.0 \\ 6.0 \\ 7.0 \\ 8.0 \end{pmatrix}.$$

A.1 Example results for symmetric systems

Upon successful execution of the solver, the result of the solution X is as follows:

```
[PARDISO]: License check was successful ...
[PARDISO]: Matrix type      : real symmetric
[PARDISO]: Matrix dimension : 8
[PARDISO]: Matrix non-zeros : 18
[PARDISO]: Abs. coeff. range: min 0.00e+00  max 1.10e+01
[PARDISO]: RHS no. 1:  min 0.00e+00  max 7.00e+00

================  PARDISO: solving a symmetric indef.  system  ================


Summary PARDISO 6.0.0: ( reorder to reorder )
=======================

Times:
======

      Time fulladj: 0.000000 s
      Time reorder: 0.000000 s
      Time symbfct: 0.000000 s
      Time parlist: 0.000000 s
      Time malloc : 0.000000 s
      Time total  : 0.000000 s total - sum: 0.000000 s

Statistics:
===========
 < Parallel Direct Factorization with #cores: >              1
 <                               and #nodes: >              1
 < Numerical Factorization with Level-3 BLAS performance >

 < Linear system Ax = b>
             #equations:                                    8
             #non-zeros in A:                               18
             non-zeros in A (%):                            28.125000
             #right-hand sides:                             1

< Factors L and U >
             #columns for each panel:                       80
             # of independent subgraphs:                    0
 < preprocessing with state of the art partitioning metis>
             #supernodes:                                   5
             size of largest supernode:                     4
             number of nonzeros in L                        29
             number of nonzeros in U                        1
             number of nonzeros in L+U                      30
             number of perturbed pivots                     0
             number of nodes in solve                       8
             mflop   for the numerical factorization:       0.000075

Reordering completed ...
Number of nonzeros in factors  = 30
Number of factorization MFLOPS = 0
================  PARDISO: solving a symmetric indef.  system  ================
```

```
Summary PARDISO 6.0.0: ( factorize to factorize )
=================================

Times:
======

      Time A to LU: 0.000000 s
      Time numfct : 0.000000 s
      Time malloc : 0.000000 s
      Time total  : 0.000000 s total - sum: 0.000000 s

Statistics:
===========
 < Parallel Direct Factorization with #cores: >              1
 <                               and #nodes: >              1
 < Numerical Factorization with Level-3 BLAS performance >

 < Linear system Ax = b>
             #equations:                                    8
             #non-zeros in A:                               18
             non-zeros in A (%):                            28.125000
             #right-hand sides:                             1

< Factors L and U >
             #columns for each panel:                       80
             # of independent subgraphs:                    0
 < preprocessing with state of the art partitioning metis>
             #supernodes:                                   5
             size of largest supernode:                     4
             number of nonzeros in L                        29
             number of nonzeros in U                        1
             number of nonzeros in L+U                      30
             number of perturbed pivots                     0
             number of nodes in solve                       8
             mflop   for the numerical factorization:       0.000075

Factorization completed ...

================  PARDISO: solving a symmetric indef.  system  ================


Summary PARDISO 6.0.0: ( solve to solve )
=================================

Times:
======

      Time solve  : 0.000000 s
      Time total  : 0.000000 s total - sum: 0.000000 s

Statistics:
===========
 < Parallel Direct Factorization with #cores: >              1
 <                               and #nodes: >              1
```

```
 < Numerical Factorization with Level-3 BLAS performance >

 < Linear system Ax = b>
             #equations:                                  8
             #non-zeros in A:                             18
             non-zeros in A (%):                          28.125000
             #right-hand sides:                           1

< Factors L and U >
             #columns for each panel:                     80
             # of independent subgraphs:                  0
 < preprocessing with state of the art partitioning metis>
             #supernodes:                                 5
             size of largest supernode:                   4
             number of nonzeros in L                      29
             number of nonzeros in U                      1
             number of nonzeros in L+U                    30
             number of perturbed pivots                   0
             number of nodes in solve                     8
             mflop   for the numerical factorization:     0.000075

Solve completed ...
The solution of the system is:
  x(1) =  -0.3168031420208231
  x(2) =  -0.4955685649709140
  x(3) =  -0.2129608358961057
  x(4) =   5.6704583348771778E-002
  x(5) =   0.8607062136425950
  x(6) =   0.3140983363592574
  x(7) =   0.4003408796176218
  x(8) =   1.4988624995368485
```

A.2 Example pardiso_sym.f for symmetric linear systems (including selected inversion)

```
C-----------------------------------------------------------------------
C       Example program to show the use of the "PARDISO" routine
C       for symmetric linear systems
C -------------------------------------------------------------------
C      This program can be downloaded from the following site:
C      http://www.pardiso-project.org
C
C  (C) Olaf Schenk, Institute of Computational Science
C      Universita della Svizzera italiana, Lugano, Switzerland.
C      Email: olaf.schenk@usi.ch
C -------------------------------------------------------------------
        PROGRAM pardiso_sym
        IMPLICIT NONE

C..     Internal solver memory pointer
        INTEGER*8 pt(64)

C..     All other variables
```

```
      INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
      INTEGER iparm(64)
      INTEGER ia(9)
      INTEGER ja(18)
      REAL*8  dparm(64)
      REAL*8  a(18)
      REAL*8  b(8)
      REAL*8  x(8)
      REAL*8  y(8)

      INTEGER i, j, idum, solver
      REAL*8  waltime1, waltime2, ddum, normb, normr

C.. Fill all arrays containing matrix data.

      DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/

      DATA ia /1,5,8,10,12,15,17,18,19/

      DATA ja
     1       /1,          3,                6,    7,
     2              2,    3,          5,
     3                    3,                           8,
     4                         4,                7,
     5                               5,    6,    7,
     6                                     6,          8,
     7                                           7,
     8                                                 8/

      DATA a
     1    /7.d0,       1.d0,              2.d0, 7.d0,
     2          -4.d0, 8.d0,        2.d0,
     3                 1.d0,                        5.d0,
     4                       7.d0,             9.d0,
     5                             5.d0, -1.d0, 5.d0,
     6                                   0.d0,       5.d0,
     7                                         11.d0,
     8                                               5.d0/

C  .. set right hand side
      do i = 1, n
        b(i) = i
      end do

C
C  .. Setup Pardiso control parameters und initialize the solvers
C     internal adress pointers. This is only necessary for the FIRST
C     call of the PARDISO solver.
