Programming for supercomputers

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
http://scemama.mooo.com
Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Note

Slides can be downloaded here:

- http://irpf90.ups-tlse.fr/files/lttc17_supercomputing.pdf
- http://irpf90.ups-tlse.fr/files/lttc17_parallelism.pdf

- If the program is not parallel, a supercomputer will be not much better than a desktop computer
- Many different architectures -> Portability of the code
- Fortunately, *all* architectures are well suited to **dense linear algebra**.
- Unfortunately, not all scientific problems are well suited to dense linear algebra.

Guidelines

1. Express your problem in a *parallelizable* algorithm
2. Use dense linear algebra where you can
3. Use sparse linear algebra if the problem is really sparse (< 1% non-zeros)
4. Avoid file I/O
5. Use standardized Application Programming Interfaces (APIs) for hot spots

Most common APIs used in High Performance Computing:

Linear Algebra

BLAS, LAPACK, ScaLAPACK

Parallelism

MPI, OpenMP, OpenACC, StarPU, GPI

Outline

I. Linear Algebra APIs
II. Parallel computing
III. OpenMP

I. Linear Algebra APIs

BLAS

- Basic Linear Algebra Subprograms
- Simple matrix and vector operations
- Support for C and Fortran.

Libraries:

- MKL (Intel)
- ESSL (OpenPower)
- CuBLAS (Nvidia)
- ATLAS, OpenBLAS, ... (multi-architecture)

Quick reference : http://www.netlib.org/blas/blasqr.pdf

Computable routine names

Ex : DGEMV = **D** ouble precision **GE** neral **M** atrix **V** ector

Prefixes

S: REAL, **C**: COMPLEX, **D**: DOUBLE PRECISION, **Z**: COMPLEX*16

Matrix types

GE: GEneral, **GB**: General Band, **SY**: SYmmetric, **SB**: Symmetric Band, **SP**: Symmetric Packed **HE**: HErmitian, **HB**: Hermitian Band, **HP**: Hermitian Packed, **TR**: TRiangular, **TB**: Triangular Band, **TP**: Triangular Packed

Level 1

Vector operations (*N* data, *N* operations)

Level 2

Matrix Vector operations (N^2 data, N^2 operations)

Level 3

Matrix Matrix operations (N^2 data, N^3 operations)

Level 1

- DROTG - setup Givens rotation
- DROTMG - setup modified Givens rotation
- DROT - apply Givens rotation
- DROTM - apply modified Givens rotation
- DSWAP - swap x and y
- DSCAL - x = a*x
- DCOPY - copy x into y
- DAXPY - y = a*x + y
- DDOT - dot product
- DSDOT - dot product with extended precision accumulation
- DNRM2 - Euclidean norm
- DZNRM2 - Euclidean norm
- DASUM - sum of absolute values
- IDAMAX - index of max abs value

Level 2

- DGEMV - matrix vector multiply
- DGBMV - banded matrix vector multiply
- DSYMV - symmetric matrix vector multiply
- DSBMV - symmetric banded matrix vector multiply
- DSPMV - symmetric packed matrix vector multiply
- DTRMV - triangular matrix vector multiply
- DTBMV - triangular banded matrix vector multiply
- DTPMV - triangular packed matrix vector multiply
- DTRSV - solving triangular matrix problems
- DTBSV - solving triangular banded matrix problems
- DTPSV - solving triangular packed matrix problems
- DGER - performs the rank 1 operation A := alpha*x*y' + A
- DSYR - performs the symmetric rank 1 operation A := alpha*x*x' + A
- DSPR - symmetric packed rank 1 operation A := alpha*x*x' + A
- DSYR2 - performs the symmetric rank 2 operation, A := alpha*x*y' + alpha*y*x' + A
- DSPR2 - performs the symmetric packed rank 2 operation, A := alpha*x*y' + alpha*y*x' + A

Level 3

- DGEMM - matrix matrix multiply
- DSYMM - symmetric matrix matrix multiply
- DSYRK - symmetric rank-k update to a matrix
- DSYR2K - symmetric rank-2k update to a matrix
- DTRMM - triangular matrix matrix multiply
- DTRSM - solving triangular matrix with multiple right hand sides

Documentation

```
subroutine DGEMM (character           TRANSA, TRANSB,
                  integer             M, N, K,
                  double precision    ALPHA,
                  double precision    A(LDA,*),
                  integer             LDA,
                  double precision    B(LDB,*),
                  integer             LDB,
                  double precision    BETA,
                  double precision    C(LDC,*),
                  integer             LDC )

! C := alpha*op( A )*op( B ) + beta*C,
! where  op( X ) is one of
!   op( X ) = X   or   op( X ) = X**T,
! alpha and beta are scalars, and A, B and C are matrices,
! with op( A ) an m by k matrix, op( B ) a k by n matrix
! and C an m by n matrix.
```

Example

```
! C := alpha*op( A )*op( B ) + beta*C,

double precision :: A(m,k), B(k,n), C(m,n)

call DGEMM('N','N', m, n, k,          &
  1.d0, A, size(A,1), B, size(B,1), &
  0.d0, C, size(C,1) )
```

Question

Why pass `size(_,1)`? Isn't it redundant with `m,n,k` ?

Answer

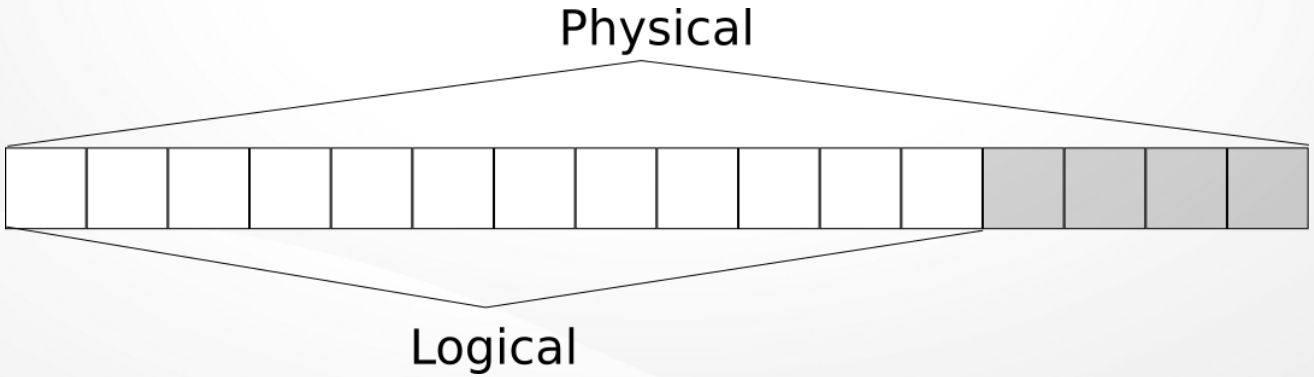
No! => Physical / Logical storage

Storage of arrays

1-dimensional

```
double precision :: A(nmax)
do i=1,n
  A(i) = ...
end do
```

- Physical array : Allocated memory, size : nmax
- Logical array : Useful part of the array, size : n with n ≤ nmax

