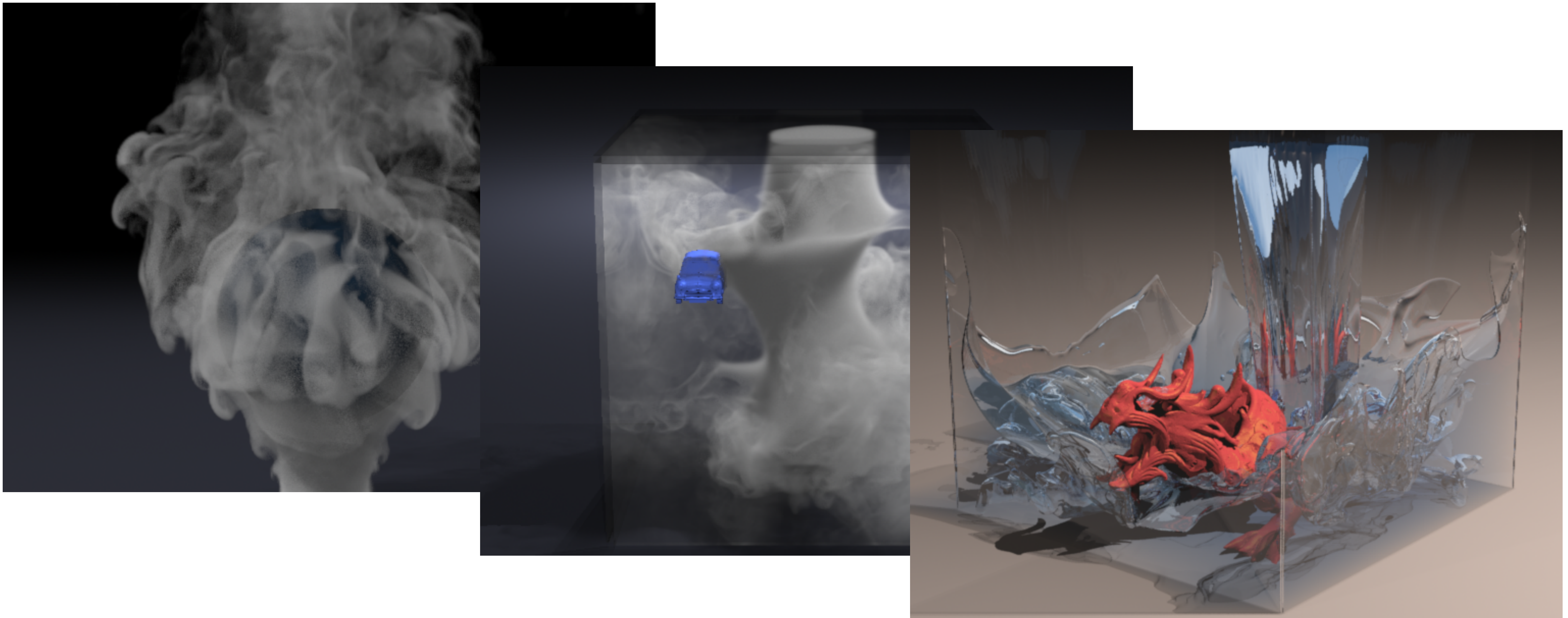


Parallel Sparse Direct Solvers
A survey of Intel MKL PARDISO

Introduction

As far as solvers for linear systems are concerned, we have so far seen:

- Iterative, sparse system solvers, e.g. Conjugate Gradients
(Pros: matrix free, parallel; Cons: Inaccurate unless iterated to convergence)
- Direct, dense system solvers, e.g. LAPACK
(Pros: Most accurate, broadly usable, parallel; Cons: Limited to small matrices)
- Triangular, dense or sparse, system solvers (BLAS Level 2/3)
(Pros: Inexpensive, parallel when dense; Cons: Limited scope, memory bound)

Introduction

Today we visit a very popular library that implements a <u>direct</u> solver, on <u>sparse</u> linear systems of equations
- Called the "PARDISO" sparse direct solver
- Initially developed as a stand-alone project (still ongoing)
- Incorporated into the Math Kernel Library
- Extremely optimized, using advanced theory and parallelism

Objective:
- Obtain a user-level understanding of how to use this library
- Review an implementation applied to one of our benchmark problems
- Understand the underlying semantics (API options, interface details)
- Have some <u>superficial</u> appreciation of the design strategies that went into creating such a library

Principle of operation (most common case)

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

PARDISO computes the Cholesky Decomposition (or a slight variation called the LDL factorization)