C
      mtype     = -2  ! unsymmetric matrix symmetric, indefinite
      solver    =  10  ! use sparse direct method

C  .. PARDISO license check and initialize solver
      call pardisoinit(pt, mtype, solver, iparm, dparm, error)
```

```
C  .. Numbers of Processors ( value of OMP_NUM_THREADS )
      iparm(3) = 1

      IF (error .NE. 0) THEN
        IF (error.EQ.-10 ) WRITE(*,*) 'No license file found'
        IF (error.EQ.-11 ) WRITE(*,*) 'License is expired'
        IF (error.EQ.-12 ) WRITE(*,*) 'Wrong username or hostname'
        STOP
      ELSE
        WRITE(*,*) '[PARDISO]: License check was successful ... '
      END IF

c  .. pardiso_chk_matrix(...)
c     Checks the consistency of the given matrix.
c     Use this functionality only for debugging purposes
      CALL pardiso_chkmatrix  (mtype, n, a, ia, ja, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

c ..  pardiso_chkvec(...)
c     Checks the given vectors for infinite and NaN values
c     Input parameters (see PARDISO user manual for a description):
c     Use this functionality only for debugging purposes
      CALL pardiso_chkvec (n, nrhs, b, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

c ..  pardiso_printstats(...)
c     prints information on the matrix to STDOUT.
c     Use this functionality only for debugging purposes

      CALL pardiso_printstats (mtype, n, a, ia, ja, nrhs, b, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
       ENDIF


C..   Reordering and Symbolic Factorization, This step also allocates
C     all memory that is necessary for the factorization

      phase     = 11      ! only reordering and symbolic factorization
      msglvl    = 1       ! with statistical information
      iparm(33) = 1       ! compute determinant

      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, ddum, ddum, error, dparm)

      WRITE(*,*) 'Reordering completed ... '
```

```
      IF (error .NE. 0) THEN
        WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      END IF

      WRITE(*,*) 'Number of nonzeros in factors   = ',iparm(18)
      WRITE(*,*) 'Number of factorization MFLOPS  = ',iparm(19)

C.. Factorization.
      phase     = 22  ! only factorization
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, ddum, ddum, error, dparm)

      IF (iparm(33).EQ.1)  THEN
            write(*,*) 'Log of determinant is  ',  dparm(33)
      ENDIF

      WRITE(*,*) 'Factorization completed ... '
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

C.. Back substitution and iterative refinement
      iparm(8)  = 1   ! max numbers of iterative refinement steps
      phase     = 33  ! only solve

      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, b, x, error, dparm)

      WRITE(*,*) 'Solve completed ...  '

      WRITE(*,*) 'The solution of the system is '
      DO i = 1, n
        WRITE(*,*) ' x(',i,') = ', x(i)
      END DO

C.. Residual test
      normb = 0
      normr = 0
      CALL pardiso_residual (mtype, n, a, ia, ja, b, x, y, normb, normr)
      WRITE(*,*) 'The norm of the residual is ',normr/normb


C.. Selected Inversion

      phase = -22
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, b, x, error, dparm)

c..  Print diagonal elements of the inverse of A = (a, ia, ja) */
      do i = 1, n
        j = ia(i)
        write(*,*) 'Diagonal',i,'-element of A^{-1} = ', a(j)
```

```
      end do

C.. Termination and release of memory
      phase     = -1           ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1              idum, nrhs, iparm, msglvl, ddum, ddum, error, dparm)
      END
```

A.3 Example pardiso_sym.c for symmetric linear systems (including selected inversion)

```
/* -------------------------------------------------------------------- */
/*      Example program to show the use of the "PARDISO" routine        */
/*      on symmetric linear systems                                     */
/* -------------------------------------------------------------------- */
/*      This program can be downloaded from the following site:         */
/*      http://www.pardiso-project.org                                  */
/*                                                                      */
/*  (C) Olaf Schenk, Institute of Computational Science                 */
/*      Universita della Svizzera italiana, Lugano, Switzerland.        */
/*      Email: olaf.schenk@usi.ch                                       */
/* -------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>



/* PARDISO prototype. */
void pardisoinit (void   *, int    *,   int *, int *, double *, int *);
void pardiso     (void   *, int    *,   int *, int *,    int *, int *,
                  double *, int    *,    int *, int *,   int *, int *,
                     int *, double *, double *, int *, double *);
void pardiso_chkmatrix  (int *, int *, double *, int *, int *, int *);
void pardiso_chkvec     (int *, int *, double *, int *);
void pardiso_printstats (int *, int *, double *, int *, int *, int *,
                           double *, int *);


int main( void )
{
    /* Matrix data. */
    int    n = 8;
    int    ia[ 9] = { 0, 4, 7, 9, 11, 14, 16, 17, 18 };
    int    ja[18] = { 0,    2,       5, 6,
                         1, 2,    4,
                            2,             7,
                               3,       6,
                                  4, 5, 6,
                                     5,    7,
                                        6,
                                           7 };
    double  a[18] = { 7.0,      1.0,           2.0, 7.0,
```

```
                          -4.0, 8.0,           2.0,
                                1.0,                      5.0,
                                     7.0,           9.0,
                                          5.0, -1.0, 5.0,
                                          0.0,       5.0,
                                              11.0,
                                                    5.0 };


    int      nnz = ia[n];
    int      mtype = -2;        /* Real symmetric matrix */

    /* RHS and solution vectors. */
    double   b[8], x[8];
    int      nrhs = 1;          /* Number of right hand sides. */

    /* Internal solver memory pointer pt,                  */
    /* 32-bit: int pt[64]; 64-bit: long int pt[64]         */
    /* or void *pt[64] should be OK on both architectures  */
    void    *pt[64];

    /* Pardiso control parameters. */
    int      iparm[64];
    double   dparm[64];
    int      maxfct, mnum, phase, error, msglvl, solver;

    /* Number of processors. */
    int      num_procs;

    /* Auxiliary variables. */
    char    *var;
    int      i, k;

