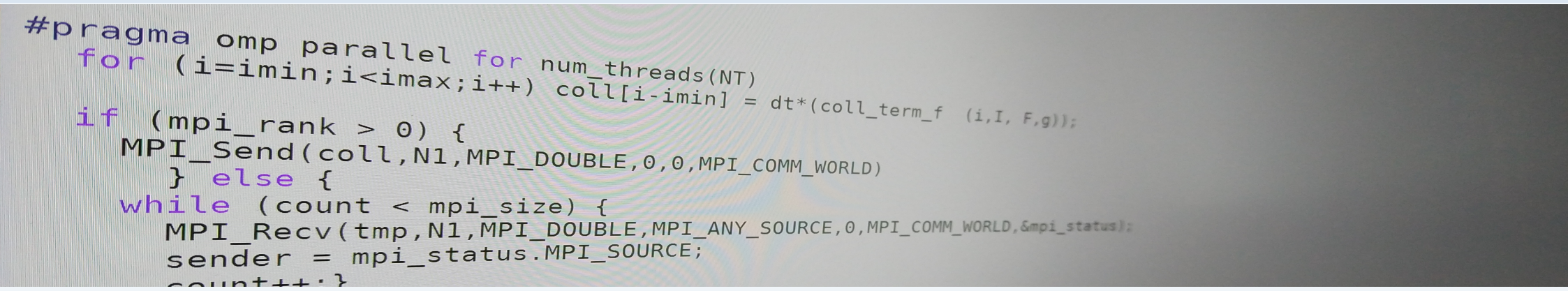
Institut de Physique Théorique

Cours de Physique Théorique



Parallel programming for physicists

FRANCOIS GELIS (IPHT) AND GREGOIRE MISGUICH (IPHT)

On Fridays 7, 14, 21, 28 June 2019, from 10:00 to 12:15.

Modern computers have a growing number of processors or 'cores'. From a few units in a simple laptop, to several thousands in big servers, their number has been growing quickly over the years. But to fully take advantage of this computing power, it is necessary to have codes or softwares being able to distribute a given task over several processors working in parallel.

These lectures will present an introduction to parallel programming in the context of scientific calculations;

- Introduction to hardware aspects ('shared' versus 'distributed' memory, communication between processors, vectorization, etc.)
- Solutions based on 'already-parallel' softwares (from linear algebra libraries to high-level computer algebra softwares)

We will then present two widely used libraries for code parallelization, OpenMP (Open MultiProcessing) and MPI (Message Passing Interface).

These lectures will be based on simple and concrete examples. They are intended for people with some basic programming knowledge (for instance in C/C++, Python or Fortran), but no prior experience with parallelization.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

Parallel Programming for Physicists

François Gelis
Grégoire Misguich

IPhT, June 7, 14, 21 & 28, 2019

Institut de
Physique Théorique
CEA/DRF Saclay

Material for the course (code, Makefiles)

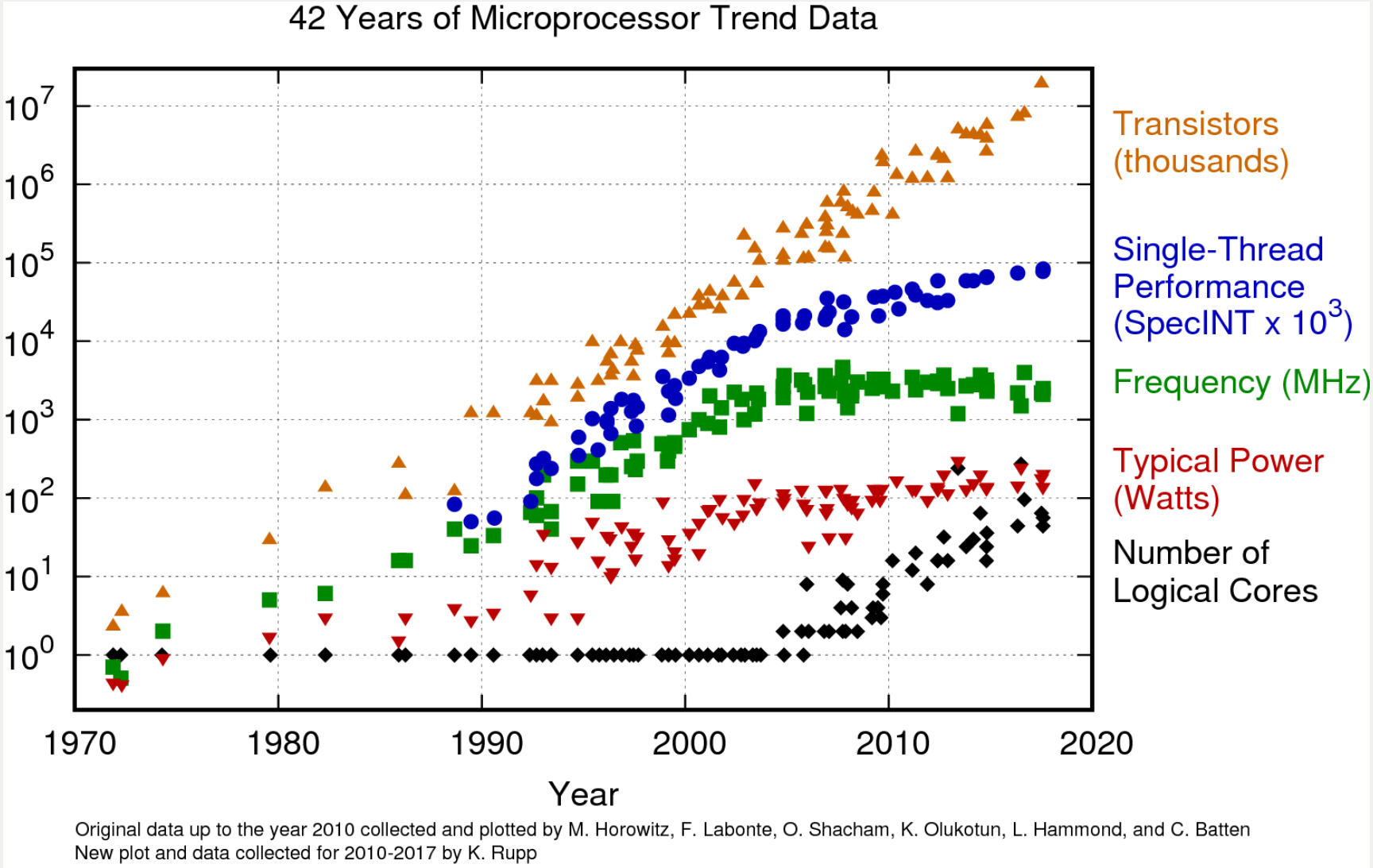
https://www.ipht.fr/Pisp/gregoire.misguich/pp.php

- uses a recent gcc/g++ compiler suite
- an example uses the fftw3 library
- for latest course: need openmpi

Outline

- Hardware Considerations
- Parallellization in Existing Software
- Shared Memory: OpenMP
- Distributed Memory: MPI
- Hybrid parallelization: OpenMPI + MPI

Hardware Considerations


42 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

My Laptop [DELL XPS 13 (2015)]

- CPU: 2.4 to 3.0 GHz, 2 cores
- Peak performance: 46 GFLOP/sec (double precision)

  (this would have placed my laptop on the top500 list of the world's most powerful computers, circa 1999...)

Let's do the math...

- At 3.0 GHz, 46 GFLOP/sec is equivalent to 15.3 FLOP/cycle

  i.e., 7.7 FLOP/cycle/core

  $\Longrightarrow$ there is a high amount of *instruction-level parallelism*
  In this case: each core uses AVX instructions to perform 4 double precision OPs at once, and has two Floating Point Units

$$3.0 \text{ GHz} \times (2 \text{ cores}) \times (2 \text{ FPUs}) \times 4(\text{AVX}) = 48 \text{ GFLOP/sec}$$

- What bandwidth do we need for this?

  To simplify, assume 1 FLOP = 2 reads + 1 write

$$\begin{aligned} 48 \text{ GFLOP/sec} =& 48 \times 8 \text{ (Bytes in a double)} \\ & \times 3 \text{ (2 reads + 1 write)} \qquad \text{GB/sec} \\ =& 1152 \text{ GB/sec} \end{aligned}$$

My Laptop [cont..]

- Memory Bandwidth:
  - from RAM: 17-20 GB/sec
  - from L2 cache: 100-140 GB/sec
  - from L1 cache: 570-980 GB/sec
- Memory Latency:
  - from RAM to CPU: 300 cycles
  - from L2 cache to CPU: 12 cycles
  - from L1 cache to CPU: 4 cycles
  - *from L2 (core 1) to L2 (core 2): 90 cycles* ⟵ *relevant for OpenMP*
- When the access pattern is regular, the CPU does some data prefetching, and these latencies are somewhat avoided
- With many cores, the needs for memory bandwidth become more severe

Cache: use correct loop order in nested loops

- Good:

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
     tmp += a[j+N*i];
   }
 }
```

- Not good:

```
for (j=0;j<N;j++){
   for (i=0;i<N;i++){
     tmp += a[j+N*i];
   }
 }
```

Cache: sometimes, "loop tiling" helps

- Example with no really ideal ordering:

```
for (i=0;i<N;i++){
     for (j=0;j<N;j++){
   b[j+N*i] = a[i+N*j];
     }
    }
```

- Improvement: slice the j loop in blocks of size B:

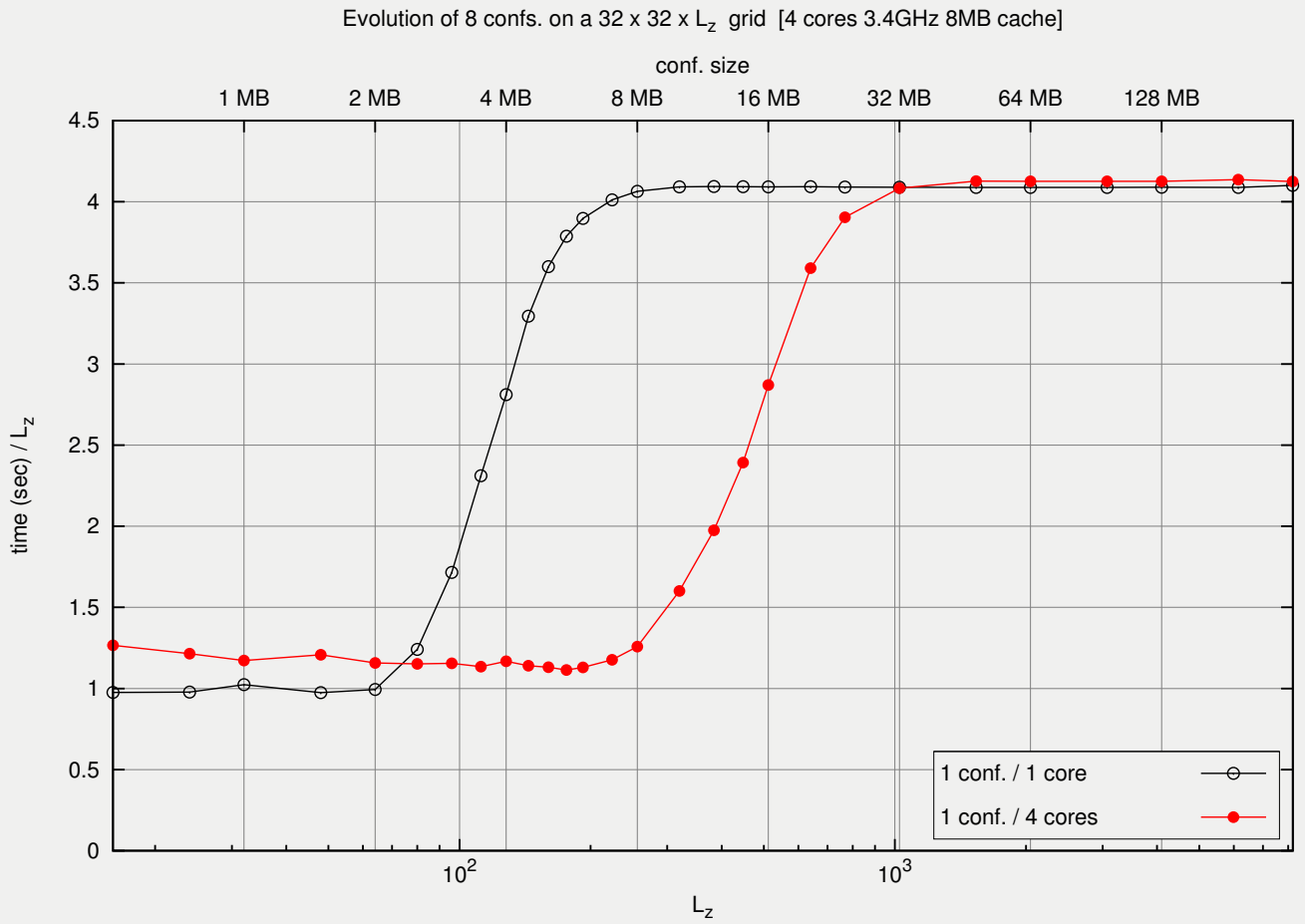
```
for (jb=0;jb<N;jb+=B){
    for (i=0;i<N;i++){
      for (j=0;j<B;j++){
   b[j+jb+N*i] = a[i+N*(j+jb)];
     }
    }
  }
```

Cache: sometimes, "loop tiling" helps

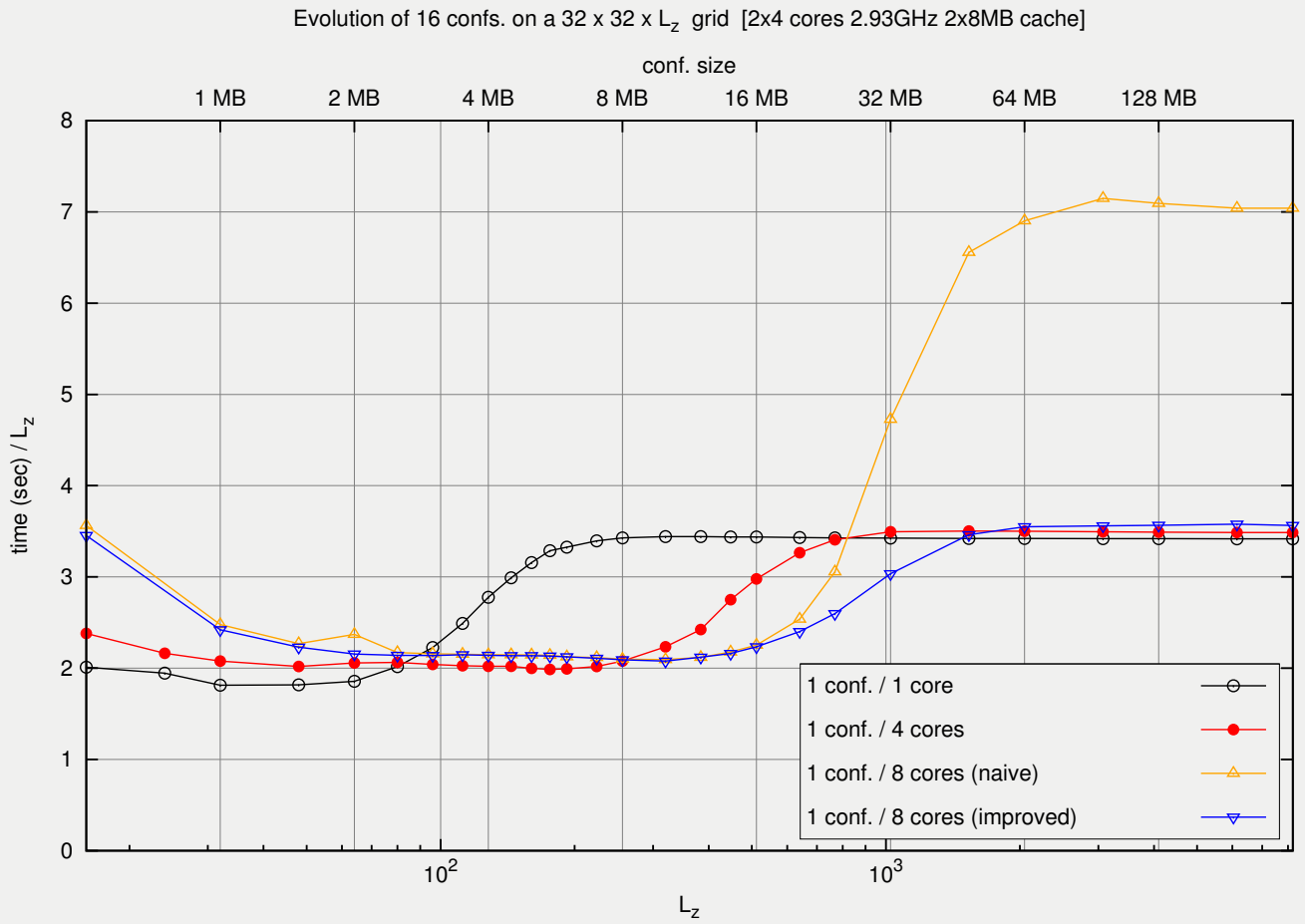
- How does it work?
  - This transformation makes the two innermost loops work on a contiguous range of size $N * B$ doubles
  - For $N = 2^{13}$, the optimal block-size is around $B = 2^4$ (empirical)
  - $2^{13} \times 2^4 \times 8$ (Bytes in a double) $= 1$ MByte = order of L2 cache

- Note: the compiler can do this transformation automatically (but perhaps not choose the best block-size)

Example: using parallelization to better use the cache



Example: using parallelization to better use the cache



Vector instructions

- SSE: 128 bit registers
  - 2 double precision
  - 4 simple precision
- AVX: 256 bit registers (most CPUs since 2015)
  - 4 double precision
  - 8 simple precision
- AVX512: 512 bit registers (only high-end CPUs)
- How to check:
  gcc -march=native -dM -E - < /dev/null | egrep ”SSE|AVX” | sort

Vectorization

- The compiler tries automatically with $-O3$, but not perfect
- Biggest obstacle: dependence among loop indices:

```
for (int j = 1; j < N; ++j){
  result[j] = 1.0/(1.0+result[j-1]);
 }
```

- Less if the dependence has a range longer than the vector length

```
for (int j = 4; j < N; ++j){
  result[j] = 1.0/(1.0+result[j-4]);
 }
```

Summary

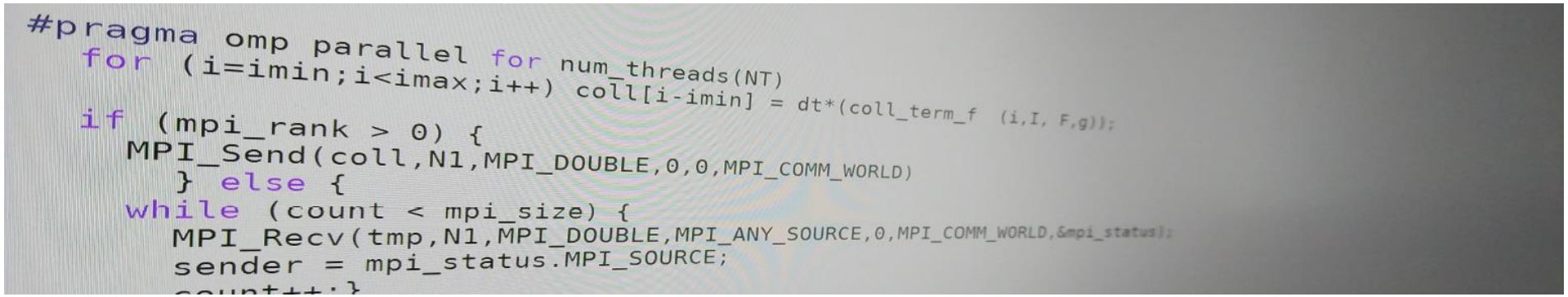
$$\text{PEAK Perf} = \underbrace{\text{Frequency}}_{\text{cannot change}} \times \underbrace{(\#\ \text{FPUs}) \times (\text{Width of vector inst.})}_{\text{well exploited only for appropriate code}} \times \underbrace{(\#\ \text{cores})}_{\text{try this?}}$$

Parallelization on shared memory:

- Very easy to implement via OpenMP, with tiny code modifications
- Should scale perfectly for independent tasks
- More tricky to split across cores tightly dependent tasks

A multicore Bash script...

