Introduction to parallel programming (for physicists)

François Gélis & Grégoire Misguich, IPhT courses, June 2019.
1. Introduction & hardware aspects (FG)
2. A few words about Maple & Mathematica
3. Linear algebra libraries
4. Fast Fourier transform
5. Python Multiprocessing
6. OpenMP
7. MPI (FG)
8. MPI+OpenMP (FG)
Numerical Linear algebra

Here: a few simple examples showing how to call some parallel linear algebra libraries in numerical calculations.
(numerical) Linear algebra

- Basic Linear Algebra Subroutines: **BLAS**
  - vector op. (=level 1)
  - matrix-vector (=level 2)
  - matrix-matrix mult. & triangular inversion (=level 3)
  - Many implementations but standardized interface
  - Discussed here: Intel MKL & OpenBlas (multi-threaded = parallelized for shared-memory architectures)

- More advanced operations Linear Algebra Package: **LAPACK**
  - ('90, Fortran 77)
    - Call the BLAS routines
    - Matrix diagonalization, linear systems and eigenvalue problems
    - Matrix decompositions: LU, QR, SVD, Cholesky
      - Many implementations

Used in most scientific softwares & libraries
(Python/Numpy, Maple, Mathematica, Matlab, ...
A few other useful libs
... for BIG matrices

- **ARPACK**
  = Implicitly Restarted Arnoldi Method (~Lanczos for Hermitian cases)
  Large scale eigenvalue problems. For (usually sparse) nxn matrices with n which can be as large as $10^8$, or even more!
  Ex: iterative algo. to find the largest eigenvalue, without storing the matrix M (just provide $v \rightarrow Mv$).
  Can be used from Python/SciPy

- **PARPACK**
  = parallel version of ARPACK
  for distributed memory architectures (the matrices are stored over several nodes)

- **ScaLAPACK**
  Parallel version of LAPACK for for distributed memory architectures
Two multi-threaded implementations of BLAS & Lapack

OpenBLAS
• Open source License (BSD)
• Based on GotoBLAS2
  [Created by GAZUSHIGE GOTO, Texas Adv. Computing Center, Univ. of Texas]
• Can be used from Fortran, C, C++, Python/Numpy, ...

Intel’s implementation (=part of the MKL lib.)
• Commercial license
• Included in *Intel® Parallel Studio XE* (compilers, libraries, ...), which is free for students
• Included in *Intel® Performance Libraries* (libraries only, without compiler), free for every one.
• Installed in most/many computing centers / intel-based clusters
• Included in *intel-pyhton*, which is free for *all* users
• Can be used from Fortran, C, C++ or Python/Numpy
Lapack/Intel-MKL

Code examples in C or FORTRAN:
https://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_lapack_examples/

Intel® Math Kernel Library LAPACK Examples
This document provides code examples for LAPACK (Linear Algebra PACKage) routines that solve problems in the following fields:

Linear Equations
Examples for several LAPACK routines that solve systems of linear equations.

Linear Least Squares Problems
Examples for some of the LAPACK routines that find solutions to linear least squares problems.

Symmetric Eigenproblems
Symmetric Eigenproblems has examples for LAPACK routines that compute eigenvalues and eigenvectors of real symmetric and complex Hermitian matrices.

Nonsymmetric Eigenproblems
Nonsymmetric Eigenproblems provides examples for ?gee, one of several LAPACK routines that compute eigenvalues and eigenvectors of general matrices.

Singular Value Decomposition
Examples for LAPACK routines that compute the singular value decomposition of a general rectangular matrix.
The routine computes the minimum-norm solution to a real linear least squares problem: minimize \( \| b - A^* x \| \) using the singular value decomposition (SVD) of \( A \). \( A \) is an \( m \)-by-\( n \) matrix which may be rank-deficient.

Several right hand side vectors \( b \) and solution vectors \( x \) can be handled in a single call; they are stored as the columns of the \( m \)-by-\( nrhs \) right hand side matrix \( B \) and the \( n \)-by-\( nrhs \) solution matrix \( X \).

The effective rank of \( A \) is determined by treating as zero those singular values which are less than \( rcond \) times the largest singular value.