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{21} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Most commonly used for symmetric matrices
- Can either be positive definite or not (takes advantage if they are)
- Also works for non-symmetric matrices (slightly less efficiently)

$$\mathbf{A}\mathbf{x} = \mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{b}$$

$$\mathbf{L}\mathbf{z} = \mathbf{b}$$

$$\mathbf{L}^T\mathbf{x} = \mathbf{z}$$

Once factorization is computed solution to Ax=b is obtained via forward/backward substitution

Benchmark - Laplacian system

Same system as we saw before
*(Small change : Since the matrix below is negative definite, we instead solve the system (-**A**)**x**=(-**b**) which is positive definite)*

$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
 & 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
 & & \ddots & \ddots & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
 & 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
 & & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
 & & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
 & \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
 & & \ddots & & & & 1 & & & 1 & -6 & 1 \\
 & & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$

Benchmark - Laplacian system

For symmetric matrices, PARDISO expects to be given only the upper-triangular part of the matrix (lower triangular entries should not be included in the CSR representations)

$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
 & 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
 & & \ddots & & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
 & 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
 & & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
 & & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
 & \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
 & & \ddots & & & & 1 & & & 1 & -6 & 1 \\
 & & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$

(but if you feel you need to give the lower-triangular part instead, just let PARDISO know you're using column-major/CSC instead of row-major/CSR)

Main routine (main.cpp)

SparseDirect/LaplacePARDISO_0_0

Contrast with:
LaplaceSolver/LaplaceSolver_1_3

```
#include "DirectSolver.h"
#include "Laplacian.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [...]
    array_t z = reinterpret_cast<array_t>(*zRaw);

    CSRMatrix matrix;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildUpperTriangularLaplacianMatrix(); // This takes a while ...
        timer.Stop("Initialization : ");
    }

    // Call PARDISO solver
    DirectSparseSolver(matrix, x, f);

    return 0;
}
```

Main routine (main.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "DirectSolver.h"
#include "Laplacian.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [...]
    array_t z = reinterpret_cast<array_t>(*zRaw);

    CSRMatrix matrix;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildUpperTriangularLaplacianMatrix(); // This takes a while ...
        timer.Stop("Initialization : ");
    }

    // Call PARDISO solver
    DirectSparseSolver(matrix, x, f);

    return 0;
}
```

*Very slightly modified: Previously we were specifying fixed values of **x** by setting them directly, now we obtain the same by slight adjustments to **f***

Main routine (main.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "DirectSolver.h"
#include "Laplacian.h"
#include "Timer.h"
#include "Utilities.h"

int main(int argc, char *argv[])
{
    using array_t = float (&) [XDIM][YDIM][ZDIM];

    float *xRaw = new float [XDIM*YDIM*ZDIM];
    [...]
    array_t z = reinterpret_cast<array_t>(*zRaw);

    CSRMatrix matrix;

    // Initialization
    {
        Timer timer;
        timer.Start();
        InitializeProblem(x, f);
        matrix = BuildUpperTriangularLaplacianMatrix(); // This takes a while ...
        timer.Stop("Initialization : ");
    }

    // Call PARDISO solver
    DirectSparseSolver(matrix, x, f);

    return 0;
}
```

These contain all the essential changes relative to: LaplaceSolver/LaplaceSolver_1_3

Laplacian matrix (Laplacian.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "CSRMatrixHelper.h"
#include "Laplacian.h"

inline int LinearIndex(const int i, const int j, const int k) { return ((i*YDIM)+j)*ZDIM+k; }

CSRMatrix BuildUpperTriangularLaplacianMatrix()
{
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 6.;
        if (i < XDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i+1, j, k) ) = -1.;
        if (j < YDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j+1, k) ) = -1.;
        if (k < ZDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k+1) ) = -1.; }

    // Need to put part of the "identity" matrix on nodes of the margin
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        if ( i == 0 || i == XDIM-1 || j == 0 || j == YDIM-1 || k == 0 || k == ZDIM-1 )
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 1.;

    return matrixHelper.ConvertToCSRMatrix();
}
```

Laplacian matrix (Laplacian.cpp)

SparseDirect/LaplacePARDISO_0_0

Storing only the upper-triangular part, in CSR format

```
#include "CSRMatrixHelper.h"
#include "Laplacian.h"

inline int LinearIndex(const int i, const int j, const int k) { return ((i*YDIM)+j)*ZDIM+k; }

