Intel® MKL Sparse Solvers

Software Solutions Group - Developer Products Division

Agenda

- ✓ Overview
- ✓ Direct Solvers
  - ✓Introduction
  - ✓PARDISO: main features
  - ✓PARDISO: advanced functionality
  - ✓DSS
  - ✓Performance data
- ✓ Iterative Solvers
- ✓ Performance Data
- ✓ Reference

Overview

For solving Systems of Linear Equations (SLAE) Intel® MKL offers:

for **Dense case (**beyond the scope of this presentation**):**

- LAPACK
- ScaLAPACK (LAPACK for distributed memory systems)

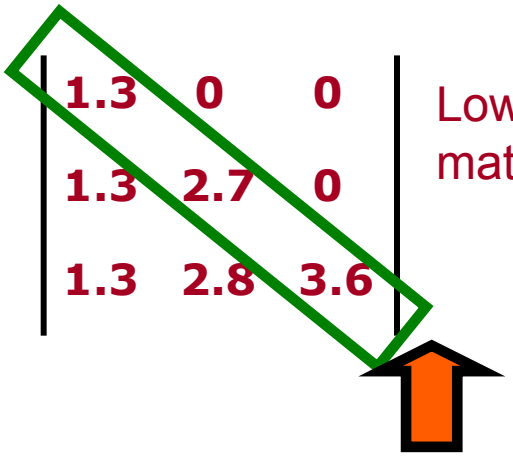
for **Sparse case:**

- Direct solvers: PARDISO / DSS
- Iterative solvers: RCI CG and FGMRES (requires matrix-vector multiplications, e.g., from Sparse BLAS, and (optionally) a preconditioner, e.g., ILU0)

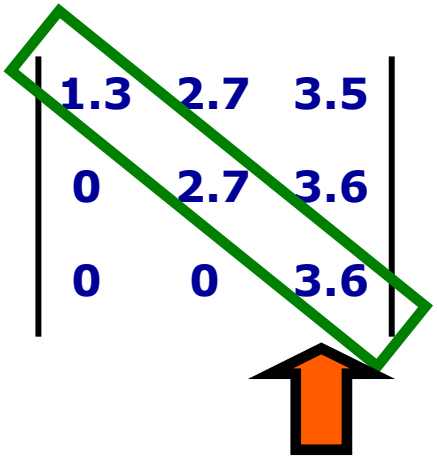
Direct solvers: Introduction

The idea behind direct methods (solvers) is the decomposition of the original matrix into simpler form. The most typical example is LU-decomposition: A=LU, where L is lower triangular, U is upper triangular matrix.

Example:



Lower triangular matrix



Upper triangular matrix

Main diagonal of the matrix

Direct solvers: Introduction

Usually, sparse direct solvers solve the task by the following stages:

- Phase 1: Fill-reduction analysis and symbolic factorization A'=P^t A P
- Phase 2: Numerical factorization A' = LU (LL^t, LDL^t)
- Phase 3: Forward and Backward solve including iterative refinements
Ax=f ➔ LUx=f ➔ Ly=f, Ux=y
- Termination and Memory Release Phase (phase≤ 0)

Direct solvers: Introduction

The following examples illustrates Fill-In and Reordering of Sparse Matrices concepts:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 6 & 0 & 12 & 0 & 0 \\ \frac{3}{4} & 0 & 0 & \frac{5}{8} & 0 \\ 3 & 0 & 0 & 0 & 16 \end{bmatrix} \quad L = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 2 & -2 & 2 & 0 & 0 \\ \frac{1}{4} & \frac{1}{-4} & \frac{1}{-2} & \frac{1}{2} & 0 \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix}$$