    double   ddum;              /* Double dummy */
    int      idum;              /* Integer dummy. */


/* -------------------------------------------------------------------- */
/* ..  Setup Pardiso control parameters.                                */
/* -------------------------------------------------------------------- */

    error = 0;
    solver=0;/* use sparse direct solver */
    pardisoinit (pt,  &mtype, &solver, iparm, dparm, &error);

    if (error != 0)
    {
        if (error == -10 )
           printf("No license file found \n");
        if (error == -11 )
           printf("License is expired \n");
        if (error == -12 )
           printf("Wrong username or hostname \n");
         return 1;
```

```
    }
    else
        printf("[PARDISO]: License check was successful ... \n");

    /* Numbers of processors, value of OMP_NUM_THREADS */
    var = getenv("OMP_NUM_THREADS");
    if(var != NULL)
        sscanf( var, "%d", &num_procs );
    else {
        printf("Set environment OMP_NUM_THREADS to 1");
        exit(1);
    }
    iparm[2]  = num_procs;

    maxfct = 1;         /* Maximum number of numerical factorizations.  */
    mnum   = 1;         /* Which factorization to use. */

    msglvl = 1;         /* Print statistical information  */
    error  = 0;         /* Initialize error flag */

/* ---------------------------------------------------------------------- */
/* ..  Convert matrix from 0-based C-notation to Fortran 1-based        */
/*     notation.                                                        */
/* ---------------------------------------------------------------------- */
    for (i = 0; i < n+1; i++) {
        ia[i] += 1;
    }
    for (i = 0; i < nnz; i++) {
        ja[i] += 1;
    }

    /* Set right hand side to i. */
    for (i = 0; i < n; i++) {
        b[i] = i + 1;
    }

/* ---------------------------------------------------------------------- */
/*  .. pardiso_chk_matrix(...)                                          */
/*     Checks the consistency of the given matrix.                      */
/*     Use this functionality only for debugging purposes               */
/* ---------------------------------------------------------------------- */

    pardiso_chkmatrix  (&mtype, &n, a, ia, ja, &error);
    if (error != 0) {
        printf("\nERROR in consistency of matrix: %d", error);
        exit(1);
    }

/* ---------------------------------------------------------------------- */
/* ..  pardiso_chkvec(...)                                              */
/*     Checks the given vectors for infinite and NaN values             */
/*     Input parameters (see PARDISO user manual for a description):    */
/*     Use this functionality only for debugging purposes               */
/* ---------------------------------------------------------------------- */
```

```
    pardiso_chkvec (&n, &nrhs, b, &error);
    if (error != 0) {
        printf("\nERROR  in right hand side: %d", error);
        exit(1);
    }

/* ---------------------------------------------------------------------- */
/* .. pardiso_printstats(...)                                             */
/*    prints information on the matrix to STDOUT.                         */
/*    Use this functionality only for debugging purposes                  */
/* ---------------------------------------------------------------------- */

    pardiso_printstats (&mtype, &n, a, ia, ja, &nrhs, b, &error);
    if (error != 0) {
        printf("\nERROR right hand side: %d", error);
        exit(1);
    }

/* ---------------------------------------------------------------------- */
/* ..  Reordering and Symbolic Factorization.  This step also allocates */
/*     all memory that is necessary for the factorization.                */
/* ---------------------------------------------------------------------- */
    phase = 11;

    pardiso (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error, dparm);

    if (error != 0) {
        printf("\nERROR during symbolic factorization: %d", error);
        exit(1);
    }
    printf("\nReordering completed ... ");
    printf("\nNumber of nonzeros in factors  = %d", iparm[17]);
    printf("\nNumber of factorization MFLOPS = %d", iparm[18]);

/* ---------------------------------------------------------------------- */
/* ..  Numerical factorization.                                           */
/* ---------------------------------------------------------------------- */
    phase = 22;
    iparm[32] = 1; /* compute determinant */

    pardiso (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error,  dparm);

    if (error != 0) {
        printf("\nERROR during numerical factorization: %d", error);
        exit(2);
    }
    printf("\nFactorization completed ...\n ");

/* ---------------------------------------------------------------------- */
```

```
/* ..  Back substitution and iterative refinement.                    */
/* ------------------------------------------------------------------ */
    phase = 33;

    iparm[7] = 1;       /* Max numbers of iterative refinement steps. */

    pardiso (pt, &maxfct, &mnum, &mtype, &phase,
             &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, b, x, &error,  dparm);

    if (error != 0) {
        printf("\nERROR during solution: %d", error);
        exit(3);
    }

    printf("\nSolve completed ... ");
    printf("\nThe solution of the system is: ");
    for (i = 0; i < n; i++) {
        printf("\n x [%d] = % f", i, x[i] );
    }
    printf ("\n\n");


/* ------------------------------------------------------------------ */
/* ... Inverse factorization.                                         */
/* ------------------------------------------------------------------ */

    if (solver == 0)
    {
        printf("\nCompute Diagonal Elements of the inverse of A ... \n");
        phase = -22;
        iparm[35]  = 1; /*  no not overwrite internal factor L */

        pardiso (pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, &idum, &nrhs,
             iparm, &msglvl, b, x, &error,  dparm);

       /* print diagonal elements */
       for (k = 0; k < n; k++)
       {
            int j = ia[k]-1;
            printf ("Diagonal element of A^{-1} = %d %d %32.24e\n", k, ja[j]-1, a[j]);
       }

    }


/* ------------------------------------------------------------------ */
/* ..  Convert matrix back to 0-based C-notation.                     */
/* ------------------------------------------------------------------ */
    for (i = 0; i < n+1; i++) {
        ia[i] -= 1;
    }
    for (i = 0; i < nnz; i++) {
        ja[i] -= 1;
```

```
    }

/* ------------------------------------------------------------------- */