CSRMatrix BuildUpperTriangularLaplacianMatrix()
{
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 6.;
        if (i < XDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i+1, j, k) ) = -1.;
        if (j < YDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j+1, k) ) = -1.;
        if (k < ZDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k+1) ) = -1.; }

    // Need to put part of the "identity" matrix on nodes of the margin
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        if ( i == 0 || i == XDIM-1 || j == 0 || j == YDIM-1 || k == 0 || k == ZDIM-1 )
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 1.;

    return matrixHelper.ConvertToCSRMatrix();
}
```

Benchmark - Laplacian system

Storing only the upper-triangular part, in CSR format

$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
& 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
& & \ddots & & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
& 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
& & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
& & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
& \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
& & \ddots & & & & 1 & & & 1 & -6 & 1 \\
& & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$

Laplacian matrix (Laplacian.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "CSRMatrixHelper.h"
#include "Laplacian.h"

inline int LinearIndex(const int i, const int j, const int k) { return ((i*YDIM)+j)*ZDIM+k; }

CSRMatrix BuildUpperTriangularLaplacianMatrix()
{
    static constexpr int matSize = XDIM * YDIM * ZDIM;
    CSRMatrixHelper matrixHelper(matSize);

    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++) {
        matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 6.;
        if (i < XDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i+1, j, k) ) = -1.;
        if (j < YDIM-2)
            matrixHelper( LinearIndex(i, j, k), Line
        if (k < ZDIM-2)
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k+1) ) = -1.; }

    // Need to put part of the "identity" matrix on nodes of the margin
    for (int i = 0; i < XDIM; i++)
    for (int j = 0; j < YDIM; j++)
    for (int k = 0; k < ZDIM; k++)
        if ( i == 0 || i == XDIM-1 || j == 0 || j == YDIM-1 || k == 0 || k == ZDIM-1 )
            matrixHelper( LinearIndex(i, j, k), LinearIndex(i, j, k) ) = 1.;

    return matrixHelper.ConvertToCSRMatrix();
}
```

For all values along the boundary create an equation like x_{ijk} = const

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "DirectSolver.h"
#include "Utilities.h"

#include "mkl.h"

#include <iostream>

void DirectSparseSolver(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    float (&f)[XDIM][YDIM][ZDIM],
    const bool writeOutput)
{
    MKL_INT n = matrix.mSize; // Matrix size
    MKL_INT mtype = 2;        // Real symmetric positive definite matrix
    MKL_INT nrhs = 1;         // Number of right hand sides
    void *pt[64];             // Internal solver memory pointer pt
                              // should be "int" when using 32-bit architectures, or "long int"
                              // for 64-bit architectures. void* should be OK in both cases
    MKL_INT iparm[64];        // Pardiso control parameters
    MKL_INT maxfct, mnum, phase, error, msglvl;
    MKL_INT i;
    float ddum;               // Scalar dummy (PARDISO needs it)
    MKL_INT idum;             // Integer dummy (PARDISO needs it)

    // Set-up PARDISO control parameters

    for ( i = 0; i < 64; i++ )
```

Developer Reference for Intel® Math Kernel Library - C

Submitted March 30, 2020

Search document...

Contents

Getting Help and Support

What's New

Notational Conventions

Overview

BLAS and Sparse BLAS Routines

LAPACK Routines

ScaLAPACK Routines

Sparse Solver Routines

Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface

pardiso

pardisoinit

pardiso_64

pardiso_getenv, pardiso_setenv

mkl_pardiso_pivot

pardiso_getdiag

pardiso_export

pardiso_handle_store

Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel® MKL PARDISO solver.

The Intel® MKL PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [Schenk00-2]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [Schenk00, Schenk01, Schenk02, Schenk03]. The parallel pivoting methods allow complete supernode pivoting to compromise 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 multiprocessing architecture.

Optimization Notice

Intel's compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

PARDISO solver (DirectSolver.cpp)

```
#include "DirectSolver.h"
#include "Utilities.h"

#include "mkl.h"

#include <iostream>

void DirectSparseSolver(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    float (&f)[XDIM][YDIM][ZDIM],
    const bool writeOutput)
{
    MKL_INT n = matrix.mSize; // Matrix size
    MKL_INT mtype = 2;        // Real symmetric positive definite matrix
    MKL_INT nrhs = 1;         // Number of right hand sides
    void *pt[64];             // Internal solver memory pointer pt
                              // should be "int" when using 32-bit architectures, or "long int"
                              // for 64-bit architectures. void* should be OK in both cases
    MKL_INT iparm[64];        // Pardiso control parameters
    MKL_INT maxfct, mnum, phase, error, msglvl;
    MKL_INT i;
    float ddum;               // Scalar dummy (PARDISO needs it)
    MKL_INT idum;             // Integer dummy (PARDISO needs it)

