Introduction to ScaLAPACK library

Karim Hasnaoui

Institut du Développement et des Ressources en Informatique Scientifique, UAR851 - Orsay
Maison de la Simulation, UAR3441 - Saclay

karim.hasnaoui@idris.fr

Outline

Outline

Introduction : What is ScaLAPACK ?

- ScaLAPACK is the Scalable Linear Algebra Package : LAPACK library in a HPC framework
- First release in February 1995 and latest in February 2022
- ScaLAPACK is mostly developed by Univ. of Tennessee ; Univ. of California, Berkeley ; Univ. of Colorado Denver ; and NAG Ltd.
- The aim of ScaLAPACK is to solve all the possible dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems
- ScaLAPACK is designed for heterogeneous computing
- Portable on any computer that supports MPI or PVM
- "Supposed to be easy to use" : LAPACK and ScaLAPACK interfaces look as similar as possible

Introduction : What is ScaLAPACK ?

- OpenSource under BSD license, free of charge and can be download from Netlib or GitHub :
  http://www.netlib.org/scalapack
  https://github.com/Reference-ScaLAPACK/scalapack/

- Commercial distributions :
  - Intel MKL (Math Kernel Library)
  - IBM ESSL (Engineering and Scientific Subroutine Library)
  - AMD ACML (AMD Core Math Library)

- Other similar projects in a parallel framework :
  - PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) :
    http://icl.cs.utk.edu/plasma/
  - MAGMA (Matrix Algebra on GPU and Multicore Architectures) :
    http://icl.cs.utk.edu/magma/

Introduction : Why this lecture ?

- ScaLAPACK seems complicated for programmers that know and don't know parallel programing

- The ScaLAPACK is a huge tool box, therefore the documentation is voluminous :
  http://netlib.org/scalapack/slug/index.html
  http://www.netlib.org/scalapack/explore-html

- Even after reading the online tutorial, you'll get a good overview about the possibilities and performances, but you won't be able to use it yet :
  http://www.netlib.org/scalapack/tutorial

- Examples are available, but they cover only few functionalites and don't show how to distribute the data between parallel processes :
  http://www.netlib.org/scalapack/examples

Introduction : About this lecture ?

- Lecture contents :
  - Learning the most important ScaLAPACK basics
  - Lecture mostly based on a tutorial
  - Discussion about performances for a practical example
- Requirements :
  - Knowledge of MPI is not required
  - Fortran 90/95
  - Simple basics in UNIX systems
- Tutorial :
  1. "Hello World ! ! !" code : how to set a grid, to compile and to run a code
  2. How to set the sub matrices sizes properly
  3. How to distribute data to parallel processes
  4. Matrix-Matrix addition and multiplication
  5. Matrix inversion
  6. Symmetric-Matrix diagonalization
  7. Final problem : $\exp(A)$
- Small discussion about performances