```
#!/bin/bash
maxjobs=8  ## adjust to the number of cores
runningjobs=0
for i in * ## modify to your needs
   do
      echo $i
      (sleep 1)& ## do something more useful here
      runningjobs=$(($runningjobs+1))
      if [ $runningjobs -ge "$maxjobs" ]
      then
        wait
        runningjobs=0
    echo " "
      fi
   done
```



Introduction to parallel programming (for physicists)

FRANÇOIS GÉLIS & GRÉGOIRE MISGUICH, IPhT courses, June 2019.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

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)

These slides (GM)

Increasing coding effort

Parallelism with Maplesoft

1. Parallel programming

   [From Maple's documentation] « Maple provides tools for two different types of parallel programming. The **Task Programming Model** enables parallelism by executing multiple tasks within a single process. The second type of parallelism comes from **the Grid package**, which enables parallelism by starting multiple processes."

   Remark: the task model is somewhat analogous Threads/OpenMP, and Grid is analogous to MPI. Both OpenMP and MPI will be presented in these lectures

2. **Automatic parallelization**

Automatic Parallelism with Maplesoft

Example of automatic parallelization: simple numerical sum

- Sequential version

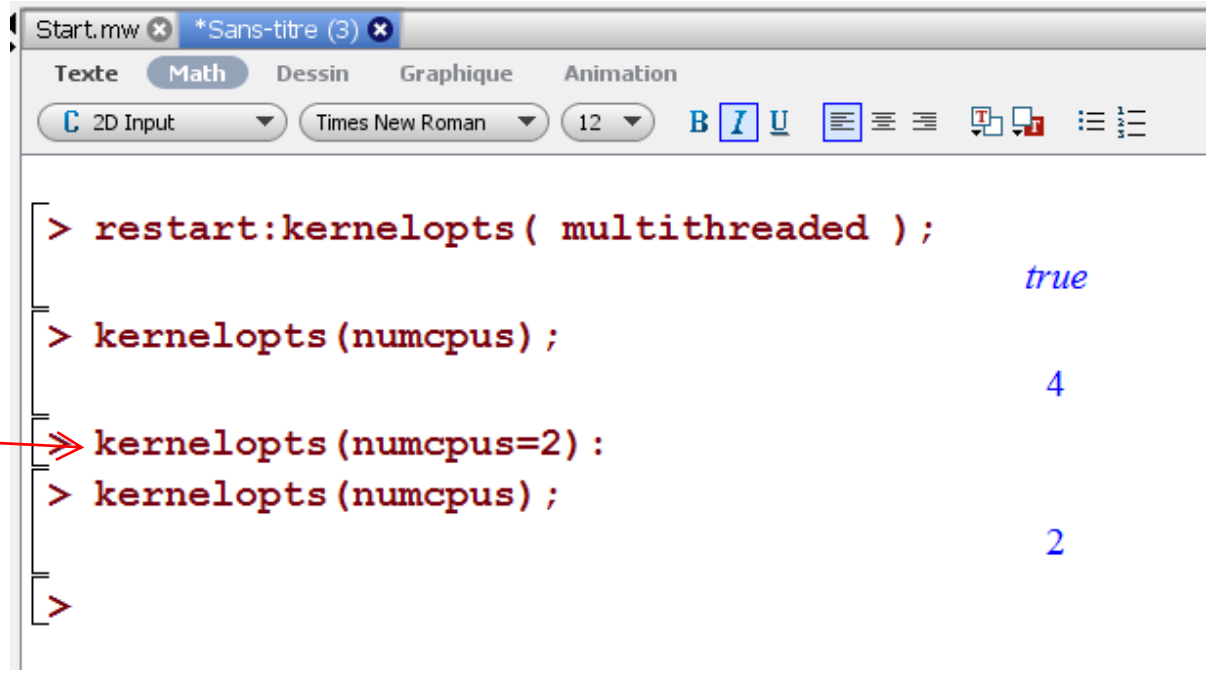
```
add(evalf(sin(i)),i=1..100000);
```

- Parallel version:

```
with(Threads);
Add(evalf(sin(i)),i=1..100000);
```

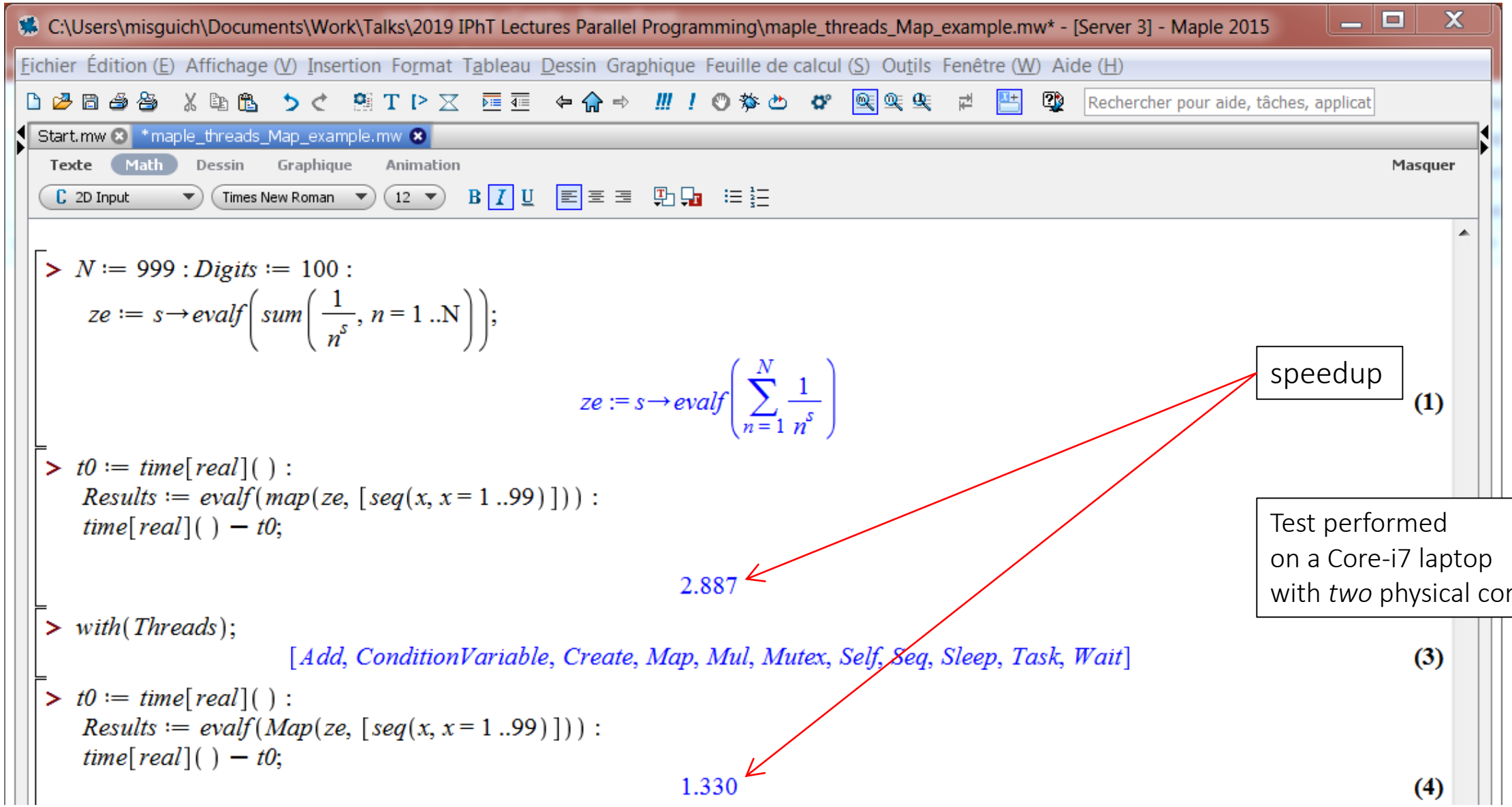
- by default Maple will create as many threads as available CPU cores
- Similar functions: **Mul**, **Seq**, and **Map**.

To select the number of CPU to be used by the multi-theaded engine:



Automatic Parallelism with Maplesoft

Example with `Map`



Automatic parallelism with Maplesoft

Example with `Add`

```
> t0 := time[real]( ) :
  add(evalf(sin(i)), i = 1 ..100000);
  time[real]( ) - t0;
```

1.847777103630379156630554301110741745559264562965641397408536842921550126002349383388906799091582245

15.803 (5)

```
> t0 := time[real]( ) :
  Add(evalf(sin(i)), i = 1 ..100000);
  time[real]( ) - t0;
```

1.847777103630379156630554301110741745559264562965641397408536842921550126002349383388906799091582245

8.406 (6)

```
>
```

speedup

Parallelism with Mathematica

Many possibilities:

Parallel Computing

The Wolfram Language provides a uniquely integrated and automated environment for parallel computing. With zero configuration, full interactivity, and seamless local and network operation, the symbolic character of the Wolfram Language allows immediate support of a variety of existing and new parallel programming paradigms and data-sharing models.

Automatic Parallelization

Parallelize — evaluate an expression using automatic parallelization
ParallelTry — try different computations in parallel, giving the first result obtained

Computation Setup & Broadcasting »

ParallelEvaluate — evaluate an expression on all parallel subkernels
DistributeDefinitions — distribute definitions to all parallel subkernels
ParallelNeeds — load the same package into all parallel subkernels

Data Parallelism »

ParallelMap ▪ **ParallelTable** ▪ **ParallelSum** ▪ **...**
ParallelCombine — evaluate expressions in parallel and combine their results

Concurrency Control »

ParallelSubmit — submit expressions to be evaluated concurrently
WaitAll — wait for all concurrent evaluations to finish
WaitNext — wait for the next of a list of concurrent evaluations to finish

Shared Memory & Synchronization »

SetSharedVariable — specify symbols with values to synchronize across subkernels
SetSharedFunction — specify functions whose evaluations are to be synchronized
$SharedVariables ▪ **$SharedFunctions** ▪ **UnsetShared** ▪ **CriticalSection**

Setup and Configuration »

LaunchKernels — launch a specified number of subkernels
$KernelCount — number of running subkernels
$KernelID ▪ **Kernels** ▪ **AbortKernels** ▪ **CloseKernels** ▪ **...**
$ProcessorCount — number of processor cores on the current computer

Multi-Processor and Multicore Computation

Compile — create compiled functions that run in parallel
Parallelization — execute compiled functions in parallel
CompilationTarget — create machine-level parallel compiled functions

GPU Computing »

CUDAFunctionLoad — load a function to run on a GPU using CUDA
OpenCLFunctionLoad — load a function to run on a GPU using OpenCL

File-Based Parallelism

FileSystemScan ▪ **FileSystemMap**

(automatic) Parallelism with Mathematica - **Parallelize**

```
Mathematica 10.4.1 for Linux x86 (64-bit)
Copyright 1988-2016 Wolfram Research, Inc.

In[1]:=  $KernelCount
Out[1]= 0

In[2]:=  AbsoluteTiming[Table[Length[FactorInteger[10^50 + n]], {n, 20}]]
Out[2]= {2.59524, {7, 4, 6, 4, 6, 4, 3, 7, 3, 6, 6, 7, 2, 5, 3, 3, 8, 3, 7, 6}}

In[3]:=  $KernelCount
Out[3]= 0

In[4]:= AbsoluteTiming[Parallelize[Table[Length[FactorInteger[10^50 + n]], {n,
20}]]]
Launching kernels...
Out[4]= {3.23171, {7, 4, 6, 4, 6, 4, 3, 7, 3, 6, 6, 7, 2, 5, 3, 3, 8, 3, 7, 6}}

In[5]:=  $KernelCount
Out[5]= 2

In[6]:=  AbsoluteTiming[Parallelize[Table[Length[FactorInteger[10^50 + n]], {n,
20}]]]
Out[6]= {1.55382, {7, 4, 6, 4, 6, 4, 3, 7, 3, 6, 6, 7, 2, 5, 3, 3, 8, 3, 7, 6}}
```

gives the number of subkernels available for parallel computations

Test done on a 2-(physical) core processor

speedup

(automatic) Parallelism with Mathematica - **ParallelMap**

```
Mathematica 10.3.0 for Linux x86 (64-bit)
Copyright 1988-2015 Wolfram Research, Inc.

In[1]:=  LaunchKernels[4]
Out[1]= {KernelObject[1, local], KernelObject[2, local], KernelObject[3, local],
KernelObject[4, local]}

In[2]:= Map[(Pause[1]; f[#]) &, {a, b, c, d}] // AbsoluteTiming
Out[2]= {4.00321, {f[a], f[b], f[c], f[d]}}

In[3]:= ParallelMap[(Pause[1]; f[#]) &, {a, b, c, d}] // AbsoluteTiming
Out[3]= {1.02335, {f[a], f[b], f[c], f[d]}}

In[4]:= CloseKernels[]

Out[4]= {KernelObject[1, local, <defunct>], KernelObject[2, local, <defunct>],
KernelObject[3, local, <defunct>], KernelObject[4, local, <defunct>]}
In[6]:= $KernelCount

Out[6]= 0

In[7]:=  LaunchKernels[2]
Out[7]= {KernelObject[5, local], KernelObject[6, local]}

In[8]:= ParallelMap[(Pause[1]; f[#]) &, {a, b, c, d}] // AbsoluteTiming
Out[8]= {2.00858, {f[a], f[b], f[c], f[d]}}
```

speedup

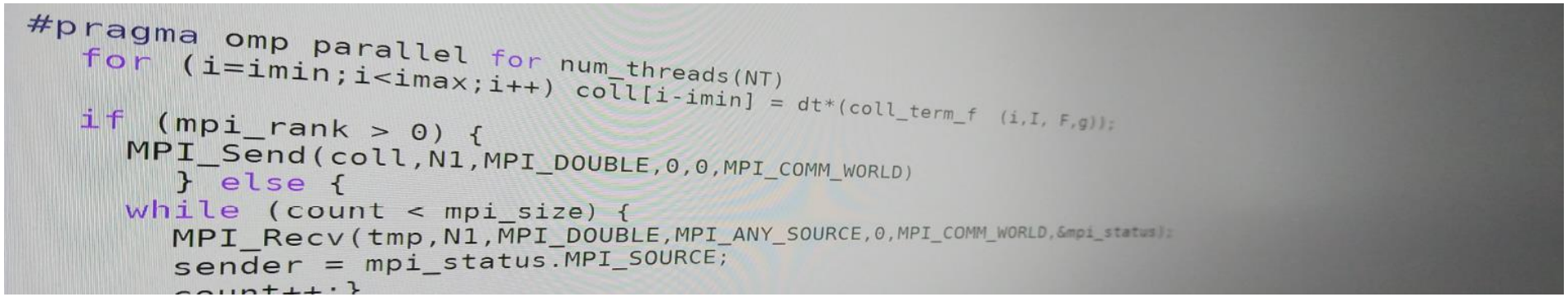
(automatic) Parallelism with Mathematica

Licence restrictions…

At IPhT (Mathematica network licence):

```
Mathematica 10.4.1 for Linux x86 (64-bit)
Copyright 1988-2016 Wolfram Research, Inc.

In[1]:=  $MaxLicenseProcesses

Out[1]= 10

In[2]:= $MaxLicenseSubprocesses

Out[2]= 80
```

8 subkernels per Process



Introduction to parallel programming (for physicists)

FRANÇOIS GÉLIS & GRÉGOIRE MISGUICH, IPhT courses, June 2019.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

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)

These slides (GM)

Numerical Linear algebra

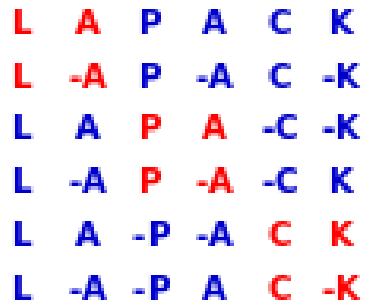
Here: a few simple examples
showing how to call some parallel
linear algebra libraries
in numerical calculations

(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→Mv).

Can be used from Python/SciPy

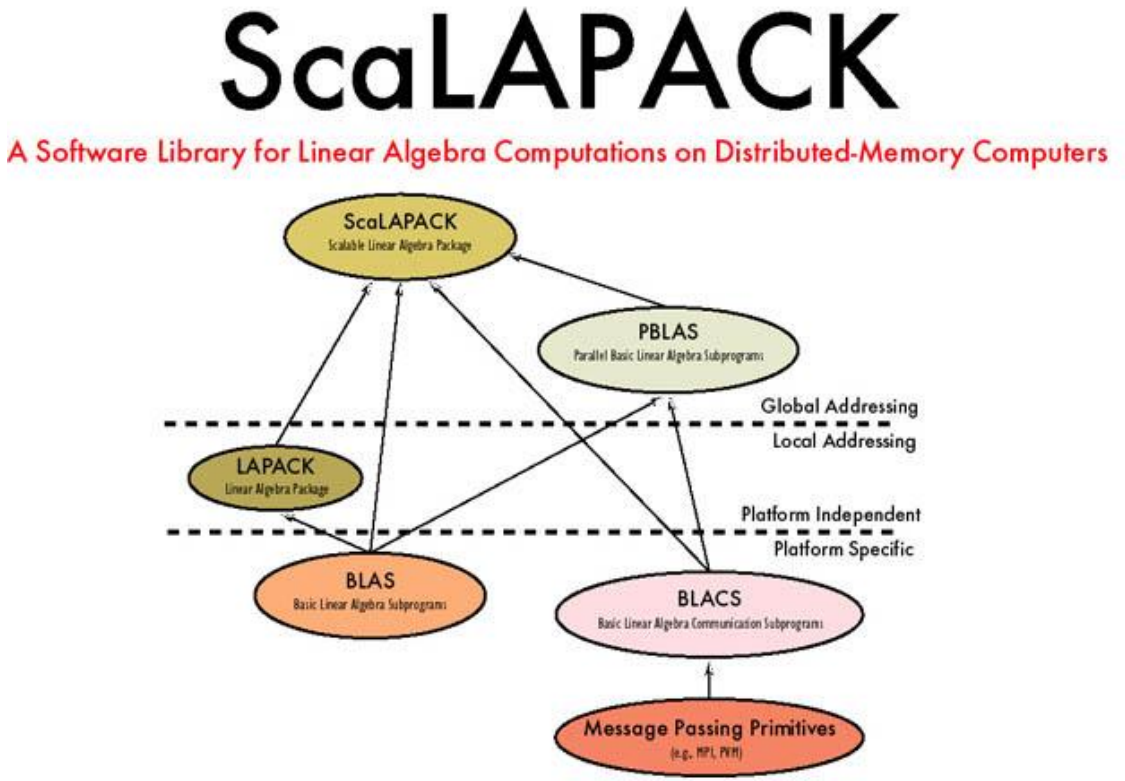
- **PARPACK**

= 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* (librairies only, without compiler), free for every one.
- Installed in most/many computing centers / intel-based clusters
- Included in `intel-python`, 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

Nonsymmetic Eigenproblems provides examples for *?geev*, 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.

Lapack-MKL example in Fortran

DGELSD/from Intel's website (part 1/3)

```
*  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
*  singular
*  values which are less than rcond times
*
*  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
*
(...)
```

Here minimize $\left\|\vec{b} - A\vec{x}\right\|_2$ with A a rectangular matrix and $\vec{b}$ a vector.

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*MIN(M,N)*NLVL + 11*MIN(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 )
      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
     $                   /
*
*      .. 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 me solved.

Lapack-MKL example in Fortran (part 3/3)

```
*     .. 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.
```

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

Actual calculation

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
```

Lapack/Intel-MKL

zheevd example in C

```c
#include <stdlib.h>
#include <stdio.h>
#include <mkl.h>
// C-wrapper to the Fortran Lapack lib. :
#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(
);
```

prints the num. of threads

```c
  // 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\n",w[N-1]);
}
```

w: array containing the eigen. vals.

Call to Lapack.
This version takes care of the workspace memory management (contrary to the Fortran version)

OpenBLAS/LAPACK

zheevd example in C

```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\n",w[N-1]);
}
```

OpenBLAS/CBLAS

dgemm example in C

```c
#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;
}
```

Matrix-matrix multiplcation (BLAS) of double precision general matrices

Specify the « order » of the matrix elements in memory:

A[i][j]=A[j+LDA*i] row-major
(C/C++ order)

A[i][j]=A[i+LDA*j] column-major
(=Fortran order)

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
```

`makefile`
$< : 1st pre-requiste (usually the source file)
$@ : target (executable name)

`-j` option: use several threads/cores to compile multiple files in parallel

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

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.
Elaspe 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.
Elaspe 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)

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

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

1 process with ~10 threads running

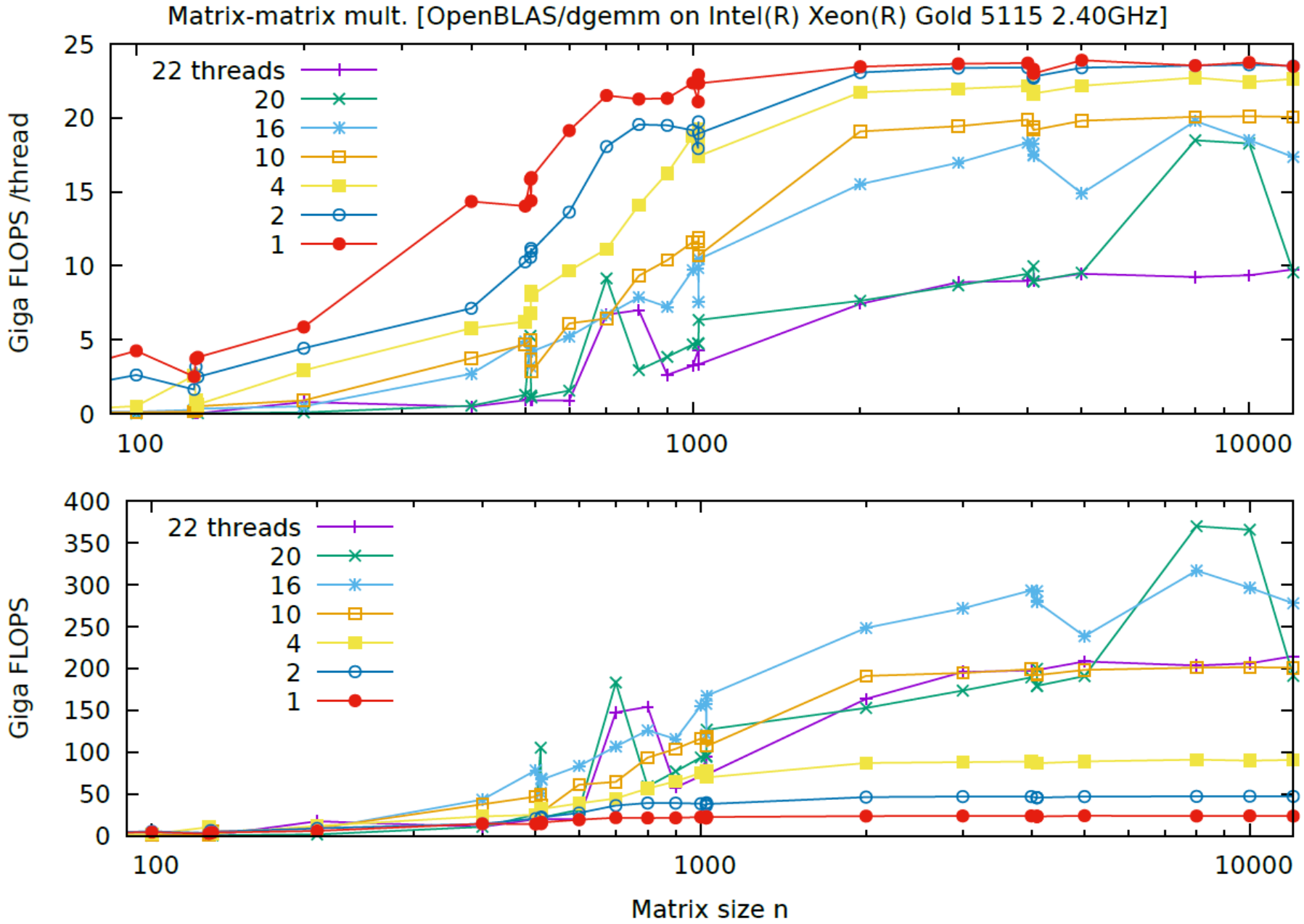
Top

misguich@totoro:~