    // Set-up PARDISO control parameters

    for ( i = 0; i < 64; i++ )
```

Search document...

Contents

Getting Help and Support

What's New

Notational Conventions

› Overview

› BLAS and Sparse BLAS Routines

› LAPACK Routines

› ScaLAPACK Routines

⌄ Sparse Solver Routines

⌄ Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface

pardiso

pardisoinit

pardiso_64

pardiso_getenv, pardiso_setenv

mkl_pardiso_pivot

pardiso_getdiag

pardiso_export

pardiso_handle_store

pardiso_handle_restore

pardiso_handle_delete

pardiso_handle_store_64

pardiso_handle_restore_64

pardiso

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

```
void pardiso (_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT *mnum, const MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n, const void *a, const MKL_INT *ia, const MKL_INT *ja, MKL_INT *perm, const MKL_INT *nrhs, MKL_INT *iparm, const MKL_INT *msglvl, void *b, void *x, MKL_INT *error);
```

Include Files

- `mkl.h`

Description

The routine `pardiso` calculates the solution of a set of sparse linear equations

```
A*X = B
```

with single or multiple right-hand sides, using a parallel LU, LDL, or LL^T factorization, where A is an n-by-n matrix, and X and B are n-by-$nrhs$ vectors or matrices.

NOTE

This routine supports the Progress Routine feature. See Progress Function for details. The case of `iparm[23]`=10 does not support this feature.

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
#include "DirectSolver.h"
#include "Utilities.h"

#include "mkl.h"

#include <iostream>

void DirectSparseSolver(
    CSRMatrix& matrix,
    float (&x)[XDIM][YDIM][ZDIM],
    float (&f)[XDIM][YDIM][ZDIM],
    const bool writeOutput)
{
    MKL_INT n = matrix.mSize; // Matrix size
    MKL_INT mtype = 2;        // Real symmetric positive definite matrix
    MKL_INT nrhs = 1;         // Number of right hand sides
    void *pt[64];             // Internal solver memory pointer pt
                              // should be "int" when using 32-bit architectures, or "long int"
                              // for 64-bit architectures. void* should be OK in both cases
    MKL_INT iparm[64];        // Pardiso control parameters
    MKL_INT maxfct, mnum, phase, error, msglvl;
    MKL_INT i;
    float ddum;               // Scalar dummy (PARDISO needs it)
    MKL_INT idum;             // Integer dummy (PARDISO needs it)

    // Set-up PARDISO control parameters

    for ( i = 0; i < 64; i++ )
```

SparseDirect/LaplacePARDISO_0_0

PARDISO solver (DirectSolver.cpp)

```
// Set-up PARDISO control parameters

for ( i = 0; i < 64; i++ )
{
    iparm[i] = 0;
}
iparm[0] = 1;           // No solver default
iparm[1] = 3;           // Parallel nested dissection
iparm[3] = 0;           // No iterative-direct algorithm
iparm[4] = 0;           // No user fill-in reducing permutation
iparm[5] = 0;           // Write solution into x
iparm[6] = 0;           // Not in use
iparm[7] = 0;           // Max numbers of iterative refinement steps
iparm[8] = 0;           // Not in use
iparm[9] = 13;          // Perturb the pivot elements with 1E-13
iparm[10] = 1;          // Use nonsymmetric permutation and scaling MPS
iparm[11] = 0;          // Not in use
iparm[12] = 0;          // Maximum weighted matching algorithm is switched-off
                        // Try iparm[12] = 1 in case of inappropriate accuracy
iparm[13] = 0;          // Output: Number of perturbed pivots
iparm[14] = 0;          // Not in use
iparm[15] = 0;          // Not in use
iparm[16] = 0;          // Not in use
iparm[17] = -1;         // Output: Number of nonzeros in the factor LU
iparm[18] = -1;         // Output: Mflops for LU factorization
iparm[19] = 0;          // Output: Numbers of CG Iterations
iparm[23] = 1;          // Two-level factorization*/
iparm[26] = 1;          // Check matrix for errors
iparm[27] = 1;          // Use float precision
```

