Presentation of XLiFE++

Eigenvalues Solver & OpenMP

Manh-Ha NGUYEN

Unité de Mathématiques Appliquées,
ENSTA - Paristech

25 Juin 2014

- Independent from external packages (BLAS, LAPACK, etc, ...)
- Capable of solving large eigenvalues problems with efficiency
- Simple utilization on the level of `TermMatrix`
- Easy to develop new eigenvalue methods

Basing on two modules:

EigenCore: Lapack-like functions for solving small size eigenvalue problem

EigenSparse: Implementation of eigenvalue solving methods for sparse matrices

- Simple and easy to use
- Lapack-like functions
- Tool for small eigenvalues problem

Components

- utils
- decomposition
- eigenSolver

1. utils
   - MatrixEigenDense dense row by row matrix
   - VectorEigenDense vector by line or column
2. decomposition
   - Henssenberg
   - QR
   - Tridiagonalization
   - Real/Complex Schur
3. eigenSolver
   - SelfAdjoint
   - GeneralizedSelfAdjoint
   - Real/Complex EigenSolver

Example:

```
HessenbergDecomposition<MatrixEigenDense<Real> > rHess(5);
Tridiagonalization<MatrixEigenDense<Real> > tridiag(5);
RealSchur<MatrixEigenDense<Real> > rSchur(5);

MatrixEigenDense<Real> rMatTest(5);
rMat.loadFromFile(dataPathTo("EigenSolverIntern/rSym5.data").c_str());

rHess.compute(rMat);
std::out << "Hessenberg matrix " << rHess.matrixH() << std::endl;
std::out << "Unitary matrix " << rHess.matrixQ() <<std::endl;

tridiag.compute(rMat);
std::out << "Unitary matrix " << tridiag.matrixQ() << "\n";

rSchur.compute(rMat);
std::out << "T matrix " << rSchur.matrixT() << std::endl;;
std::out << "Unitary matrix " << rSchur.matrixU() << std::endl;
```

```
SelfAdjointEigenSolver<MatrixEigenDense<Real> > eigStd(20);

MatrixEigenDense<Real> rMat(20);
rMat.loadFromFile(dataPathTo("EigenSolverIntern/rSymPos20.data").c_str());

Vector<Real> rEigVal = eigStd.eigenvalues();
MatrixEigenDense<Real> rEigVec = eigStd.eigenvectors();
std::out << "Eigen values " << rEigVal << "\n";
std::out << "Eigen vectors " << rEigVec << "\n";
```

Implementation of different eigenvalues solver methods. Up to now, XLIFE++ has had two methods corresponding different cases

- Krylov-Schur: Effective in computing eigenvalues in the end of the spectrum of matrix A if these eigenvalues are well seperated from the remaining spectrum or if it is applied to a shifted and inverted matrix operator $(A - \sigma I)^{-}1$. Like Arpack, it is is capable of solving large scale symmetric, non-symmetric, and generalized eigenproblems
- Davidson: Capable of solving standard and generalized symmetric eigen problems. In some cases, it could find eigenvalues and the corresponding eigenvalues faster than Krylov-Schur.

Implementation of different eigenvalues solver methods. Up to now, XLIFE++ has had two methods corresponding different cases

- Krylov-Schur: Effective in computing eigenvalues in the end of the spectrum of matrix A if these eigenvalues are well seperated from the remaining spectrum or if it is applied to a shifted and inverted matrix operator $(A - \sigma I)^{-}1$. Like Arpack, it is is capable of solving large scale symmetric, non-symmetric, and generalized eigenproblems
- Davidson: Capable of solving standard and generalized symmetric eigen problems. In some cases, it could find eigenvalues and the corresponding eigenvalues faster than Krylov-Schur.

Implementation of different eigenvalues solver methods. Up to now, XLIFE++ has had two methods corresponding different cases

- Krylov-Schur: Effective in computing eigenvalues in the end of the spectrum of matrix A if these eigenvalues are well seperated from the remaining spectrum or if it is applied to a shifted and inverted matrix operator $(A - \sigma I)^{-}1$. Like Arpack, it is is capable of solving large scale symmetric, non-symmetric, and generalized eigenproblems
- Davidson: Capable of solving standard and generalized symmetric eigen problems. In some cases, it could find eigenvalues and the corresponding eigenvalues faster than Krylov-Schur.