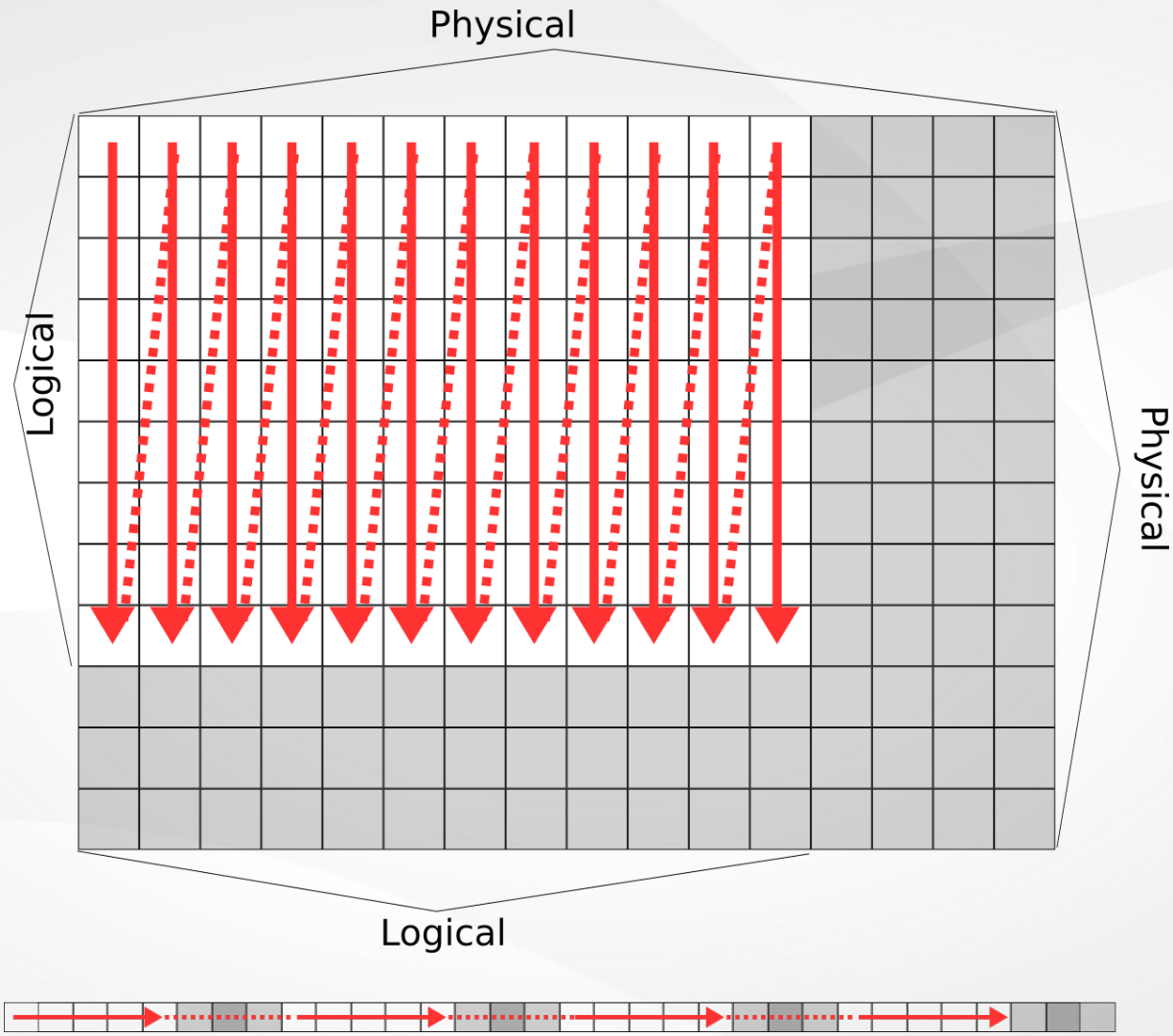


2-dimensional

```
double precision :: A(nmax,mmax)
do j=1,m
  do i=1,n
    A(i,j) = ...
  end do
end do
```

Equivalent in memory to

```
double precision :: A(nmax*mmax)
do j=1,m
  do i=1,n
    A( (j-1)*nmax + i ) = ...
  end do
end do
```


Physical
Logical
Physical
Logical

For efficiency : Inner-most loop should be the 1st index

Not like this:

```
do i=1,n
  do j=1,m
    A(i,j) = ...
  end do
end do
```

Like this:

```
do j=1,m
  do i=1,n
    A(i,j) = ...
  end do
end do
```

LAPACK

- Linear Algebra PACKage
- Support for C and Fortran.
- Combines efficient BLAS routines in high-level algorithms (LU, Cholesky, QR, SVD, Diagonalization, ...)
- Naming convention similar to BLAS

Libraries:

- MKL (Intel)
- ESSL (OpenPower)
- MAGMA (Nvidia)
- PLASMA (multi-core)
- ScaLAPACK (Distributed)

http://www.netlib.org/lapack

Example : Diagonalize a symmetric matrix

```
subroutine dsyev (character          JOBZ,
                  character          UPLO,
                  integer            N,
                  double precision, dimension( lda, * ) A,
                  integer            LDA,
                  double precision, dimension( * )      W,
                  double precision, dimension( * )      WORK,
                  integer            LWORK,
                  integer            INFO )

!  DSYEV computes all eigenvalues and, optionally, eigenvectors
!  real symmetric matrix A.
```

Exercise: Compute the determinant of a matrix

Hints :

- LU Factorization : A = P L U

- Product of the diagonal elements of U
- Determine number of permutations in P

Solution

```
call DGETRF(na, na, A, size(A,1), ipiv, info)
if (info /= 0) stop 'Error in determinant'
det=1.d0
j=0
do i=1,na
  if (ipiv(i) /= i) j=j+1
  det=det*A(i,i)
end do
if (iand(j,1) /= 0)  then
  det_l=-det_l
end if
```

III. Parallel computing

Experiment : The Human Parallel Machine

- Sort a deck of cards
- Do it as fast as you can

- I did it *alone* in 5'25" (325 seconds)
- Each one of you is a "human compute node"
- How fast can all of you sort the same deck?