Factorization without permutation

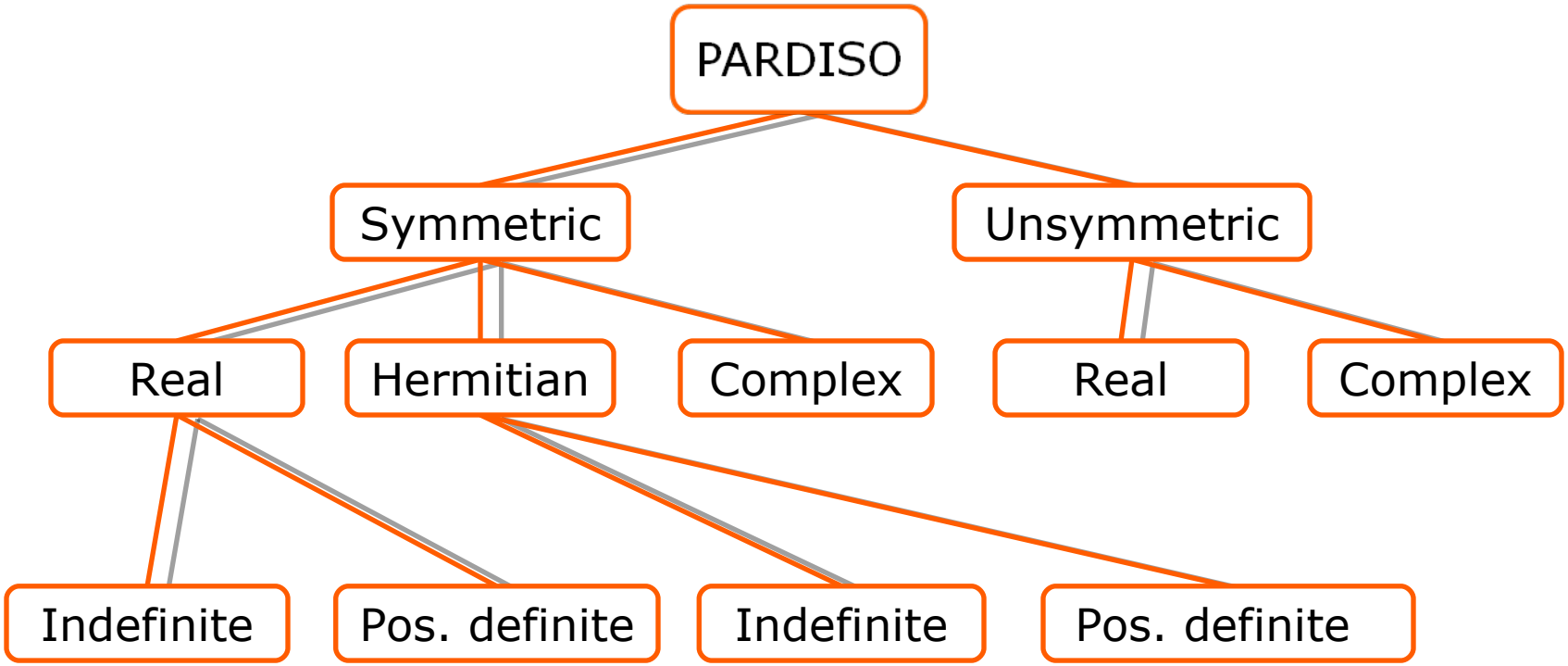
$$B = \begin{bmatrix} 16 & 0 & 0 & 0 & 3 \\ 0 & \frac{1}{2} & 0 & 0 & \frac{3}{2} \\ 0 & 0 & 12 & 0 & 6 \\ 0 & 0 & 0 & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix} \quad L = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 2\sqrt{3} & 0 & 0 \\ 0 & 0 & 0 & \frac{\sqrt{10}}{4} & 0 \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix}$$

Factorization after permutation

Permuting A to B minimizes fill-in.

PARDISO: main features

The PARDISO/DSS solvers support a wide range of sparse matrix types:



$A = LDL^T$ $A = LL^T$ $A = LDL^H$ $A = LL^H$ $A = LU$

PARDISO: main features

Pardiso *supports CSR format only !*

- ***values***, ***columns***, and ***rowIndex***

  are arrays contain all the storage information
- ***values –*** *A* real or complex array that contains the non-zero entries of *A*. The non-zero values of *A* are mapped into the *values* array using the row major, upper triangular storage mapping.
- ***columns*** - Element *i* of the integer array *columns* contains the number of the column in *A* that contained the value in *values*(*i*).
- ***rowIndex*** - Element *j* of the integer array *rowIndex* gives the index into the values array that contains the first non-zero element in a row *j* of *A*. The length of the *values* and *columns* arrays is equal to the number of non-zeros in *A*.