Outline

Generalities : ScaLAPACK contents

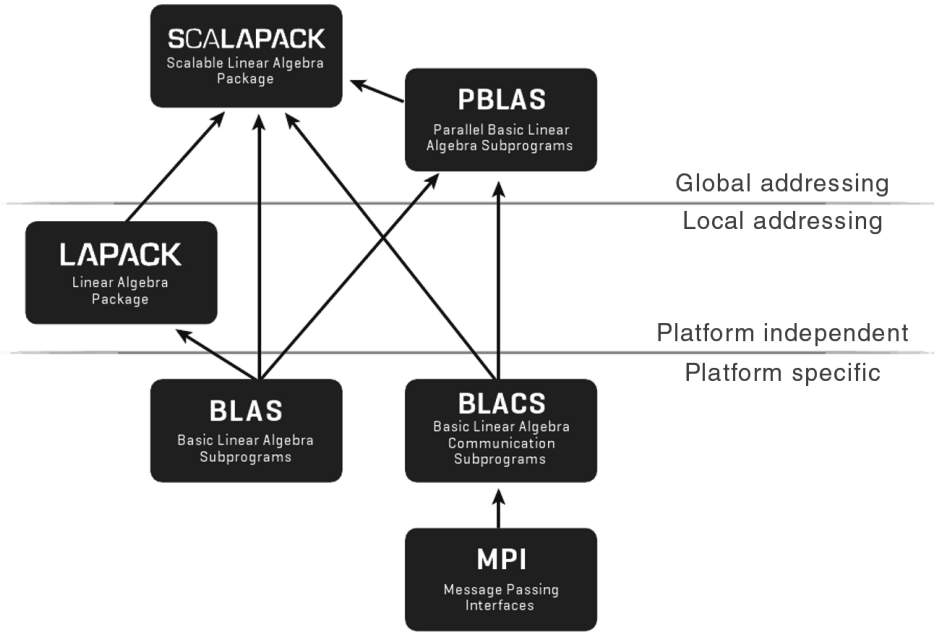


Figure taken from netlib.org

- BLACS (Basic Linear Algebra Communication Subprograms) : MPI communications for linear algebra operations
- BLAS (Basic Linear Algebra Subprograms) : Basic linear algebra operations such as dot-products, matrix-vector multiplication, and matrix-matrix multiplication
- PBLAS is the parallel version of BLAS
- LAPACK (Linear Algebra Package) : Contains other linear algebra operations

Generalities : How MPI/BLACS processes are organized ?

Example with 12 processes on a 3x4 grid :

0	1	2	3
4	5	6	7
8	9	10	11

- Here it is assumed that one process is bound to one core
- Processes are embedded in a two-dimensional grid
- The grid is set with **BLACS** subroutines (Tutorial part 1)
- Information about processes can be obtained with **BLACS** subroutines too
- For each process, data are obtained from the master process or set locally

Generalities : How data are organized on the grid process ?

Example : a 5x5 matrix partitioned in 2x2 blocks on a 2x2 BLACS grid

$$\left[\begin{array}{ccc|cc} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ \hline a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ \hline a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|cc} a_{1,1} & a_{1,2} & a_{1,5} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,5} & a_{2,3} & a_{2,4} \\ a_{5,1} & a_{5,2} & a_{5,5} & a_{5,3} & a_{5,4} \\ \hline a_{3,1} & a_{3,2} & a_{3,5} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,5} & a_{4,3} & a_{4,4} \end{array}\right]$$

- The user must choose the size of block data that will be communicated between processes
- The data must be distributed within a 2D cyclic-cyclic scheme
- The first step is to calculate the matrix sizes for each process (Tutorial part 2)
- The second step is to distribute properly the data to each process while respecting the 2D cyclic-cyclic scheme (Tutorial part 3)

Generalities : Why do the 2D cyclic-cyclic scheme must be used ?

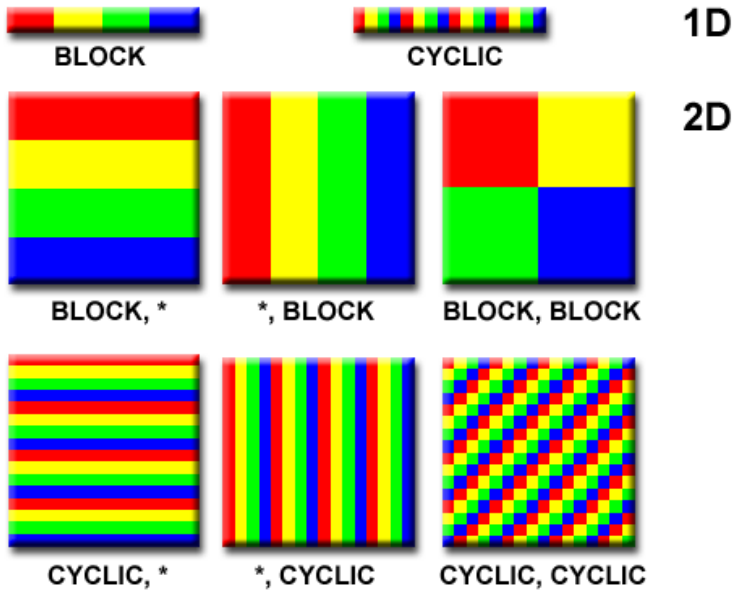


Figure taken from the LLNL parallel computing tutorial

- There are few ways to distribute data
- The most natural way is to use the 2D block-block scheme
- The 2D cyclic-cyclic scheme is much more efficient because it reduces the latences and communications among processes
- The ScaLAPACK subroutines are better optimized if the 2D cyclic-cyclic scheme is used

Generalities : How the linear algebra operations are finally done ?