```
top - 11:20:56 up 5 days, 19:04,  7 users,  load average: 3.56, 1.75, 0.74
Tasks: 547 total,   2 running, 545 sleeping,   0 stopped,   0 zombie
Cpu(s): 41.6%us,  0.1%sy,  0.0%ni, 58.2%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65876860k total,  9536948k used, 56339912k free,  1465352k buffers
Swap: 65535996k total,        0k used, 65535996k free,  4382076k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
75562 misguich  20   0 2799m 2.3g  616 R 999.0  3.6   0:53.36 dgemm_example.e
 3559 nrpe      20   0 43596 1432 1012 S  0.3  0.0   0:06.47 nrpe
 3694 root      20   0 62852  40m 1060 S  0.3  0.1   3:53.52 pbs_server
75387 misguich  20   0 17508 1664  972 R  0.3  0.0   0:00.53 top
    1 root      20   0 21456 1624 1300 S  0.0  0.0   0:03.70 init
```

OpenBLAS performance

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

High-performance BLAS

Now *OpenBLAS*

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 subprogrms

ACM Transactions on Mathematical Software 34(3):Article 12 · May 2008. DOI: 10.1145/1356052.1356053

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

Example of (symmetric) matrix diagonalization

Can call OpenBLAS or MKL

```
#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;
```

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

Matrix-matrix multiplication

```
#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;
}
```

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

compilation

```
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
```

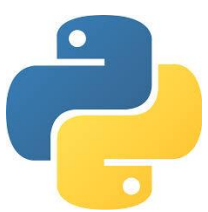
Armadillo: C++ library for linear algebra & scientific computing

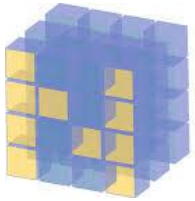
checking what linear algebra library that is actually used

```
$ ldd diag.exe
linux-vdso.so.1 =>  (0x00007ffeddd7c000)
libarmadillo.so.9 => /usr/local/install/armadillo-9.200.6/lib64/libarmadillo.so.9 (0x00007f2c30549000)
libstdc++.so.6 => /usr/local/install/gcc-4.8.0/lib64/libstdc++.so.6 (0x00007f2c3023f000)
libm.so.6 => /lib64/libm.so.6 (0x000000338f200000)
libgcc_s.so.1 => /usr/local/install/gcc-4.8.0/lib64/libgcc_s.so.1 (0x00007f2c30010000)
libc.so.6 => /lib64/libc.so.6 (0x000000338e200000)
libmkl_rt.so => /opt/intel/composer_xe_2013.5.192/mkl/lib/intel64/libmkl_rt.so
(0x00007f2c2fb02000)
libhdf5.so.6 => /usr/lib64/libhdf5.so.6 (0x0000003390a00000)
libz.so.1 => /lib64/libz.so.1 (0x000000338ee00000)
/lib64/ld-linux-x86-64.so.2 (0x000000338de00000)
libdl.so.2 => /lib64/libdl.so.2 (0x000000338ea00000)
```

MKL used here

BLAS & Lapack from Python/Numpy





L A P A C K
L -A P -A C -K
L A P A -C -K
L -A P -A -C K
L A -P -A C K
L -A -P A C -K

Python → Numpy → Linalg → LAPACK → BLAS

Different possible implementations

- standard BLAS/LAPACK (Netlib)
- ATLAS (Automatically Tuned Linear Algebra Software)
- OpenBLAS
- MKL
- …

Multi-threaded /parallel

Python/Numpy linalg

Check the version of Lapack & BLAS numpy is linked to:

```
>>> import numpy as np
>>> np.__config__.show()
```

```
Python 2.7.5 (default, Mar 20 2015, 15:33:03)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-11)] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> import numpy as np
>>> np.__config__.show()
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

https://software.intel.com/en-us/articles/installing-intel-free-libs-and-python-apt-repo

(…)

```
> sudo apt-get install intelpython3
```

The corresponding numpy will be using the MKL lib.

Python/Numpy linalg

Matrix diagonalization example

```
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 environnent variable (bash shell):

```
> export OMP_NUM_THREADS=4
```

Performance *vs* #of cores & matrix dim.

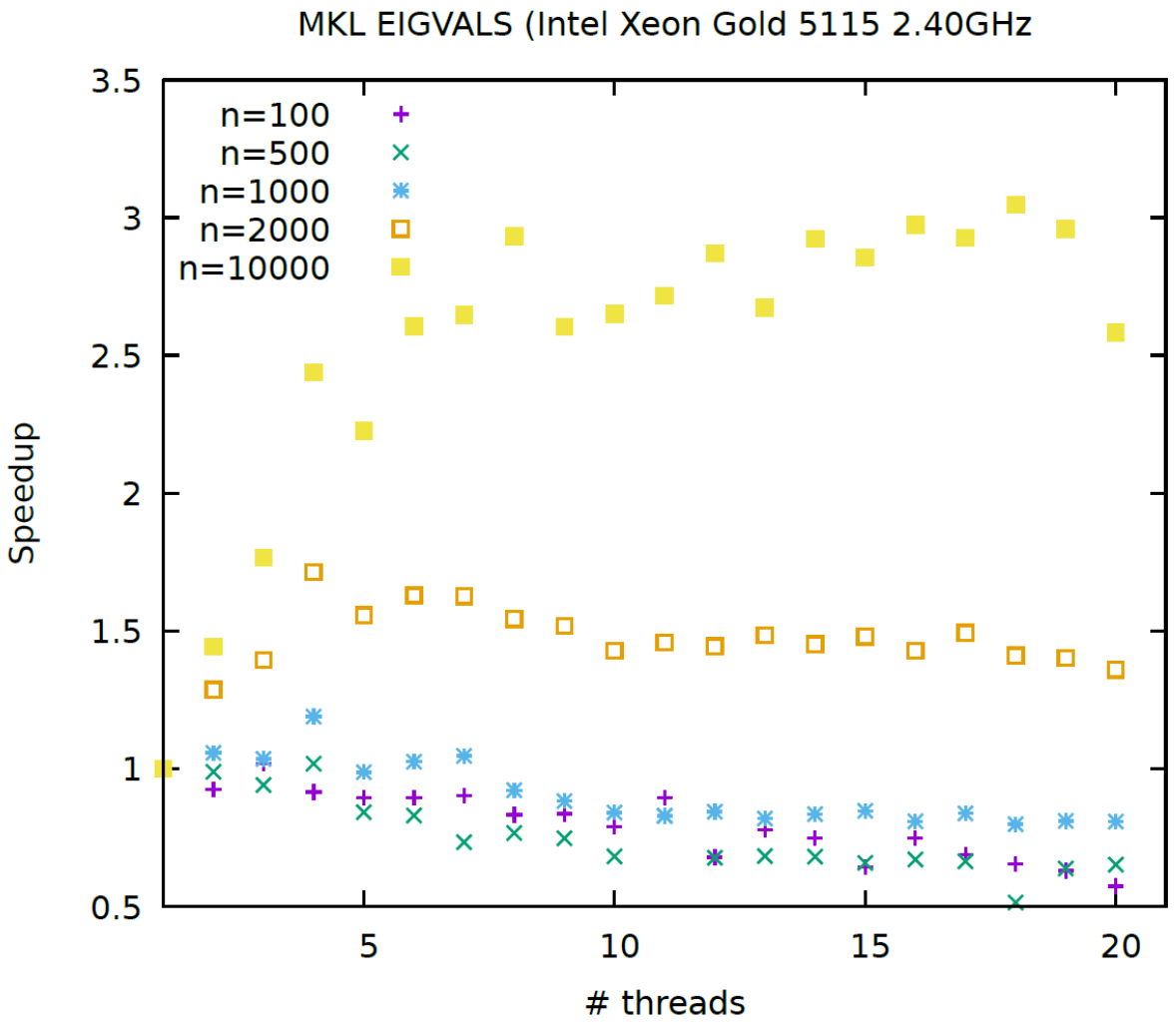
Matrix diagonalization

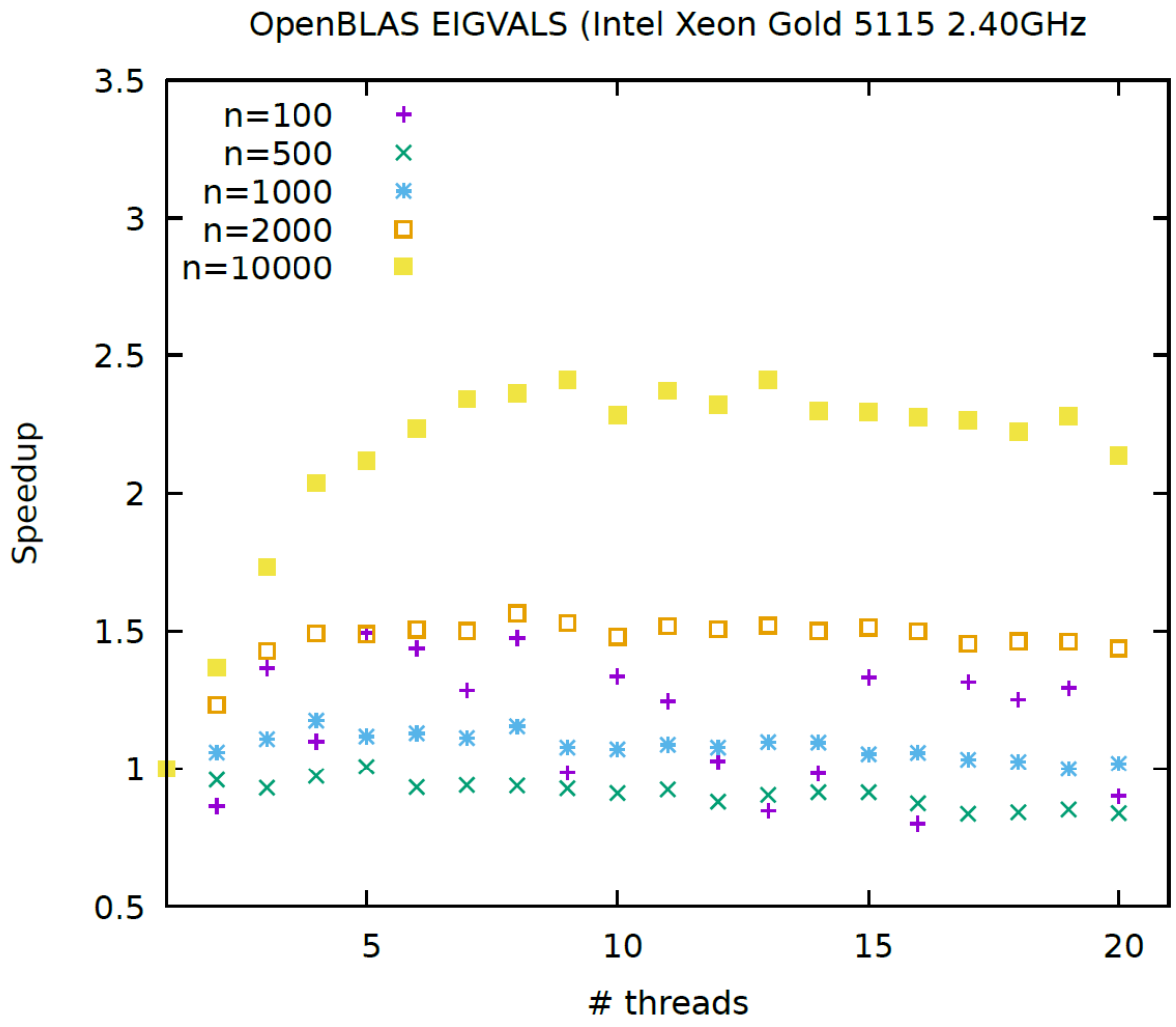
```
import numpy as np
import numpy.random as npr
import time

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.

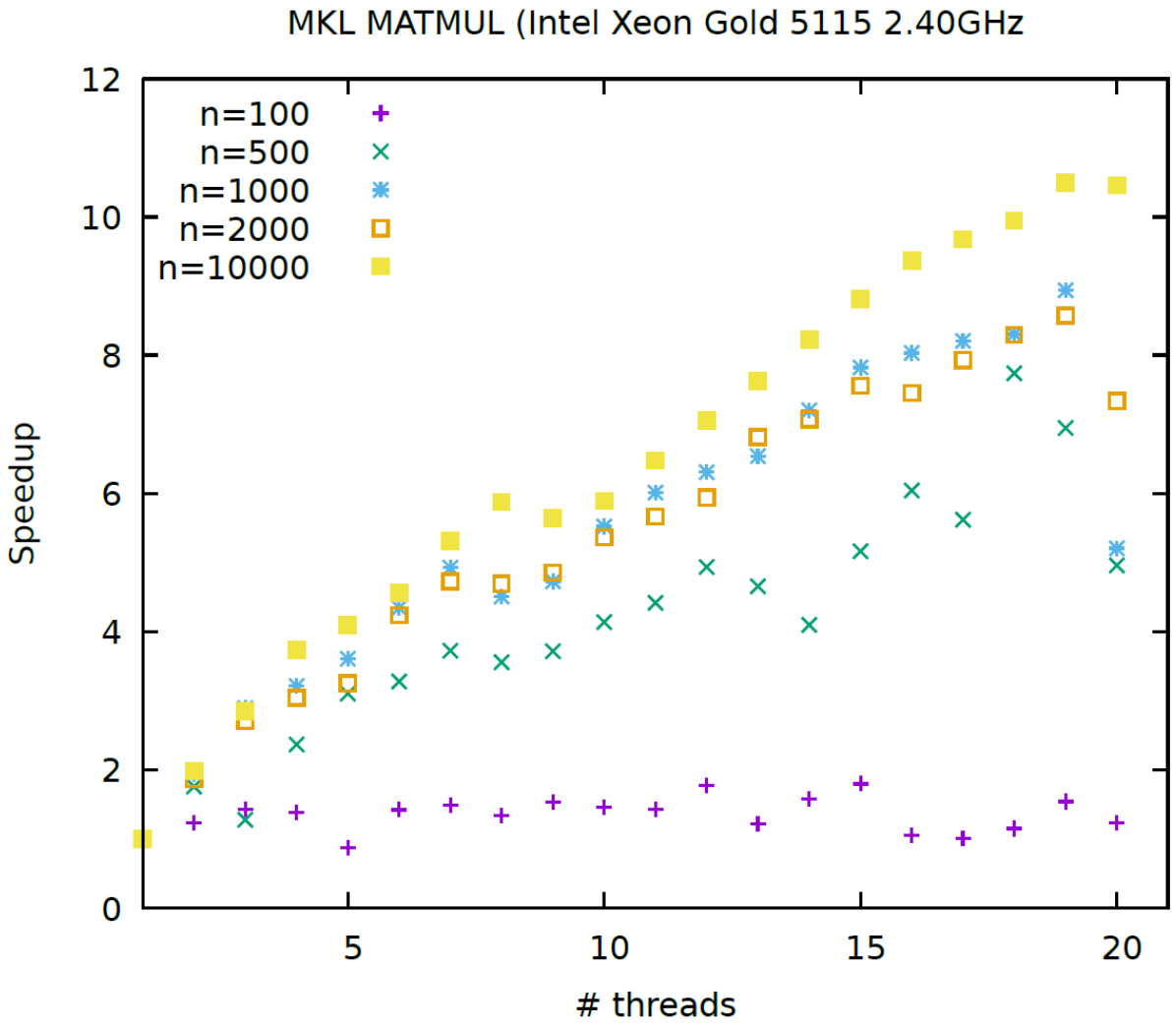
General matrix diagonalization

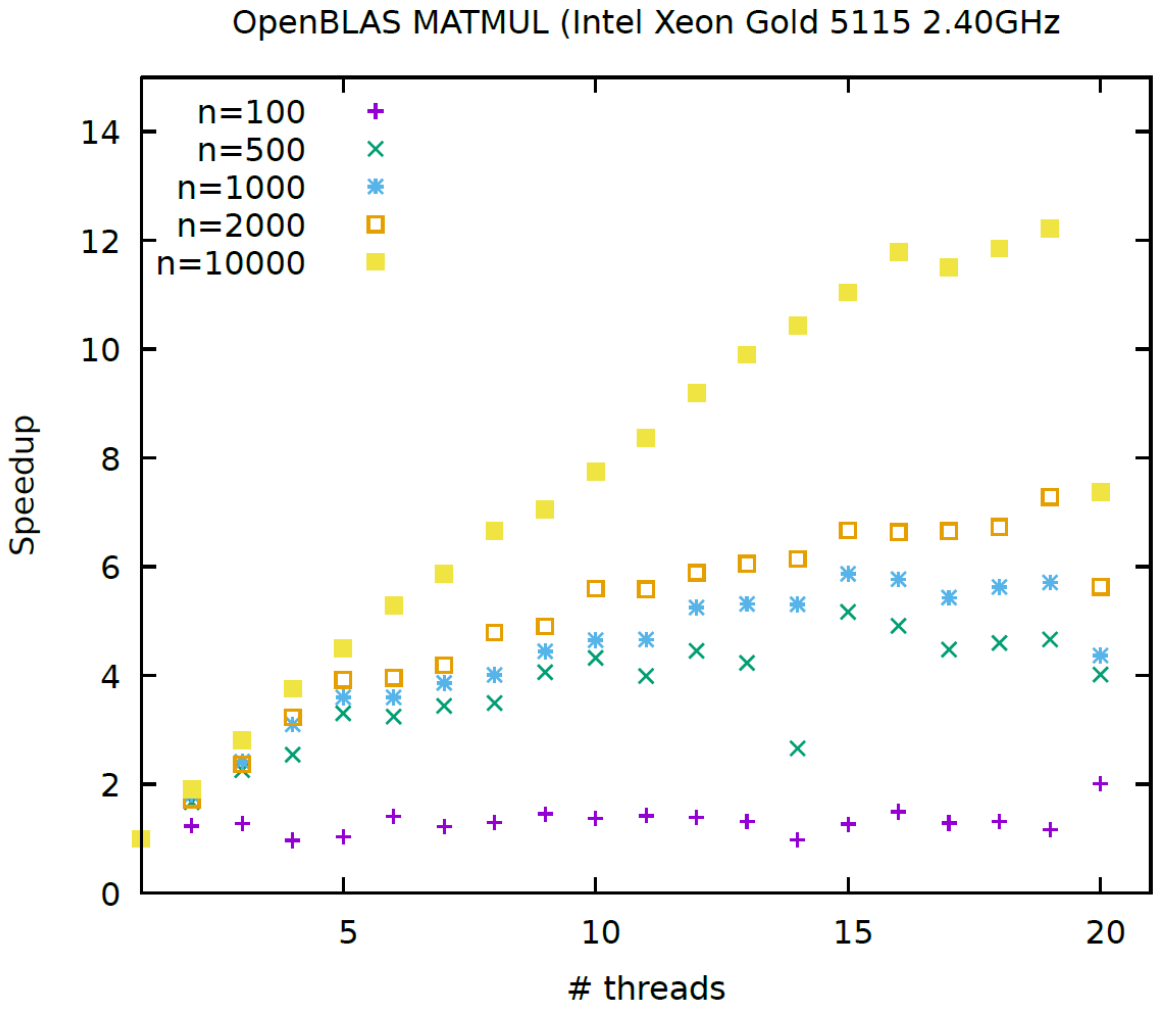




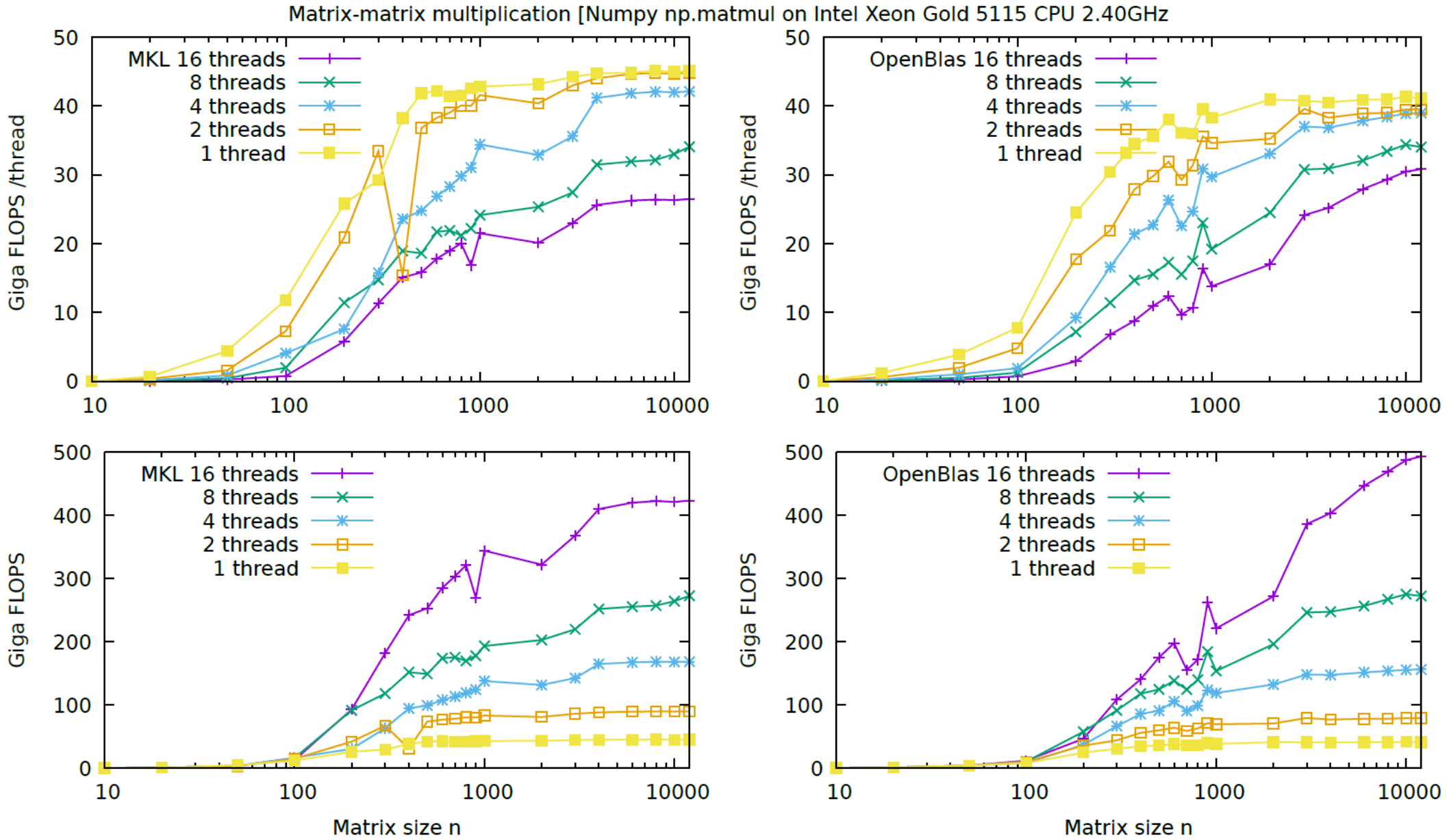
Speedup *vs* #of cores & matrix dim.