PARDISO: main features

- Symmetrical matrix

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ * & \frac{1}{2} & * & * & * \\ * & * & \frac{1}{2} & * & * \\ * & * & * & \frac{5}{8} & * \\ * & * & * & * & 16 \end{bmatrix}$$

values = (9 3/2 6 3/4 3 1/2 1/2 5/8 16)
columns = (1 2 3 4 5 2 3 4 5)
rowIndex = (1 6 7 8 9 10)

- Unsymmetrical matrix - zeros added to make matrix structurally symmetric

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{bmatrix}$$

values = (1 -1 -3 -2 5 0 4 6 4 -4 2 7 8 0 -5)
columns = (1 2 4 1 2 5 3 4 5 1 3 4 2 3 5)
rowIndex = (1 4 7 10 13 16)

PARDISO: main features

- **First**, the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right).
- **Second**, no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix. The second restriction implies that when dealing with symmetric or structurally symmetric matrices that have zeros on the diagonal, the zero diagonal elements must be explicitly represented in the *values* array.
- **Third -** supported one-based indexing only (true for version 10.2 inclusive)

- **Tips:**

matrix checker (iparm(27))– checks the CSR format consistency. Checker switches off by the default

PARDISO: main features

fill-in reducing ordering.

- *iparm(2) controls the fill-in reducing ordering for the input* matrix.
- If *iparm(2) = 0, the minimum degree algorithm is applied*.
- If *iparm(2) = 2, the solver uses the nested dissection* algorithm from the METIS package.
- If *iparm(2) = 3, the parallel (OpenMP) version of the* nested dissection algorithm is used. It can decrease the time of computations on multi-core computers, especially when the time of the PARDISO Phase 1 is significant for your task.
- The default value of *iparm(2) is 2.*

user permutation.

- This parameter (*iparm**(5)***) controls whether the user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms.

PARDISO: main features

Factorization: scaling, matching, pivoting, boosting

- *During factorization phase PARDISO performs LU decomposition of input matrix. In order to improve accuracy and stability of solving process PARDISO can apply some transformations of original matrix. For example, scaling (PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1), matching (*PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal*), pivoting and boosting (*when the factorization algorithm reaches a point where it cannot factorize the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy*)*

PARDISO: main features

Solving: allowable error, iterative refinement

Each algorithm of solving system of equations calculate solution with some accuracy. The residual is dependent on algorithm and on the initial matrix properties. Namely, let x is the ideal solution of system Ax=f. So, x = A^-1 f. But due to inaccuracies of digital computations we obtain x' = x' = (L'U')$^{-1}$ f. In order to improve accuracy of obtained solution PARDISO can perform several iteration for refinement solution.

The solver will not perform more than the absolute value of iparm(8) steps of iterative refinement and will stop the process if a satisfactory level of accuracy of the solution in terms of backward error has been achieved.

PARDISO: Solving of systems with many RHS

Sometimes solving of a system with many right hand sides (RHS) is required. There are two possible ways of doing so.

First way is to solve system of equation for each RHS consecutively. In this situation the total time of solving will increased in N times as compared to solving step for single RHS where N is the number of RHS's to be solved.

*Second variant is to solve all RHS at once (parameter **nrhs** in PARDISO interface is dedicated to pass the number of solving vectors). In this way PARDISO not only will use BLAS / LAPACK functions for finding the several solutions at once but it will do it in parallel on available threads.*

PARDISO: Solving several matrices at once

More complicated situation is in solving several matrices at the same time. As usually there are two different situations.

First one, when all each matrices solving separately from another's. In this situation one should use unique handle for each solving matrix. This way is more universal but it is more wasteful. Namely each handle stores many working arrays for each matrix which are equals in the case when matrices have the same sparsity structure.

Second situation when many matrices with the same sparsity structures need to be solved. In this situation PARDISO can solve these matrices more effectively with the use of single handle. In this case maxfct is the maximal number of factors with identical nonzero sparsity structure that the user would like to keep at the same time in memory, and mnum is actual matrix number.