Example Program Results

```
DGELSD Example Program Results

Minimum norm solution
-0.69 -0.24 0.06
-0.80 -0.08 0.21
 0.38  0.12 -0.65
 0.29 -0.24 0.42
 0.29  0.35 -0.30

(...)
```

Naming convention of the LAPACK routines: \( XYYZZZ \)

\( X \): \( S \) REAL, \( D \) DOUBLE PRECISION, \( C \) COMPLEX, \( Z \) COMPLEX*16 or DOUBLE COMPLEX

\( YY \): \( BD \) bidiagonal, \( DI \) diagonal, \( GB \) general band, \( GE \) general, \( GG \) general matrices, generalized problem (i.e., a pair of general matrices), \( GT \) general tridiagonal, (...)

\( ZZZ \): Type of computation. Here \( LSD \) stands for minimum norm solution to a linear least squares problem using the singular value decomposition of \( A \) and a divide and conquer method.
**Lapack-MKL example in Fortran**

(part 2/3)

```
* .. Parameters ..
INTEGER M, N, NRHS
PARAMETER ( M = 4, N = 5, NRHS = 3 )
INTEGER LDA, LDB
PARAMETER ( LDA = M, LDB = N )
INTEGER LWMAX
PARAMETER ( LWMAX = 1000 )

* .. Local Scalars ..
INTEGER INFO, LWORK, RANK
DOUBLE PRECISION RCOND

* .. Local Arrays ..
* IWORK dimension should be at least
* 3*M*N*NLVL + 11*M*N,
* Where
* NLVL = MAX( 0,INT(LOG_2( MIN(M,N)/(SMLSIZ+1))))+1
* and SMLSIZ = 25
INTEGER IWORK( 3*M*0+11*M )
DOUBLE PRECISION A( LDA, N ), B( LDB, NRHS ), S( M ),
WORK( LWMAX )
```

**External Subroutines**

EXTERNAL DGELSD
EXTERNAL PRINT_MATRIX

**Intrinsic Functions**

INTRINSIC INT, MIN

Lapack routines require you to provide some memory space to work (in the form of array(s)). Here: WORK, double precision array of size LWORK(see next slide).

M, N: size of A
NRHS: number of vectors $\vec{b}$ for which the problem must be solved.

DATA A/
$ 0.12, -6.91, -3.33, 3.97,$
$ -8.19, 2.22, -8.94, 3.33,$
$ 7.69, -5.12, -6.72, -2.74,$
$ -2.26, -9.08, -4.40, -7.92,$
$ -4.71, 9.96, -9.98, -3.20$

DATA B/
$ 7.30, 1.33, 2.68, -9.62, 0.00,$
$ 0.47, 6.58, -1.71, -0.79, 0.00,$
$ -6.28, -3.42, 3.46, 0.41, 0.00$

*/
Lapack-MKL example in Fortran

*... Executable Statements...
WRITE(*,*)'DGELSD Example Program Results'
*  Negative RCOND means using default (machine precision) value
RCOND = -1.0
*
*  Query the optimal workspace.
*  LWORK = -1
CALL DGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK, LWORK, IWORK, INFO )
LWORK = MIN( LWMAX, INT( WORK(1) ) )
*
*  Solve the equations A*X = B.
*  CALL DGELSD( M, N, NRHS, A, LDA, B, LDB, S, RCOND, RANK, WORK, LWORK, IWORK, INFO )
*
*  Check for convergence.
*
IF( INFO.GT.0 ) THEN
WRITE(*,*)'The algorithm computing SVD failed to converge;
WRITE(*,*)'the least squares solution could not be computed.'
STOP
END IF
*
*  Print minimum norm solution.
*
CALL PRINT_MATRIX( 'Minimum norm solution', N, NRHS, B, LDB )
*
*  Print effective rank.
*
WRITE(*,'(/A,I6)')' Effective rank = ', RANK
*
*  Print singular values.
*
CALL PRINT_MATRIX( 'Singular values', 1, M, S, 1 )
STOP
END
*
*  End of DGELSD Example.

SVD failed to converge;
the least squares solution could not be computed.

Effective rank = 4

Minimum norm solution
-0.69  -0.24   0.06
-0.80  -0.08  -0.21
 0.38   0.12  -0.65
 0.29  -0.24   0.42
 0.29   0.35 -0.30

Singular values
18.66  15.99  10.01   8.51

Compilation and output:

$ ifort -mkl DGELSD_example.f
$ ./a.out

DGELSD Example Program Results

Minimum norm solution
-0.69  -0.24   0.06
-0.80  -0.08  -0.21
 0.38   0.12  -0.65
 0.29  -0.24   0.42
 0.29   0.35 -0.30

Effective rank = 4

Singular values
18.66  15.99  10.01   8.51

First call to DGELSD with LWORK=-1 → LAPACK returns the optimal size LWORK of the workspace array WORK.