Disappointing result !

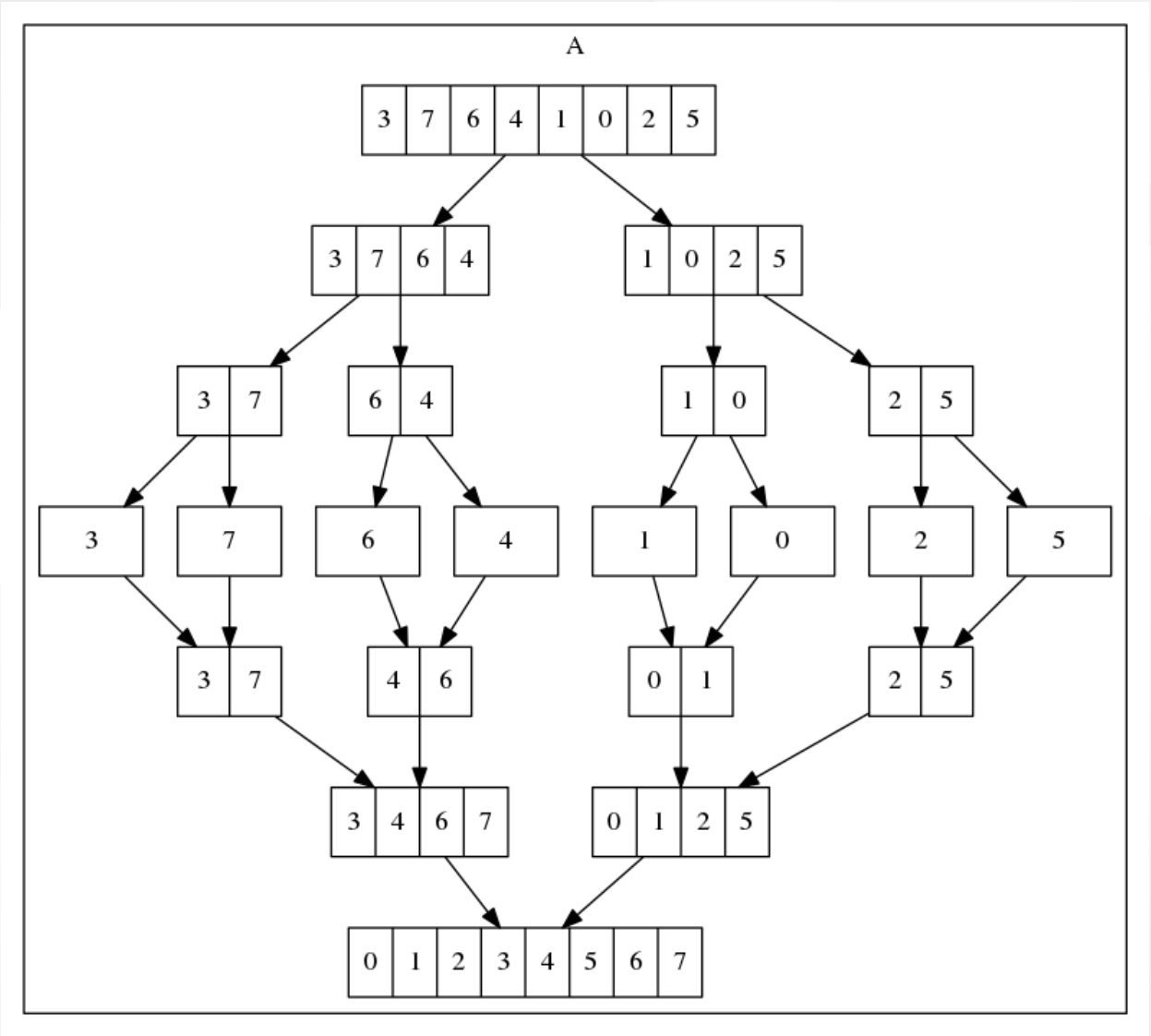
- You were many more people than me, but you didn't go many times faster !