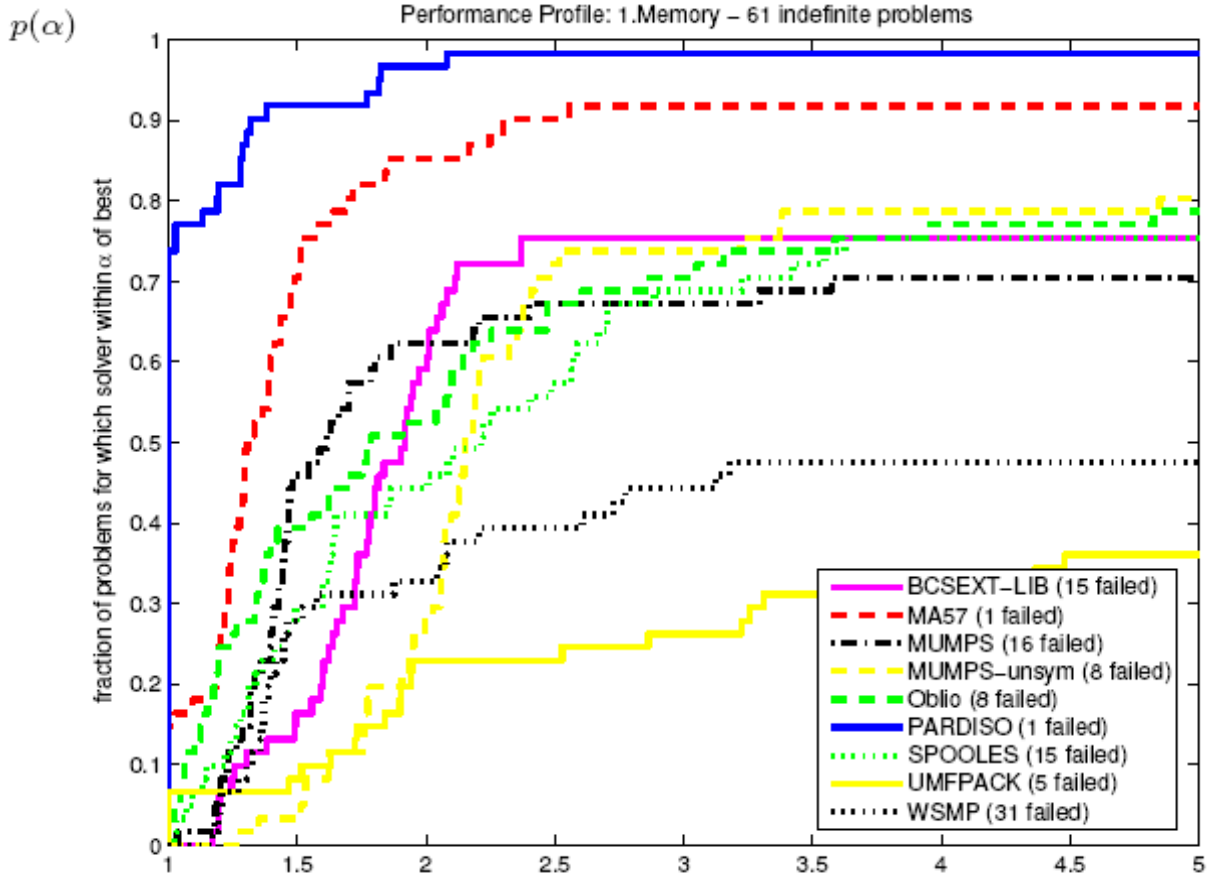
PARDISO: Using PARDISO as preconditioner (CG, etc)

Next situation when many SLAU should be solved consecutively and next matrix is only slightly differs form previous one. Applying factorization phase can not be so optimal in this case because the obtained solution will be slightly differs form previous one. So iterative algorithm could be the most effective in this situation. Namely, PARDISO already has LU decomposition for previous matrix and this decomposition can be used for constructing the effective preconditioner for finding the new solution. Please see description of parameters iparm ***(4) - preconditioned CGS***

PARDISO: OOC mode

Some tasks can't be solved in default (In-Core) PARDISO mode because obtained LU factors are not fit in RAM. For these situations the Out-Of-Core mode is exists in PARDISO (see description of iparm(60)). In this mode PARDISO can store most intermediate information and LU factors on disk. In this mode PARDISO can efficiently solve tasks which required 3-5 and more times more memory than available RAM.