- Once data is distributed within a 2D cyclic-cyclic scheme, the linear algebra operations can be performed with ScaLAPACK and/or PBLAS subroutines (Tutorial part 4 - 7)

- The ScaLAPACK and PBLAS subroutines take care about the communications between processes within a given context

- ScaLAPACK and PBLAS subroutines almost look similar to the LAPACK and BLAS subroutines

  Example for matrix multiplication :
  - BLAS : `call dgemm(M, N, A(IA,JA), lda...)`
  - PBLAS : `call pdgemm(M, N, A, IA, JA, desca...)`

- Most of LAPACK and BLAS subroutines have been transcribed

- Once the linear algebra operations are done, and only if it is needed, the user must take care of redistributing all the data to the master process

Generalities : Summary about all the steps

The linear algebra operations are done in these multiple steps :

1. The grid is set with **BLACS**
2. The size of block data that will be communicated between processes must be set
3. Each process calculates the dimension of the local matrix by using **BLACS** subroutines and declares the local matrices
4. The master process sends the sub blocs within a cyclic distribution to each process by using **BLACS** subroutines
5. Linear algebra operations are performed by using the **ScaLAPACK/PBLAS** subroutines in a parallel framework
6. Once the local calulations are done, and if it is needed, the master process receives the sub blocs from each processes by using **BLACS** subroutines

Outline

Tutorial - 0 : How to install the ScaLAPACK library

- For Debian distribution and its derivatives :
  `apt-get install libscalapack-openmpi1 libblacs-openmpi1 libscalapack-mpi-dev libblacs-mpi-dev scalapack-doc blacs-mpi-test scalapack-mpi-test scalapack-test-common`

- For RedHat distribution and its derivatives :
  `yum install scalapack-common.x86_64 scalapack-openmpi.x86_64 scalapack-openmpi-devel.x86_64 scalapack-openmpi-static.x86_64`

- The ScaLAPACK library can be compiled from scratch and the sources are downloadable from the following link :
  http://www.netlib.org/scalapack

- For Windows, please read the following documentation :
  http://icl.cs.utk.edu/lapack-for-windows/scalapack

- The other way is to install the Intel Compiler with the Intel MKL :
  https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html

Tutorial - 1 : The “hello world ! ! !” code

Aim of this part :

- How to set and release a **BLACS** grid
- How to get information about BLACS grid and processes
- How to compile and execute the code that uses ScaLAPACK library

Tutorial - 1 : The "hello world ! ! !" code

How to set the BLACS grid ?

- The BLACS grid is initialized with :

```
call blacs_gridinit(ictxt, order, nprow, npcol)
```

nprow and **npcol** are the number of rows and columns of the grid
ictxt is the MPI/BLACS context
order indicates how to map processes to BLACS grid

- BLACS communications are tied to a context. The user can define its own context or get the default system context via :

```
call blacs_get(0, 0, ictxt)
```

- The **order** indicates how to map processes to BLACS grid.
There are 3 choices :
  - 'Row' : Use row-major natural ordering
  - 'Col' : Use column-major natural ordering
  - 'Else' : Use row-major natural ordering

Tutorial - 1 : The "hello world ! ! !" code

How to release the BLACS grid ?

- The grid must be released when it is no longer needed :

```
call blacs_gridexit(ictxt)
```

- When a process has finished all use of BLACS, all contexts and memory that BLACS have allocated must be released :

```
call blacs_exit(0)
```

How to get information about BLACS grid and processes ?

- The rank process and the number of total processes are obtained with :

```
call blacs_pinfo(rank, nb_procs)
```

- The process indexes are obtanied with :

```
call blacs_gridinfo(ictxt, nprow, npcol, myrow, mycol)
```

where **myrow** and **mycol** are respectively the row and column indexes

Tutorial - 1 : The "hello world ! ! !" code

How to compile and execute a code that uses ScaLAPACK library ?

- For the standard ScaLAPACK library, for compilation with the adapted library linking is the following :

```
mpif90 my_source.f90 -o my_binary -llapack
-lscalapack-openmpi -lblas -lf77blas -lcblas
-lblacsF77init-openmpi -lblacs-openmpi -lblacsF77init-open
```

- For this tutorial, the Intel MKL will be used. The compilation with the adapted library linking is defined by :

```
mpif90 my_source.f90 -o my_binary -L${MKLROOT}/lib/intel64
-lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential
-lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm
```

  Depending of the Intel MKL version, the proper library linking can be checked with the following link :
  https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

- The code can be run with the following command :

```
mpirun -n number_of_processes my_binary
```