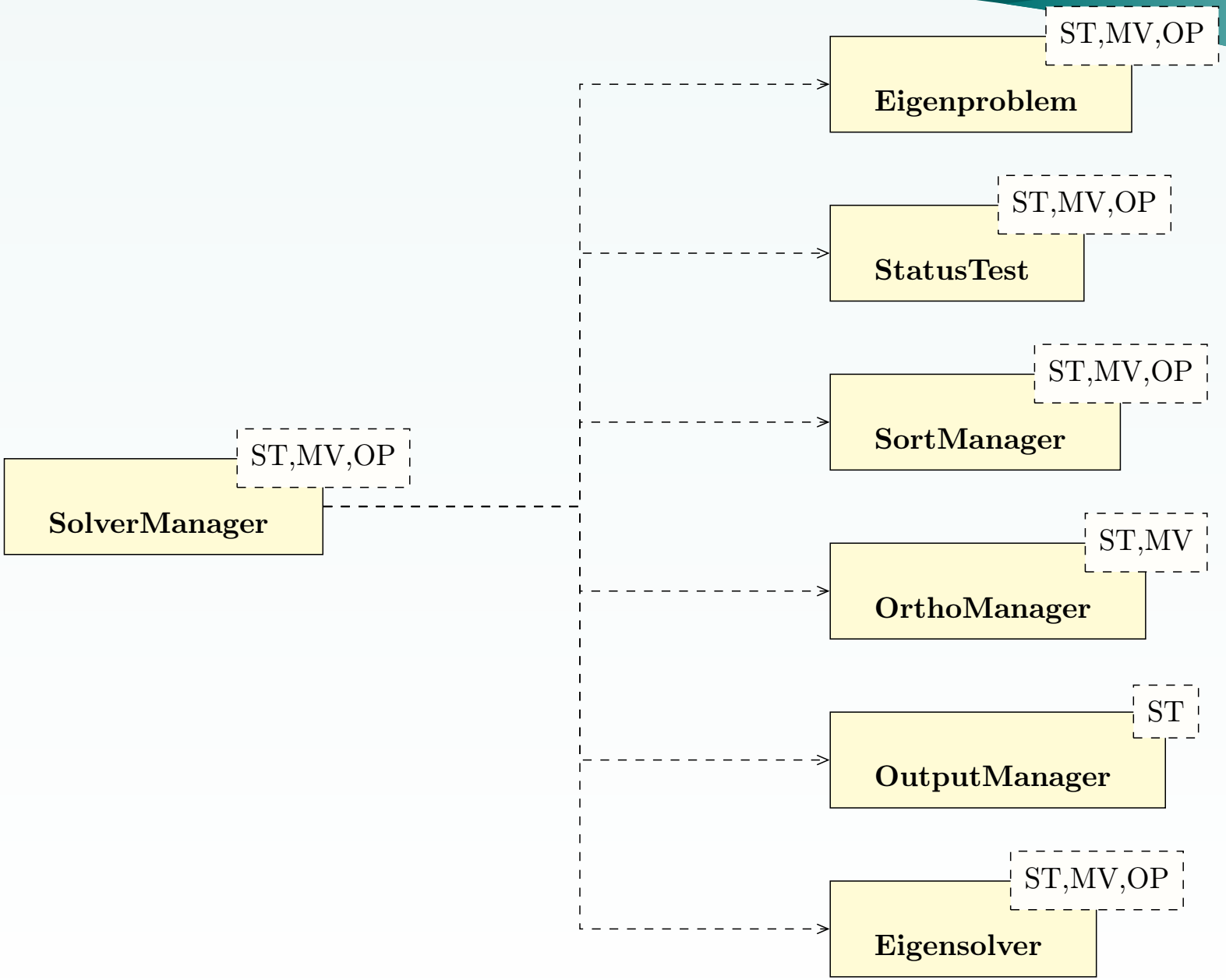
Implementation of different eigenvalues solver methods. Up to now, XLIFE++ has had two methods corresponding different cases

- Krylov-Schur: Effective in computing eigenvalues in the end of the spectrum of matrix A if these eigenvalues are well seperated from the remaining spectrum or if it is applied to a shifted and inverted matrix operator $(A - \sigma I)^{-}1$. Like Arpack, it is is capable of solving large scale symmetric, non-symmetric, and generalized eigenproblems
- Davidson: Capable of solving standard and generalized symmetric eigen problems. In some cases, it could find eigenvalues and the corresponding eigenvalues faster than Krylov-Schur.