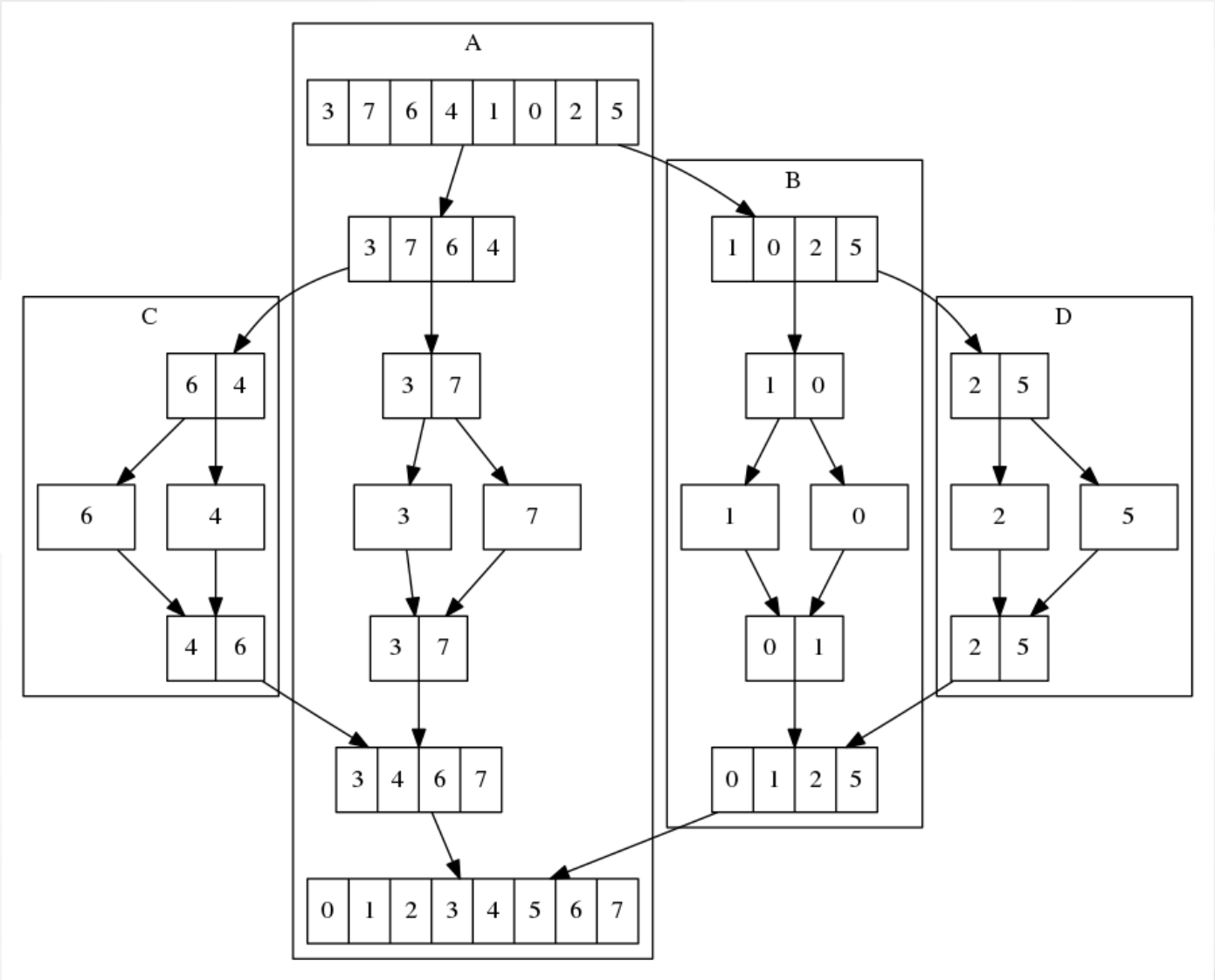
=> Try the *Merge sort*

Recursive agorithm :

```
sort( x(1:1) ) = x(1:1)

sort( A(1:n) ) =  merge (
    sort( A(1:n/2) ), sort( A(n/2+1:n) ) )
```


A
3 7 6 4 1 0 2 5
3 7 6 4
1 0 2 5
3 7
6 4
1 0
2 5
3
7
6
4
1
0
2
5
3 7
4 6
0 1
2 5
3 4 6 7
0 1 2 5
0 1 2 3 4 5 6 7


A
3 7 6 4 1 0 2 5
3 7 6 4
1 0 2 5
B
C
D
6 4
3 7
1 0
2 5
6
4
3
7
1
0
2
5
4 6
3 7
0 1
2 5
3 4 6 7
0 1 2 5
0 1 2 3 4 5 6 7

Better, but still not perfect

- Moving the cards around the room takes time (communication)
- Sorting the sub-piles is super-fast (computation)

=> Algorithm is bounded by communication : Difficult to scale

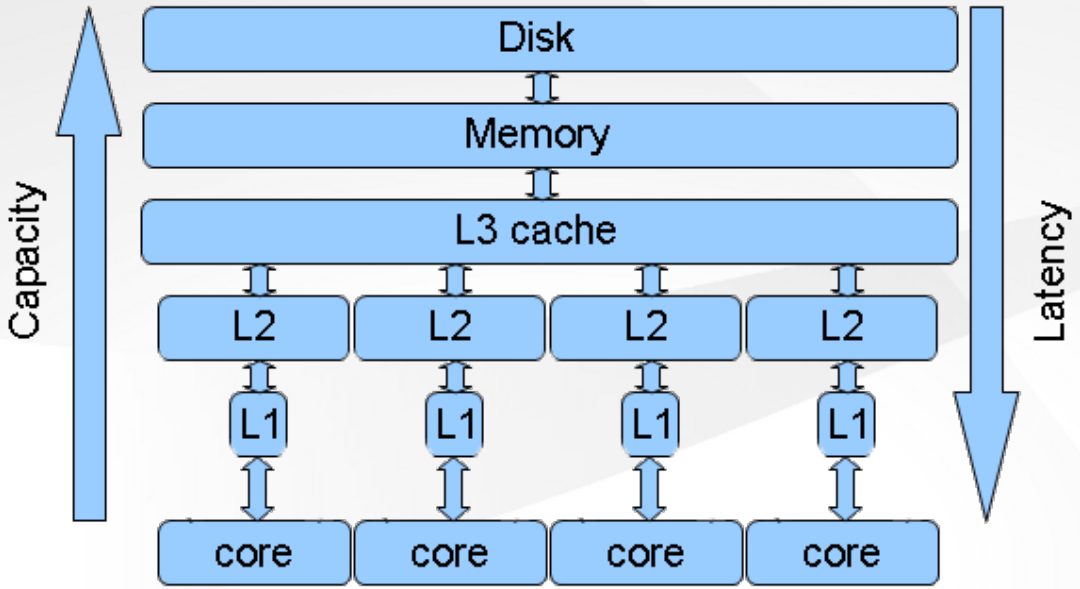
If the assignment was:

- Consider the function $f(a,b,x) = a.x^5 - b.x^3/a$
- Each card has a value for *a*, *b* and *x*
- Evaluate *f(a,b,x)* for each card and write the result on the card
- Sort the results

Same communication pattern, but more computational effort => better scaling.

Important

Difficulty is data movement (communication), not computation



Link with parallelism:

- You can imagine a parallel machine as a set of people working together
- Moving data is much more time consuming than computing
- Moving data far away takes more time than moving it close
- Communication times are usually not uniform
- Obtaining an ideal speedup is usually *very* difficult

Exercise: Implement the merge sort algorithm in Fortran

```
recursive subroutine sort(A,n)
  implicit none
  integer                , intent(in   ) :: n
  double precision, intent(inout) :: A(n)
! ...
! Condition to stop recursion
  if (...) then
    return
  else
!   ...
    call sort(...)
!   ...
    call sort(...)
!   ...
  end if
end
```

Solution

```
recursive subroutine sort(A,n)
  implicit none
  integer                , intent(in   ) :: n
  double precision, intent(inout) :: A(n)
  integer                                :: iA, iB, iC, m
  double precision                       :: B(n/2+1), C(n-n/2+1)
  if (n == 1) then
    return
  end if
  m=n/2
! 1) Divide
  B(1:m) = A(1:m)
  call sort(B,m)
  C(1:n-m) = A(m+1:n)
  call sort(C,n-m)
! ...
```

```
! 2) Merge
  B(m+1) = huge(1.d0) ; C(n-m+1) = huge(1.d0)
  iB=1 ; iC=1
  do iA=1,n
    if (B(iB) < C(iC)) then
      A(iA) = B(iB) ; iB = iB+1
    else
      A(iA) = C(iC) ; iC = iC+1
    end if
  end do

end
```

Parallel computing

When solving a problem, multiple calculations can be carried out concurrently. If multiple computing hardware is used, concurrent computing is called **parallel computing**.