Actual calculation
```c
#include <stdlib.h>
#include <stdio.h>
#include <mkl.h>
#include <mkl_lapacke.h>

int main()
{
    const int n = 2000;
    printf("Matrix size=%i\n",N);
    printf("Number of threads=%i\n",mkl_get_max_threads());

    MKL_INT N = n, LDA = n, info, i, j;
    double w[N];
    MKL_Complex16* a;
    a = malloc(N*LDA*sizeof(MKL_Complex16));

    srand(999); /* dense random matrix */
    for (i=0; i < N; i++)
        for (j = 0; j < N; j++)
            a[i*LDA+j].real = rand(), a[i*LDA+j].imag = rand();

    info = LAPACKE_zheevd(LAPACK_ROW_MAJOR, 'V', 'L', N, a, LDA, w);

    /* Check for convergence */
    if (info > 0) {
        printf("The algorithm failed to compute eigenvalues.\n" );
        exit( 1 );
    }

    /* Print the extreme eigenvalues */
    printf("Smallest eigen value=%6.2f\n",w[0]);
    printf("Largest value=%6.2f\n",w[N-1]);
}
```

// LAPACKE_zheevd: computes all eigenvalues and eigenvectors of a complex Hermitian matrix A using divide and conquer algorithm

info = LAPACKE_zheevd( LAPACK_ROW_MAJOR, 'V', 'L', N, a, LDA, w );
```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
// C-wrapper to the Fortran Lapack lib.
#include <lapacke.h>

int main()
{
    const int N = 2000;

    printf("Matrix size=%i\n",N);
    printf("Number of threads=%i\n",omp_get_max_threads());

    int LDA = N, info, i, j;
    double w[N];
    lapack_complex_double* a;
    a = malloc(N*LDA*sizeof(lapack_complex_double));

    srand(999); /* Dense random matrix */
    for (i=0; i < N; i++)
        for(j = 0; j<N; j++)
            a[i*LDA+j] = lapack_make_complex_double(rand(), rand());

    // LAPACKE_zheevd: computes all
    // eigenvalues and eigenvectors of a
    // complex Hermitian matrix A using divide
    // and conquer algorithm
    info = LAPACKE_zheevd(LAPACK_ROW_MAJOR, 'V', 'L', N, a, LDA, w);

    /* Check for convergence */
    if (info > 0) {
        printf("The algorithm failed to compute eigenvalues.\n" );
        exit(1);
    }

    /* Print the extreme eigenvalues */
    printf("Smallest eigen value=%6.2f\n",w[0]);
    printf("Largest value=%6.2f\n",w[N-1]);
}
```
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>

int main() {
    int N=10000, N2, i, j;
    N2=N*N;

    //Memory allocation for the arrays:
    double *A, *B, *C;
    A = (double *)malloc( N2*sizeof( double ) );
    B = (double *)malloc( N2*sizeof( double ) );
    C = (double *)malloc( N2*sizeof( double ) );

    for (i = 0; i < (N2); i++)
        A[i] = (double)(i+1),
        B[i] = (double)(-i-1),
        C[i] = 0.0;

    printf ("Computing matrix product using OpenBLAS dgemm function via CBLAS interface ...\n");
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0, A, N, B, N, 1.0, C, N);
    printf ("done.\n\n");
    return 0;
}
Compilation of the OpenBLAS examples – use of make & makefile

# specify below the path to the OpenBLAS library files (like libopenblas.a and cblas.h)
OPEN_BLAS_LIB= ../OpenBLAS

# specify below the path to the lapacke.h header file
LAPACKE_INC= ../OpenBLAS/lapack-netlib/LAPACKE/include

dgemm_example_mini.exe: dgemm_example_mini.c
    gcc $< -o $@ -L $(OPEN_BLAS_LIB) -I $(OPEN_BLAS_LIB) -lopenblas -pthread
zheevd_example.exe: zheevd_example.c
    gcc $< -o $@ -I $(LAPACKE_INC) -L $(OPEN_BLAS_LIB) -lopenblas -fopenmp -lgfortran

clean:
    \rm *.exe

all: dgemm_example.exe zheevd_example.exe

To compile: make dgemm_example.exe or make all or make -j 2 all

- $<: 1st pre-requisite (usually the source file)
- $@: target (executable name)
- \rm * .exe: remove executable files
- -j option: use several threads/cores to compile multiple files in parallel
Check parallelism (1)