Tutorial - 1 : The “hello world ! ! !” code

Work to do :

- Compile and run the code
- Same procedure as before with different number of processes
  Warning : **nb_procs = nprow * npcol**
- Bonus : Modify the **order** parameter by changing it from “row-major” ordering to “column-major” ordering, and check how the processes are mapped on the grid

Tutorial - 2 : How to set the sub matrix sizes properly

Aim of this part :

- How to calculate the sub matrix sizes properly
- Learning how to set the descriptor for each process

Tutorial - 2 : How to set the sub matrices sizes properly

Example : a 5x5 matrix partitioned in 2x2 blocks on a 2x2 BLACS grid

$$\left[\begin{array}{ccc|cc} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ \hline a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ \hline a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{array}\right] \Longrightarrow \left[\begin{array}{ccc|cc} a_{1,1} & a_{1,2} & a_{1,5} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,5} & a_{2,3} & a_{2,4} \\ a_{5,1} & a_{5,2} & a_{5,5} & a_{5,3} & a_{5,4} \\ \hline a_{3,1} & a_{3,2} & a_{3,5} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,5} & a_{4,3} & a_{4,4} \end{array}\right]$$

- The user must choose the size of block data that will be communicated among processes
- The dimension is about **mb** * **nb** where **mb** and **mb** are called the blocking factors
- How to calculate the sub matrix sizes as a function of the global matrix size, **mb** and **mb**, and for a given process ?

$\Longrightarrow$ the NUMROC subroutine does it ! ! !

Tutorial - 2 : How to set the sub matrices sizes properly

- The NUMROC subroutine computes the number of rows or columns of a distributed matrix owned by a given process

- The NUMROC subroutine is called as the following :

```
call numroc(m, mb, iproc, isrcproc, procs)
```

m is the dimension of the global matrix
mb is the blocking factor
iproc is the coordinate of the process whose local array row or column
isrcproc is the coordinate of the process whose local array row or column
nprocs is the total number processes over which the matrix is distributed

- Example for a global **m * n** matrix distributed on a **loc_m * loc_n** local matrices, the local dimensions can be calculated such as :

```
loc_m = call numroc(m, mb, myrow, rsrc, nprow)
loc_n = call numroc(n, nb, mycol, csrc, npcol)
```

Tutorial - 2 : How to set the sub matrix sizes properly

- The user must determine descriptors which are used by ScaLAPACK and PBLAS subroutines to encapsulate information on distributed matrices

- A descriptor has 9 parameters :

  desc(1) = **dtype** = 1
  desc(2) = **ictxt** (BLACS context)
  desc(3) = **m** (Number of rows for the global matrix)
  desc(4) = **n** (Number of columns for the global matrix)
  desc(5) = **mb** (Row blocking factor)
  desc(6) = **nb** (Column blocking factor)
  desc(7) = **rsrc** (Index over which the first row of the matrix is distributed)
  desc(8) = **csrc** (Index over which the first column of the matrix is distributed)
  desc(9) = **lld** (Leading dimension of the matrix storing the local blocks of the distributed matrix)

- The descriptor can be initialized as the following :

```
call descinit(desc, m, n, mb, nb, rsrc, csrc, ictxt, lld, info)
```

Tutorial - 2 : How to set the sub matrix sizes properly

Work to do :

- Calculate the local dimensions of the sub matrices by implementing the **numroc** subroutine
- Initialize the descriptor by using the **descinit** subroutine
  Tip : **lld = loc_m**
- For a given topology of the **BLACS** grid, modify the value of the blocking factors and check how the sub matrix sizes change

Tutorial - 3 : How to distribute data to parallel processes

Aim of this part :

- How to distribute data within a 2D cyclic-cyclic scheme among processes

- Extras :
  - Notations
  - Point to point communications
  - Collective communications

Tutorial - 3 : How to distribute data to parallel processes

- As it has been presented in the previous section, the data must be distributed within a 2D cyclic-cyclic scheme

- One way is to use collective and/or point to point communicators by addressing manually the data in order to obtain the 2D cyclic-cyclic scheme
  $\Longrightarrow$ Extremely hard to write and algorithms become rapidly unreadable

- Since the last versions of ScaLAPACK, there are subroutines that facilitate this type of distribution