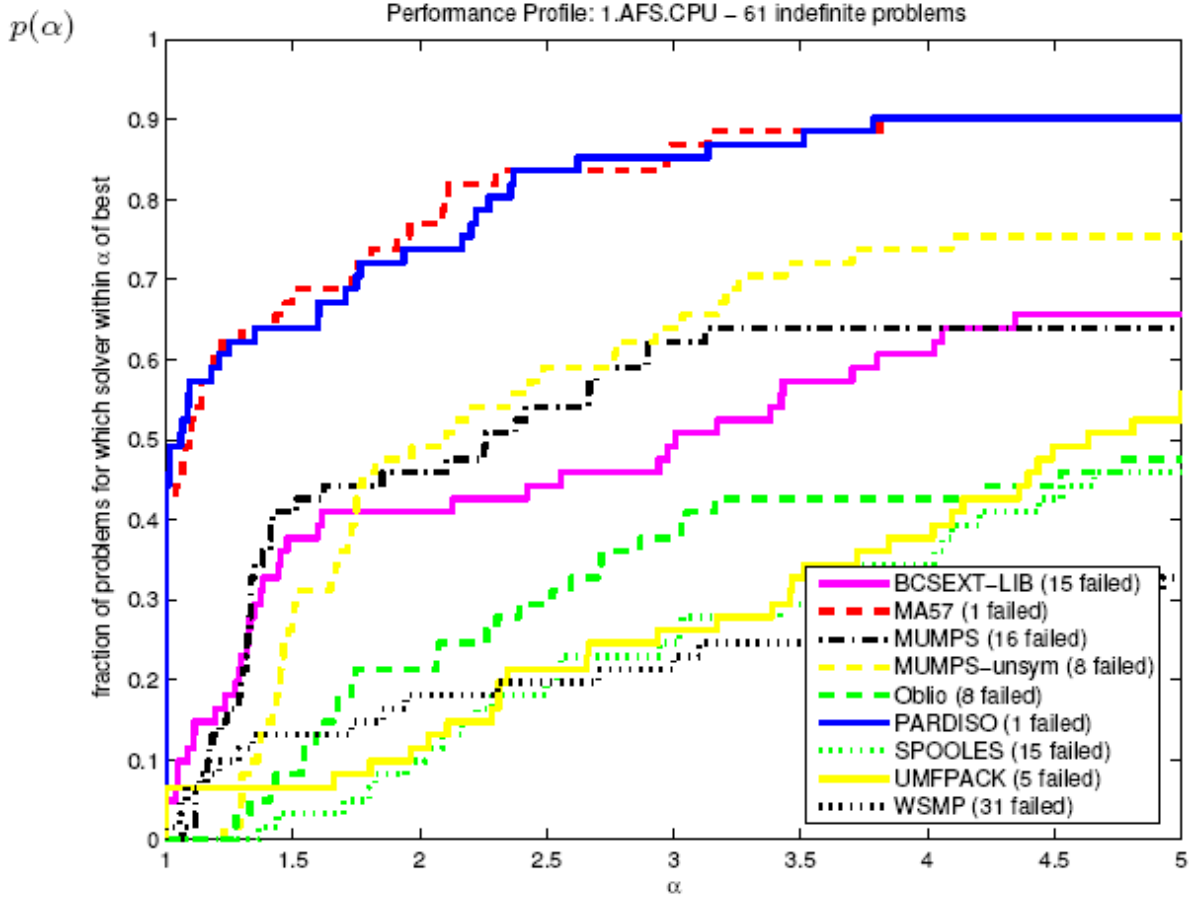
PARDISO Memory Utilization



This figure indicates a high correlation between the total memory used and the numbers of nonzeros in the factors …, with PARDISO requiring the least memory, followed by MA57 well above the rest