Many levels of parallelism:

- Distributed, Loosely-coupled : Computing grids (shell scripts)
- Distributed, Tightly-coupled : Supercomputers (MPI,...)
- Shared memory : OpenMP, threads
- Hybrid: wth accelerators like GPUs, FPGAs, Xeon Phi, etc
- Socket-level : Shared cache
- Instruction-level : superscalar processors
- Bit-level : vectorization

All levels of parallelism can be exploited in the same code, but every problem is not parallelizable at all levels.

Data access is *slow* with respect to computation:

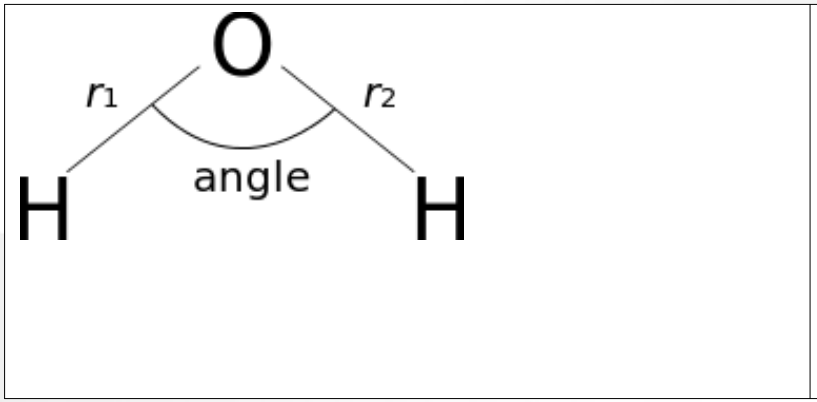
Operation	Latency (ns)	Scaled
Int ADD	0.3	1 s
FP ADD	0.9	3 s
FP MUL	1.5	5 s
L1 cache	1.2	4 s
L2 cache	3.5	12 s
L3 cache	13	43 s
RAM	79	4 min 23 s
Infiniband	1 200	1 hr 7 min
Ethernet	50 000	1 day 22 hr
Disk (SSD)	50 000	1 day 22 hr
Disk (15k)	2 000 000	23 days 4 hr

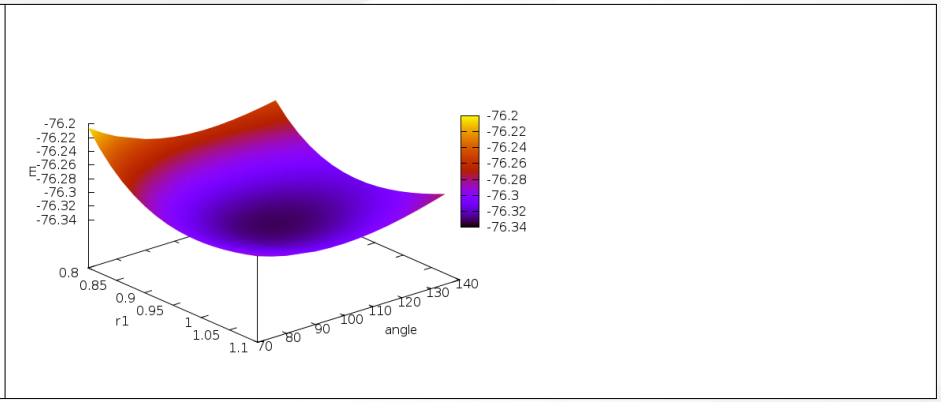
Arithmetic intensity : Flops/memory access

Loosely coupled parallelism

- Well adapted to **all** parallel machines
- 99.9999% parallel speedup can be reached

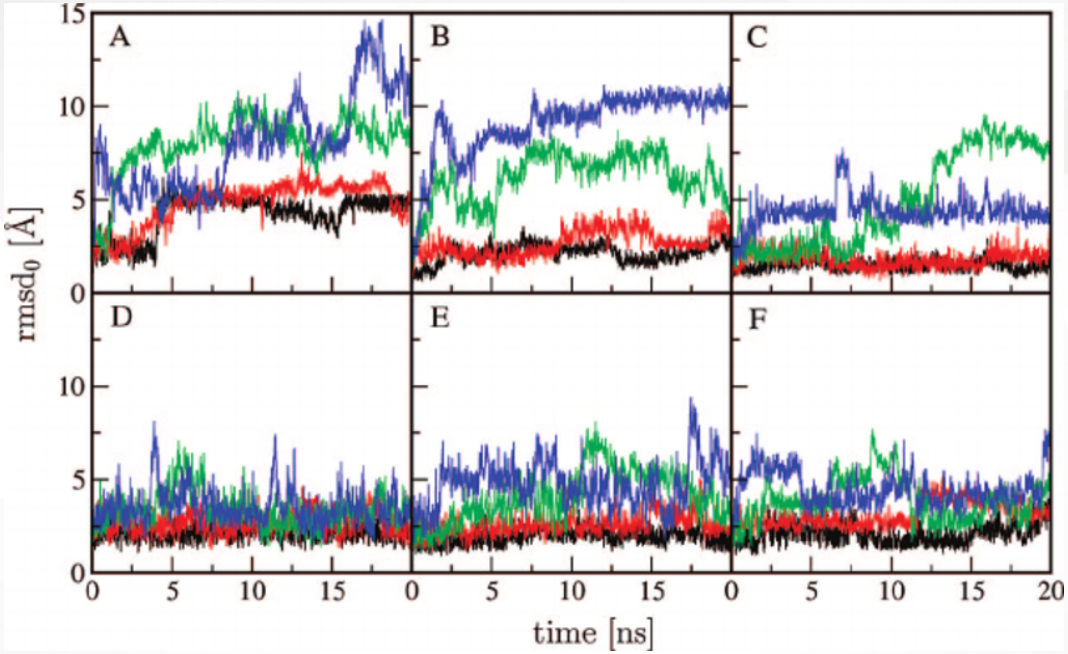
Example : Compute the potential energy surface of the Water molecule.





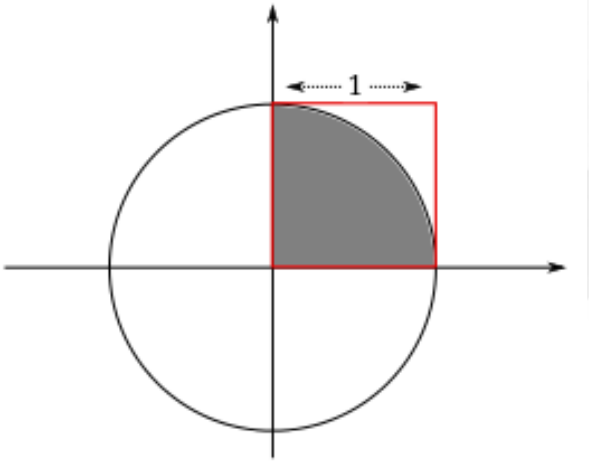
- 3 coordinates : r1, r2, angle
- 50 values for each coordinate -> 125 000 points
- Each point is independent : 125 000 independent runs

Example : Molecular dynamics



- Multiple independent trajectories
- Different initial conditions
- Post-processing to compute the quantities of interest

Example : Compute the value of $\pi$with a Monte Carlo algorithm.