/* ..  Termination and release of memory.                              */
/* ------------------------------------------------------------------- */
    phase = -1;                 /* Release internal memory. */

    pardiso (pt, &maxfct, &mnum, &mtype, &phase,
             &n, &ddum, ia, ja, &idum, &nrhs,
             iparm, &msglvl, &ddum, &ddum, &error,  dparm);

    return 0;
}
```

A.4 Example pardiso_sym_schur.cpp for Schur-complement computation for symmetric linear systems

```
/* ------------------------------------------------------------------- */
/*      Example program to show the use of the "PARDISO" routine       */
/*      for evaluating the Schur-complement                            */
/* ------------------------------------------------------------------- */
/*      This program can be downloaded from the following site:        */
/*      http://www.pardiso-project.org                                 */
/*                                                                     */
/*  (C) Drosos Kourounis, Institute of Computational Science           */
/*      Universita della Svizzera Italiana.                            */
/*      Email: drosos.kourounis@usi.ch                                 */
/* ------------------------------------------------------------------- */

// C++ compatible

#include <fstream>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <stdlib.h>

using namespace std;

/* PARDISO prototype. */
extern "C" void pardisoinit_d(void   *, int    *,   int *, int *, double *, int *);
extern "C" void pardiso_d(void   *, int    *,   int *, int *,    int *, int *,
                  double *, int    *,    int *, int *,   int *, int *,
                     int *, double *, double *, int *, double *);
extern "C" void pardiso_chkmatrix_d(int *, int *, double *, int *, int *, int *);
extern "C" void pardiso_chkvec_d(int *, int *, double *, int *);
extern "C" void pardiso_printstats_d(int *, int *, double *, int *, int *, int *, doub
extern "C" void pardiso_get_schur_d(void*, int*, int*, int*, double*, int*, int*);

void dumpCSR(const char* filename, int n, int* ia, int* ja, double* a)
{
  fstream fout(filename, ios::out);
  fout << n << endl;
  fout << n << endl;
```

```
  fout << ia[n] << endl;

  for (int i = 0; i <= n; i++)
  {
    fout << ia[i] << endl;
  }

  for (int i = 0; i < ia[n]; i++)
  {
    fout << ja[i] << endl;
  }

  for (int i = 0; i < ia[n]; i++)
  {
    fout << a[i] << endl;
  }

  fout.close();
}


void printCSR(int n, int nnz, int* ia, int* ja, double* a)
{
  cout << "rows: " << setw(10) << n << endl;
  cout << "nnz : " << setw(10) << nnz << endl;

  if (nnz == n*n)
  {
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++)
      {
        cout << setw(10) << a[i*n + j];
      }
      cout << endl;
    }
  }
  else
  {
    for (int i = 0; i < n; i++)
    {
      for (int index = ia[i]; index < ia[i+1]; index++)
      {
        int j = ja[index];
        cout << setw(10) << "(" << i << ", " << j << ") " << a[index];
      }
      cout << endl;
    }
  }
}


void shiftIndices(int n, int nonzeros, int* ia, int* ja, int value)
{
```

```
  int i;
    for (i = 0; i < n+1; i++)
    {
        ia[i] += value;
    }
    for (i = 0; i < nonzeros; i++)
    {
        ja[i] += value;
    }
}




int main( void )
{
    /* Matrix data.
     */

    int    n =10;
    int    ia[11] = { 0, 6, 11, 15, 19, 22, 26, 29, 32, 33, 34};
    int    ja[34] = { 0, 2, 5, 6, 8, 9, 1, 2, 4, 8, 9, 2, 7, 8,
                      9, 3, 6, 8, 9, 4, 8, 9, 5, 7, 8, 9, 6, 8,
                      9, 7, 8, 9, 8, 9};

    double  a[34] = { 7.0,      1.0,           2.0, 7.0,     17.0, 8.5,
                         -4.0, 8.0,      2.0,                6.0, 3.0,
                               1.0,                      5.0, 6.0, 3.0,
                                    7.0,           9.0,     16.0, 8.0,
                                         0.0,               -4.0,-2.0,
                                              3.0,      8.0,18.0, 9.0,
                                                  11.0,     12.0, 6.0,
                                                        5.0, 4.0, 2.0,
                                                             0.0,
                                                                  0.0};



    int      nnz = ia[n];
    int      mtype = -2;        /* Real unsymmetric matrix */

    dumpCSR("matrix_sample_mtype_2.csr", n, ia, ja, a);
    printCSR(n, nnz, ia, ja, a);

    /* Internal solver memory pointer pt,                  */
    /* 32-bit: int pt[64]; 64-bit: long int pt[64]         */
    /* or void *pt[64] should be OK on both architectures  */
    void    *pt[64];

    /* Pardiso control parameters. */
    int      iparm[65];
    double   dparm[64];
    int      solver;
    int      maxfct, mnum, phase, error, msglvl;
```

```
    /* Number of processors. */
    int      num_procs;

    /* Auxiliary variables. */
    char    *var;
    int      i;

    double   ddum;              /* Double dummy */
    int      idum;              /* Integer dummy. */

/* -------------------------------------------------------------------- */
/* ..  Setup Pardiso control parameters and initialize the solvers      */
/*     internal adress pointers. This is only necessary for the FIRST   */
/*     call of the PARDISO solver.                                      */
/* ---------------------------------------------------------------------*/

    error = 0;
    solver = 0; /* use sparse direct solver */
    pardisoinit_d(pt,  &mtype, &solver, &iparm[1], dparm, &error);

    if (error != 0)
    {
        if (error == -10 )
           printf("No license file found \n");
        if (error == -11 )
           printf("License is expired \n");
        if (error == -12 )
           printf("Wrong username or hostname \n");
         return 1;
    }
    else
        printf("[PARDISO]: License check was successful ... \n");


    /* Numbers of processors, value of OMP_NUM_THREADS */
    var = getenv("OMP_NUM_THREADS");
    if(var != NULL)
        sscanf( var, "%d", &num_procs );
    else
    {
        printf("Set environment OMP_NUM_THREADS to 1");
        exit(1);
    }
    iparm[3]  = num_procs;
    iparm[11] = 1;
    iparm[13] = 0;


    maxfct = 1;         /* Maximum number of numerical factorizations.  */
    mnum   = 1;         /* Which factorization to use. */

    msglvl = 1;         /* Print statistical information  */
    error  = 0;         /* Initialize error flag */
```

```
/* ------------------------------------------------------------------- */
/* ..  Convert matrix from 0-based C-notation to Fortran 1-based       */