- For real matrices, sending a global **matrix_A** to local sub matrices **matrix_A_loc** can be done as the following :

```
call pdgemr2d(m, n, matrix_A, ia, ja, desc_A, matrix_A_loc,
ia_loc, ja_loc, desc_A_loc, ictxt)
```

  where **ia**, **ja**, **ia_loc** and **ja_loc** are the global row and column indexes which point to the beginning of matrix A and its submatrices

Tutorial - 3 : How to distribute data to parallel processes

Work to do :

- Distribute a global **matrix_A** within a 2D cyclic-cyclic scheme to the local sub matrices **matrix_A_loc** by using the **pdgemr2d** subroutine
- By choosing a given process, check localy if the sub matrix presents a 2D cyclic-cyclic distribution
- Resend back all the sub matrices to a global matrix, and check if it is equal to the original global matrix
- Tips :
  - One descriptor must be set for the global matrix
  - A descriptor must be set for each sub matrices
  - For this case it is better to set also two grids : one which contains only one process for the global matrix and another one which contains all the processes for the sub matrices
  - The matrix can be initialized with the **init_matrix** subroutine

Tutorial - 3 (Extras) : Notations

Notations : The communication subroutines that will be presented on the next slides with _ are type-dependent routines

_	Meaning
I	Integer data is to be operated on
S	Single precision real data is to be operated on
D	Double precision real data is to be operated on
C	Single precision complex data is to operated on
Z	Double precision complex data is to operated on

Tutorial - 3 (Extras) : Point to point communications

Point to point communication subroutines :

- A given process sends a matrix **A** and a given process receives it. Therefore, the send and receive subroutines must be used together

- Send subroutines :
  `_gesd2d(ictxt, m, n, A, lda, rdest, cdest)`

  `_trsd2d(ictxt, uplo, diag, m, n, A, lda, rdest, cdest)`

- Receive subroutines :
  `_gerv2d(ictxt, m, n, A, lda, rsrc, csrc)`

  `_trrv2d(ictxt, uplo, diag, m, n, A, lda, rsrc, csrc)`

- **ge** and **tr** denote subroutines for general and triangular matrices

- Additional information about the functional specifications can be found at the following link : http://www.netlib.org/blacs/BLACS/QRef.html

Tutorial - 3 (Extras) : Collective communications

Collective communication subroutines :

- A given process sends a matrix **A** to all the processes. Therefore, the broadcast send and receive subroutines must be used together

- Broadcast send subroutines :
  `_gebs2d(ictxt, scope, top, m, n, A, lda)`

  `_trbs2d(ictxt, scope, top, uplo, diag, m, n, A, lda)`

- Broadcast receive subroutines :
  `_gebr2d(ictxt, scope, top, m, n, A, lda, rsrc, csrc)`

  `_trbr2d(ictxt, scope, top, uplo, diag, m, n, A, lda, rsrc, csrc)`

- **ge** and **tr** denote subroutines for general and triangular matrices

- Additional information about the functional specifications can be found at the following link : http://www.netlib.org/blacs/BLACS/QRef.html

Tutorial - 4 : Matrix-Matrix addition and multiplication

Aim of this part :

- Parallel Matrix-Matrix addition by just distributing the matrices on processes

- Parallel Matrix-Matrix multiplication by using the **PBLAS** subroutine **pdgemm**

- **pdgemm** evaluates **C** $= \alpha$ * **A** * **B** $+ \beta$ * **C** and it works as the following :

```
call pdgemm(transa, transb, m, n, k, alpha, matrix_A_loc, ia,
ja, desc_loc_A, matrix_B_loc, ib, jb, desc_B_loc, beta,
matrix_C_loc, ic, jc, desc_C_loc)
```

  where **transa** and **transb** can take 3 values :
  - 'N' or 'n' to keep the matrix as the same
  - 'T' or 't' to transpose the matrix
  - 'C' or 'c' to conjugate the matrix

- Extras : **PBLAS** notations and subroutines

Tutorial - 4 : Matrix-Matrix addition and multiplication

Work to do :

- Distribute the global matrices **matrix_A** and **matrix_B** within a 2D cyclic-cyclic scheme to the local sub matrices **matrix_A_loc** and **matrix_B_loc** by using the **pdgemr2d** subroutine
- Once the global matrices are distributed, the linear algebra operations **C = A + B** and/or **C = A * B** can be done locally
- When the linear algebra operations are done, redistribute the sub matrices **matrix_C_loc** to a global matrix **matrix_C** owned by the master process
- Check if the results are correct by comparing your results with the usual scalar fortran subroutine **matmul**
- Tip : The matrices can be initialized randomly with the **init_matrix** subroutine
- Bonus : By placing wisely some timers, evaluate the gain in performance for large matrices between the scalar and parallel subroutines
- Bonus : Try the Matrix-Vector product by using the **pdgemv** subroutine