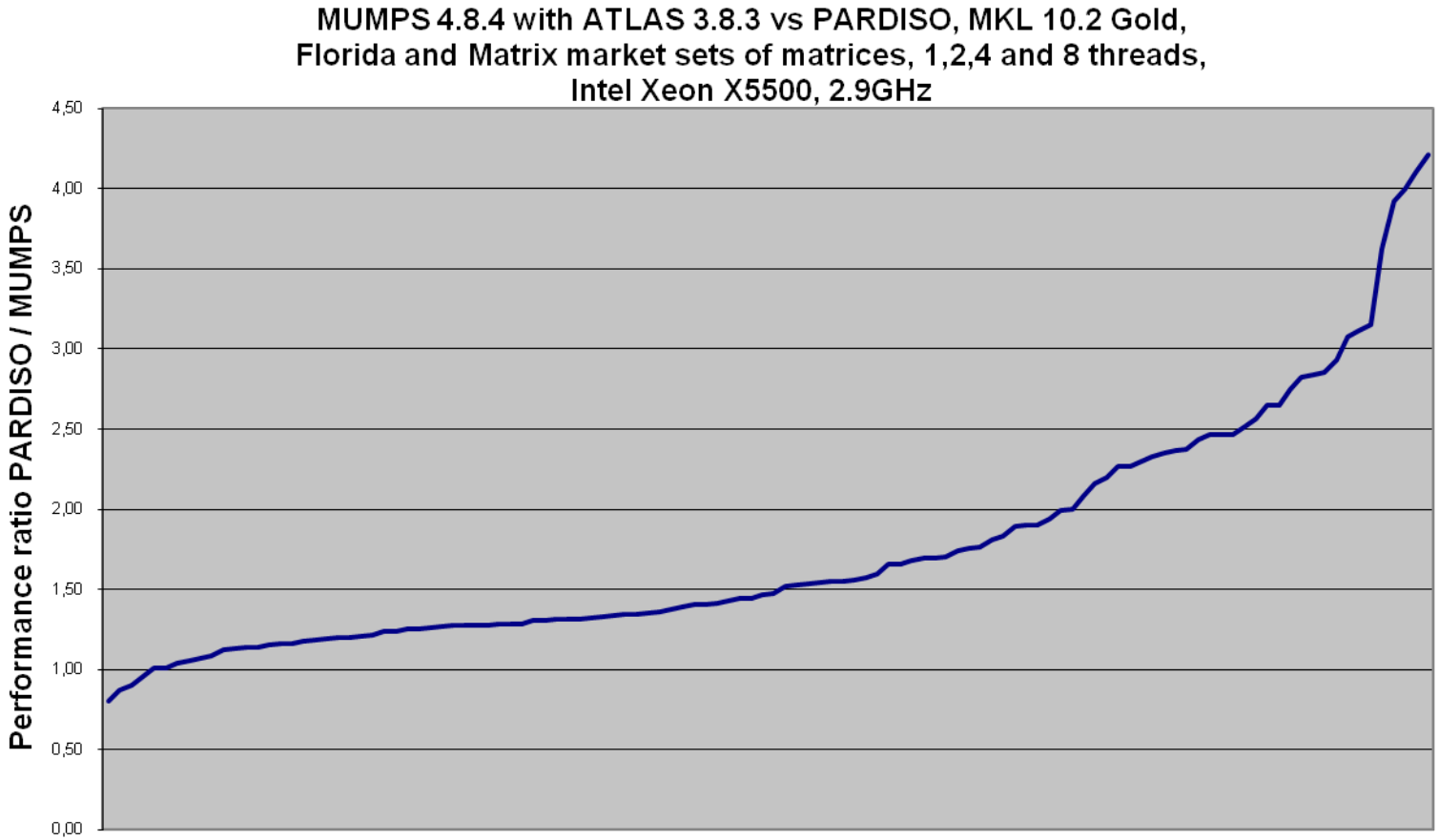
ftp://ftp.numerical.rl.ac.uk/pub/reports/ghsRAL200505.pdf

Performance profile, CPU time for the complete solution.