Matrix-matrix multiplication



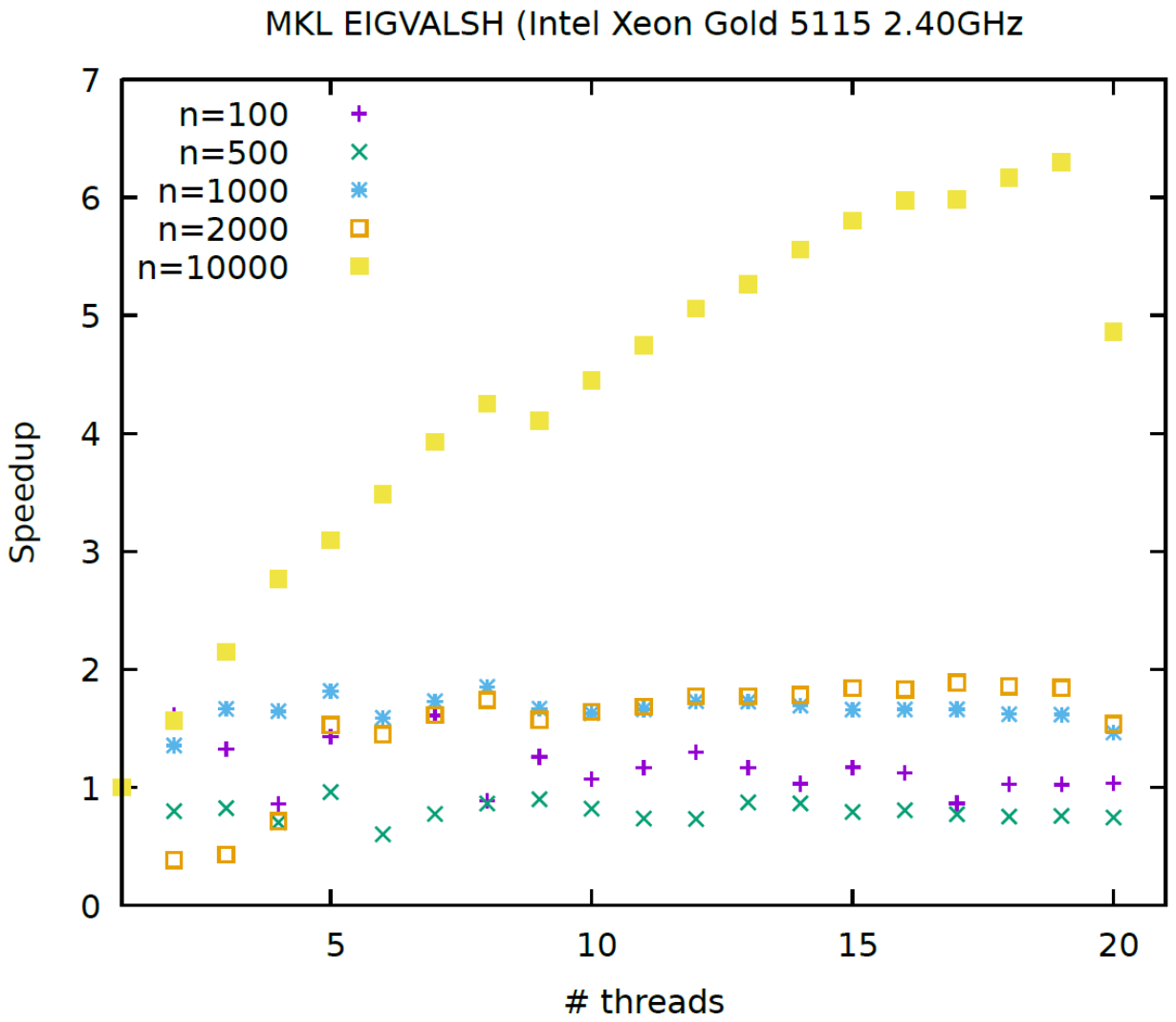


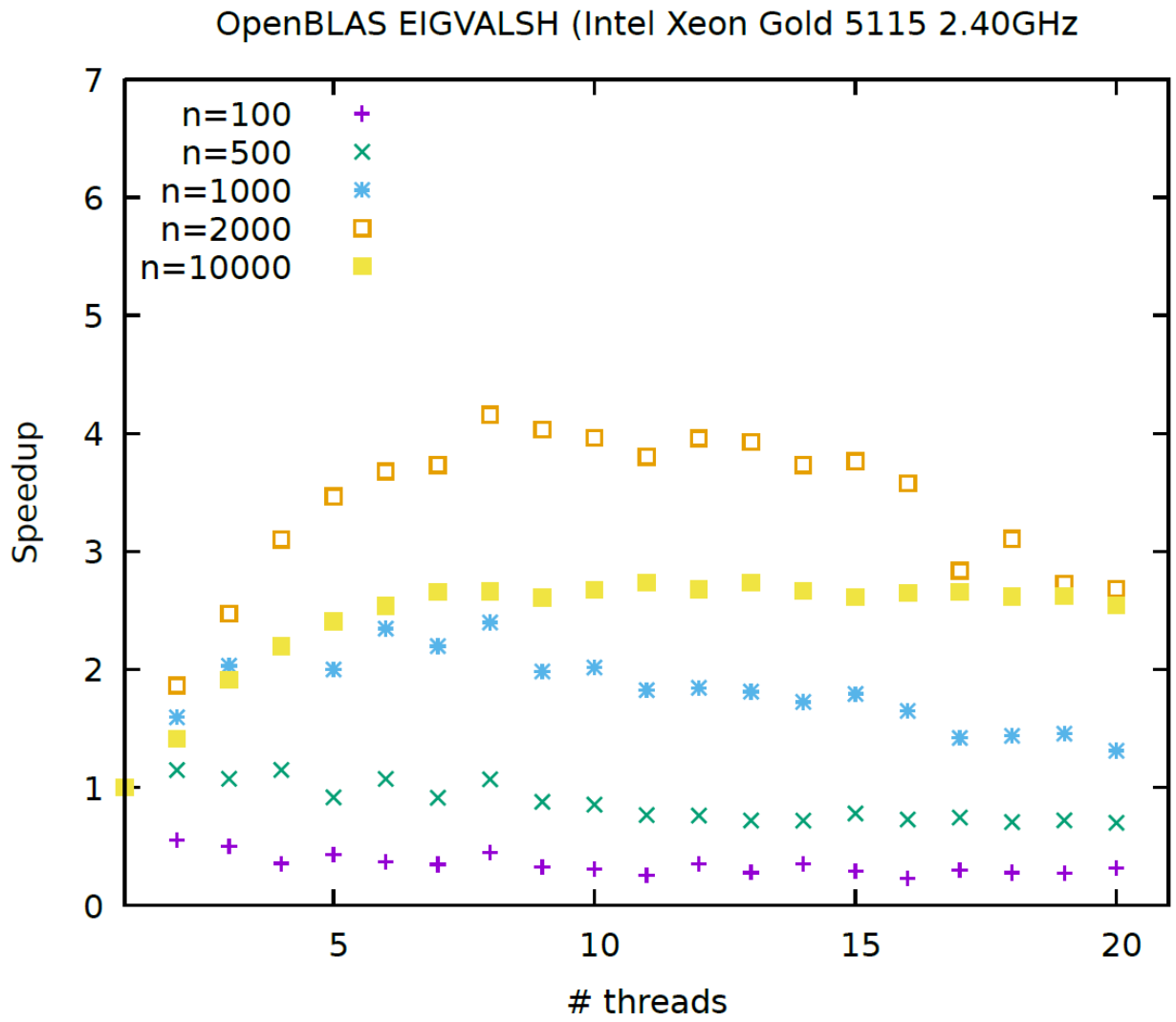
Matrix-matrix multiplication - OpenBLAS & MKL performance



Speedup *vs* #of cores & matrix dim.

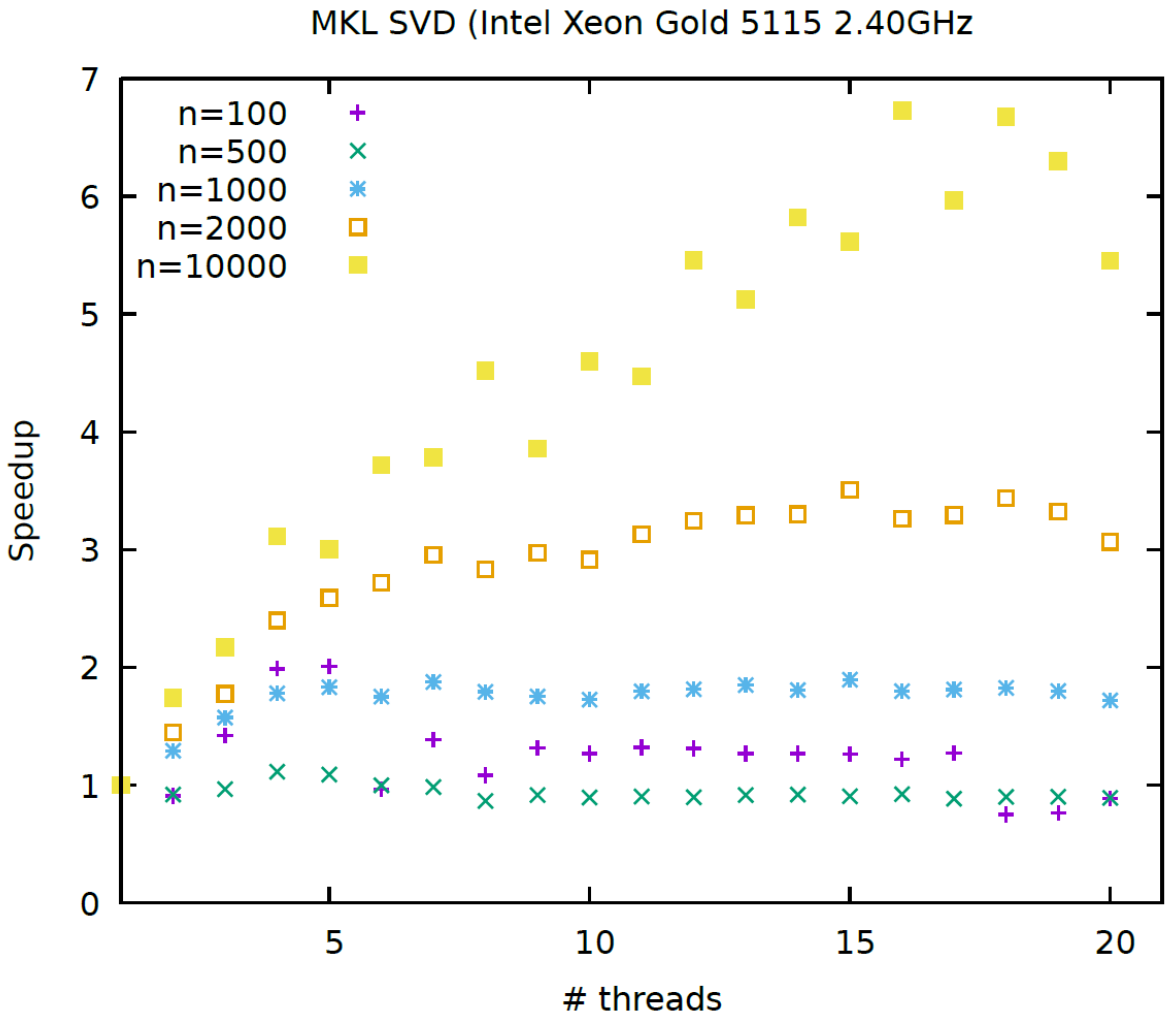
Hermitian matrix diagonalization

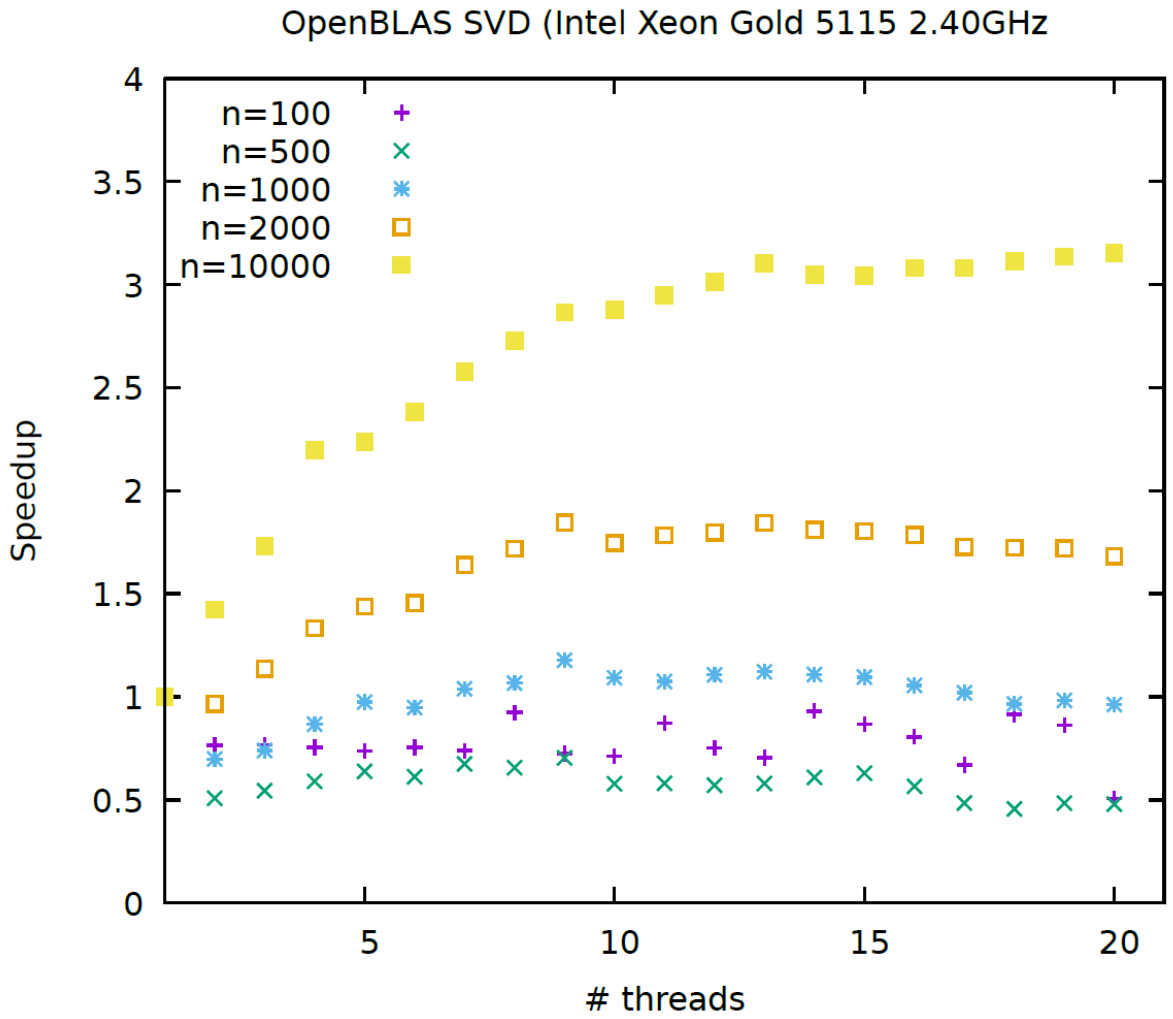


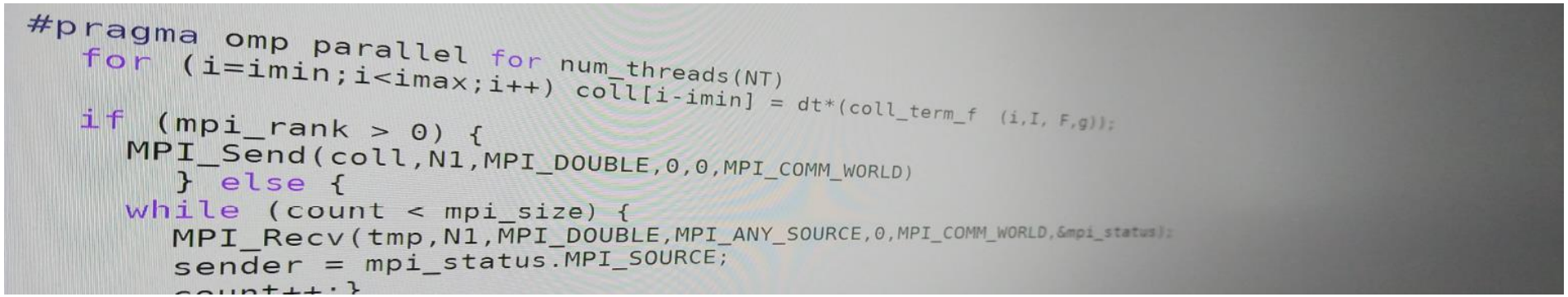


Speedup *vs* #of cores & matrix dim.

Matrix Singular Value Decomposition







Introduction to parallel programming (for physicists)

FRANÇOIS GÉLIS & GRÉGOIRE MISGUICH, IPhT courses, June 2019.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

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)

These slides (GM)

(discrete) Fourier transform

- FFTW: a high performance implementation (in C)
  Fastest Fourier transform *in the West*
- http://www.fftw.org/
- Multi-threaded
- portable
- Open source (GPL license)
- Can be used from Python (pyFFTW)

(discrete) Fourier transform example using multi-threaded FFTW v3

```c
#include <complex.h>
#include <fftw3.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define L (1<<25)
#define NT 2

void main(void){

  short unsigned int seed[3];
  fftw_complex * A;
  seed[0] = 123; seed[1] = 456;
  seed[2] = 789; seed48(seed);

  A =
(fftw_complex*)fftw_malloc(L*sizeof(fftw_compl
ex));
  int i;
  for (i=0;i<L;i++) A[i] =
(fftw_complex)(drand48()+I*drand48());

   fftw_init_threads();

  fftw_complex *in    =
(fftw_complex*)fftw_malloc(L*sizeof(fftw_co
ex));
  fftw_complex *out   =
(fftw_complex*)fftw_malloc(L*sizeof(fftw_co
ex));

  fftw_plan_with_nthreads(NT);
  fftw_plan FP =
fftw_plan_dft_1d(L,in,out,1,FFTW_ESTIMATE);
  fftw_free(in);
  fftw_free(out);

  fftw_complex *fft   = (fftw_complex
*)fftw_malloc(L*sizeof(fftw_complex));

fftw_execute_dft(FP,A,fft);

}
```