Tutorial - 4 (Extras) : **PBLAS** notations and subroutines

The **PBLAS** naming scheme follows almost the same convention of the **BLAS** subroutine : **P_XXYYY**

_	**Data type**
I	Integer
S	Real
D	Double precision
C	Complex
Z	Double complex

Additional information about the functional specifications can be found at the following link : http://www.netlib.org/scalapack/explore-html

Tutorial - 4 (Extras) : **PBLAS** notations and subroutines

The **PBLAS** naming scheme follows almost the same convention as the **BLAS** subroutine : **P_XXYYY**

XX	**Matrix type**
GE	GEneral
HE	HErmitian
SY	SYmetric
TR	TRiangular

Additional information about the functional specifications can be found at the following link : http://www.netlib.org/scalapack/explore-html

Tutorial - 4 (Extras) : **PBLAS** notations and subroutines

The **PBLAS** naming scheme follows almost the same convention of the **BLAS** subroutine : **P_XXYYY**

YYY	**Operation**
MM	Matrix-Matrix product
MV	Matrix-Vector product
R	Rank-1 update for a matrix
R2	Rank-2 update for a matrix
Rk	Rank-k update for a symmetric/hermitian matrix
R2k	Rank-2 update for a symmetric/hermitian matrix
SM	Solves a system of linear equations for a matrix of rhs
SV	Solves a system of linear equations for a rhs vector

Additional information about the functional specifications can be found at the following link : http://www.netlib.org/scalapack/explore-html

Tutorial - 5 : Matrix inversion

Aim of this part :

- To invert a general real matrix by using the **ScaLAPACK** subroutine **pdgesv**

- **pdgesv** solve the linear system **A*x = B** and it is used as the following :

```
call pdgesv(m, n, matrix_A_loc, ia, ja, desc_A_loc, ipiv,
matrix_B_loc, ib, jb, desc_B_loc, info)
```

  where **ipiv** is a vector which contains the pivoting information. Its dimension is **loc_m + mb**

  The solution **x** of the linear system **A*x = B** is returned in the **B** matrix

Tutorial - 5 : Matrix inversion

Work to do :

- By using the **pdgesv** subroutine, find the inverse of the matrix **A** by solving the linear system $\mathbf{A} * \mathbf{A}^{-1} = \mathbb{1}$
- Distribute the global matrices within a 2D cyclic-cyclic scheme to their local sub matrices by using the **pdgemr2d** subroutine
- Once the linear system has been solved, redistribute the sub matrices for the reverse matrix to its global matrix owned by the master process
- Tip : The global matrix A^{-1} must be initialized by the identity matrix
- After solving the linear system, check if the product $\mathrm{A} * \mathrm{A}^{-1}$ gives the identity matrix

Tutorial - 6 : Symmetric-Matrix diagonalization

Aim of this part :

- Find the eigenvalues and eigenvectors for a real symmetic matrix by using the **ScaLAPACK** subroutine **pdsyev**

- **pdsyev** can be called as the following :

```
call pdsyev(jobz, uplo, m, matrix_A_loc, ia, ja, desc_A_loc,
matrix_eigenvalues, matrix_eigenvectors_loc, ib, jb,
desc_eigen_loc, work, lwork, info)
```

where **work** is the local workspace and **lwork** its dimension

jobz can take 2 choises :
- 'N' Compute the eigenvalues only
- 'V' Compute the eigenvalues and the eigenvectors

uplo specifies whether the upper or lower triangular part of the matrix A is stored. It can take 2 choises :
- 'U' Upper triangular
- 'L' Lower triangular

Tutorial - 6 : Symmetric-Matrix diagonalization

Work to do :

- By using the **pdsyev** subroutine, find the eigenvalues and eigenvectors of a symmetric matrix **A**
- Distribute the global matrix **A** within a 2D cyclic-cyclic scheme to its local sub matrices by using the **pdgemr2d** subroutine
- Once the matrix of eigenvectors has been found, redistribute the sub matrices to the global matrix of eigenvectors owned by the master process
- Tips :
  - The vector that contains the eigenvalues is found completely by all the processes
  - The predifined subroutine **init_sym_matrix** initializes symmetricly and randomly the matrix **A**