pardiso_setenv
mkl_pardiso_pivot
pardiso_getdiag
pardiso_export
pardiso_handle_store
pardiso_handle_restore
pardiso_handle_delete
pardiso_handle_store_64
pardiso_handle_restore_64
pardiso_handle_delete_64
Intel® MKL PARDISO Parameters in Tabular Form
pardiso iparm Parameter
PARDISO_DATA_TYPE
› Parallel Direct Sparse Solver for Clusters Interface
› Direct Sparse Solver (DSS) Interface Routines
› Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)
› Preconditioners based on Incomplete LU Factorization Technique
› Sparse Matrix Checker Routines
› Graph Routines
› Extended Eigensolver Routines
› Vector Mathematical

pardiso iparm Parameter

This table describes all individual components of the Intel® MKL PARDISO *iparm* parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description	
iparm[0] input	Use default values.	
	0	*iparm*[1] - *iparm*[63] are filled with default values.
	≠0	You must supply all values in components *iparm*[1] - *iparm*[63].
iparm[1] input	Fill-in reducing ordering for the input matrix. **CAUTION** You can control the parallel execution of the solver by explicitly setting the `MKL_NUM_THREADS` environment variable. If fewer OpenMP threads are available than specified, the execution may slow down instead of speeding up. If `MKL_NUM_THREADS` is not defined, then the solver uses all available processors.	
	0	The minimum degree algorithm [Li99].
	2*	The nested dissection algorithm from the METIS package [Karypis98].
	3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® MKL PARDISO Phase 1 takes significant time. **NOTE** Setting *iparm*[1] = 3 prevents the use of CNR mode (*iparm*[33] > 0) because Intel® MKL PARDISO uses dynamic parallelism.
iparm[2]	Reserved. Set to zero.	
iparm[3] input	Preconditioned CGS/CG. This parameter controls preconditioned CGS [Sonn89] for nonsymmetric or structurally symmetric matrices and Conjugate-Gradients for symmetric matrices. *iparm*[3] has the form *iparm*[3]= 10*L+K.	

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
// Set-up PARDISO control parameters

for ( i = 0; i < 64; i++ )
{
    iparm[i] = 0;
}
iparm[0] = 1;          // No solver default
iparm[1] = 3;          // Parallel nested dissection
iparm[3] = 0;          // No iterative-direct algorithm
iparm[4] = 0;          // No user fill-in reducing permutation
iparm[5] = 0;          // Write solution into x
iparm[6] = 0;          // Not in use
iparm[7] = 0;          // Max numbers of iterative refinement steps
iparm[8] = 0;          // Not in use
iparm[9] = 13;         // Perturb the pivot elements with 1E-13
iparm[10] = 1;         // Use nonsymmetric permutation and scaling MPS
iparm[11] = 0;         // Not in use
iparm[12] = 0;         // Maximum weighted matching algorithm is switched-off
                       // Try iparm[12] = 1 in case of inappropriate accuracy
iparm[13] = 0;         // Output: Number of perturbed pivots
iparm[14] = 0;         // Not in use
iparm[15] = 0;         // Not in use
iparm[16] = 0;         // Not in use
iparm[17] = -1;        // Output: Number of nonzeros in the factor LU
iparm[18] = -1;        // Output: Mflops for LU factorization
iparm[19] = 0;         // Output: Numbers of CG Iterations
iparm[23] = 1;         // Two-level factorization*/
iparm[26] = 1;         // Check matrix for errors
iparm[27] = 1;         // Use float precision
```

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
iparm[8] = 0;           // Not in use
iparm[9] = 13;          // Perturb the pivot elements with 1E-13
iparm[10] = 1;          // Use nonsymmetric permutation and scaling MPS
iparm[11] = 0;          // Not in use
iparm[12] = 0;          // Maximum weighted matching algorithm is switched-off
                        // Try iparm[12] = 1 in case of inappropriate accuracy
iparm[13] = 0;          // Output: Number of perturbed pivots
iparm[14] = 0;          // Not in use
iparm[15] = 0;          // Not in use
iparm[16] = 0;          // Not in use
iparm[17] = -1;         // Output: Number of nonzeros in the factor LU
iparm[18] = -1;         // Output: Mflops for LU factorization
iparm[19] = 0;          // Output: Numbers of CG Iterations
iparm[23] = 1;          // Two-level factorization*/
iparm[26] = 1;          // Check matrix for errors
iparm[27] = 1;          // Use float precision
iparm[34] = 1;          // Use zero-based indexing
maxfct = 1;             // Maximum number of numerical factorizations.
mnum = 1;               // Which factorization to use.
msglvl = 1;             // Print statistical information in file
error = 0;              // Initialize error flag

// Initialize the internal solver memory pointer. This is only
// necessary for the FIRST call of the PARDISO solver
for ( i = 0; i < 64; i++ )
{
    pt[i] = 0;
}
```