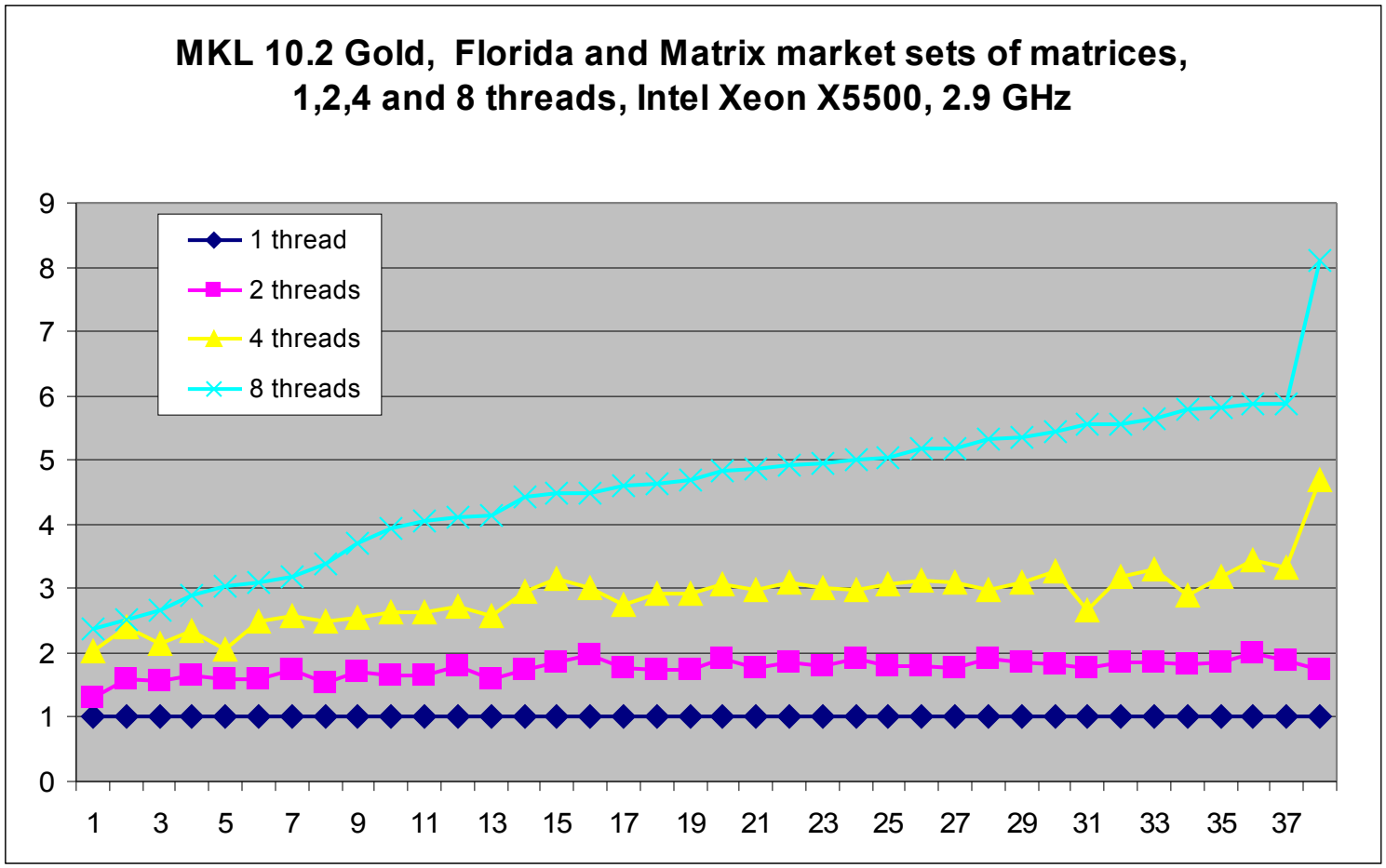
ftp://ftp.numerical.rl.ac.uk/pub/reports/ghsRAL200505.pdf

Intel® MKL PARDISO* performance comparison: best results in more than 95% of test cases



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations

Intel® MKL PARDISO* scalability results: up to linear speedup on 4 and 8 cores



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations

Main features of Intel MKL PARDISO

- Utilizes Intel MKL BLAS and LAPACK and uses shared-memory parallelism to improve numerical factorization performance.
- SMP system **only** (true for version 10.3 inclusive)
- Solves wide class of SLAE as compared with iterative solvers.
- Minimizes RAM in use despite it is a direct solver.
- Up to linear speedup on multicore systems.
- Additional simplified interface DSS (Direct Sparse Solver) available from C and Fortran user codes.

Additional functionality of Intel(R) MKL PARDISO

- Reordering input matrix A. 3 options are available: choose one of two effective internal algorithms or provide specific reordering vector.
- Built-in CG algorithm with preconditioner
- Boosting ill-conditioned matrices
- Iterative refinement
- ILP64 interface (dealing with more than 2*10^9 elements).
- In-Core, Hybrid and Out-Of-Core modes
- Supported single or double precision modes (by the default – double)
- 32bit Pardiso can solves task for 250*10^6 double nonzero elements of L factors (true for SPD matrixes and for MKL 10.2 update 2 inclusively)

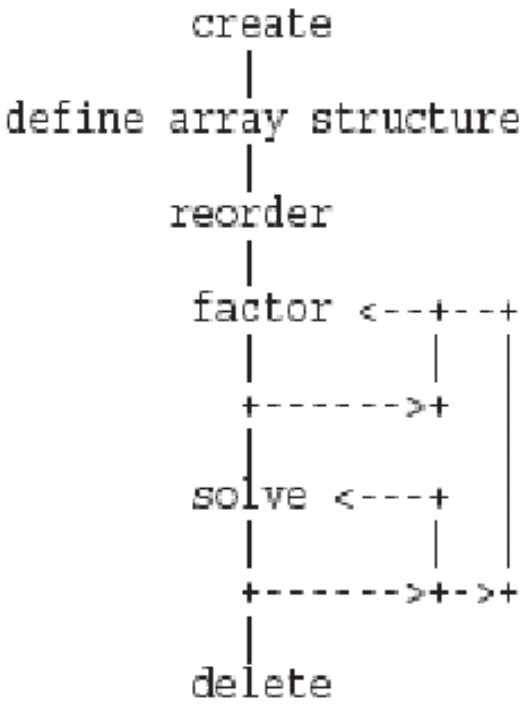
Direct Sparse Solver (DSS)