Tutorial - 7 : Main problem

The aim of this problem is to calculate **exp(A)** for a general matrix with the Taylor expansion :

$$\exp A = \mathbb{1} + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \ldots + \frac{A^n}{n!}$$

- Distribute the global matrix **A** within a 2D cyclic-cyclic scheme to its local sub matrices by using the **pdgemr2d** subroutine
- Apply the Taylor expansion by using the **dgemm** subroutine
- Tips :
  - The identity matrix $\mathbb{1}$ can be initialized locally or initialized by the master process and then distributed to local sub matrices
  - A temporary matrix is needed locally in order to calculate each order
- Bonus : Evaluate **exp(A)** for a symmetric matrix by finding the eigenvalues and eigenvectors such as :

$$\exp A = U e^a U^{-1}$$

Outline

Discussion about performances : Benchmark conditions

Compiler and libraries used :

- Code compiled with the Intel compiler 2018.3
- Intel MPI 2018 & Intel MKL 2018 (Intel ScaLAPACK version)

Machine used :

- Ada @ IDRIS : 332 IBM x3750-M4 nodes + InfiniBand links
- One node based on 4 Intel(R) Xeon(R) Sandy Bridge E5-4650
- Each CPU is clocked @ 2.7GHz, contains 8 cores and 20Mo of cache memory

Algorithm conditions used :

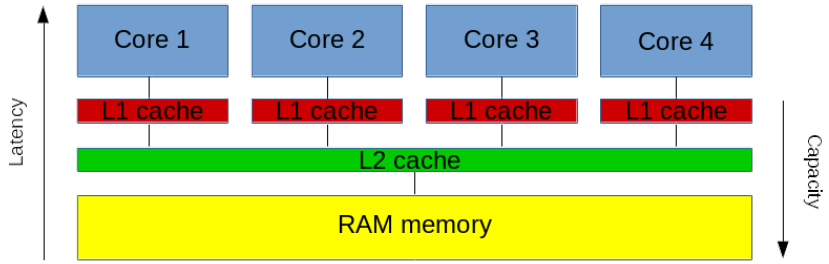
- The ScaLAPACK/MPI algortihm will be compared to the scalar expansion which uses the **matmul** Fortran subroutine for matrix multiplications
- The Taylor expansions is done till the 50th order for complex numbers

Discussion about performances : Influence of core numbers and grid topology

Nb cores	Topology	Time [s] (N=1000)	Time [s] (N=5000)
2	1x2	10.6	1250.6
	2x1	11.7	1251.4
4	2x2	5.9	651.8
	1x4	5.7	631.1
	4x1	5.9	651.0
8	2x4	3.2	347.3
	4x2	3.3	351.6
	1x8	3.2	347.0
	8x1	3.7	356.7
16	4x4	1.8	181.3
	2x8	1.8	178.9
	8x2	1.8	183.7
	1x16	2.0	192.2
	16x1	2.3	193.1
32	4x8	1.1	92.4
	8x4	1.1	96.2
	2x16	1.0	96.3
	16x2	1.2	100.3
	1x32	2.0	119.0
	32x1	2.4	132.2

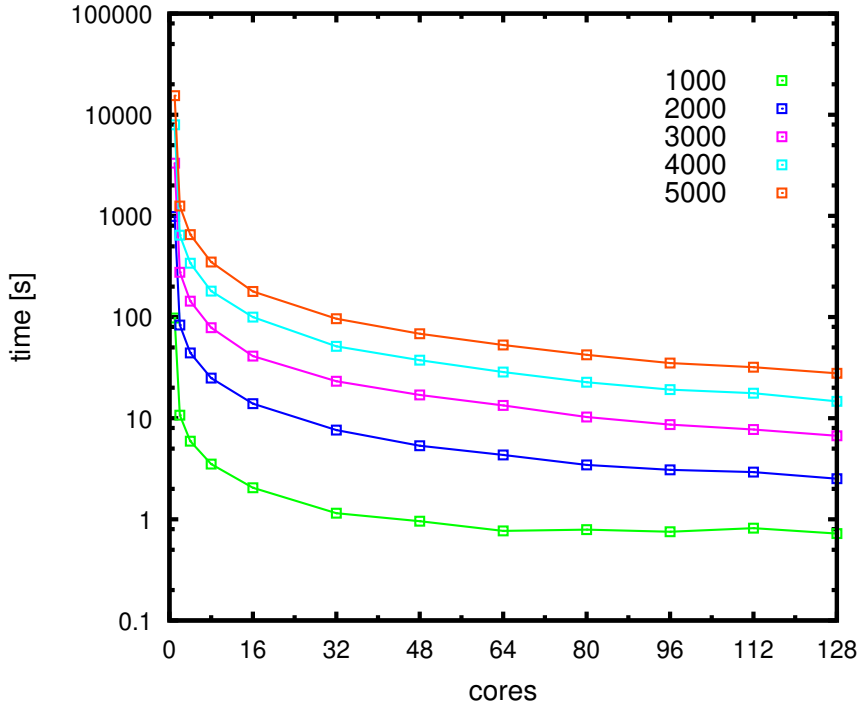
Discussion about performances : Influence of core numbers and grid topology