Num. of points

Num of threads

Allocate the input array

Random input

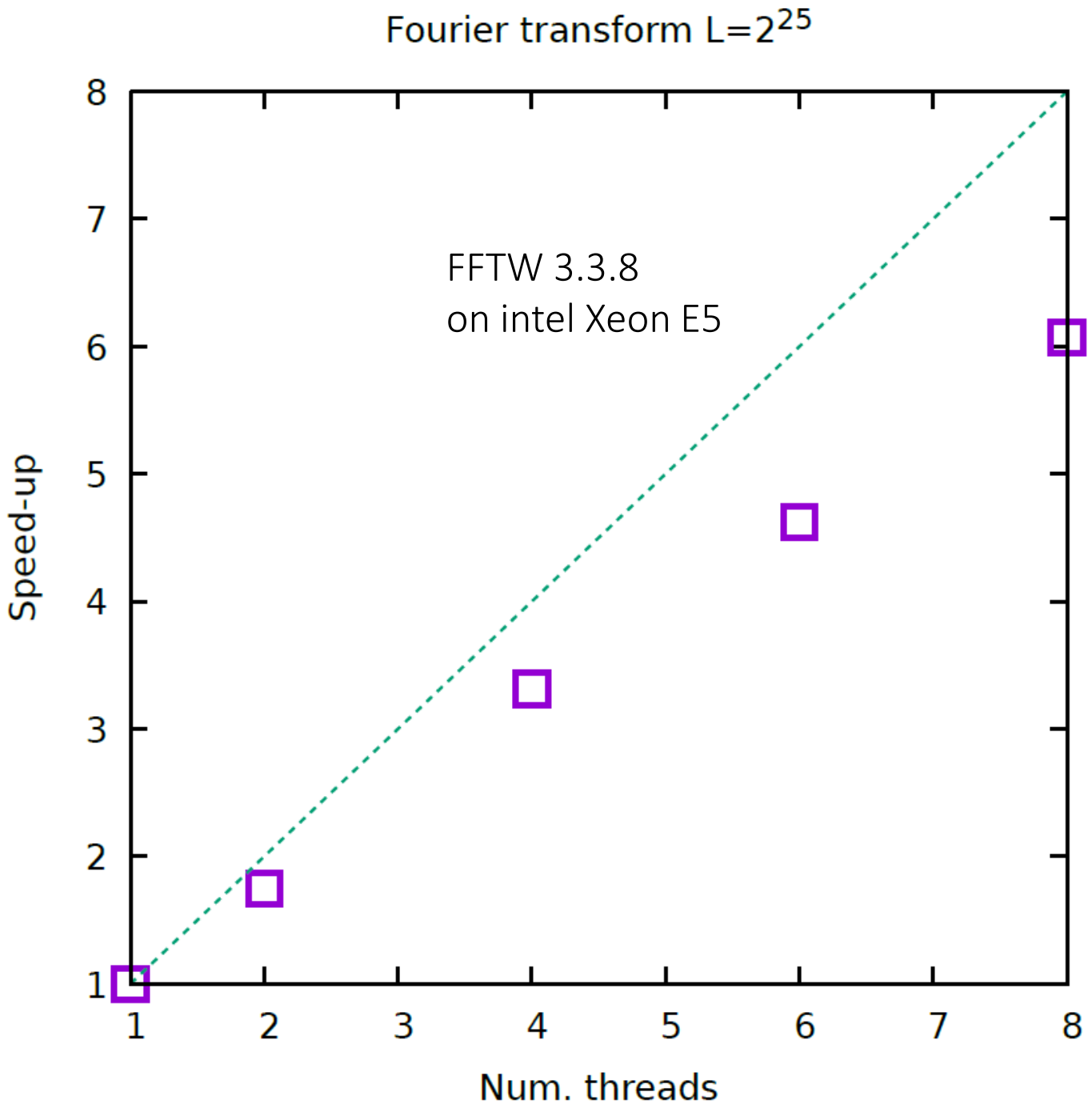
Actual FFT

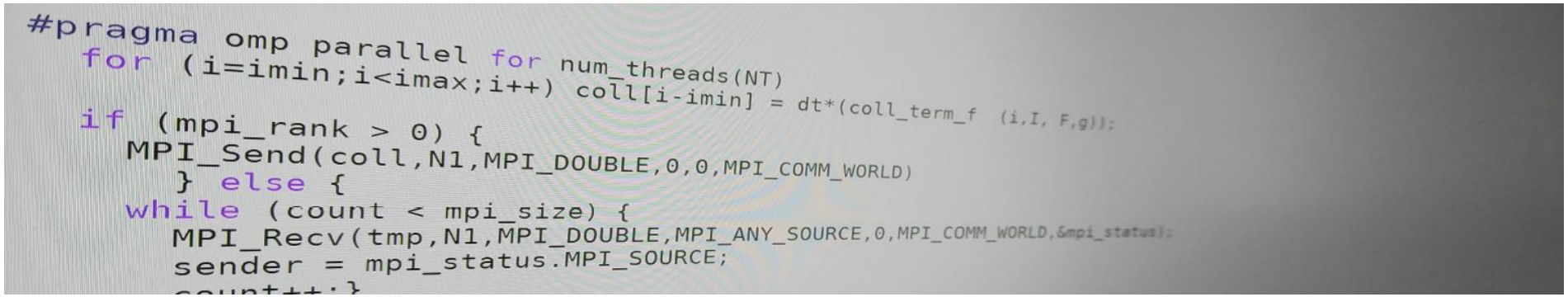
Output

Workspaces, and fft « plan »

Fourier transform

Speed-up with threads using FFTW3





Introduction to parallel programming (for physicists)

FRANÇOIS GÉLIS & GRÉGOIRE MISGUICH, IPhT courses, June 2019.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

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)

These slides (GM)

Parallel programing in Python

Threads *versus* Processes

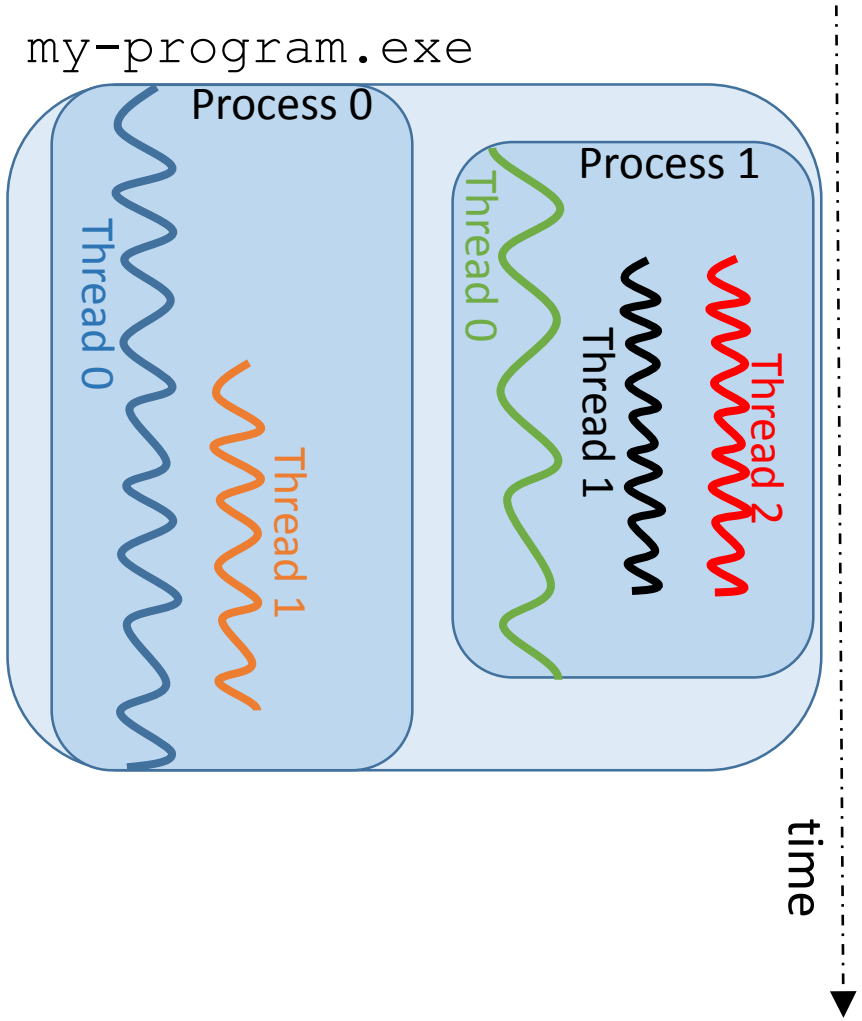
A given application/program may run different processes and different threads

Process

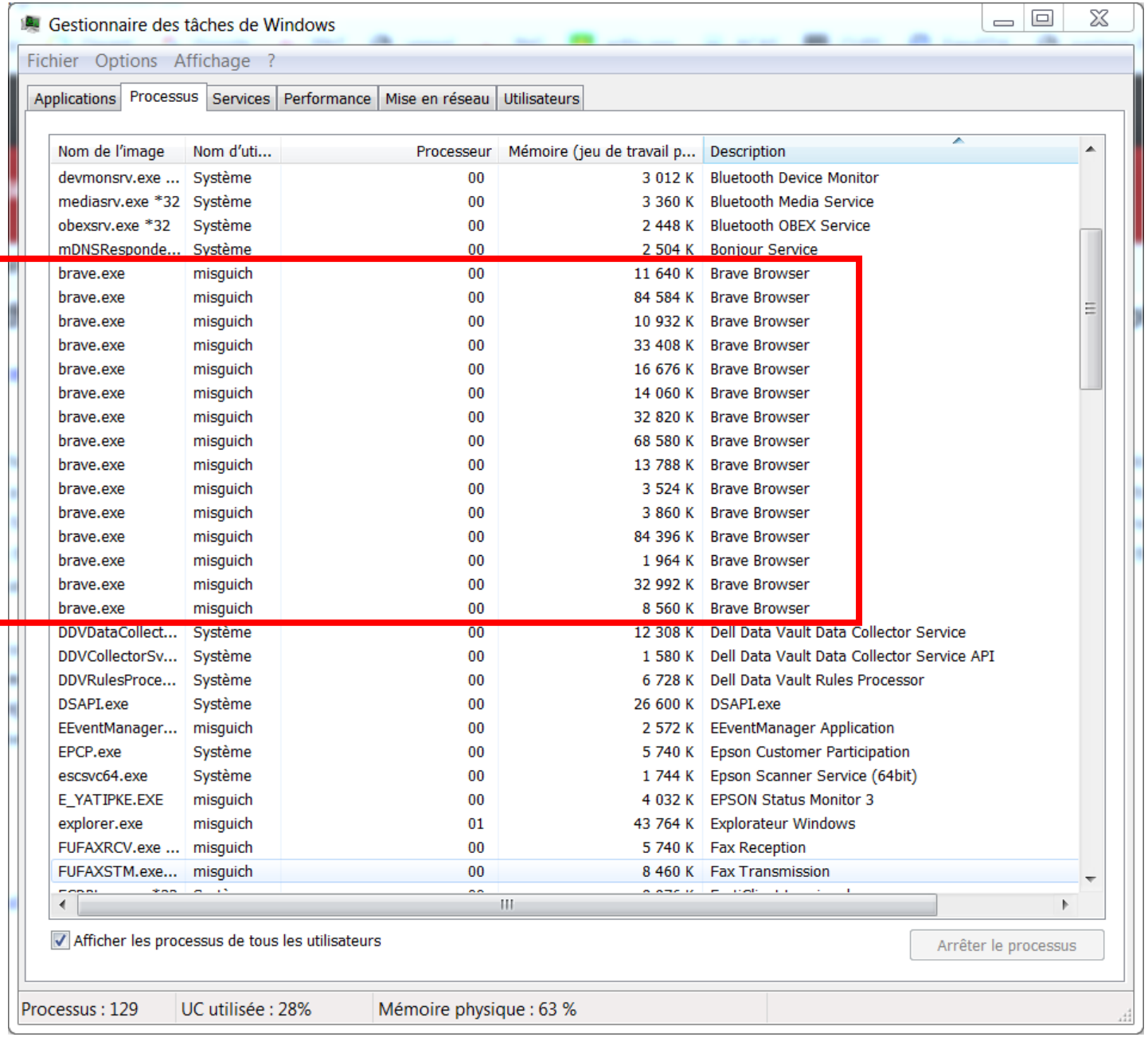
- Each process has its own memory space.
  → they are somewhat independent
- Switching from one process requires some interaction with the operating system → slow switching

Thread

- Threads (associated to a given process) share the same memory space
  → Threads can share information easily /quickly
- There is no memory "protection" between the threads of the same process → responsibility of the programmer
- Threads have little information of their own
  → faster to create than processes



Example of application using several processes



Parallel programing in Python

- Thread library (`import threading`)
  Can start several threads, but they will not run simultaneously (because of the Global Interpreter Lock – a.k.a. GIL).

  Can be useful for some I/O tasks (because the CPU will be waiting for some remote server, etc.), but not really for computations.
  … will not be discussed here.

- Multiprocessing library (`import multiprocessing`)
  Allows to perform tasks simultaneously (using *processes* instead of *threads*)
  We will present a few examples using:
  - `Process, Queue`
  - `Pool, map, imap`

Python/Multiprocessing

Multiprocessing.Process

```
import os, math
from multiprocessing import Process,Queue

def my_function(r,q):
    proc = os.getpid()
    i1=r[0]**3
    i2=r[1]**3
    print('Process #{0} will sum from {1}
to {2}'.format(proc,i1,i2))
    sum=0.0
    for i in xrange (i1,i2):
       sum+=math.sin(i)
    q.put((i1,i2,sum))
```

```
ranges = [ [1,100], [201,300],
[300,400],[401,500] ]
list_of_procs = []
q=Queue()
for r in ranges:
    p = Process(target=my_function,
args=(r,q))
    list_of_procs.append(p)
    p.start()
for p in list_of_procs:
    p.join()

results=[q.get() for p in list_of_procs]
print(results)
```

q: Object where each process can write its result

Python/Multiprocessing

Multiprocessing.Process

Processes do **not** share memory, which means that the global variables are *copied*, hence their value in the original process do not change.

This example *does not work* (i.e. output is [0,0,0,0])

```
import os, math
from multiprocessing import Process

A=[0,0,0,0]

def my_function(i):
      A[i]=A[i]+1
      return

list_of_procs = []
for r in range(4):
    p = Process(target=my_function, args=(r,))
    list_of_procs.append(p)
    p.start()

for p in list_of_procs:
    p.join()

print(A)
```

Python/multiprocessing

Multiprocessing.Pool

```
import numpy as np
import multiprocessing as mp
import numpy.random as npr
from sympy import *

def gsrm(i):
        n=200
        A = npr.randn(n,n)
        ev=np.linalg.eigvals(A)
        return i,ev[0].real

p=mp.Pool(processes=4)

print(p.map(gsrm,range(9)))

results=p.imap_unordered(gsrm,range(9))
for r in results:
        print (r)
```

Blocks untill all task are finished

Does not block

Python/multiprocessing

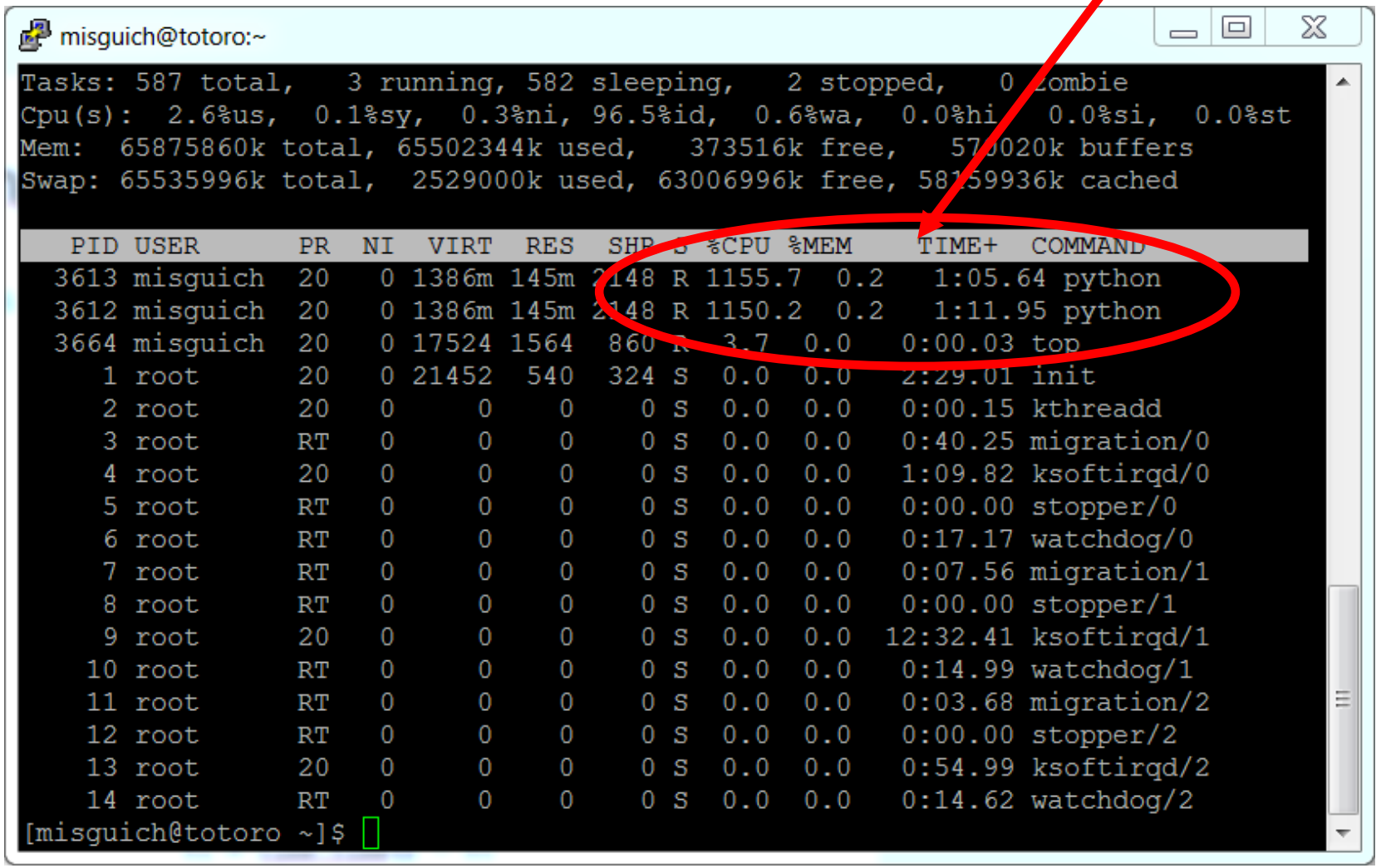
Multiprocessing.Pool & linear algebra: processes versus threads

```
import numpy as np
import multiprocessing as mp
import numpy.random as npr
import time


def smallest_ev(M):
        ev=np.linalg.eigvals(M)
        return ev[0].real

p=mp.Pool(processes=2)
#Two random matrices:
n=2000
A=npr.randn(n,n)
B=npr.randn(n,n)
t0 = time.time()
results=p.imap_unordered(smallest_ev,
[A,B])
t1 = time.time() - t0
print(" Time after .imap_unordered=%0.4f
s" % (t1))
for r in results:
        print (r)
t1 = time.time() - t0
print(" Time after print results=%0.4f s"
% (t1))

t0 = time.time()
print(smallest_ev(A))
print(smallest_ev(B))
t1 = time.time() - t0
print(" Sequential time=%0.4f s" % (t1))
```

Python/multiprocessing

Multiprocessing.Pool &
linear algebra: processes versus threads

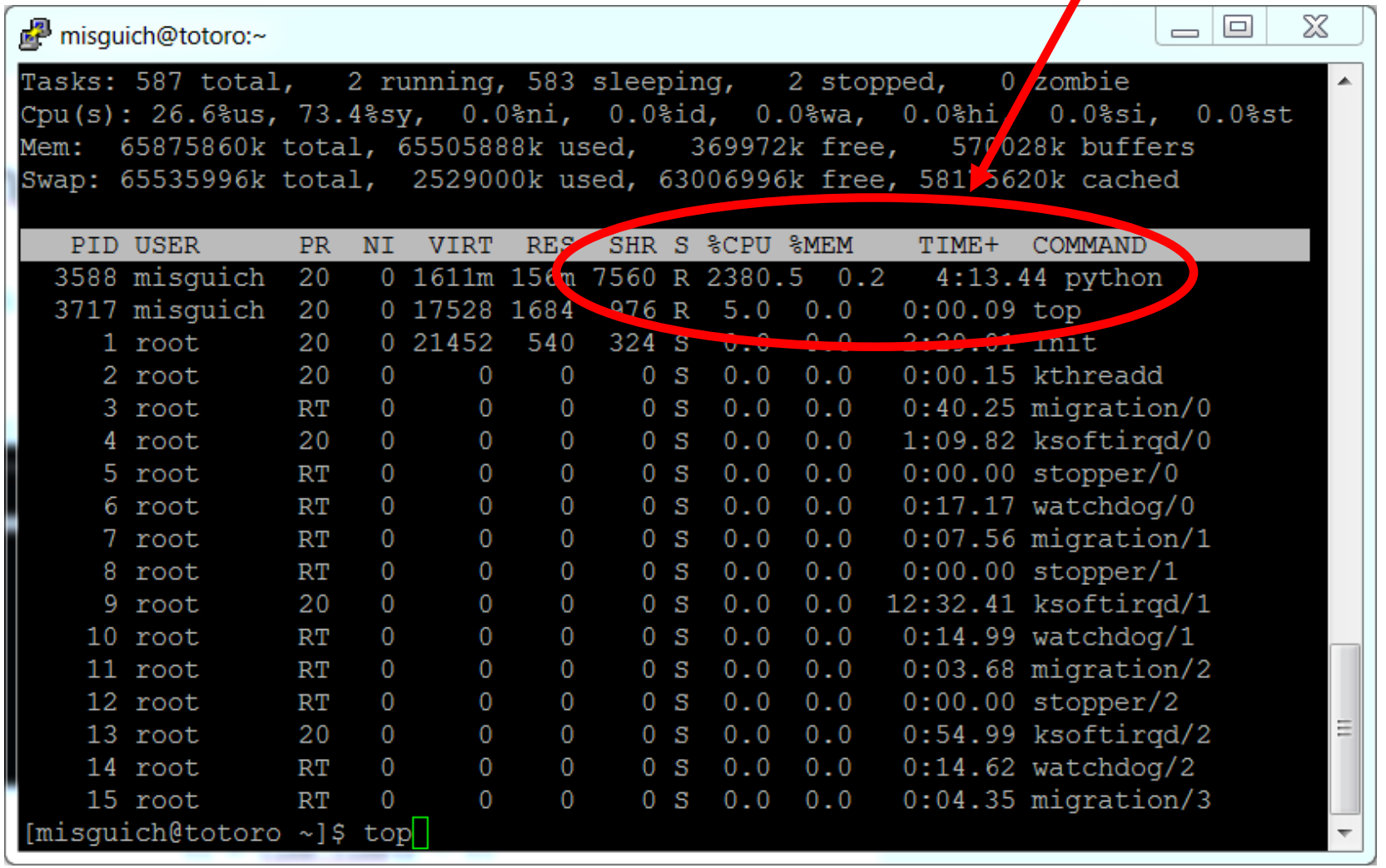
2 workers (=processes), each one
using ~12 threads (and 12 CPU cores)
[mutli-threaded linear algebra lib.]