Krylov-Schur is used as a default method to search for eigenvalues and the corresponding eigenvectors.

EigenSparse, like other libraries of XLIFE++, utilizes trait classes to define interfaces for the scalar field, multi-vector and matrix operators. Only three template parameters should be provided, could one use the available eigenvalues solver of XLIFE++ with ease.
Three essential template parameters:

1. scalar type, describing the field over which the vectors and operators are defined
2. multivector type over the given scalar field, providing a data structure that denotes a collection of vectors
3. operator type over the given scalar field, providing a linear operators used to define eigenproblems


ST,MV,OP
Eigenproblem
ST,MV,OP
StatusTest
ST,MV,OP
SortManager
ST,MV,OP
SolverManager
ST,MV
OrthoManager
ST
OutputManager
ST,MV,OP
Eigensolver

1. SolverManager:
   - encapsulate all functionalities of eigenSparse
   - enable an easy-to-use interface to eigenSparse
2. Eigenproblem defines a minimum interface expected of an eigenvalue problem
   - matrix A and M of $Ax = \lambda Mx$
   - initial vector
   - number of eigenvalues to be computed
3. StatusTest defines some criterion to stop solver iteration as provided
   - some convergence statisfied
   - some parts of current solutions are accurate enough and removed from iteration (locking)
   - solver has performed a sufficient or excessive number of iterations
4. SortManager performs the sorting in a specific manner
5. OrthoManager provides different approaches of orthogonalization and orthonormalization (Euclidean inner product, M-inner product, ...etc)
6. OutputManager controls what output is printed and where output is printed to with regard to the verbosity.
7. EigenSolver implement specific eigenvalues solving methods. Up to now, there are two projection methods implemented:
   - Krylov-Schur: Krylov subspace and Schur decomposition
   - Davidson: non-Krylov subspace

Example:

1. compute eigenvalues and the corresponding eigenvector in regular mode

```
EigenElements eigs=eigenSolver(A, B, 10,"SM");
```

2. in several cases, eigenvalues close to the reasonable shift σ are expected

```
EigenElements eigs=eigenSolver(A, B, 10, Complex(10.0,0.0);
```

OpenMP(Open Multi-Processing) is:

- shared memory processing (SMP)
- simple to use
- capable of paralleling with a reasonable performance

In XLIFE++, two consuming-time operations have been paralleled with OpenMP

- matrix-vector multiplication
- matrix factorization

Matrix-vector multiplication largely serves for

- iterative solver
- eigensolver

Among sparse matrix format, compressed storage row (CSR) is naturally suitable for parallelizing sparse matrix-vector multiplication (SpMV).
Common data structure of a $m \times n$ CSR matrix

- values[nz] store the value of each non-zero element in matrix A
- colIndex[nz] stores the column index of each element in *val[nz]* array
- rowPointer[m+1] stores the index of the first non-zero element of each row and *rowPointer[m] = nz*

Constraint on the problem of load balancing, scheduling overhead and synchronization overheads, it's better to apply the *row partitioning* scheme: The matrix will be partitioned into blocks of row by the number of threads.

Matrix-factorization mainly serves for direct solver $Ax = y$
XLiFE++ has supported some popular factorization methods: LU, LDLt and LDL*; all of them have a parallelized version with OpenMP. Because of factorization algorithm, only skyline (or band or profile) storage is suitable to be factorized.
Common data structure of a $m \times n$ skyline matrix

- values[nz] store the value of diagonal, value of each non-zero row of lower part then value of non-zero column of upper part in matrix A
- colPointer[n] stores the index of the first non-zero element of each column of the upper part
- rowPointer[m] stores the index of the first non-zero element of each row of the lower part