/*     notation.                                                       */
/* ------------------------------------------------------------------- */
    shiftIndices(n, nnz, ia, ja, 1);


/* ------------------------------------------------------------------- */
/*  .. pardiso_chk_matrix(...)                                         */
/*     Checks the consistency of the given matrix.                     */
/*     Use this functionality only for debugging purposes              */
/* ------------------------------------------------------------------- */
    pardiso_chkmatrix_d(&mtype, &n, a, ia, ja, &error);
    if (error != 0)
    {
        printf("\nERROR in consistency of matrix: %d", error);
        exit(1);
    }


/* ------------------------------------------------------------------- */
/* ..  Reordering and Symbolic Factorization.  This step also allocates */
/*     all memory that is necessary for the factorization.             */
/* ------------------------------------------------------------------- */

    int nrows_S = 2;
    phase       = 12;
    iparm[38]   = nrows_S;

    int nb = 0;

    pardiso_d(pt, &maxfct, &mnum, &mtype, &phase,
              &n, a, ia, ja, &idum, &nb,
              &iparm[1], &msglvl, &ddum, &ddum, &error,  dparm);

    if (error != 0)
    {
        printf("\nERROR during symbolic factorization: %d", error);
        exit(1);
    }
    printf("\nReordering completed ...\n");
    printf("Number of nonzeros in factors  = %d\n", iparm[18]);
    printf("Number of factorization MFLOPS = %d\n", iparm[19]);
    printf("Number of nonzeros is   S      = %d\n", iparm[39]);

/* ------------------------------------------------------------------- */
/* ..  allocate memory for the Schur-complement and copy it there.     */
/* ------------------------------------------------------------------- */
    int nonzeros_S = iparm[39];

    int* iS        = new int[nrows_S+1];
    int* jS        = new int[nonzeros_S];
```

```
    double* S      = new double[nonzeros_S];

    pardiso_get_schur_d(pt, &maxfct, &mnum, &mtype, S, iS, jS);


/* ------------------------------------------------------------------- */
/* ..  Convert matrix from 1-based Fortan notation to 0-based C        */
/*     notation.                                                       */
/* ------------------------------------------------------------------- */
    shiftIndices(n, nnz, ia, ja, -1);

/* ------------------------------------------------------------------- */
/* ..  Convert Schur complement from Fortran notation held internally  */
/*     to 0-based C notation                                           */
/* ------------------------------------------------------------------- */
    shiftIndices(nrows_S, nonzeros_S, iS, jS, -1);

    printCSR(nrows_S, nonzeros_S, iS, jS, S);

/* ------------------------------------------------------------------- */
/* ..  Termination and release of memory.                              */
/* ------------------------------------------------------------------- */
    phase = -1;                 /* Release internal memory. */

    pardiso_d(pt, &maxfct, &mnum, &mtype, &phase,
            &n, &ddum, ia, ja, &idum, &idum,
            &iparm[1], &msglvl, &ddum, &ddum, &error,  dparm);


    delete[] iS;
    delete[] jS;
    delete[] S;


    return 0;
}
```

A.5 Example iterative solver pardiso_iter_sym.f for symmetric linear systems

```
C-----------------------------------------------------------------------
C       Example program to show the use of the "PARDISO" routine
C       for symmetric linear systems
C -------------------------------------------------------------------
C      This program can be downloaded from the following site:
C      http://www.pardiso-project.org
C
C  (C) Olaf Schenk, Institute of Computational Science
C      Universita della Svizzera italiana, Lugano, Switzerland.
C      Email: olaf.schenk@usi.ch
C -------------------------------------------------------------------

        PROGRAM pardiso_sym
        IMPLICIT NONE
```

```
C..      Internal solver memory pointer for 64-bit architectures
C..      INTEGER*8 pt(64)
C..      Internal solver memory pointer for 32-bit architectures
C..      INTEGER*4 pt(64)
C..      This is OK in both cases
         INTEGER*8 pt(64)

C..      All other variables
         INTEGER maxfct, mnum, mtype, phase, n, nrhs, error, msglvl
         INTEGER iparm(64)
         INTEGER ia(9)
         INTEGER ja(18)
         REAL*8  dparm(64)
         REAL*8  a(18)
         REAL*8  b(8)
         REAL*8  x(8)

         INTEGER i, idum, solver
         REAL*8  waltime1, waltime2, ddum

C.. Fill all arrays containing matrix data.

         DATA n /8/, nrhs /1/, maxfct /1/, mnum /1/

         DATA ia /1,5,8,10,12,15,17,18,19/

         DATA ja
     1        /1,          3,                6,    7,
     2              2,    3,          5,
     3                    3,                           8,
     4                         4,                7,
     5                               5,    6,    7,
     6                                     6,          8,
     7                                           7,
     8                                                 8/

         DATA a
     1     /7.d0,       1.d0,              2.d0, 7.d0,
     2           -4.d0, 8.d0,        2.d0,
     3                  1.d0,                        5.d0,
     4                        7.d0,             9.d0,
     5                              5.d0, 1.d0, 5.d0,
     6                                    0.d0,       5.d0,
     7                                          11.d0,
     8                                                5.d0/

C  .. set right hand side
       do i = 1, n
         b(i) = 1.d0
       end do
C
C  .. Setup Pardiso control parameters und initialize the solvers
C     internal adress pointers. This is only necessary for the FIRST
C     call of the PARDISO solver.
```

```
C
      mtype     = -2  ! unsymmetric matrix symmetric, indefinite
      solver    =  1  ! use sparse direct method

c  .. pardiso_chk_matrix(...)
c     Checks the consistency of the given matrix.
c     Use this functionality only for debugging purposes
      CALL pardiso_chkmatrix  (mtype, n, a, ia, ja, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

c ..  pardiso_chkvec(...)
c     Checks the given vectors for infinite and NaN values
c     Input parameters (see PARDISO user manual for a description):
c     Use this functionality only for debugging purposes
      CALL pardiso_chkvec (n, nrhs, b, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

c ..  pardiso_printstats(...)
c     prints information on the matrix to STDOUT.