PARDISO solver (DirectSolver.cpp)

```
}

// 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,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during symbolic factorization");

std::cout << "Reordering completed ... " << std::endl;
std::cout << "Number of nonzeros in factors = " <<  iparm[17] << std::endl;
std::cout << "Number of factorization MFLOPS = " << iparm[18] << std::endl;

// Numerical factorization
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during numerical factorization");

std::cout << "Factorization completed ... " << std::endl;

// Back substitution and iterative refinement
phase = 33;
iparm[7] = 0;         // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

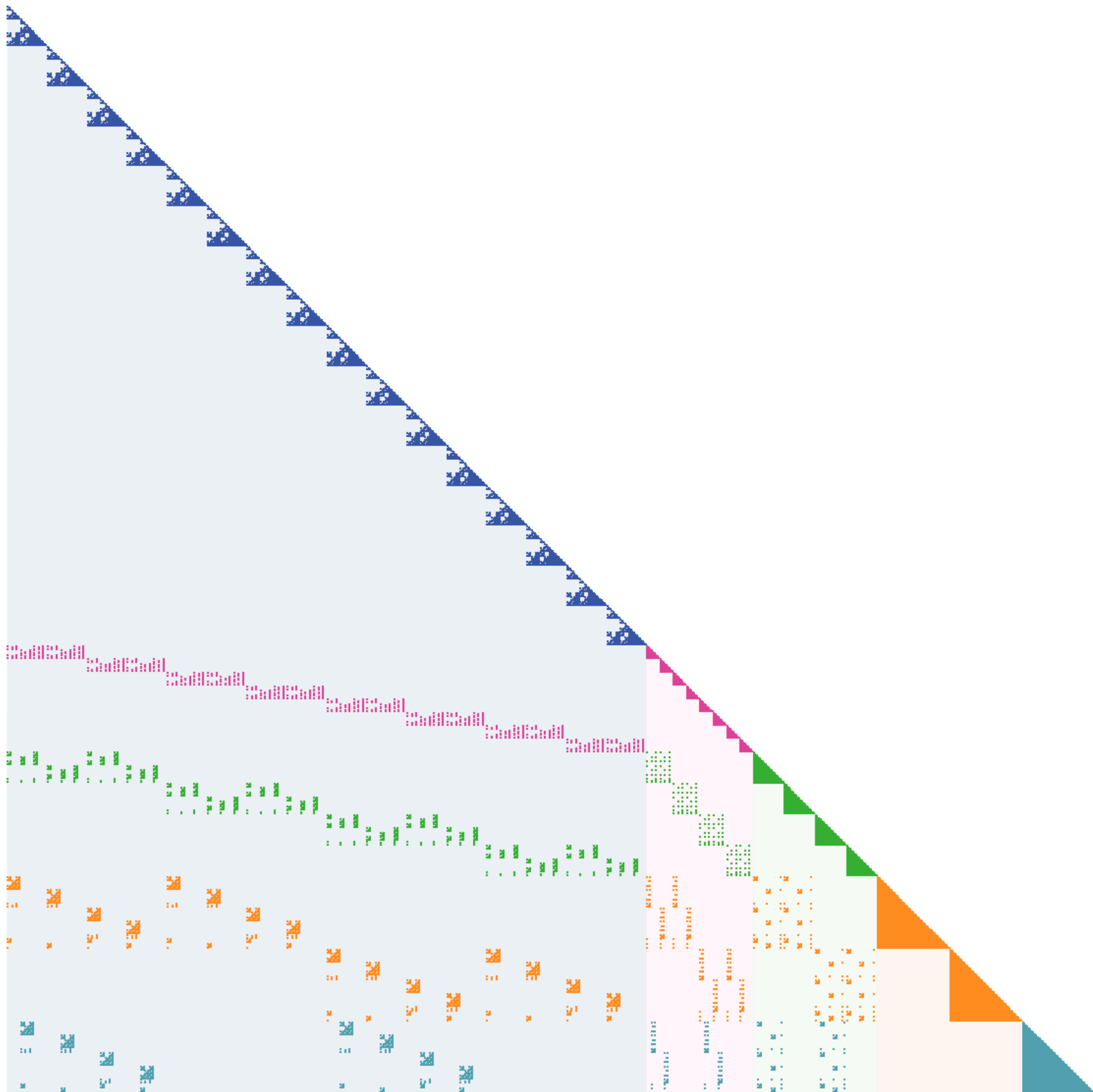
PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
}

// 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,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during symbolic factorization");

std::cout << "Reordering completed ... " << std::endl;
std::cout << "Number of nonzeros in factors = " <<  iparm[17] << std::endl;
std::cout << "Number of factorization MFLOPS = " << iparm[18] << std::endl;

// Numerical factorization
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during numerical factorization");

std::cout << "Factorization completed ... " << std::endl;

// Back substitution and iterative refinement
phase = 33;
iparm[7] = 0;         // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

Benchmark - Laplacian system

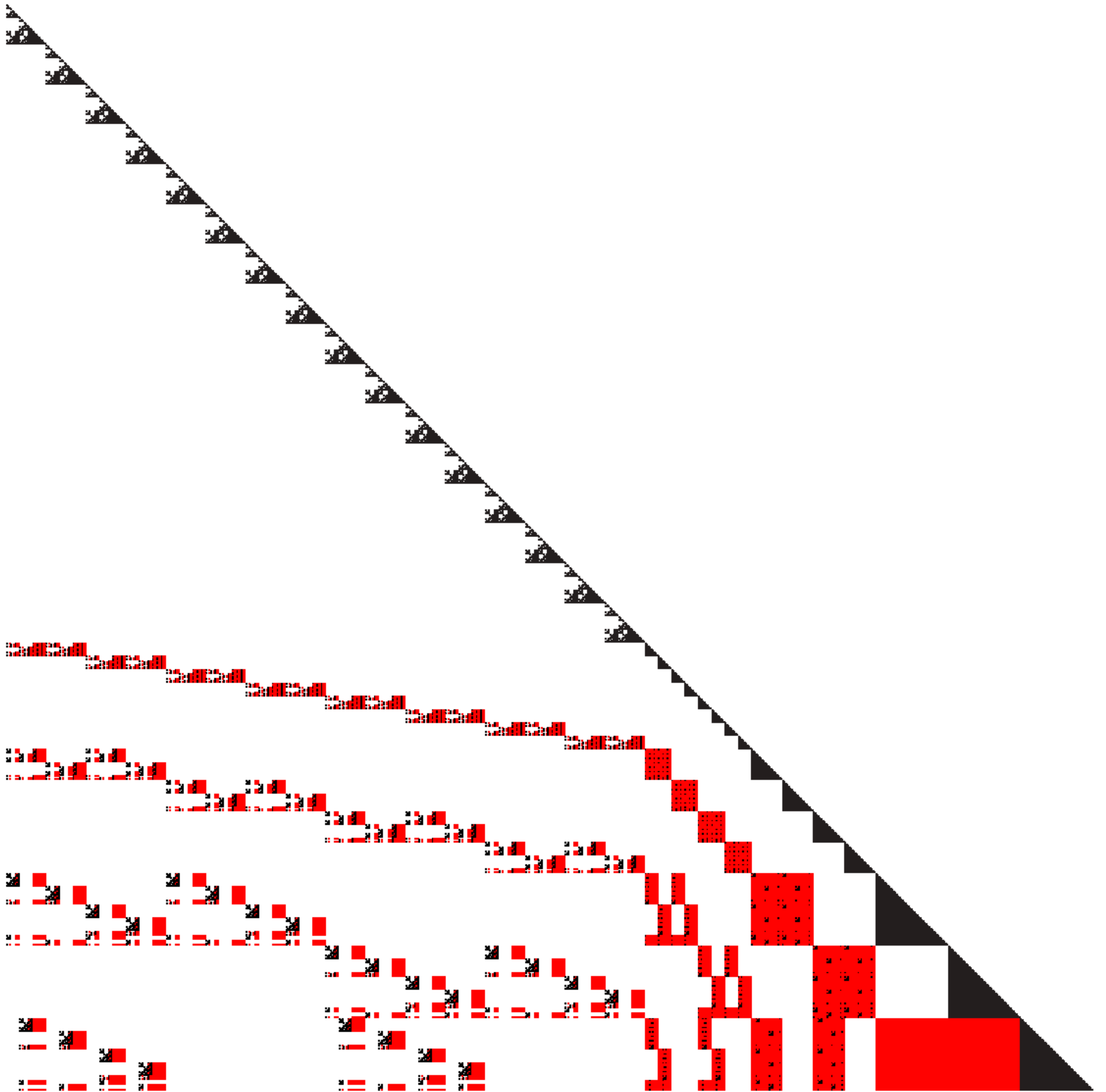
$$
\begin{pmatrix}
-6 & 1 & & & 1 & & & & 1 & & & \\
1 & -6 & 1 & & & 1 & & & & \ddots & & \\
 & 1 & -6 & 1 & & & \ddots & & & & \ddots & \\
 & & \ddots & \ddots & \ddots & & & \ddots & & & & 1 \\
1 & & & \ddots & \ddots & \ddots & & & \ddots & & & \\
 & 1 & & & \ddots & \ddots & \ddots & & & \ddots & & \\
 & & \ddots & & & \ddots & \ddots & \ddots & & & 1 & \\
 & & & \ddots & & & \ddots & \ddots & \ddots & & & 1 \\