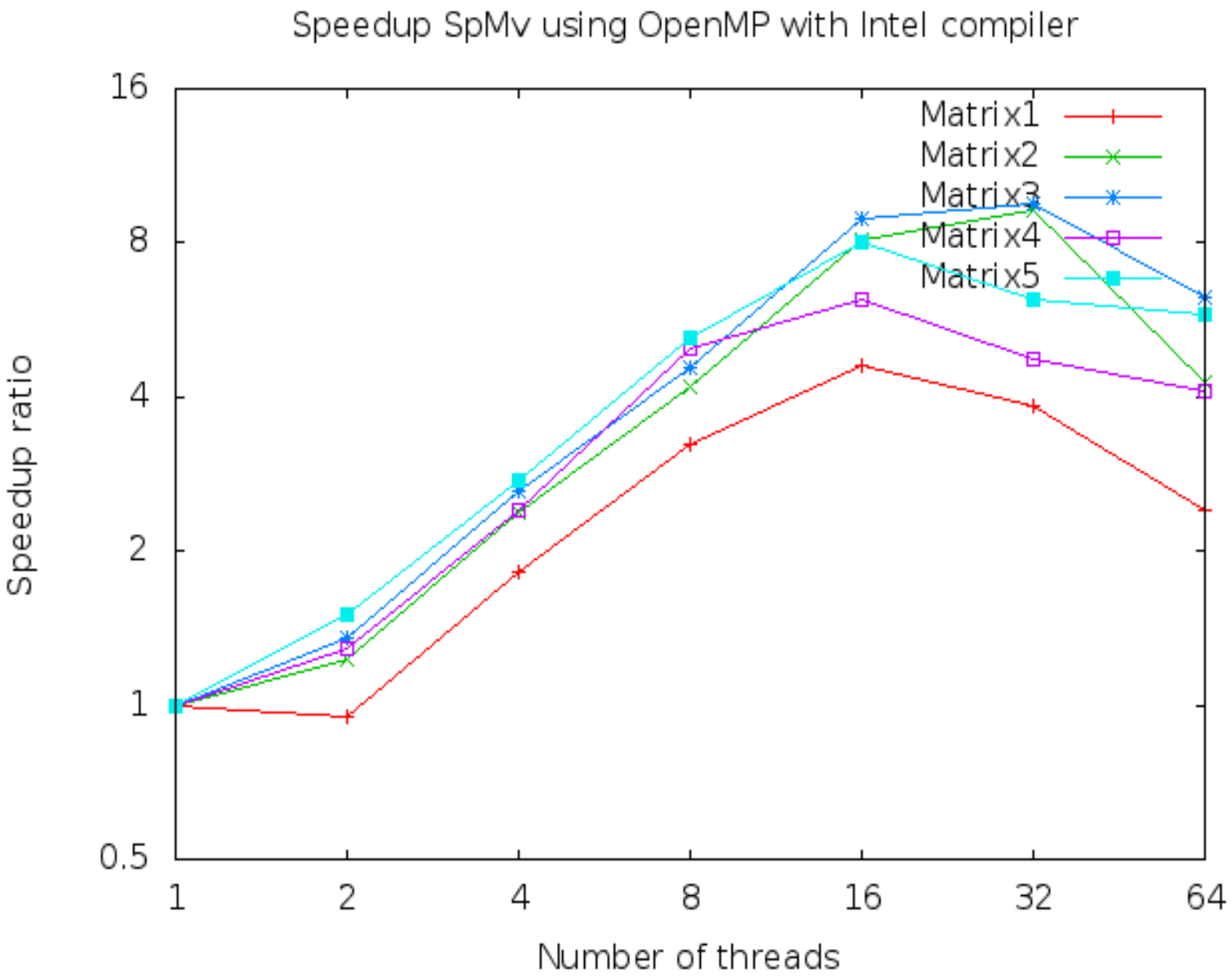
The idea behind multi-threaded factorization is simple: Instead of implementing the factorization element by element as in the serial version, we make the algorithm work on block by block. For each iteration, block on diagonal is processed then block on the row and column corresponding to this diagonal block.