c     Use this functionality only for debugging purposes

      CALL pardiso_printstats (mtype, n, a, ia, ja, nrhs, b, error);
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
       ENDIF

C  .. PARDISO license check and initialize solver
      call pardisoinit(pt, mtype, solver, iparm, dparm, error)
      IF (error .NE. 0) THEN
        IF (error.EQ.-10 ) WRITE(*,*) 'No license file found'
        IF (error.EQ.-11 ) WRITE(*,*) 'License is expired'
        IF (error.EQ.-12 ) WRITE(*,*) 'Wrong username or hostname'
        STOP
      ELSE
        WRITE(*,*) '[PARDISO]: License check was successful ... '
      END IF

C  .. Numbers of Processors ( value of OMP_NUM_THREADS )
      iparm(3) = 1

      phase     = 12  ! set up of the preconditioner
      msglvl    = 1

      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, ddum, ddum, error, dparm)

      WRITE(*,*) 'Set up completed ... '
```

```
      IF (error .NE. 0) THEN
         WRITE(*,*) 'The following ERROR was detected: ', error
        STOP
      ENDIF

C  .. Solve with SQMR
      phase     = 33  ! only solve

c  ..Set initial solution to zero
      DO i = 1, n
        x(i) = 0
      END DO

      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja,
     1              idum, nrhs, iparm, msglvl, b, x, error, dparm)

      WRITE(*,*) 'Solve completed ...  '

      WRITE(*,*) 'The solution of the system is '
      DO i = 1, n
        WRITE(*,*) ' x(',i,') = ', x(i)
      END DO

C..   Termination and release of memory
      phase     = -1           ! release internal memory
      CALL pardiso (pt, maxfct, mnum, mtype, phase, n, ddum, idum, idum,
     1              idum, nrhs, iparm, msglvl, ddum, ddum, error, dparm)
      END
```

B Examples for sparse symmetric linear systems on distributed-memory architectures

Appendices B.1 gives an example in C for solving symmetric indefinite linear systems with PARDISO on distributed-memory architectures. Other example are available on the PARDISO web page.

B.1 Example pardiso_sym_mpp.c for the solution of symmetric linear systems on distributed memory architecutes

```
/* -------------------------------------------------------------------- */
/*      This program can be downloaded from the following site:         */
/*      http://www.pardiso-project.org                                  */
/*                                                                      */
/*  (C) Olaf Schenk, Institute of Computational Science                 */
/*      Universita della Svizzera italiana, Lugano, Switzerland.        */
/*      Email: olaf.schenk@usi.ch                                       */
/* -------------------------------------------------------------------- */

static int mpipardiso_driver(
        int n,
        int nnz,
        int* ia,
        int* ja,
        double *va,
```

```
        double *b,
        MPI_Comm comm)
{
    int mpi_rank;
    int mpi_size;

    MPI_Comm_rank(comm, &mpi_rank);
    MPI_Comm_size(comm, &mpi_size);
    if (mpi_rank == 0)
    {
        /*
         * Master process
         */

        double *x;
        double time_analyze = 0.0;
        double time_factorize = 0.0;
        double time_solve = 0.0;

        void  *handle[64];
        int    iparam[64];
        double dparam[64];
        int ido;
        int nb = 1; /* Only one right hand side.                */
        int Nmax_system = 1;
        int matrix_number = 1;
        int matrix_type = -2; /* Sparse symmetric matrix is indefinite.*/
        int solver = 0;       /* Use sparse direct method */
        int perm_user = 0;    /* Dummy pointer for user permutation.   */
        int msglvl = 1;
        int ierror = 0;       /* Initialize the Pardiso error flag     */

        /*
         * Setup Pardiso control parameters und initialize the solver's
         * internal address pointers. This is only necessary for the FIRST
         * call of the PARDISO solver.
         *
         * PARDISO license check and initialize solver
         */
        memset(dparam, 0, sizeof(dparam));
        memset(handle, 0, sizeof(handle));
        pardisoinit_(handle, &matrix_type, &solver, iparam, dparam, &ierror);

        if (ierror)
            return pardiso_error(ierror, iparam);

        IPARAM(3) = 8;          /* maximal number of OpenMP tasks */
        IPARAM(8) = 0;          /* no iterative refinement */
        IPARAM(51) = 1;         /* use MPP factorization kernel */
        IPARAM(52) = mpi_size; /* number of nodes that we would like to use */

        printf(FMT_PARDISO_IPARAM, "IPARAM(3)", IPARAM(3),
                "Number of OpenMP tasks per host");
        printf(FMT_PARDISO_IPARAM, "IPARAM(11)", IPARAM(11), "Scaling");
```

```
printf(FMT_PARDISO_IPARAM, "IPARAM(13)", IPARAM(13), "Matching");
printf(FMT_PARDISO_IPARAM, "IPARAM(52)", IPARAM(52), "Number of hosts");

/* Additional initialization for PARDISO MPI solver */
pardiso_mpi_init_c_(handle, comm, &ierror);
if (ierror)
    return pardiso_error(ierror, iparam);

/* Allocate work arrays */
x = malloc(n * sizeof(double));
if (x == NULL)
    malloc_error();

/* Analysis */
time_analyze -= seconds();

/*
 * Start symbolic factorization of the PARDISO solver.
 */
ido = 11; /* perform only symbolic factorization */
pardiso_(handle, &Nmax_system, &matrix_number, &matrix_type, &ido, &n,
        va, ia, ja, &perm_user, &nb, iparam, &msglvl, b, x, &ierror, dparam);

if (ierror)
    return pardiso_error(ierror, iparam);

time_analyze += seconds();
printf(TIME_FMT, "time analyze", time_analyze);

printf(FMT_PARDISO_IPARAM, "IPARAM(18)", IPARAM(18),
        "Number of LU nonzeros");
printf(FMT_PARDISO_IPARAM, "IPARAM(19)", IPARAM(19), "Number of FLOPS");
printf(FMT_PARDISO_IPARAM, "IPARAM(15)", IPARAM(15), "Analysis memory");
printf(FMT_PARDISO_IPARAM, "IPARAM(16)", IPARAM(16), "Structure memory");
printf(FMT_PARDISO_IPARAM, "IPARAM(17)", IPARAM(17), "Factor memory");

/*
 * Start numerical factorization of the PARDISO solver.