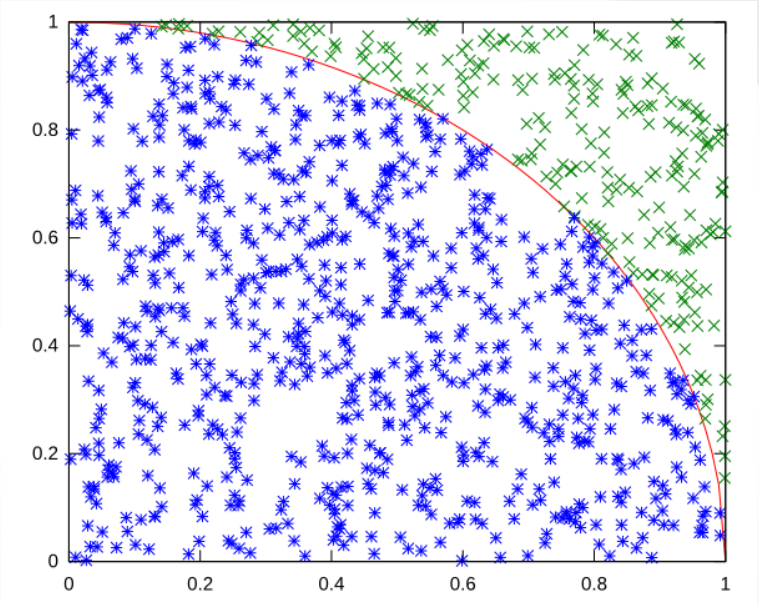


- The surface of the circle is πr^2=> For a unit circle, the surface is π
- The function in the red square is $y = \sqrt{1 - x^2}$ (the circle is $\sqrt{x^2 + y^2} = 1$)
- The surface in grey corresponds to

$$\int_0^1 \sqrt{1 - x^2}\, dx = \pi/4$$

To compute this integral, a Monte Carlo algorithm can be used:

- Points (x, y) are drawn randomly in the unit square.
- Count how many times the points are inside the circle
- The ratio (inside)/(inside+outside) is $\pi/4$.

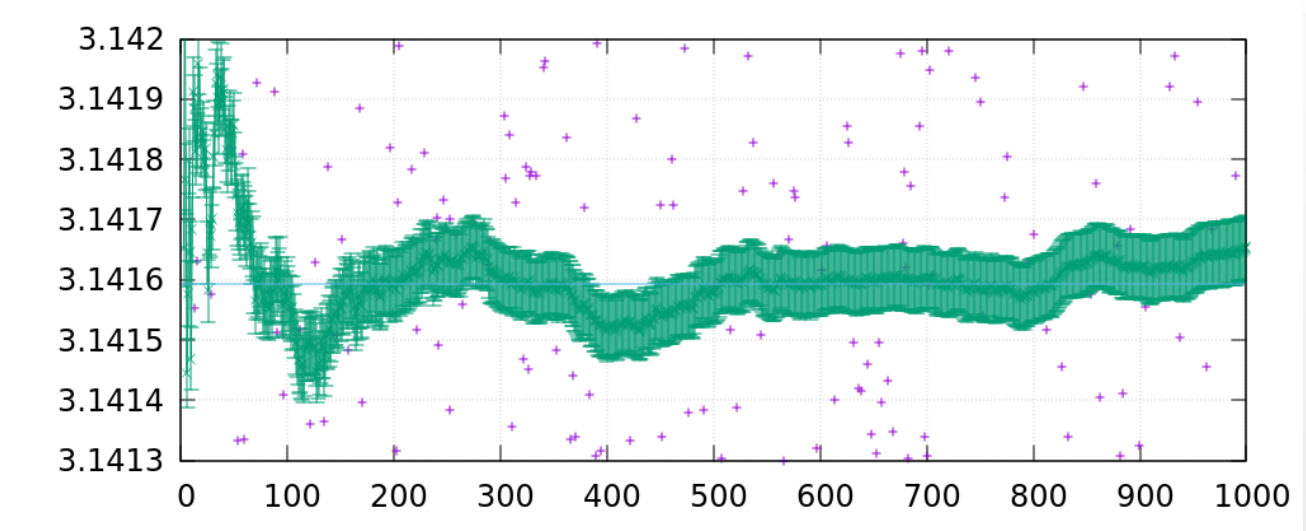


Optimal algorithm:

- Each CPU core computes the its own average
- All M results obtained on different CPU cores are independent, so they are Gaussian-distributed random variables (central-limit theorem) with average $\langle X \rangle$ and the variance σ^2

$$\pi \sim \langle X \rangle = \frac{1}{M} \sum_{i=1}^{M} X_i \qquad \sigma^2 = \frac{1}{M-1} \sum_{i=1}^{M} (X_i - \langle X \rangle)^2$$

to compute the statistical error as $\delta\pi = \sigma / \sqrt{M}$



Exercise: Write a Fortran program that computes $\pi$with the Monte Carlo algorithm.

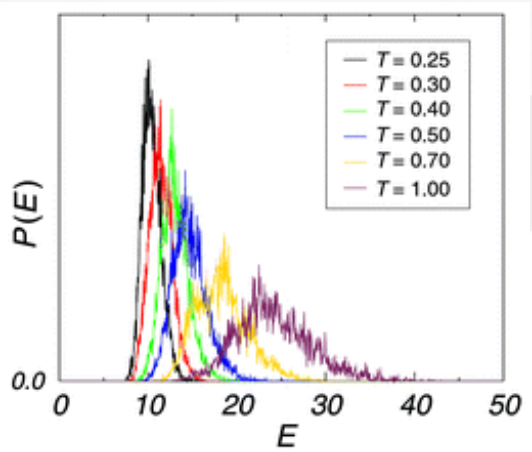
```
program pi_monteCarlo()
  implicit none
  double precision :: x,y, pi
! ...
! Initialize random number generator
  call random_seed()
! ...
  call random_number(x)
  call random_number(y)
! ...
  print *,  pi
end
```