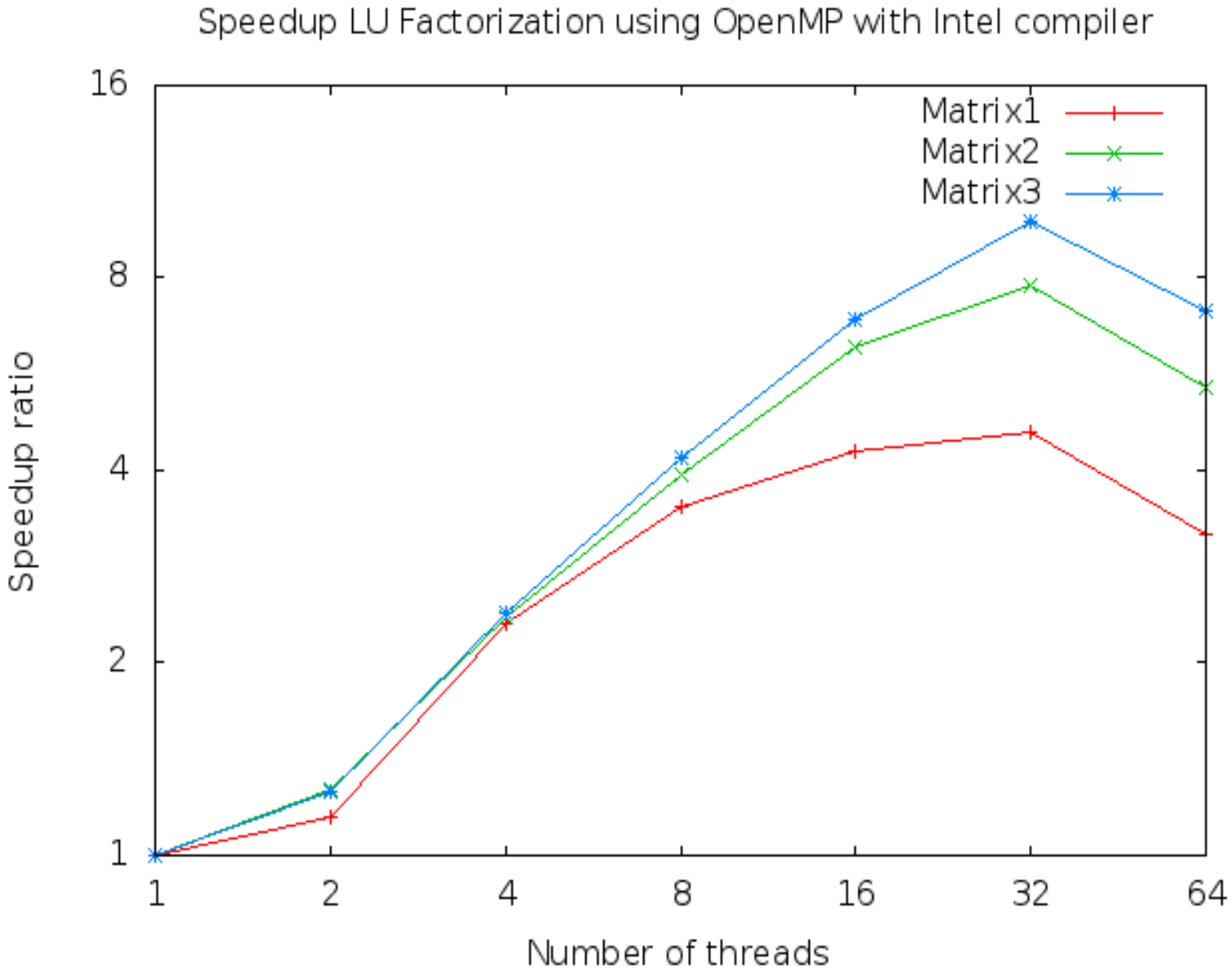
Some results with OpenMP

1. Test matrices

Matrix	*Domain*	*ElementType*	*NoSubdivision*	*Order*	*Size*	*Nnz*
Matrix1	Rectangle	Triangle	20	2	40401	464601
Matrix2	Rectangle	Quadrangle	15	3	51076	1267876
Matrix3	Rectangle	Quadrangle	20	3	90601	2253001
Matrix4	Cube	Tetrahedron	6	2	230945	6422657
Matrix5	Cube	Hexahedron	10	2	1030301	64481201

2. Environment
   - Medoc server: 32 physical cores with Hyperthreading (64 logical cores), 110 GB RAM
   - Intel compiler 12.0.5 with GNU 4.6
3. Compilation options
   - -opt-prefetch -O3


Speedup SpMv using OpenMP with Intel compiler
Speedup ratio
Number of threads
16
8
4
2
1
0.5
1
2
4
8
16
32
64
Matrix1
Matrix2
Matrix3
Matrix4
Matrix5


Speedup LU Factorization using OpenMP with Intel compiler
Speedup ratio
Number of threads
Matrix1
Matrix2
Matrix3
16
8
4
2
1
1
2
4
8
16
32
64