$ make dgemm_example.exe
gcc dgemm_example.c -o dgemm_example.exe -L ../OpenBLAS -I ../OpenBLAS -lopenblas -fopenmp -lrt

$ export OMP_NUM_THREADS=1; ./dgemm_example.exe 2000
Initializing the matrices ... done.
Computing matrix product using OpenBLAS dgemm function via CBLAS interface... done.
Elapsed time (s): 0.719543
22.2308 GFlops

$ export OMP_NUM_THREADS=10; ./dgemm_example.exe 2000
Initializing the matrices ... done.
Computing matrix product using OpenBLAS dgemm function via CBLAS interface... done.
Elapsed time (s): 0.0814542
196.38 GFlops

Close to the peak power of the CPU (here Xeon E5-2630 v2 @ 2.60GHz / 15360 KB Cache). Check with cat /proc/cpuinfo)
Check parallelism (2)

$ export OMP_NUM_THREADS=10; ./dgemm_example.exe 10000
Initializing the matrices ... done.
Computing matrix product using OpenBLAS sgemm function via CBLAS interface... done.
Elapsed time (s): 9.45623
211.49 GFlops

Environment variable to specify the # of threads (if not specified in the code)

1 process with ~10 threads running
OpenBLAS performance of `dgemm` (from C)
Anatomy of High-Performance Matrix Multiplication

KAZUSHIGE GOTO
The University of Texas at Austin
and
ROBERT A. VAN DE GEIJN
The University of Texas at Austin

We present the basic principles which underlie the high-performance implementation of the matrix-matrix multiplication that is part of the widely used GotoBLAS library. Design decisions are justified by successively refining a model of architectures with multilevel memories. A simple but effective algorithm for executing this operation results. Implementations on a broad selection of architectures are shown to achieve near-peak performance.

Categories and Subject Descriptors: G.4 [Mathematical Software]: —Efficiency

General Terms: Algorithms; Performance

Additional Key Words and Phrases: linear algebra, matrix multiplication, basic linear algebra subprograms

Armadillo: C++ library for linear algebra & scientific computing

Example of (symmetric) matrix diagonalization

Can call OpenBLAS or MKL

```cpp
#include <iostream>
#include <armadillo>
using namespace arma;

int main() {
    const int N = 5000;
    size_t dim = N;
    mat A(dim, dim, arma::fill::randu);
    vec eigval;
    mat eigvec;
    eig_sym(eigval, eigvec, A);
    cout << "1st eigenvalue = " << eigval[0] << " \tLast = " << eigval[dim - 1] << endl;
    return 0;
}
```
#include <iostream>
#include <armadillo>
using namespace arma;

int main() {
    const int N = 10000;
    size_t dim = N;
    mat A(dim, dim, arma::fill::randu);
    mat B(dim, dim, arma::fill::randu);
    mat C = A * B;
    return 0;
}
ARMA_INC=/usr/local/install/armadillo-9.200.6/include
ARMA_LIB=/usr/local/install/armadillo-9.200.6/lib64/

diag.exe: diag.cpp
    g++ -std=gnu++11 $< -o $@ -I $(ARMA_INC) -L $(ARMA_LIB) -larmadillo

mult.exe: mult.cpp
    g++ -std=gnu++11 $< -o $@ -I $(ARMA_INC) -L $(ARMA_LIB) -larmadillo
Armadillo: C++ library for linear algebra & scientific computing

checking what linear algebra library that is actually used

$ ldd diag.exe
linux-vdso.so.1 => (0x00007ffedd7c000)
libarmadillo.so.9 => /usr/local/install/armadillo-9.200.6/lib64/libarmadillo.so.9 (0x00007ff2c30549000)
libstdc++.so.6 => /usr/local/install/gcc-4.8.0/lib64/libstdc++.so.6 (0x00007ff2c3023f000)
libm.so.6 => /lib64/libm.so.6 (0x000000338f200000)
libgcc_s.so.1 => /usr/local/install/gcc-4.8.0/lib64/libgcc_s.so.1 (0x00007ff2c30010000)
libc.so.6 => /lib64/libc.so.6 (0x000000338e200000)
libmkl_rt.so => /opt/intel/composer_xe_2013.5.192/mkl/lib/intel64/libmkl_rt.so (0x00007f2c2f020000)
libhdf5.so.6 => /usr/lib64/libhdf5.so.6 (0x00000003390a0000)
libz.so.1 => /lib64/libz.so.1 (0x000000033ee0000)
/lib64/ld-linux-x86-64.so.2 (0x000000033de0000)
libdl.so.2 => /lib64/libdl.so.2 (0x000000033ea0000)

MKL used here
**BLAS & Lapack from Python/Numpy**

Python → Numpy → Linalg → LAPACK → BLAS

Different possible implementations
- standard BLAS/LAPACK (Netlib)
- ATLAS (Automatically Tuned Linear Algebra Software)
- OpenBLAS
- MKL
- ...