Solution

```
subroutine pi_monteCarlo()
  implicit none
  double precision :: x, y, pi
  integer :: i, nmax = 10**8
  pi = 0.d0
  call random_seed()
  do i=1, nmax
    call random_number(x)
    call random_number(y)
    if (x*x + y*y <= 1.d0) then
      pi = pi+1.d0
    end if
  end do
  print *,  4.d0*pi/dble(nmax)
end
```

Tightly coupled parallelism

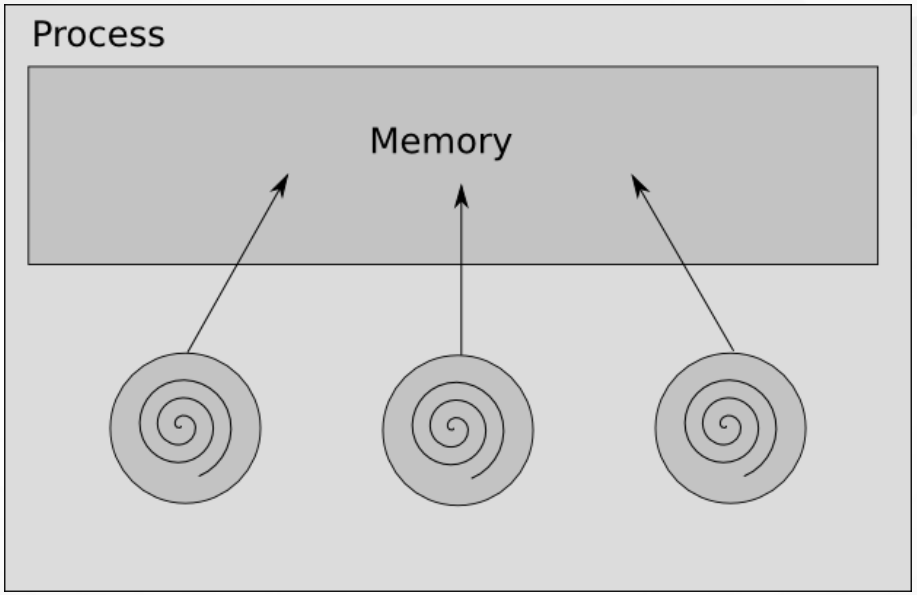
Example : Parallel tempering Molecular dynamics



- Multiple trajectories, each one at a specific temperature
- Different initial conditions
- Periodically exchange the temperatures between tajectories => coupling
- Coupling at every *N* steps => Need for fast communication
- MPI for distributed programming (see J. Cuny's session later in the week)

II. OpenMP

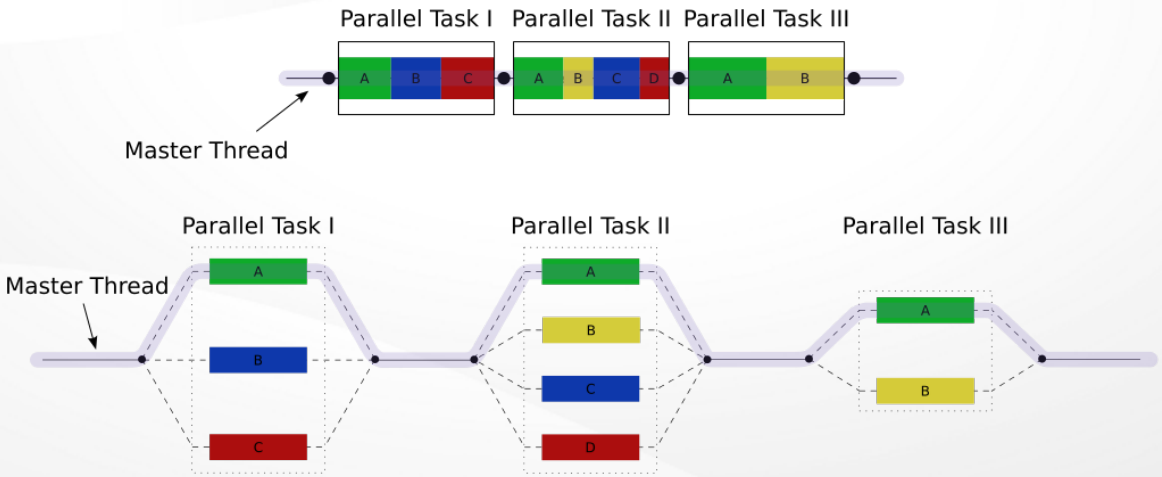
Very tighltly coupled parallelism.



- Low latency network latency : ~1.2 microsecond
- Random memory access : ~100 nanoseconds

OpenMP is an extension of programming languages that enable the use of multi-threading to parallelize the code using directives given as comments. The *same* source code can be compiled with/without OpenMP.

```
!$OMP PARALLEL DO DEFAULT(SHARED)  PRIVATE(i)
do i=1,n
  A(i) = B(i) + C(i)
end do
!$OMP END PARALLEL DO
```



By Wikipedia user A1 - w:en:File:Fork_join.svg, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=32004077

- `!$OMP PARALLEL` starts a new multi-threaded section. Everything inside this block is executed by *all* the threads
- `!$OMP DO` tells the compiler to split the loop among the different threads (by changing the loop boundaries for instance)
- `!$OMP END DO` marks the end of the parallel loop. It contains an implicit synchronization. After this line, all the threads have finished executing the loop.
- `!$OMP END PARALLEL` marks the end of the parallel section. Contains also an implicit barrier.
- `DEFAULT(SHARED)` : all the variables (A,B,C) are in shared memory by default
- `PRIVATE(i)` : the variable i is private to every thread

Other important directives:

- `!$OMP CRITICAL ... !$OMP END CRITICAL` : all the statements in this block are protected by a lock
- `!$OMP TASK ... !$OMP END TASK` : define a new task to execute

- `!$OMP BARRIER` : synchronization barrier
- `!$OMP SINGLE ... !$OMP END SINGLE` : all the statements in this block are executed by a single thread
- `!$OMP MASTER ... !$OMP END MASTER` : all the statements in this block are executed by the master thread
- `omp_get_thread_num()` : returns the ID of the current running thread
- `omp_get_num_threads()` : returns the total number of running threads
- `OMP_NUM_THREADS` : Environment variable (shell) that fixes the number of threads to run

Important

- Multiple threads *can read* at the same address
- Multiple threads **must not write** at the same address