1 & & & & \ddots & & & \ddots & \ddots & \ddots & & \\
 & \ddots & & & & \ddots & & & 1 & -6 & 1 & \\
 & & \ddots & & & & 1 & & & 1 & -6 & 1 \\
 & & & 1 & & & & 1 & & & 1 & -6
\end{pmatrix}
\mathbf{x} = \mathbf{b}
$$



PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
}

// 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,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during symbolic factorization");

std::cout << "Reordering completed ... " << std::endl;
std::cout << "Number of nonzeros in factors = " <<  iparm[17] << std::endl;
std::cout << "Number of factorization MFLOPS = " << iparm[18] << std::endl;

// Numerical factorization
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during numerical factorization");

std::cout << "Factorization completed ... " << std::endl;

// Back substitution and iterative refinement
phase = 33;
iparm[7] = 0;          // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```



PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
}

// 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,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during symbolic factorization");

std::cout << "Reordering completed ... " << std::endl;
std::cout << "Number of nonzeros in factors = " <<  iparm[17] << std::endl;
std::cout << "Number of factorization MFLOPS = " << iparm[18] << std::endl;

// Numerical factorization
phase = 22;
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
    matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
    &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
if ( error != 0 )
    throw std::runtime_error("PARDISO error during numerical factorization");

std::cout << "Factorization completed ... " << std::endl;

// Back substitution and iterative refinement
phase = 33;
iparm[7] = 0;         // Max numbers of iterative refinement steps
PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
```