Python/multiprocessing

Multiprocessing.Pool &
linear algebra: processes versus threads

Single Python processe
using ~24 threads (and 24 CPU cores)



Python/multiprocessing

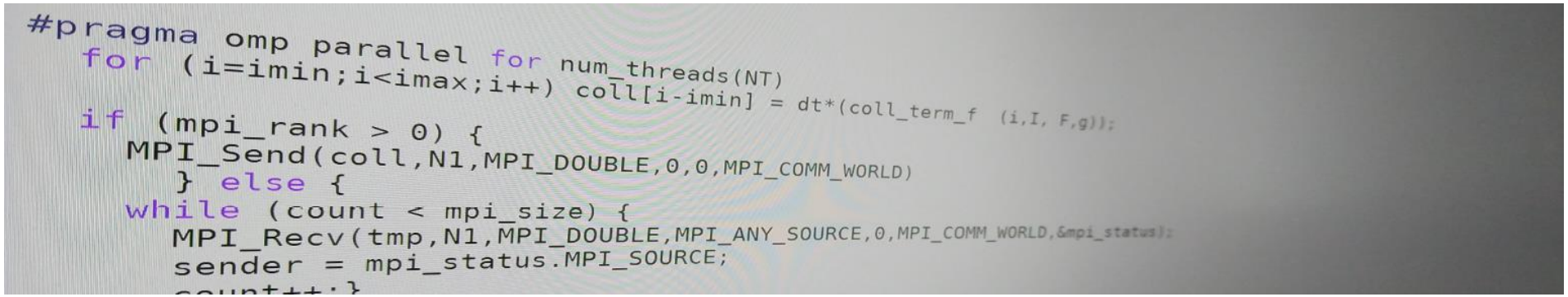
Multiprocessing.Pool & Sympy

```
import multiprocessing as mp
from sympy import *

p=mp.Pool(processes=3)
x = symbols('x')
print(p.map(integrate,[x,x**2,x**3]))
```

Python/multiprocessing

Multiprocessing.Pool & Sympy

```
import multiprocessing as mp
from sympy import *
import sys

x = symbols('x')
A= [ exp(x),sin(x),cos(x),cosh(x),sinh(x)]
def my_func(f):
      count=0
      while (count<10000) :
            f=diff(f,x)
            count=count+1
      return f
# Pass the wanted number of process as command-
line argument
np=int(sys.argv[1])
p=mp.Pool(processes=np)
results=p.map(my_func,A)
print(results)
```



Introduction to parallel programming (for physicists)

FRANÇOIS GÉLIS & GRÉGOIRE MISGUICH, IPhT courses, June 2019.

IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

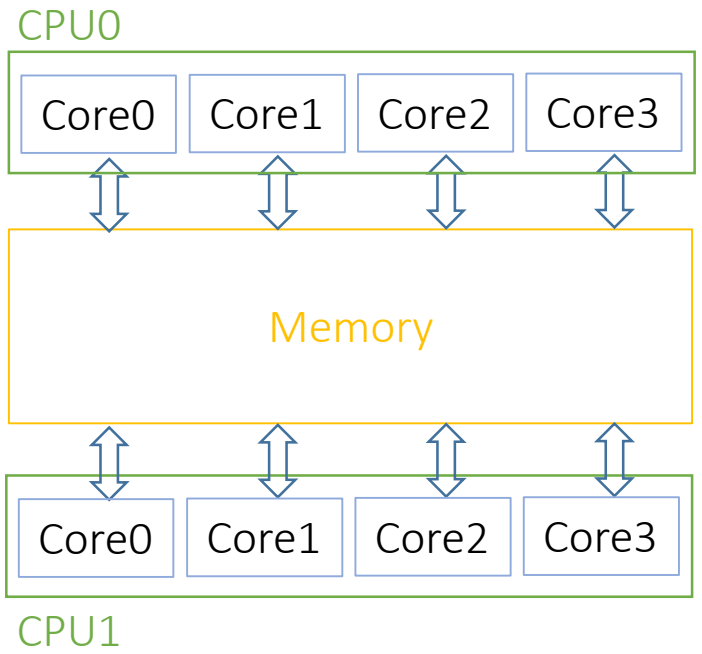
1. Introduction & hardware aspects (FG)
2. A few words about Maple & Mathematica
3. Linear algebra libraries
4. Fast Fourier transform
5. Python Multiprocessing
6. OpenMP
7. MPI (FG)
8. MPI+OpenMP (FG)

These slides (GM)

OpenMP

OpenMP

- Based on threads
- For shared-memory architectures
- Standardized
- Mature (goes back to the 90's)
- Portable (supported by many compilers, systems and languages)
- Efficient, with minimal programmation effort ☺

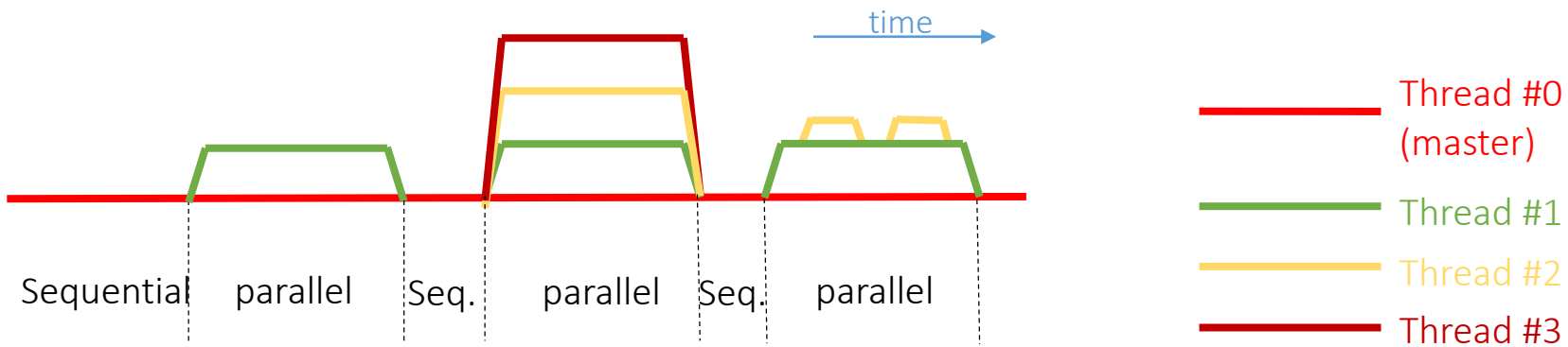


OpenMP

Basic idea:

Add compiler *directives* to your existing sequential code (written in C, C++ or Fortran) to tell:

- Which instructions should executed in parallel
- How to distribute (and synchronize) the instructions over the threads
- How to distribute/share the data over the threads



‘Hello world’ with OpenMP

Fotran example. Compile with

```
> ifort -fopenmp hello.f90
```

Parallel section
=creation of a *team* of threads

```
program hello
USE OMP_LIB
    PRINT *,"Hello, I am thread #", OMP_GET_THREAD_NUM()
    PRINT *,"a.k.a. the master thread"
!$OMP PARALLEL NUM_THREADS(3)
    PRINT *,"I am thread #", OMP_GET_THREAD_NUM()
!$OMP END PARALLEL

end program hello
```

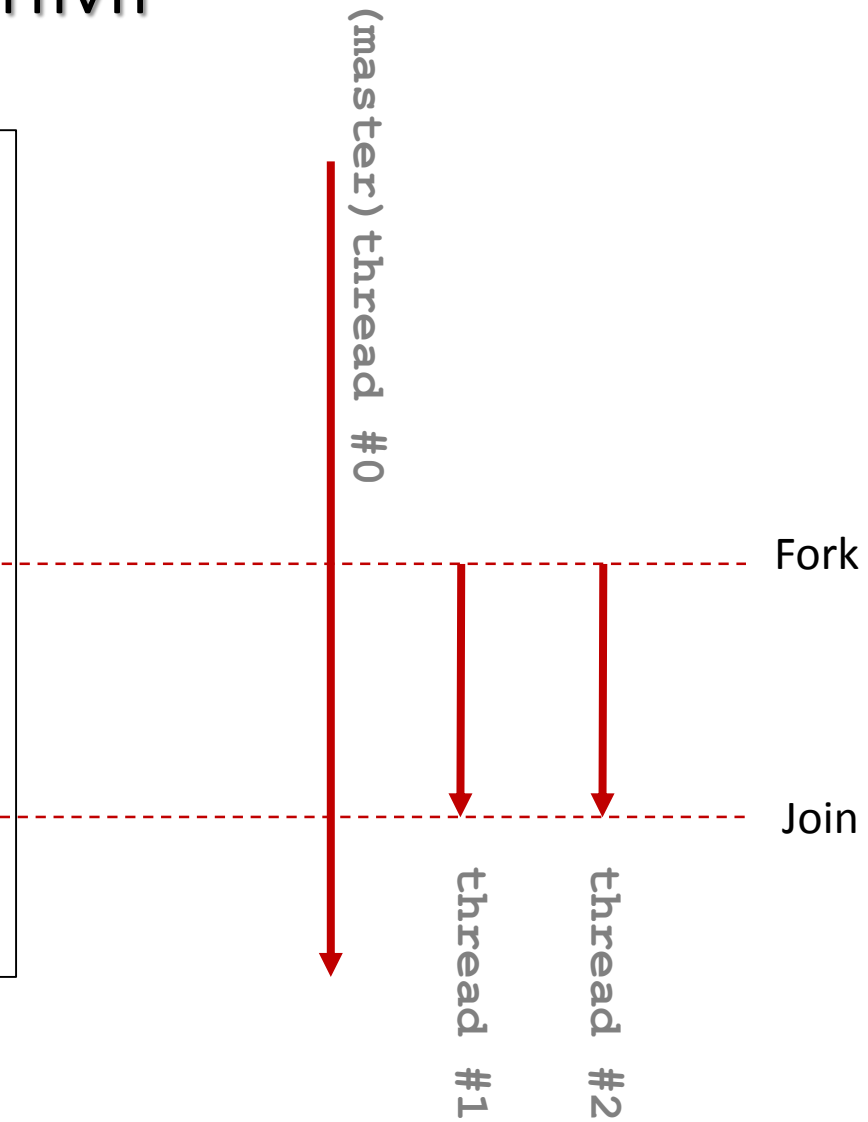
OpenMP compiler directives

Output:

```
[misguich@totoro OpenMP]$ ./a.out
 Hello, I am thread #           0
 a.k.a. the master thread
 I am thread #           0
 I am thread #           1
 I am thread #           2
```

‘Hello world’ with OpenMP

```
program hello
USE OMP_LIB
     PRINT *,"Hello, I am thread
#", OMP_GET_THREAD_NUM()
     PRINT *,"a.k.a. the master
thread"
!$OMP PARALLEL NUM_THREADS(3)
     PRINT *,"I am thread #",
OMP_GET_THREAD_NUM()

!$OMP END PARALLEL

end program hello
```



Implicit barrier at the end of the parallel section
(wait until all the threads in the team have reach this point)

‘Hello world’ with OpenMP

C++ example. Compile with: `icc -fopenmp hello.cpp`

```
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel num_threads(3)
{
printf("Hello, I am thread
%d/%d\n",omp_get_thread_num(),omp_get_num_threads());
}
}
```

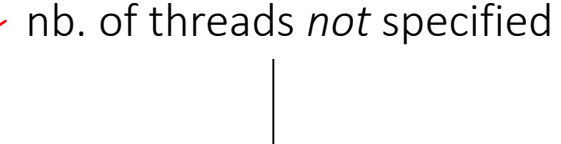
Output:

```
 ./hello-cpp.exe
Hello, I am thread 0/3
Hello, I am thread 2/3
Hello, I am thread 1/3
```

Setting the number of threads

Via the environment variable `OMP_NUM_THREADS`

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel num_threads(3)
  { printf("%d ",omp_get_thread_num()); }
}
```



Output:

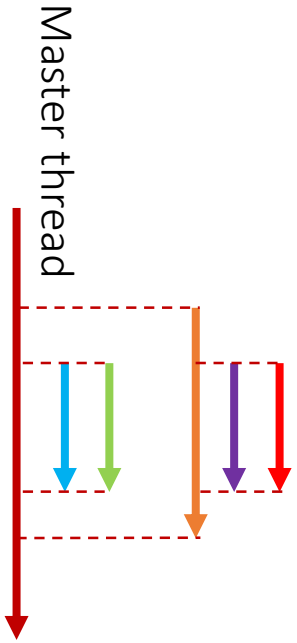
```
> export OMP_NUM_THREADS=5
> ./env-var.exe
2 0 3 1 4
```

One can also use: `omp_set_num_threads()` to override the value of the environment variable.

Nested

```
#include <stdio.h>
#include <omp.h>
int main() {
  omp_set_nested(1);
  #pragma omp parallel num_threads(2)
  {
    #pragma omp parallel num_threads(3)
    printf("Hello, world\n");
  }
  return 0;
}
```

Allows *one* nesting level



Code ouput:

```
$ ./nested.exe
Hello, I am #0
Hello, I am #0
Hello, I am #1
Hello, I am #2
Hello, I am #2
Hello, I am #1
```

Remarks:

- If `omp_set_nested(0)` → only 2 prints
- There are two teams, and in each team the threads are numbered 0,1,2 .

Sections

```
#include <stdio.h>
#include <unistd.h>

int main () {

long int i1,i2,imax=1e9;
double sum1=0,sum2=0;

#pragma omp parallel sections
num_threads(3)
 {

#pragma omp section
   {//---- Task 1 ----
     for (i1=0;i1<imax;i1++)
     sum1+=i1;
     printf("task1 done.\n");
   }

#pragma omp section
   {//---- Task 2 ----
     for (i2=0;i2<imax;i2++)
     sum2-=i2;
     printf("task2 done.\n");
   }

#pragma omp section
   {// Task 3 (monitors Tasks 1 & 2)
    for (;i1<imax && i2<imax;) {
     sleep(1);
     printf("i1=%ld i2=%ld\n", i1,
i2);
    }
   }
 }
return 0;
}
```

Sections

```
$ time ./sections.exe
i1=115696211 i2=45202371
i1=237163286 i2=78929005
i1=357525610 i2=115805983
i1=477927610 i2=152408869
i1=598161099 i2=189138856
i1=718318718 i2=225871473
i1=838702317 i2=262691740
i1=959165968 i2=299535339
task1 done.
i1=1000000000 i2=538152420
task2 done.

real    0m10.354s
user    0m18.702s
sys     0m0.000s
```

Using shared variables slows down

Compare the code & output below with the previous example – here **loop indices** are **private** variables

```
#include <stdio.h>

int main () {
long int imax=1e9;
double sum1=0,sum2=0; //Shared
#pragma omp parallel sections
num_threads(3)
 {
#pragma omp section
   {//---- Task 1 ----

     for (long int i1=0;i1<imax;i1++)
     sum1+=i1;
     printf("task1 done.\n");
   }
#pragma omp section
   {//---- Task 2 ----

     for (long int i2=0;i2<imax;i2++)
     sum2-=i2;
     printf("task2 done.\n");
   }
 }
printf("sum=%g\n",sum1+sum2);
return 0;
}
```

→ 5 times faster

```
 time ./sections_fast.exe
task2 done.
task1 done.
sum=0

real    0m1.959s
user    0m3.260s
sys     0m0.002s
```

Using shared variables slows down

Compare the code & output below with the previous examples – here **loop indices** and **sums** are **private** vars.

```
#include <stdio.h>

int main () {
long int imax=1e9;

#pragma omp parallel sections
num_threads(3)
 {
#pragma omp section
   {//---- Task 1 ----
     double sum1=0;//Private
     for (long int i1=0;i1<imax;i1++)
     sum1+=i1;
     printf("task1 done.\n");
   }
#pragma omp section
   {//---- Task 2 ----
     double sum2=0; //Private
     for (long int i2=0;i2<imax;i2++)
     sum2-=i2;
     printf("task2 done.\n");
   }
 }
return 0;
}
```

→ Almost 10 times faster than with global variables for **sum1** and **sum2**. But these sums are now lost when exiting the parallel region.

```
 time
./sections_fast.exe
task1 done.
task2 done.

real    0m0.332s
user    0m0.658s
sys     0m0.001s
```

shared, private

Inside a parallel region:

- **Shared** variables can be read and written by all the threads.
  Be careful with potential *race conditions*. If two threads simultaneously write at the same memory location (variable), or if a threads reads it while another one writes on it, the result is potentially random (possibility of corrupted data). There will be no error message !

- If a variable is **private**, each thread has its own copy. If a variable existed with the same name before the parallel construct, it is not affected when exiting the parallel region.
- By default, variables declared outside the parallel regions are shared, and those declared inside are private.
- When entering a parallel region, the private variables are not initialized. In C++ they are created using the default constructor

firstprivate, lastprivate

- **Firstprivate**: special case of private variable, where each local copy is initialized from the value of the variable with the same name before the beginning of the parallel region

- **Lastprivate**: special case of private variable for parallel section or parallel for, where, at the end of the parallel region, the variable with the same name outside the parallel region gets the value of local copy of the thread doing the last iteration (or last section).

```
int a=1;
#pragma omp parallel firstprivate(a)
{
// Each thread has its own copy of a,
// initialize to 1.
}
// Here a is 1 again, whatever the
// threads did with their local
// copies of a.
```

```
int a=1;
#pragma omp parallel lastprivate(a)
{
#pragma omp section
a=2;
#pragma omp section
a=3;
}
// Here a is 3
```

Shared, private, firstprivate, lastprivate

```
#include <stdio.h>
int main() {int a=1,b=1,c=1;
#pragma omp parallel num_threads(4)
{
#pragma omp sections firstprivate(b) lastprivate(c)
{
#pragma omp section
b=0;
#pragma omp section
a=a+1;
#pragma omp section
c=b+1;
#pragma omp section
c=b+3;
} }

printf("a=%d b=%d c=%d\n",a,b,c);
return 0;
}
```

Can you predict the output ?
(there is a trap)

Shared, private, firstprivate, lastprivate

```
#include <stdio.h>
int main() {int a=1,b=1,c=1;
#pragma omp parallel num_threads(4)
{
#pragma omp sections firstprivate(b) lastprivate(c)
{
#pragma omp section
b=0;
#pragma omp section
a=a+1;
#pragma omp section
c=b+1;
#pragma omp section
c=b+3;
} }

printf("a=%d b=%d c=%d\n",a,b,c);
return 0;
}
```

Outputs:

```
$ ./shared_private.exe
a=2 b=1 c=4
$ ./shared_private.exe
a=2 b=1 c=3
```

→ We randomly get **c=3** or **c=4** !
Explanation: it sometimes happens that the same thread executes the sections #1 and then #4.

atomic

Ensures that a (single) memory location is not updated simultaneously by >1 threads

```
#include <stdio.h>

int main () {
long int imax=1e9;
double sum=0;//Shared
#pragma omp parallel sections
num_threads(3)
 {
#pragma omp section
   {//---- Task 1 ----
     double sum1=0;//Private
     for (long int i1=0;i1<imax;i1++)
     sum1+=i1;
#pragma omp atomic
     sum+=sum1;
printf("task1 done.\n");
   }
#pragma omp section
   {//---- Task 2 ----
     double sum2=0; //Private
     for (long int i2=0;i2<imax;i2++)
     sum2-=i2;
#pragma omp atomic
     sum+=sum2;
     printf("task2 done.\n");
   }
 }
printf("sum=%g\n",sum);
return 0;
}
```

Required, to ensure that threads do not attempt to update the shared variable **sum** simultaneously.

Critical and atomic

Intructions or blocks which must be executed *one thread at a time*

```
#pragma omp parallel
{
…
#pragma omp atomic
//Single memory location update
      sum+=...;

#pragma omp critical
       {
// block, executed one thread at a time
// if a thread reaches this block while
// another one is already executing it,
// it will wait that the 1st one finishes
       }
}
```

Faster, use this whenever possible

shared variable

Heavier/slower synchronization machinery

For loops

with OpenMP

For loops – basic example 1, filling a shared array

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
int main() {
long int n=1e9;
double *A=new double [n];
printf("n=%li\n",n);
#pragma omp parallel
{
#pragma omp for
      for (long int i=0;i<n;i++) {
                A[i]=sin(i);
      }
}
printf("%g",A[999]);
}
```

Beginning of the parallel section

The loop iterations are distributed to the threads

Parallel loop index is private by default

All the threads have access to all the array elements. `A` is a shared variable

Implicit barrier at the end of the loop (unless one specifies `nowait`)

Check the speed-up

```
$ export OMP_NUM_THREADS=1;time ./for.exe
n=1000000000
-0.0264608
real    0m14.434s
user    0m12.879s
sys     0m1.543s

$ export OMP_NUM_THREADS=12;time ./for.exe
n=1000000000
-0.0264608
real    0m1.423s
user    0m13.830s
sys     0m1.718s
```

Remark: if the printf statement at the end is removed and code compiled with icc:

```
n=1000000000
real    0m0.004s
user    0m0.001s
sys     0m0.002s
```

→ without `printf`, the compiler (here `icc`) has completely removed the loop !

Check the speed-up

Remove the printf statement and compile with `icc`:

```
 $ icc for.cpp -fopenmp
 $ time ./a.out
n=1000000000

real    0m0.203s
user    0m0.008s
sys     0m0.002s
```

→ without `printf`, the compiler (here `icc`) has completely removed the loop !