Example : Matrix product

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```
do j=1,N
  do i=1,N
    C(i,j) = 0.d0
    do k=1,N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end do
  end do
end do
```

Parallelization in 2 minutes:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```
!$OMP PARALLEL DO DEFAUT(SHARED) PRIVATE(i,j,k)
do j=1,N
  do i=1,N
    C(i,j) = 0.d0
    do k=1,N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end do
  end do
end do
!$OMP END PARALLEL DO
```

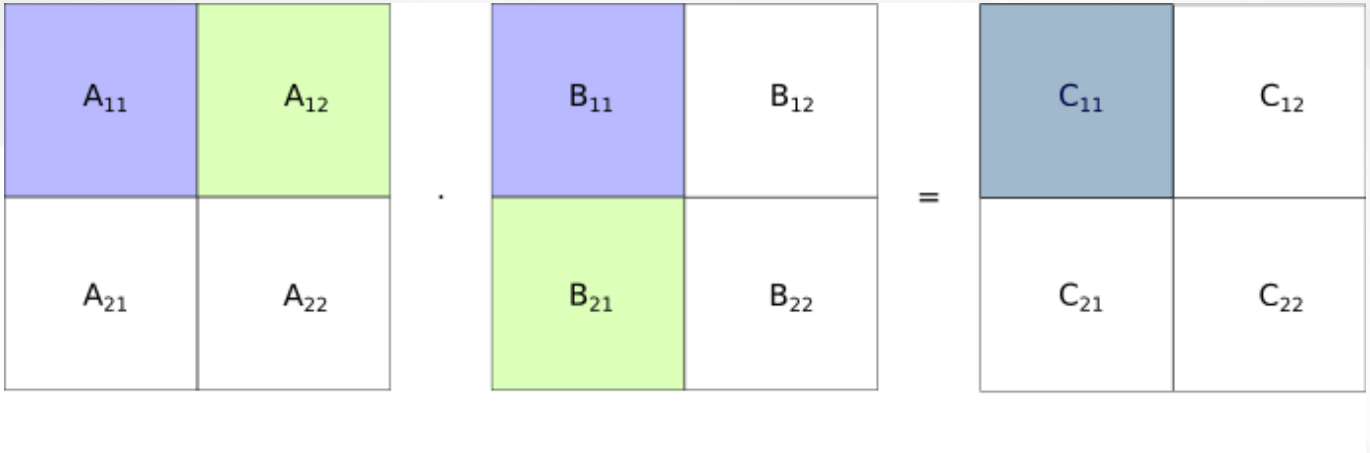
Improvement:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

```
!$OMP PARALLEL DEFAUT(SHARED) PRIVATE(i,j,k)
!$OMP DO COLLAPSE(2)
do j=1,N
  do i=1,N
    C(i,j) = 0.d0
    do k=1,N
      C(i,j) = C(i,j) + A_transposed(k,i) * B(k,j)
    end do  ! better memory access --^
  end do
end do
!$OMP END DO
!$OMP END PARALLEL
```

Divide and Conquer algorithm

The final matrix can be split, such that each CPU core builds part of it.



$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

The large *N* x *N* matrix product can be performed by doing 8 smaller matrix products of size *N*/2 x *N*/2, that can be done simultaneously by 8 CPUs.

```
step=sze/2
!$OMP DO COLLAPSE(2)
do i1=1,sze,step
  do j2=1,sze,step
    istart(1) = i1
    jstart(2) = j2
    iend(1) = istart(1)+step-1
    jend(2) = jstart(2)+step-1
    do j1=1,sze,step
      jstart(1) = j1
      istart(2) = j1
      jend(1) = jstart(1)+step-1
```

```
      iend(2) = istart(2)+step-1

      ! Compute the submatrix product
      call dgemm('N','N',                          &
          1+iend(1)-istart(1),                     &
          1+jend(1)-jstart(1),                     &
          1+jend(2)-jstart(2),                     &
          1.d0, A(istart(1),jstart(1)),sze,        &
          B(istart(2),jstart(2)),sze,              &
          1.d0, C(istart(1),jstart(2)),sze )
    enddo
  enddo
enddo
!$OMP END DO
```

Recursive variant:

```
recursive subroutine divideAndConquer(A,B,C,sze,ie1,je2)

if ( (ie1 < 200).and.(je2 < 200) ) then
  call DGEMM
else

  !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
  call divideAndConquer( & !  +-------+   +---+---+   +---+---+
      A(1,1),              & !  |   X   |   |   |   |   | X |   |
      B(1,1),              & !  +-------+ . + X |   + = +---+---+
      C(1,1),              & !  |       |   |   |   |   |   |   |
      sze,                 & !  +-------+   +---+---+   +---+---+
      ie1/2,               & !      A           B           C
      je2/2)

  !$OMP END TASK
```

```
!$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
call divideAndConquer( & !  +-------+   +---+---+   +---+---+
    A(1,1),              & !  |   X   |   |   |   |   |   | X |
    B(1,1+je2/2),        & !  +-------+ . |   | X | = +---+---+
    C(1,1+je2/2),        & !  |       |   |   |   |   |   |   |
    sze,                 & !  +-------+   +---+---+   +---+---+
    ie1/2,               & !      A           B           C
    je2-(je2/2))
!$OMP END TASK

!$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
call divideAndConquer( & !  +-------+   +---+---+   +---+---+
    A(1+ie1/2,1),        & !  |       |   |   |   |   |   |   |
    B(1,1),              & !  +-------+ . | X |   | = +---+---+
    C(1+ie1/2,1),        & !  |   X   |   |   |   |   | X |   |
    sze,                 & !  +-------+   +---+---+   +---+---+
    ie1-(ie1/2),         & !      A           B           C
    je2/2)
```

```
  !$OMP END TASK

  !$OMP TASK SHARED(A,B,C,sze) FIRSTPRIVATE(ie1,je2)
  call divideAndConquer( & !  +-------+   +---+---+   +---+---+
      A(1+ie1/2,1),      & !  |       |   |   |   |   |   |   |
      B(1,1+je2/2),      & !  +-------+ . |   | X | = +---+---+
      C(1+ie1/2,1+je2/2),& !  |   X   |   |   |   |   |   | X |
      sze,               & !  +-------+   +---+---+   +---+---+
      ie1-(ie1/2),       & !      A           B           C
      je2-(je2/2))

  !$OMP END TASK
  !$OMP TASKWAIT

end if

end
```

```
subroutine mat_prod(A,B,C,LDA,m,n)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP SINGLE
call divideAndConquer(A,B,C,LDA,m,n)
!$OMP END SINGLE NOWAIT
!$OMP TASKWAIT
!$OMP END PARALLEL
```

.