 */
time_factorize = -seconds();

ido = 22;
pardiso_(handle, &Nmax_system, &matrix_number, &matrix_type, &ido, &n,
        va, ia, ja, &perm_user, &nb, iparam, &msglvl, b, x, &ierror,
        dparam);

if (ierror)
    return pardiso_error(ierror, iparam);

time_factorize += seconds();
printf(TIME_FMT, "time_factorize", time_factorize);

printf(FMT_PARDISO_IPARAM, "IPARAM(14)", IPARAM(14), "Perturbed pivots");
printf(FMT_PARDISO_IPARAM, "IPARAM(22)", IPARAM(22), "Positive pivots");
```

```
printf(FMT_PARDISO_IPARAM, "IPARAM(23)", IPARAM(23), "Negative pivots");
printf(FMT_PARDISO_DPARAM, "DPARAM(33)", DPARAM(33), "Determinant");

/*
 * Start forward/backward substitution of the PARDISO solver.
 */
time_solve = -seconds();

ido = 33;
pardiso_(handle, &Nmax_system, &matrix_number, &matrix_type, &ido, &n,
        va, ia, ja, &perm_user, &nb, iparam, &msglvl, b, x, &ierror,
        dparam);

if (ierror)
    return pardiso_error(ierror, iparam);

time_solve += seconds();
printf(TIME_FMT, "time_solve", time_solve);

/*
 * Compute relative residual
 */
SSP_print_accuracy(n, ia, ja, va, b, x);
{
    double norm_b = 0.0;
    int i;
    for (i = 0; i < n; ++i)
        norm_b += b[i] * b[i];
    norm_b = sqrt(norm_b);

    printf(FMT_NORM, "norm(b)", norm_b);

    double norm_x = 0.0;
    for (i = 0; i < n; ++i)
        norm_x += x[i] * x[i];
    norm_x = sqrt(norm_x);

    printf(FMT_NORM, "norm(x)", norm_x);

}

/*
 * Free PARDISO internal memory
 */
ido = -1;
pardiso_(handle, &Nmax_system, &matrix_number, &matrix_type, &ido, &n,
        va, ia, ja, &perm_user, &nb, iparam, &msglvl, b, x, &ierror,
        dparam);

if (ierror)
    return pardiso_error(ierror, iparam);

/* Additional finalization for MPI solver */
pardiso_mpi_finalize_(handle, &ierror);
```

```
        if (ierror)
            return pardiso_error(ierror, iparam);

        free(x);
    }
    else /* mpi_rank != 0 */
    {
        /*
         * Worker process
         */
        void *handle[64];
        int isolve;
        int ierror = 0;

        /* Initialization for PARDISO MPI solver */
        pardiso_mpi_init_c_(handle, comm, &ierror);
        if (ierror)
            return 1;

        /* Symbolic factorization */
        pardiso_mpi_worker_(handle, &ierror);
        if (ierror)
            return 1;

        /* Numerical factorization */
        pardiso_mpi_worker_(handle, &ierror);
        if (ierror)
            return 1;

        /* Triangular solve */
        pardiso_mpi_worker_(handle, &ierror);
        if (ierror)
            return 1;

        /* Free solver memory */
        pardiso_mpi_worker_(handle, &ierror);

        /* Finalization for MPI solver */
        pardiso_mpi_finalize_(handle, &ierror);
    }

    return 0;
}
```

C Release notes

```
PARDISO - Release Notes - version 6.1.0
---------------------------------------

PARDISO Contents
----------------
PARDISO - a direct and iterative sparse linear solver library - is a
tuned math solver library designed for high performance on homogeneous
```

```
multicore machines and cluster of multicores including Intel Xeon, Intel
Itanium, AMD Opteron, and IBM Power processors, and includes both 32-bit
and 64-bit library versions.  Different versions are available for Linux
MAC OSX, and Windows 64-bit operating systems.

A full suite of sparse direct linear solver routines for all kind of
sparse matrices is provided, and key datastructures have been designed
for high performance on various multicore processors and cluster of multicore
processors in 64-bit modes.

A selected suite of iterative linear solvers for real and complex symmetric
indefinite matrices is provided that takes advantage of a new algebraic
multilevel incomplete factorization that is especially designed for good
performance in very large-scale applications.

New Features
------------
New features of release PARDISO 6.1.0:

(o) Added acceleration of block orderings for symmetric indefinite matrices.
(o) Significantly improved the reordering time for matrices including dense columns.

New features of release PARDISO 6.0.0:

(o) Added METIS 5.1 as additional preprocessing method.
(o) Improved scalability for factorization on higher number of cores.
(o) Added out-of-core option for real and complex symmetric indefinite matrices.
(o) New internal data structure to simpify future developments.

New features of release PARDISO 5.0.0:

(o) Switch to host-free license for all 64-bit libraries. This allows all users
    to use the PARDISO software within a cluster environment.
(o) Full support of multi-threaded Schur-complement computations for all
    kind of matrices.
(o) Full support of multi-threaded parallel selected inversion to compute selected
    entries of the inverse of A.
(o) Faster multi-threaded code for the solution of multiple right hand sides.
(o) Full support of 32-bit sequential and parallel factorizations for all kind
    of matrices.

New features of release PARDISO 4.1.3:
(o) Bug fix in the computation of the log of the determinant.

New features of release PARDISO 4.1.2:
(o) Support of different licensing types
    - Evaluation license  for academic and commercial use.
    - Academic license (host-unlocked, user-locked, 1 year).
    - Commercial single-user license (host-unlocked, user-locked, 1 year).
    - Commercial license (host-unlocked, user-unlocked, redistributable, 1 year).

New features of release PARDISO 4.1.0:

(o) New MPI-based numerical factorization and parallel forward/backward
```

```
    substitution on   distributed-memory  architectures  for  symmetric
    indefinite matrices. PARDISO 4.1.0 has the unique feature among all
    solvers  that  it  can compute  the  exact  bit-identical  solution
    on multicores and cluster of multicores.  Here are some results for
    a nonlinear FE  model with  800'000 elements from automobile  sheet
    metal forming simulations.