For loops – basic example 2, performing a sum [BUG]

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
int main() {
long int n=1e9;
double sum=0;
#pragma omp parallel
{
#pragma omp single
printf("I am #%d in a team of %d
threads\n",
       omp_get_thread_num(),
       omp_get_num_threads());
double local_sum=0; // private
variable
#pragma omp for
       for (long int i=0;i<n;i++) {
      double x=i*1.0/n,y=sqrt(1-x*x);
      local_sum+=y;
     }
sum+=local_sum;
```

```
}
printf("Pi~%.15f\n",4*sum/n);
}
```

Output:

```
I am #1 in a team of 24
threads
Pi~2.639864229928272
```

Why is the result (completely) wrong ?

For loops – basic example 2, performing a sum

```
#include <stdio.h>
#include <omp.h>
#include <math.h>
int main() {
long int n=1e9;
double sum=0;
#pragma omp parallel
{
#pragma omp single
printf("I am #%d in a team of %d
threads\n",
       omp_get_thread_num(),
       omp_get_num_threads());
double local_sum=0; // private
variable
#pragma omp for
       for (long int i=0;i<n;i++) {
      double x=i*1.0/n,y=sqrt(1-x*x);
      local_sum+=y;
     }
#pragma omp atomic
sum+=local_sum;
}
printf("Pi~%.15f\n",4*sum/n);
}
```

```
$ ./omp-for2.exe
I am #20 in a team of 24 threads
Pi~3.141592655589728
```

atomic : Update of a single memory location, executed **one thread at a time**

For loops

with reduction

```
#include <stdio.h>
#include <omp.h>
#include <math.h>

int main() {
long int n=1e9;
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (long int i=0;i<n;i++) {
        double x=i*1.0/n,y=sqrt(1-x*x);
        sum+=y;
}
printf("Pi~%.15f\n",4*sum/n);
}
```

`parallel` and `for` directives merged in a single line

`sum` is a shared variable before the parallel section. In the `parallel for` loop a private copy of `sum` is created for each thread. At the end of the loop the private copies are combined using the operation '+'.

Possible reduction operators:

+ - * & | ^ && ||

For loops

with `reduction` in Fortran

```
program pi
USE OMP_LIB
INTEGER  n, i
DOUBLE PRECISION sum, x
n=1e9
!$OMP PARALLEL DO REDUCTION(+:sum) private(x)
do i=0,n-1
  x=(i*1.0)/n
  sum=sum+sqrt(1-x*x)
enddo
!$OMP END PARALLEL DO
print *, sum/n*4
end program pi
```

```
$make pi_f90.exe
gfortran -fopenmp pi_f90.f90 -o  pi_f90.exe
$ export OMP_NUM_THREADS=10;time ./pi_f90.exe
   3.14159265555533034
real    0m2.115s
user    0m21.133s
sys     0m0.000s
$ export OMP_NUM_THREADS=1;time ./pi_f90.exe
   3.14159265555977323
real    0m19.771s
user    0m19.774s
sys     0m0.000s
```

Note the slightly different results
(round off errors)

For loops

`schedule` clause

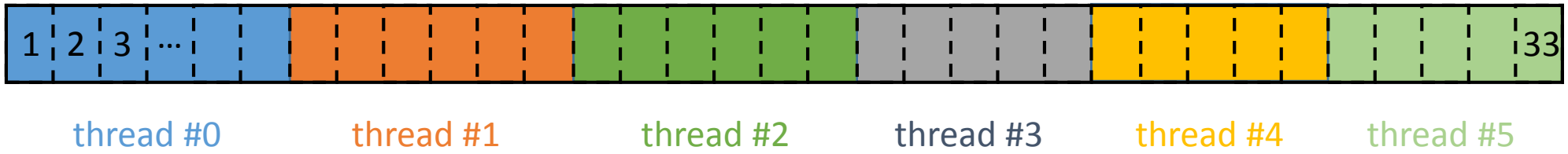
```
#pragma omp for schedule(  type  , [chunk_size] )
```

`static`
`dynamic`
`guided`
`runtime`
`auto` (will not be discussed here)

For loops

Schedule(static):

- Iterations are divided into ‘chunks’ of size `chunk_size` and distributed cyclically to the threads.
- If the `chunk_size` is not specified, the iterations are divided into (almost) equal chunks, and each thread executes one chunk (example below).



Example above:

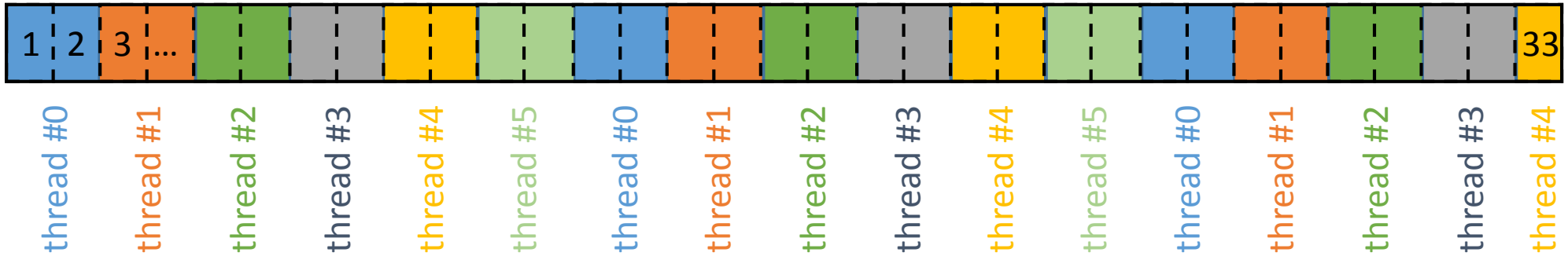
```
#pragma omp parallel num_threads(6)
{
#pragma omp for schedule(static)
for (int i=0;i<33;i++)
…
}
```

For loops

Schedule(static)

Another example:

```
#pragma omp parallel num_threads(6)
{
#pragma omp for schedule(static,2)
for (int i=0;i<33;i++)
…
}
```



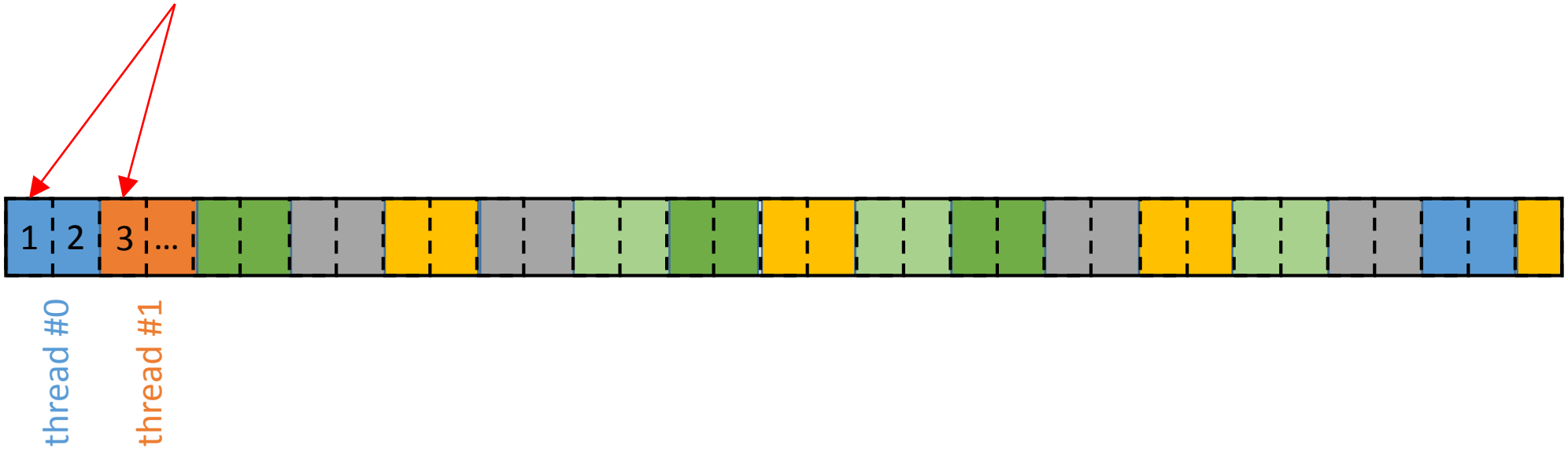
Advantage of large chunks: less overhead, cache friendly
Advantage of small chunks: better load balance between the threads

For loops

Schedule(dynamic)

- The iterations are divided in chunks of size `chunk_size` (the last one can be smaller) .
- When a thread is idle, it is assigned a new chunk (first come, first served).
- If `chunk_size` is not specified, it is set to 1.

1 and 3: long tasks.



For loops

Schedule(guided)

- Similar to dynamic, but the chunk size is initially large, and decreased gradually
- The size of a chunk is proportional to the number of remaining iterations, divide by the number of threads)
- `chunk_size` specifies the minimum size of the chunks. If not specified, this minimal size is set to 1.

For loops

Schedule(runtime)

- The scheduling method is decided only during the execution (=runtime), according to the environment variable `OMP_SCHEDULE` (or using `omp_set_schedule(…)`);

```
#include <stdio.h>
#include <omp.h>

int main() {
int n=6;int A[n];
#pragma omp parallel num_threads(2)
{
int id=omp_get_thread_num();
#pragma omp for schedule(runtime)
for (int i=0;i<n;i++) A[i]=id;
}
for (int i=0;i<n;i++)
printf("A[%d]=%d\n",i,A[i]);
}
```

```
$ export OMP_SCHEDULE=guided,1
$ ./for-schedule.exe
A[0]=0
A[1]=0
A[2]=0
A[3]=0
A[4]=0
A[5]=1
A[6]=1
A[7]=1
A[8]=0
A[9]=1
```

For loops

with `collapse` clause

```
( …)
int main() {
int i,j,k,l,m;
double sum=0;
#pragma omp parallel for collapse(4)
reduction(+:sum) private(i,j,k,l,m)
for (i=0;i<2;i++)
for (j=0;j<2;j++)
for (k=0;k<2;k++)
for (l=0;l<2;l++)
for (m=0;m<2;m++)
sum+=f(i,j,k,l,m);
printf("sum=%g\n",sum);
}
```

Transformed by the compiler into a single (parallelized) loop with 2*2*2*2*2=32 iterations

nowait

An example

```
#pragma omp parallel
{
#pragma omp for nowait
for (i=1; i<n; i++)
    b[i] = a[i] + a[i-1];
#pragma omp for
for (i=0; i<m; i++)
    c[i] = f ( a[i] );
}
…
```

Once a thread of the team has finished working on first loop, it can start working on the next one.

I absence of the `nowait` option, there is an implicit barrier at the end of the loop. All the threads will wait that all the loop iterations are completed before going on. Same implicit barrier at the end of `sections` or `single` directives.

barrier

An example

```
int i,id;
double a[nt],b[nt];
#pragma omp parallel private(i,id) shared(a,b) num_threads(nt)
{
      id=omp_get_thread_num();
      b[id]=0;
      a[id]= big_calculation(id);
#pragma omp barrier

#pragma omp for
      for (i=0; i<nt; i++)
            b[i]= another_calculation( a[(i+1)% nt], a[i] );
}
```

All threads wait here until they have all reach this point. This guaranties that all the `a[i]` are computed before proceeding.

Thread safety

A function is said to be **thread safe**, when it does what it is expected to do even when executed concurrently by several threads.

Examples: `cout` in C++, or `srand`

```
#include <omp.h>
#include <iostream>
int main() {
#pragma omp parallel num_threads(4)
  {
  std::cout<<"I am the thread#"
  <<omp_get_thread_num()<<"\n";
  }
}
```

```
./cout.exe
I am the thread #I am the thread #I am the thread #3
1
2
I am the thread #0
```

Random numbers

Basic `rand()` is **not thead-safe**

```
…
int main() {
#pragma omp parallel num_threads(4)
{
#pragma omp critical
    {
    int id=omp_get_thread_num();
    srand(id+2019);
    int r=rand();
    printf("Thread #%d, r=%d\n",id,r);
    }
}}
```

Thread-dependent *seed*

```
$ ./srand.exe
Thread #2, r=48485172
Thread #1, r=1644198542
Thread #0, r=1028584130
Thread #3, r=105705637
$ ./srand.exe
Thread #1, r=1644198542
Thread #0, r=105705637
Thread #3, r=1341262422
Thread #2, r=1028584130
$ ./srand.exe
Thread #1, r=881877917
Thread #3, r=1341262422
Thread #0, r=105705637
Thread #2, r=1644198542
```

- We could have (naively) expected to get always the same 4 integers, but that is not the case
- Reason: `rand` has some internal state variables → the different calls from different threads “interfere”.

Random numbers

`drand48()` is thead-safe. Example which estimates π.

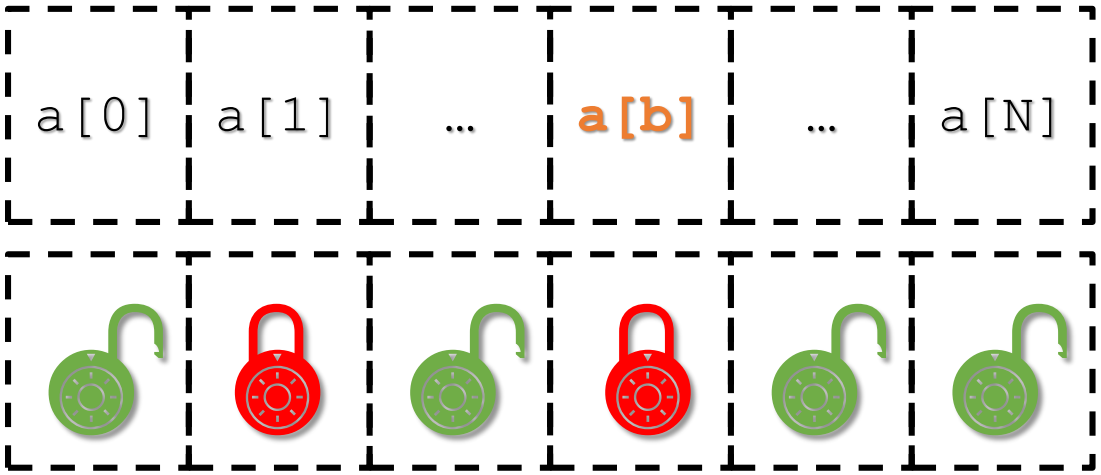
```
#include ...

int main (int argc, char *argv[])
{
  int i, count, N,rseed;
  struct drand48_data buffer;
  double x, y;
  N  = atoi(argv[1]);
  count = 0;

#pragma omp parallel default(none)
private(x, y, i, rseed, buffer)
shared(N,count)
{
int rseed = omp_get_thread_num();
srand48_r (rseed, &buffer);
```

```
#pragma omp for reduction(+:count)
  for (i=0; i<N; i++) {
    drand48_r (&buffer, &x);
    drand48_r (&buffer, &y);
    if (x*x + y*y <= 1.0) count++;
  }
}
double pi= 4.0*count/N;
printf("Pi~ %g\n",pi);
}
```

buffer : private variable (of type `drand48_data`) to store the internal state of each random generator (one for each thread)

Locks

Protecting data (sometimes more flexible than `atomic` & `critical`)




```
l=lock associated to a[b]
omp_set_lock( &l );
    a[b]= ... ;
omp_unset_lock(&l);
```

Access to **`a[b]`** is blocked for the other threads. But they can still access concurrently the other elements of the array `a`. Note: only the thread which has set a lock can unset it.

```
#pragma omp critical
{
a[b]= ... ;
}
```

Array elements can be updated by a single thread at a time. The access to this line (hence the whole array) is impossible if one thread is already there.

Locks

(classic) histogram example

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

const int Nbins=10;
int f(int i) {
  //. . .
  return b;
}
int main() {
  const int Nsamples=1e6;
  int histo[Nbins];
  omp_lock_t locks[Nbins];
  for (int b=0;b<Nbins;b++) {
    histo[b]=0;
    omp_init_lock(locks+b);
  }
```

Initialiazing an array of locks

```
#pragma omp parallel for
  for (int i=0;i<Nsamples;i++) {
    int b=f(i);
    omp_set_lock(locks+b);
       histo[b]+=1;
    omp_unset_lock(locks+b);
  }
  int total=0;
  for (int b=0;b<Nbins;b++)
    total+=histo[b],
      printf("histo[%d]=%ld\n",b,
             histo[b]);
      printf("total=%ld/%ld\n",
              total,Nsamples);
}
```

Lock

another histogram example: for each bin `b`, make the list of all 'configurations' `c` such that energy(c) ∈ `b`.

```
(… #include …)
#define N 4  // N: numb. of "spins"

double energy(long int c) {
 // 'energy' function
 // c: configuration coded in
 // binary (integer)
 // …
  return e;
}

int main() {
  const int Nbins=N*N;
  const int Nconf=1<<N; // =2^N
  vector<vector<int> > histo(Nbins
);
  omp_lock_t locks[Nbins];
  for (int b=0;b<Nbins;b++)
    omp_init_lock(locks+b);

// Parallel loop over spins config.
#pragma omp parallel for
  for (int c=0;c<Nconf;c++) {
        double ener=energy(c);
        int b=int(ener); // bin
num. associated to ener
        omp_set_lock(locks+b);
        histo[b].push_back(c);
        omp_unset_lock(locks+b);
  }
}
```

Declare & initialize an array of locks (one lock for each 'bin')

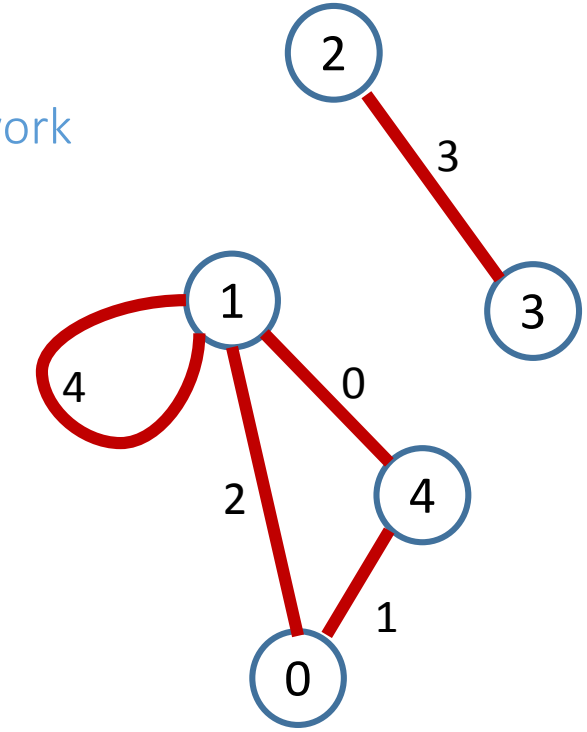
Here the use of `atomic` would have not been possible, since `vector::push_back(…)` is not an "atomic" statement. `critical` would have been possible, but slower.

Lock

Applications: computing the number of neighbors of a given node in a network

```
for (i=0; i<Nv; i++)
      omp_init_lock(&locks[i]);

#pragma omp parallel for
for (j=0; j<Nb j++){
  omp_set_lock(&locks[bondA[j]]);
  omp_set_lock(&locks[bondB[j]]);
  degree[bondA[j]]++;
  degree[bondB[j]]++;

  omp_unset_lock(&locks[bondA[j]]);
  omp_unset_lock(&locks[bondB[j]]);
}
```

BUG !



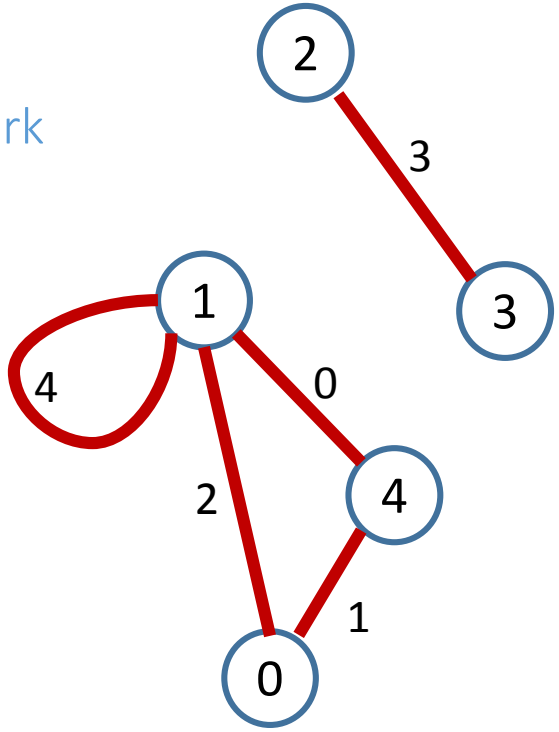

```
Nv: number of nodes=5
Nb: number of bonds=5
bondA[0]=1;bondsB[0]=4;
bondA[1]=4;bondsB[1]=0;
bondA[2]=0;bondsB[2]=1;
bondA[3]=3;bondsB[3]=2;
bondA[4]=1;bondsB[4]=1;
```

Lock

Applications: computing the number of neighbors of a given node in a network

```
for (i=0; i<Nv; i++)
      omp_init_lock(&locks[i]);

#pragma omp parallel for
for (j=0; j<Nb j++){
  omp_set_lock(&locks[bondA[j]]);
  degree[bondA[j]]++;
  omp_unset_lock(&locks[bondA[j]]);

  omp_set_lock(&locks[bondB[j]]);
  degree[bondB[j]]++;
  omp_unset_lock(&locks[bondB[j]]);
}
```

OK !




```
Nv: number of nodes=5
Nb: number of bonds=5
bondA[0]=1;bondsB[0]=4;
bondA[1]=4;bondsB[1]=0;
bondA[2]=0;bondsB[2]=1;
bondA[3]=3;bondsB[3]=2;
bondA[4]=1;bondsB[4]=1;
```

Tasks

Useful to parallelized « irregular » problems, unbounded loops, or recursive algorithms. (since OpenMP3).

- Each time a thread reaches a task directive, the corresponding unit of work is added to a queue, and that thread can continue.
- A thread of the team (the same or another one) will execute the task (now of later).
- All the tasks created by any thread in the current team will be completed before exiting the parallel region.