Multi-threaded / parallel
Check the version of Lapack & BLAS numpy is linked to:

```python
>>> import numpy as np
>>> np.__config__.show()
```
```plaintext
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c

blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c

openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
    define_macros = [('HAVE_CBLAS', None)]
    language = c
```
Python/Numpy linalg

What if my numpy version is not linked to a parallel linear algebra lib.?

• Install OpenBLAS:
  > sudo apt-get install libopenblas-dev

  (this will hopefully replace the previous BLAS lib. by OpenBLAS in Numpy)

• Or the Intel Python distribution


  (...)  
  > sudo apt-get install intelpython3

The corresponding numpy will be using the MKL lib.
Matrix diagonalization example

```python
import numpy as np
import numpy.random as npr
import time
npr.seed(2019)
n=3000
A = npr.randn(n,n)
t = time.time()
v = np.linalg.eigvals(A)
td = time.time() - t
print("Time=%0.4f s" % (td))
```

Specify the #of threads using an environment variable (bash shell):
> export OMP_NUM_THREADS=4
import numpy as np
import numpy.random as npr
import time

npr.seed(2019)
sizes=[10,20,100,200,300,400,500,600,700,800,900,1000,2000,4000,8000,10000,12000]

with open("Eigvals.dat","w") as my_file:
    my_file.write("#n\ttime\n")
    for n in sizes:
        A = npr.randn(n,n)
        t = time.time()
        v=np.linalg.eigvals(A)
        td = time.time() - t
        print(" Eigenvals of (%d,%d) matrix in %0.4f s" % (n, n, td))
        my_file.write(str(n)+"\t"+str(td)+"\n")

Eigenvals of (10,10) matrix in 0.0008 s
Eigenvals of (20,20) matrix in 0.0002 s
Eigenvals of (50,50) matrix in 0.0006 s
Eigenvals of (100,100) matrix in 0.0359 s
Eigenvals of (200,200) matrix in 0.0393 s
Eigenvals of (300,300) matrix in 0.0996 s
Eigenvals of (400,400) matrix in 0.1522 s
Eigenvals of (500,500) matrix in 0.2451 s
Eigenvals of (600,600) matrix in 0.3967 s
Eigenvals of (700,700) matrix in 0.4587 s
Eigenvals of (800,800) matrix in 0.5750 s
Eigenvals of (900,900) matrix in 0.6680 s
Eigenvals of (1000,1000) matrix in 0.7734 s
Eigenvals of (2000,2000) matrix in 2.4721 s
Eigenvals of (3000,3000) matrix in 7.4194 s
Eigenvals of (4000,4000) matrix in 14.2675 s
Eigenvals of (6000,6000) matrix in 39.1304 s
Eigenvals of (8000,8000) matrix in 68.1390 s
Eigenvals of (10000,10000) matrix in 109.9833 s
Eigenvals of (12000,12000) matrix in 195.2025 s
Performance vs # of cores & matrix dim.

General matrix diagonalization

**MKL EIGVALS (Intel Xeon Gold 5115 2.40GHz)**

**OpenBLAS EIGVALS (Intel Xeon Gold 5115 2.40GHz)**
Speedup vs # of cores & matrix dim.

Matrix-matrix multiplication
Matrix-matrix multiplication - OpenBLAS & MKL performance

Matrix-matrix multiplication [Numpy np.matmul on Intel Xeon Gold 5115 CPU 2.40GHz]

- MKL 16 threads
- 8 threads
- 4 threads
- 2 threads
- 1 thread

- OpenBLAS 16 threads
- 8 threads
- 4 threads
- 2 threads
- 1 thread

Giga FLOPS/thread vs. Matrix size n
Speedup vs # of cores & matrix dim.

Hermitian matrix diagonalization
Speedup vs # of cores & matrix dim.

Matrix Singular Value Decomposition

MKL SVD (Intel Xeon Gold 5115 2.40GHz)

OpenBLAS SVD (Intel Xeon Gold 5115 2.40GHz)