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
    // Numerical factorization
    phase = 22;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during numerical factorization");

    std::cout << "Factorization completed ... " << std::endl;

    // Back substitution and iterative refinement
    phase = 33;
    iparm[7] = 0;          // Max numbers of iterative refinement steps
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, static_cast<void*>(&f[0][0][0]), &x[0][0][0], &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during solution phase");

    std::cout << "Solve completed ... " <<std::endl;

    // Termination and release of memory.
    phase = -1;            // Release internal memory
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        &ddum, matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);

    if (writeOutput) WriteAsImage("x", x, 0, 0, XDIM/2);
}
```

PARDISO solver (DirectSolver.cpp)

SparseDirect/LaplacePARDISO_0_0

```
    // Numerical factorization
    phase = 22;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during numerical factorization");

    std::cout << "Factorization completed ... " << std::endl;

    // Back substitution and iterative refinement
    phase = 33;
    iparm[7] = 0;          // Max numbers of iterative refinement steps
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, static_cast<void*>(&f[0][0][0]), &x[0][0][0], &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during solution phase");

    std::cout << "Solve completed ... " <<std::endl;

    // Termination and release of memory.
    phase = -1;            // Release internal memory
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        &ddum, matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);

    if (writeOutput) WriteAsImage("x", x, 0, 0, XDIM/2);
}
```

SparseDirect/LaplacePARDISO_0_0

PARDISO solver (DirectSolver.cpp)

```
    // Numerical factorization
    phase = 22;
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during numerical factorization");

    std::cout << "Factorization completed ... " << std::endl;

    // Back substitution and iterative refinement
    phase = 33;
    iparm[7] = 0;          // Max numbers of iterative refinement steps
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        matrix.GetValues(), matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, static_cast<void*>(&f[0][0][0]), &x[0][0][0], &error);
    if ( error != 0 )
        throw std::runtime_error("PARDISO error during solution phase");

    std::cout << "Solve completed ... " <<std::endl;

    // Termination and release of memory.
    phase = -1;            // Release internal memory
    PARDISO (pt, &maxfct, &mnum, &mtype, &phase, &n,
        &ddum, matrix.GetRowOffsets(), matrix.GetColumnIndices(),
        &idum, &nrhs, iparm, &msglvl, &ddum, &ddum, &error);

    if (writeOutput) WriteAsImage("x", x, 0, 0, XDIM/2);
}
```