    CPUs per node:
       4 x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz (8 cores in total)
    Memory per node:
       12 GiB, Interconnect: Infiniband 4xQDR

    (t_fact: factorization in  seconds, t_solve= solve in seconds)

    PARDISO 4.1.0 (deterministic) :
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    t_fact  | 1 core   4 cores   8 cores
    ---------+--------+---------+---------
    1 host  | 92.032    23.312    11.966
    2 hosts | 49.051    12.516     7.325
    4 hosts | 31.646     8.478     5.018

    t_solve | 1 core   4 cores   8 cores
    ---------+--------+---------+---------
    1 host  |  2.188     0.767     0.545
    2 hosts |  1.205     0.462     0.358
    4 hosts |  0.856     0.513     0.487

    Intel MKL 10.2 (non-deterministic):
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    t_fact  | 1 core   4 cores   8 cores
    --------+--------+---------+---------
    1 host  | 94.566    27.266    14.018

    t_solve | 1 core   4 cores   8 cores
    ---------+--------+---------+---------
    1 host  | 2.223      2.183     2.207

    - The MPI version is only available for academic research purposes.

(o) New host-unlimited licensing meachnism integrated into PARDISO 4.1.0.
    We now have two different options available:
    - a time-limited user-host locked license, and
    - a time-limited user-locked (host-free) license.

(o) 32-bit sequential and parallel factorization and solve routines for
    real unsymmetric  matrices (matrix_type = 11). Mixed-precision refinement
    can used for these 32-bit sparse direct factorizations.

(o) Additional routines that can check the input data (matrix, right-hand-side)
    (contribution from Robert Luce, TU Berlin)

New features of release 4.0.0 of PARDISO since version 3.3.0:
```

```
(o) Due to the new features the interface to PARDISO and PARDISOINIT has
     changed! This version is not backward compatible!

(o) Reproducibility of exact numerical results on multi-core architectures.
    The solver is now able to compute the exact bit identical solution
    independent on the number of cores without effecting the scalability.
    Here are some results for a nonlinear FE model with 500'000 elements.

    Intel MKL PARDISO 10.2
    1 core  - factor: 17.980 sec., solve: 1.13 sec.
    2 cores - factor:  9.790 sec., solve: 1.13 sec.
    4 cores - factor:  6.120 sec., solve: 1.05 sec.
    8 cores - factor:  3.830 sec., solve: 1.05 sec.

   U Basel PARDISO 4.0.0:
   1 core  - factor: 16.820 sec., solve: 1.09 sec.
   2 cores - factor:  9.021 sec., solve: 0.67 sec.
   4 cores - factor:  5.186 sec., solve: 0.53 sec.
   8 cores - factor:  3.170 sec., solve: 0.43 sec.

   This method is currently only working for symmetric indefinite matrices.


(o) 32-bit sequential and parallel factorization and solve routines for
    real symmetric indefinite matrices, for symmetric complex matrices
    and for structurally symmetric matrices. Mixed-precision refinement
    is used for these 32-bit sparse direct factorizations.

(o) Internal 64-bit integer datastructures for the numerical factors allow
    to solve very large sparse matrices with over 2^32 nonzeros in the sparse
    direct factors.

(o) Work has been done to significantly improve the parallel performance of
    the sparse direct solver which results in a much better scalability for
    the numerical factorization and solve on multicore machines. At the same
    time, the workspace memory requirements have been substantially reduced,
    making the PARDISO direct routine better able to deal with large problem
    sizes.

(o) Integration of a parallel multi-threaded METIS reordering that helps to
    accelerate the reordering phase (Done by to Stefan Roellin, ETH Zurich)

(o) Integration of a highly efficient preconditioning method that is
    based on a multi-recursive incomplete factorization scheme and
    stabilized with a new graph-pivoting algorithm. The method have been
    selected by the SIAM Journal of Scientific Computing as a very important
    milestone in the area of new solvers for symmetric indefinite matrices
    and the related paper appeared as a SIGEST SIAM Paper in  2008.

    This preconditioner is highly effective for large-scale matrices with
    millions of equations.

    [1] O. Schenk, M. Bollhoefer, and R. Roemer, On large-scale
    diagonalization techniques for the Anderson model of localization.
```

```
    Featured SIGEST paper in the SIAM Review selected "on the basis of its
    exceptional interest to the entire SIAM community".
    SIAM Review 50 (2008), pp. 91--112.

(o) Support of 32-bit and 64-bit Windows operating systems (based on
    Intel Professional Compiler Suite and the Intel MKL Performance
    Library)

(o) A new extended interface to direct and iterative solver.
    Double-precision parameters are passed by a dparm array to the solver.
    The interface allow for greater flexibility in the storage of input
    and output data within supplied arrays through the setting of increment
    arguments.

(o) Note that the interface to PARDISO and PARDISOINIT has changed and that
    this version is not backward compatible.

(o) Computation of the determinant for symmetric indefinite matrices.

(o) Solve A^Tx=b using the factorization of A.

(o) The solution process e.g. LUx=b can be performed in several phases that
    the user can control.

(o) This version of PARDISO is compatible with the interior-point optimization
    package IPOPT version 3.7.0

(o) A new matlab interface has been added that allows a flexible use of all
     direct and iterative solvers.

Contributions
-------------
The following colleagues have contributed to the solver (in alphabetical order):

Peter Carbonetto  (UBC Vancouver, Canada).
Radim Janalik     (USI Lugano, Switzerland
George Karypis    (U Minnesota, US)
Arno Liegmann     (ETHZ, Switzerland)
Esmond Ng         (LBNL, US)
Stefan Roellin    (ETHZ, Switzerland)
Michael Saunders  (Stanford, US)


Applicability
-------------
Different PARDISO versions are provided for use with the GNU compilers
gfortran/gcc,  with the Intel ifort compiler, and (for use under Solaris)
with the Sun f95/cc compilers.


Required runtime libraries under Microsoft Windows
--------------------------------------------------
PARDISO version 4.0.0 and later link with the standard runtime library
provided by the Microsoft Visual Studio 2008 compilers. This requires
```

```
that the machine PARDISO runs on either has VS2K8 installed (or the Windows
SDK for Windows Server 2008), or the runtime libraries can be separately
downloaded from the appropriate Microsoft platform links provided below:

Visual Studio 2K8 Redist:

x86
<http://www.microsoft.com/downloads/details.aspx?FamilyID=9B2DA534-3E03-4391-8A4D-074B9F2BC1BF&displayla

x64
<http://www.microsoft.com/downloads/details.aspx?familyid=BD2A6171-E2D6-4230-B809-9A8D7548C1B6&displayla


Bug Reports
-----------
Bugs should be reported to info@pardiso-project.org with the string
"PARDISO-Support" in the subject line.
```