Tasks

A simple/classic recursive example: Fibonacci

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
long int fib(int);
int main(){
int n=50;
#pragma omp parallel
  {
#pragma omp single
     {
printf("Fib(%d) = %ld\n",n,
fibonacci(n));
     }
  }
}
```

```
long int fibonacci(int n){
  long int i, j;
  if(n < 2) return n;
  if (n<20) {
      return fib(n-1)+fib(n-2);
  }
  else
   {
#pragma omp task shared (i)
          i= fibonacci(n-1);
#pragma omp task shared (j)
          j= fibonacci(n-2);
#pragma omp taskwait
     }
     return (i+j);
}
```

Only **one** thread of the team calls **fibonacci**.

Cutoff needed for performance, to avoid creating too many very small tasks.

Recursive calls → exploration of a binary tree

wait that both tasks above are finished before returning the result `i+j`

Tasks

A simple/classic recursive example: Fibonacci

Code output:

```
$export OMP_NUM_THREADS=4
$time ./fib.exe
Fib(50) = 12586269025

real    0m44.913s
user    1m57.810s
sys     0m0.005s

$export OMP_NUM_THREADS=12
$time ./fib.exe
Fib(50) = 12586269025

real    0m22.571s
user    2m2.061s
sys     0m0.010s
```

```
$export OMP_NUM_THREADS=24
$time ./fib.exe
Fib(50) = 12586269025

real    0m12.613s
user    2m16.977s
sys     0m0.011s

$export OMP_NUM_THREADS=30
$time ./fib.exe
Fib(50) = 12586269025

real    0m11.632s
user    2m18.980s
sys     0m0.132s
```

In this example it is advantageous to start more threads than the number of physical cores
(12 physical cores in the runs above). This is because the threads are idle part of the time.

Note: This recursive algorithm is (of course) not the efficient way to compute the Fibonacci number.
The Fibonacci sequence is here just an excuse to explore a binary tree recursively

Tasks+Locks

A simple/classic recursive example: Fibonacci

```
… #include …

long int fibonacci(int);
const int N=90;
omp_lock_t locks[N+1];
long int fib[N+1];

int main(){
for (int i=0;i<=N;i++)
omp_init_lock(locks+i);
for (int i=0;i<=N;i++) fib[i]=-1;
#pragma omp parallel
  {
#pragma omp single
  printf("Fib(%d) = %ld\n",N,
  fibonacci(N));
  }
}
```

```
long int fibonacci(int n){
    omp_set_lock(locks+n);
    if (fib[n]==-1) {//not yet computed
    if (n < 2) fib[n]=n;
    else {
            long int i, j;
#pragma omp task shared (i)
            i=fibonacci(n-1);
#pragma omp task shared (j)
            j=fibonacci(n-2);
#pragma omp taskwait
            fib[n]=i+j;
    }
 }
    omp_unset_lock(locks+n);
    return fib[n];
}
```

Distributed Memory Parallelization

Distributed memory "computers"

- Organization of a computer cluster:
  - one front-end node for compilation and administration tasks
  - many nodes for computations, not directly accessible
  - computation tasks submitted via a batch system
- The nodes are connected via a network
- Types of network connections:
  - 1 Gb/sec ethernet (125 MB/sec, latency around 1 ms)
  - 10 Gb/sec ethernet (bandwidth $\times 10$, similar latency)
  - Infiniband (latency around 1 microsec)
- Communications are slow (compared to shared memory)
- Communications must be handled explicitly in the program

Example of task submission script (for PBS/Torque)

Listing 1: *script.pbs*

```
#PBS —S /bin/bash
#PBS —N boltzmann
#PBS —e job.err
#PBS —o job.log
#PBS —m abe
#PBS —M francois.gelis@ipht.fr
#PBS —l nodes=32:ppn=16

module load openmpi/1.6.4
cd $PBS_O_WORKDIR

mpirun —npernode 1 ./my_program
```

- This example will start 32 copies of *my_program* (one per node)
- Then, do: *qsub script.pbs*
- Other commands: *qstat, qdel*
- This is sufficient to start independent tasks on several nodes
- Non-interactive: I/O to files only

Message Passing Interface (MPI)

- MPI provides high level functions to exchange data between jobs on several nodes, that hide the network details
- Standardized since 1994: various implementations with compatible calls (openmpi, mpich, mpi/LAM, ...)
- Can be used from FORTRAN, C, C++, Python, ...
- Can be used in addition to OpenMP
- Most common functions:
  - Initialization: *MPI_Init, MPI_Comm_size, MPI_Comm_rank*
  - Cleanup: *MPI_Finalize*
  - Basic data exchange: *MPI_Send, MPI_Recv* (plus some variants)
  - Check communication status: *MPI_Wait, MPI_Test*
  - Collective communication: *MPI_Bcast, MPI_Gather, MPI_Scatter*
  - Synchronization: *MPI_Barrier*
- In C,C++: wrapper script around the compiler (mpicc) to load transparently the appropriate libraries/include files

Toy example (each job prints its rank)

Listing 2: *hello.c*

```c
#include <stdio.h>
#include <mpi.h>

int  main(int argc,char *argv[]){
   int  rank,size;
   MPI_Init(&argc,&argv);                    //   Initialize   MPI
   MPI_Comm_size(MPI_COMM_WORLD,&size); // Number of CPUs
   MPI_Comm_rank(MPI_COMM_WORLD,&rank); // Rank in the pool of CPUs
   fprintf(stderr,"I  am node %d in a pool of %d nodes\n",rank,size);
   MPI_Finalize();                           //  Finalize  just  before exit
   return 0;
}
```

Listing 3: *hostfile.sh*

```
localhost slots=16
```

- Compile with: *mpicc -o hello hello.c*
- Run locally with: mpirun -np 10 –hostfile hostfile.sh ./hello

Communication domain

- *MPI_COMM_WORLD* is the set made of all the nodes
- One may define other “communication domains” (i.e., subsets)
- This is useful to restrict collective communications to a subgroup of the nodes

Basic communication

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int rank,size;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) {
    srand(123+rank);       // Safe because each job has a fully private memory space
    int random = rand();
    fprintf(stderr,"node %d generated number %d\n",rank,random);
    MPI_Send(&random,1,MPI_INT,0,0,MPI_COMM_WORLD); // Send to rank=0
  } else {
    int count = 1, tmp;
    while (count<size) {
      MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,NULL); // Receive
      fprintf(stderr,"Node 0 received number %d\n",tmp);
      count++;
    }}

  MPI_Finalize();
  return 0;
}
```

- Note: the receiver does not know which node sent the data

Basic communication, keeping track of sender

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int rank, size;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) {
    srand(123+rank);
    int random = rand();
    fprintf(stderr,"node %d generated number %d\n",rank,random);
    MPI_Send(&random,1,MPI_INT,0,0,MPI_COMM_WORLD);
  } else {
    int count = 1, tmp, sender;
    while (count<size) {
      MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
      sender = mpi_status.MPI_SOURCE;   // Use status variable to discover who is the sender
      fprintf(stderr,"Node 0 received number %d from node %d\n",tmp,sender);
      count++;
    }}

  MPI_Finalize();
  return 0;
}
```

MPI types

MPI Data type	C Data Type
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double

- MPI provides functions for defining additional types (e.g., a block out of a block representation of a matrix)

Source, Destination and Tag

- MPI_Recv(Buf,Size,Type,**Source**,**Tag**,MPI_COMM_WORLD,&status)
- MPI_Send(Buf,Size,Type,**Dest**,**Tag**,MPI_COMM_WORLD,&status)
- **Source** : specifies the origin
  Source=*MPI_ANY_SOURCE* : unspecified origin
- **Dest** : specifies the destination
- **Tag** : integer that labels a specific communication channel
  Must match between sender and receiver, unless the receiver uses *MPI_ANY_TAG*
- When using MPI_ANY_SOURCE and/or MPI_ANY_TAG, Source and Tag may be obtained afterwards from the *status* variable, as
  - status.MPI_SOURCE
  - status.MPI_TAG

Beware: MPI_Send, MPI_Recv are blocking functions

```c
#include <stdio.h>
#include <mpi.h>

// Each node passes to the next a value received from the previous one
// Will lock, because of a "chicken and egg problem"

int main(int argc,char *argv[]){
  int rank, size, tmp;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // Each job posts a Receive, then a Send
  MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
  fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  if (rank==0) tmp=-1;
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

Avoiding deadlocks by ordering Sends/Receives

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int rank, size, tmp;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) { // Nodes>0: post a (blocking) Receive
    MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
    fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  }

  if (rank==0) tmp=-1; // Everybody: Send "tmp" to next node
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  if (rank==0) { // Node 0: final Reveive
    MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
    fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  }

  MPI_Finalize();
  return 0;
}
```

Variants of MPI_Send

- *MPI_Send* : blocks until the array containing the data to be sent out can be reused safely (does not imply that the data has already reached its destination, since it could be buffered)
- This function has several variants
  - *MPI_Ssend* : synchronous send; blocks until a matching receive has been posted. Avoids buffering, but higher risk of locking
  - *MPI_Isend* : immediate send; non-blocking, but one cannot reuse the array safely. Should be used in conjunction with *MPI_Wait* or *MPI_Test* to check when it is safe to reuse the array
  - *MPI_Bsend* : buffered send; returns immediately, and the array can be reused immediately. Degrades performance

Variants of MPI_Recv

- *MPI_Recv* : blocks until the array in which the data to be received arrives is ready to be used
- This function has one variant
  - *MPI_Irecv* : immediate receive; non-blocking, but one cannot use the array yet. Should be used in conjunction with *MPI_Wait* or *MPI_Test* to check when it is safe to use the array
- Pros/cons of non-blocking communication:
  - allows to perform other communications or computations instead of waiting
  - can reduce latency by posting receives early
  - less chances of locking
  - BUT: one needs to test whether the operation has completed

Usage of MPI_Wait and MPI_Test

- *MPI_Wait* : blocking function

```
MPI_Request request;
MPI_Status status;
//
MPI_Irecv(Buf,Size,Type,Source,Tag,MPI_COMM_WORLD,&request);
//
//  do something useful (that does not use Buf)
//  now, assume that we need Buf
//
MPI_Wait(&request,&status); // blocks until receive is  completed
```

- *MPI_Test* : non-blocking polling function

```
MPI_Request request;
MPI_Status status;
int  done = FALSE;
//
MPI_Irecv(Buf,Size,Type,Source,Tag,MPI_COMM_WORLD,&request);
//
while (done==FALSE) {
   //  do something useful (that does not use Buf)
   MPI_Test(&request,&done,&status); // returns immediately
 }
```

- *MPI_Waitall* : waits for a set of communications

Simple example (with multiple pending requests)

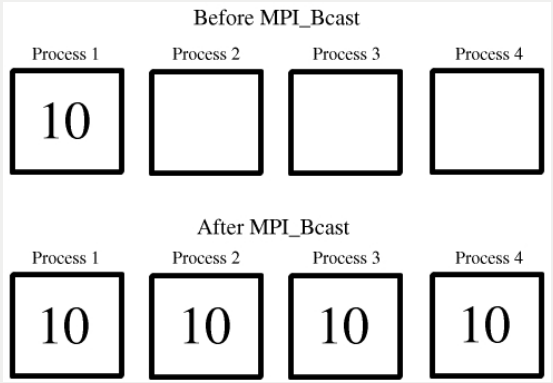
```c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[]){
  int size, rank, next, prev, buf[2], tag1=1, tag2=2;
  MPI_Request requests[4];
  MPI_Status status[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = (rank-1+size)%size;
  next = (rank+1)%size;
  // Initiate a bunch of communications (they all return immediately)
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &requests[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &requests[1]);
  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &requests[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &requests[3]);
  //
  // do something else, then wait for completion of all communications
  //
  MPI_Waitall(4, requests, status);
  printf("Task %d communicated with tasks %d & %d\n",rank,prev,next);

  MPI_Finalize();
}
```

Avoiding deadlocks with asynchronous communication

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc,char *argv[]){
  int rank, size, tmp, final;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // Node 0 posts an early non-blocking receive, others post a blocking one
  if (rank==0) {MPI_Irecv(&final,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&request);tmp = -1;}
  else {MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);}

  // Everybody: pass value to the next node
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  if (rank==0) MPI_Wait(&request,&status); // Node 0 must check that Receive has completed
  fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,status.MPI_SOURCE);

  MPI_Finalize();
  return 0;
}
```

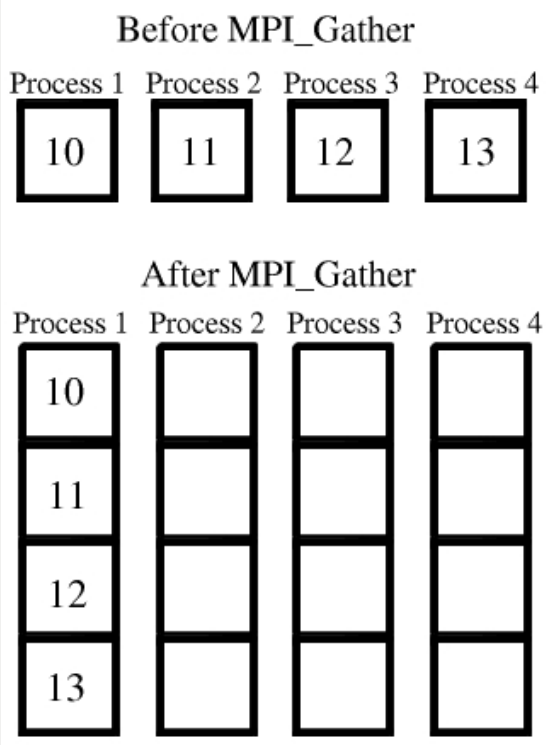
Collective communication: Broadcast

- MPI_Bcast(Buf,N,Type,0,MPI_COMM_WORLD)



Collective communication: Gather

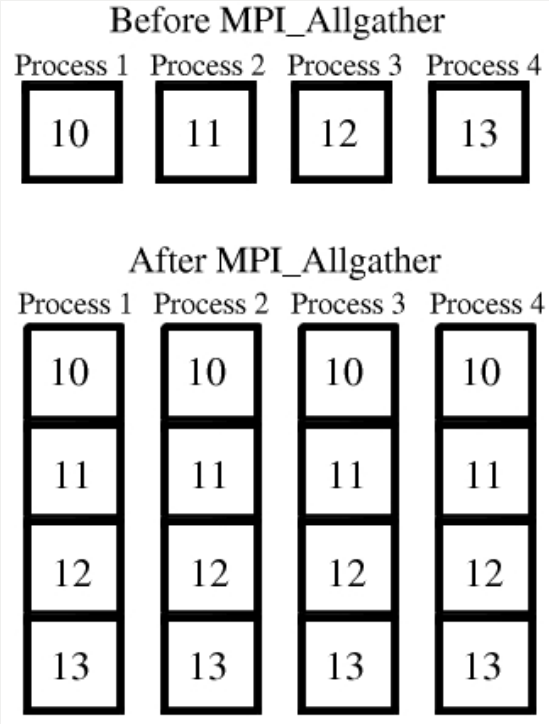
- MPI_Gather(Source,N,Type,Dest,N,Type,0,MPI_COMM_WORLD)



- Variant: MPI_Gatherv: gather variable size chunks of data, and place them at variable offsets in the Dest array

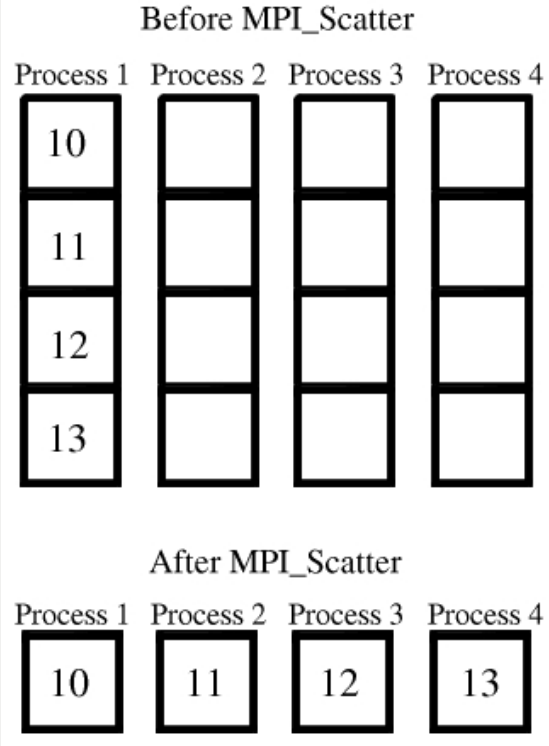
Collective communication: AllGather

- MPI_Allgather(Source,N,Type,Dest,N,Type,MPI_COMM_WORLD)



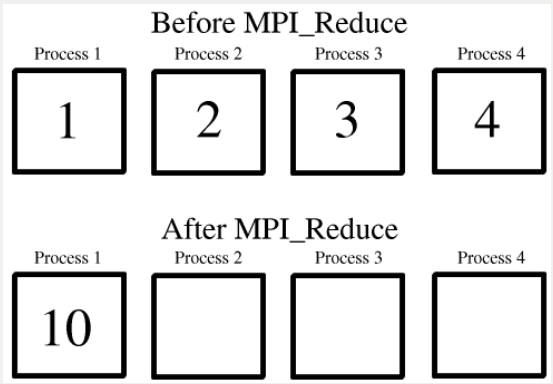
Collective communication: Scatter

- MPI_Scatter(Source,N,Type,Dest,N,Type,o,MPI_COMM_WORLD)



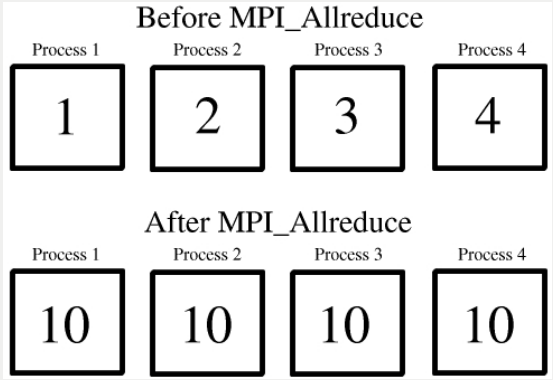
Global reductions: Reduce

- MPI_Reduce(Source,Dest,N,Type,Op,o,MPI_COMM_WORLD)
- Op = MPI_SUM, MPI_PROD, etc...



Global reductions: AllReduce

- MPI_Allreduce(Source,Dest,N,Type,Op,MPI_COMM_WORLD)



Hybrid: MPI+OpenMP

- Superimposing MPI and OpenMP parallelization poses no problem
- In general, each MPI job should fork in a number of threads equal to the number of physical cores on one node
- Various hybrid scenarios are possible

Easiest: independent computations on each node

- Each MPI job performs a (lengthy) computation, independently of the other jobs
- These computations are parallelized using OpenMP
- Output is sent back to MASTER node for final processing (e.g., averaging, saving in a file, etc...). For this, the MASTER node forks a special thread that collects this output
- Input data:
  - Monte-Carlo: input is a random configuration, generated locally on each node (make sure each MPI job uses its own RNG seed)
  - Alternative: the MASTER node sends tasks taken from a list to each node as soon as a result returns

Threads on MASTER node

- Receiving/processing computed results is a very light task compared to the computations $\Rightarrow$ the MASTER node can also perform computations without significant penalty

- Call *omp_set_nested(1)* on all nodes

- Create two parallel sections:
  - First section: executed only by MASTER (empty on other nodes)
    Post MPI_Recv to receive results, post-process results, MPI_Send to send tasks, etc...
  - Second section: executed by everybody
    performs actual computations $\Rightarrow$ further fork in $\mathrm{N}_{\mathrm{cores}}$ threads

Harder: Two-level slicing of the computation domain

- Large array to be processed (e.g., evolved in time)
- Divide the array in $\mathrm{N}_{\mathrm{cores}} \times \mathrm{N}_{\mathrm{threads}}$ slices, and assign a slice to each thread

- Case 1: evolution is "almost" local, i.e. depends at most on a few neighboring sites (e.g., discretization of a Laplacian)

  At the beginning of each timestep, each thread must be given a copy of the layers just before and just after its slice

- Case 2: evolution is completely non-local (update of a point i depends on all other points)

  2.a. Array is small enough: each node can have its own copy of the full array (it must be refreshed at the beginning of each timestep)

  2.b. Array is too large: communications will probably make parallelization very inefficient

Case study: deterministic Boltzmann solver

Boltzmann Equation

- Two-body elastic interactions
- Spatially homogenous
- Isotropic particle distribution
- Scattering amplitude may be momentum dependent

$$\partial_t f_1 = \frac{1}{E_1} \int_{\mathbf{p}_{2,3,4}} \left|M(1,2,3,4)\right|^2 \left[f_3 f_4 (1+f_1)(1+f_2) - f_1 f_2 (1+f_3)(1+f_4)\right]$$

- Collision integral reduces to 4-dim integral thanks to momentum conservation and isotropy

Sketch of the Algorithm

- Discretize $|\mathbf{p}| \quad \rightarrow \quad p_i \quad (0 \leq i < N)$

- At each time-step:

  1. For each i, compute Δf_i (given by a 4-dim integral $\Rightarrow$ slow)
  2. Check if $f_i + \Delta f_i \geq 0$ (for all i)
     If FALSE, re-run the previous loop with a smaller timestep
  3. Do $f_i + \Delta f_i \rightarrow f_i$ and return to step 1

- Note: cost of computing Δf_i not uniform (50% variation)
  Thus, we expect that the parallelization of the "big loop" will not be perfectly efficient (the computation time will align to that of the slowest bins)

Sequential version (sketch of the relevant bit of code)

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
for (i=0;i<N;i++) df[i] = dt*C(i,f); // depends on f[k] with k!=i; computation of C(i,f) very slow

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

- Large fraction of the time spent in the first loop
- For a reasonable grid size: 30 minutes/timestep
- Typical evolution: 2000 timesteps (1000 hours, or 42 days...)

OpenMP version

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
#pragma omp parallel for num_threads(NT) schedule(dynamic) // <——— ONLY ADDITION
for (i=0;i<N;i++) df[i] = dt*C(i,f);

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

- Define *NT* (number of threads) in the scope of the function
- Add *-fopenmp* to compiler options
- Better use of threads if *NT* divides *N*

MPI+OpenMP version

```
// Core of the function that evolves f[i]

int n = N/size, imin = rank*n, imax = (rank+1)*n;                    // <-- workload of each mode
double *df=NULL, *local_df = (double *)malloc(n*sizeof(double)); // <-- Node-local storage
if (rank==0) df = (double *)malloc(N*sizeof(double));                // <-- MASTER needs a buffer for df[i]

#pragma omp parallel for num_threads(NT) schedule(dynamic)
for (i=imin;i<imax;i++) local_df[i] = dt*C(i,f);

MPI_Gather(local_df,N1,MPI_DOUBLE,df,N1,MPI_DOUBLE,0,MPI_COMM_WORLD); // <-- Gather partial results on MASTER
if (rank==0) {status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;}
MPI_Bcast(&status,1,MPI_INT,0,MPI_COMM_WORLD);                       // <-- Broadcast test result
if (status==1) return 1;

if{rank==0}{for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);}
MPI_Bcast(f,N,MPI_DOUBLE,0,MPI_COMM_WORLD);                          // <-- Broadcast updated f[i]
free(local_df);
return 0;
```

- *size* and *rank* needed in the update function
- Add *#include <mpi.h>* in the header
- Output to files done by MASTER node only

Timings

- Running time:
  - Sequential: 1830 seconds/timestep
  - OpenMP (16 cores): 154 seconds/timestep
    ($\times 12$ speedup, 75% efficiency)
  - MPI+OpenMP (32 nodes $\times$ 16 cores): 6 seconds/timestep
    ($\times 306$ speedup, 60% efficiency)
    Computation time reduced from 1000 hours to 3 hours 16 minutes

- Coding time:
  - Sequential: one week (about 600 lines of code)
  - +OpenMP: +one minute (+1 line)
  - +MPI: +one hour (+10 lines)