DSS

is the alternative to the PARDISO interface

- This is the group of user-callable routines that are used in the step-by-step solving process.
- In-Core and Out-Of-Core modes
- Supported single or double precision modes (by the default is double)
- Available from C and Fortran user codes

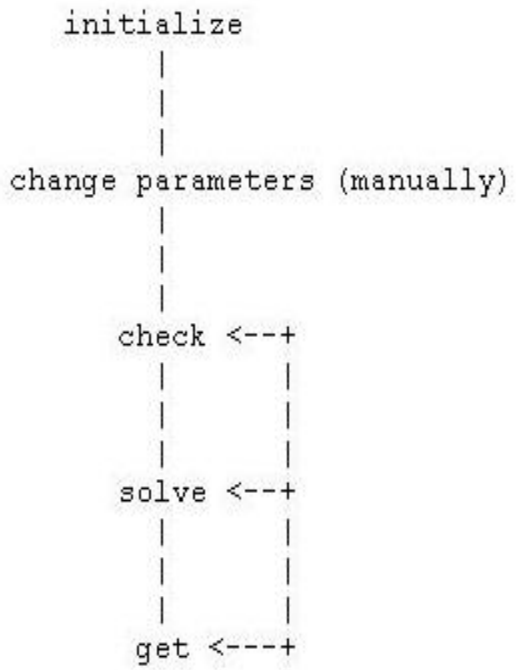
The solving process is conceptually divided into six phases:



Iterative Solvers (ISS)

- The idea behind iterative solvers is the usage of matrix-vector multiplications to build an approximation to the solution (that is why, the methods are called iterative). For arbitrary vector r0 (often, r0=b), the method computes couple of vectors consistently and uses those vectors to build an approximate solution.
- ISS interface based on the reverse communication interface (RCI)
- RCI ISS is a group of user-callable routines that are used in the step-by-step solving process

Typical Order for Invoking RCI ISS interface Routines



Intel Software Products

Software Solutions Group Developer Products Division

intel

Iterative Solvers (ISS)

Intel® MKL provides 2 RCI ISS interface implementations:

- RCI Conjugate Gradient Solver (RCI CG)

- used for symmetric positive definite matrices
- If the input matrix is not SPD type → failure or wrong solution..
- There 2 versions:
  - for 1 for system of equations with single right hand side, and
  - multiple right hand sides

- RCI Flexible Generalized Minimal RESidual Solver(RCI FGMRES).

  non-symmetric indefinite (non-degenerate) type. If the input is degenerate → failure

Iterative Solvers (ISS) - Preconditioners

- ISS use **preconditioners** (or in other words, **accelerators**) to deal with the ill-conditioned problems. Basically, the idea of preconditioning is to solve

  $\tilde{A}\tilde{x} = \tilde{b}$ **instead of Ax=b, where**

  $$\tilde{A} = B^{-1/2} A B^{-1/2}; \tilde{x} = B^{1/2} x; \tilde{b} = B^{-1/2} b$$

  Knowing the properties of the problem (customer always knows them), it is possible to reduce the condition number issues by constructing sufficiently good preconditioner (matrix B).
- in some cases **preconditioners** can reduce the number of iterations dramatically and thus lead to better solver performance.

Iterative Solvers (ISS) - Preconditioners

- ILU0 (less efficient for ill-conditioned systems and less computations required)
- ILUT (more efficient for ill-conditioned systems and more computations required). Valid since version 10.0.

Notes:

- ILU* preconditioners are based on modifications of LU-decomposition that came from direct solvers.
- A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial guess, and even ruin the convergence
- can be used alone or together with the Intel MKL RCI FGMRES
- Initial matrix must be csr format. Important: *diagonal element must be in structure even it is equal zero!*
- don't recommend use these preconditioners with MKL RCI CG solver cause unsymmetrical resulting preconditioner matrix.

Reference

- Intel® MKL product page:
  http://software.intel.com/en-us/intel-mkl/

- Intel® MKL Forum :
  http://software.intel.com/en-us/forums/intel-math-kernel-library/

- Intel® MKL Knowledge Base:
  http://software.intel.com/en-us/forums/intel-math-kernel-library/

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101