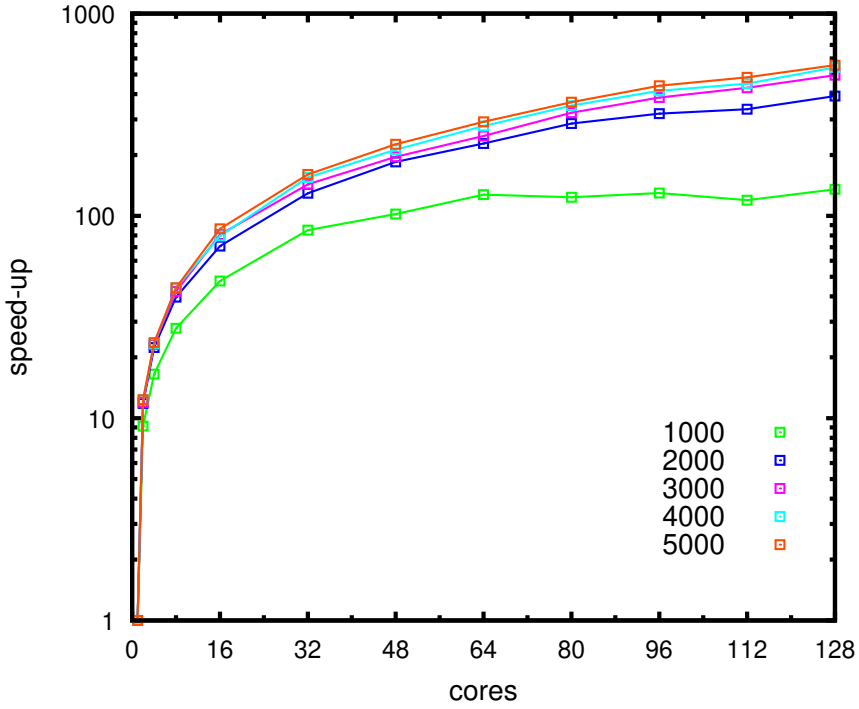
- Latencies are much lower if the data are kept in the cache memory



- The grid topology must be chosen carefully in order to store the data as much as possible in the cache memory
- The **ScaLAPACK/PBLAS pzgemm** has been designed to keep the data as much as possible in the cache memory along a row discretization
- Benchmarks must be done each time when the architecture of the machine is changed

Discussion about performances : How does the algorithm scale ?





- Benchmark done with the best topology
- Good scalability for large matrices and large number of cores
- exp(K) can be calculated in few minutes with only 2 nodes

References

- ScaLAPACK Users' Guide :
  ScaLAPACK Users' Guide, L. S. Blackford & al, Society for Industrial & Applied Mathematics (1997)
  http://www.netlib.org/scalapack/slug/index.html

- ScaLAPACK and PBLAS subroutines specifications :
  http://www.netlib.org/scalapack/explore-html

- BLACS subroutines specifications :
  http://www.netlib.org/blacs/BLACS/QRef.html

- IBM ESSL documentation :
  https://www.ibm.com/support/knowledgecenter/SSNR5K_4.2.0/pessl.v4r2_welcome.html
  https://publib.boulder.ibm.com/epubs/pdf/s3806992.pdf

- Intel MKL documentation :
  https://software.intel.com/en-us/mkl/documentation
  https://software.intel.com/en-us/articles/mkl-reference-manual

- Most decent tutorial found :
  https://info.gwdg.de/wiki/doku.php?id=wiki:hpc:scalapack