```
#pragma omp parallel for num_threads(NT)
  for (i=imin;i<imax;i++) coll[i-imin] = dt*(coll_term_f (i,I, F,g));

  if (mpi_rank > 0) {
    MPI_Send(coll,N1,MPI_DOUBLE,0,0,MPI_COMM_WORLD)
      } else {
    while (count < mpi_size) {
      MPI_Recv(tmp,N1,MPI_DOUBLE,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
      sender = mpi_status.MPI_SOURCE;
```

Introduction to parallel programming (for physicists)

François Gélis & Grégoire Misguich, IPhT courses, June 2019.

université
PARIS-SACLAY

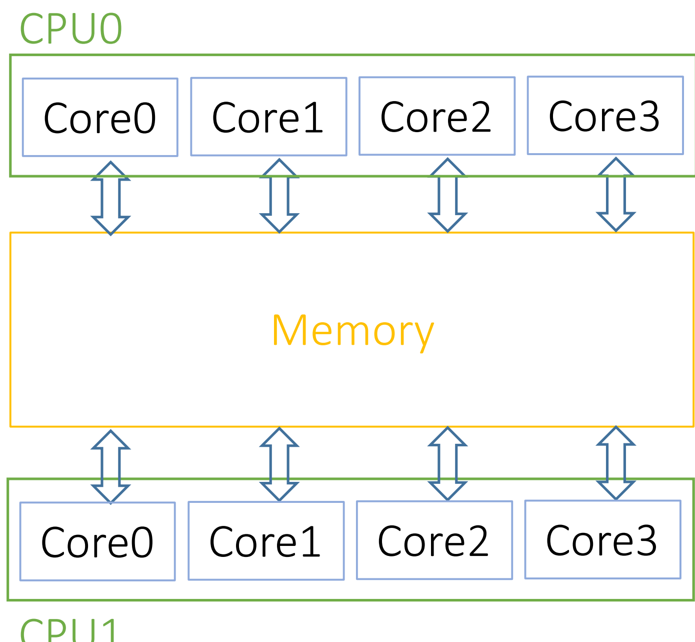
IPhT, CEA-Saclay
Itzykson room
courses.ipht.fr

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

- 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 ☺

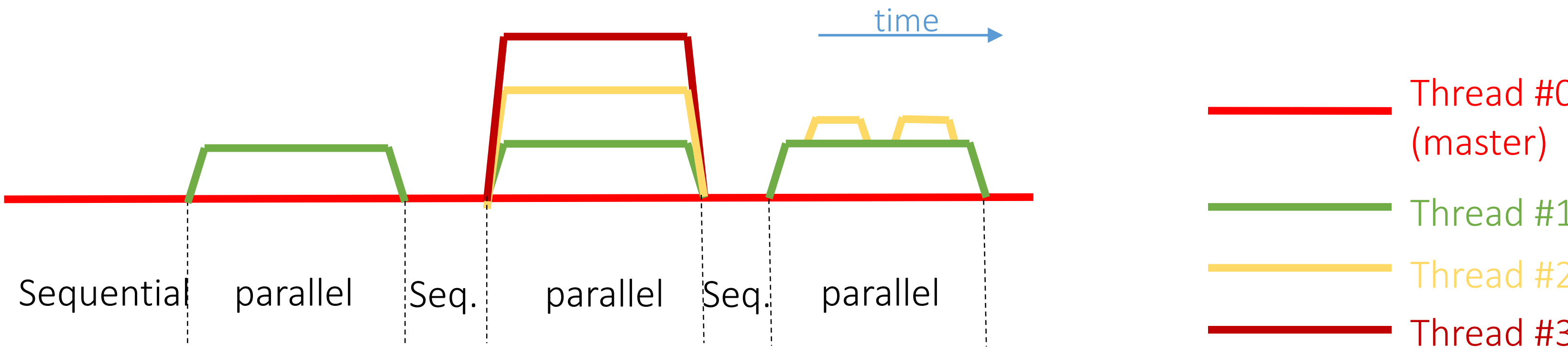


OpenMP

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()); }
}
```

nb. of threads *not* specified (the `num_threads(3)` clause is struck through)

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

```c
#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

Master thread

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

```c
#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;
}
```

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

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

→ 5 times faster

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
   3.1415926555533034
real    0m2.115s
user    0m21.133s
sys     0m0.000s
$ export OMP_NUM_THREADS=1;time ./pi_f90.exe
   3.1415926555977323
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] )
```

`type`:

- `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:

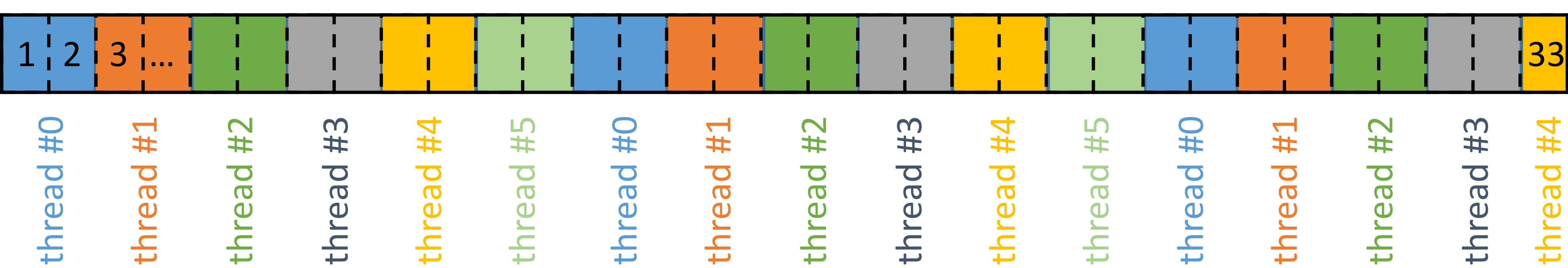
```
#pragma omp parallel num_threads(6)
{
#pragma omp for schedule(static)
for (int i=0;i<33;i++)
…
}
```

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++)
…
}
```



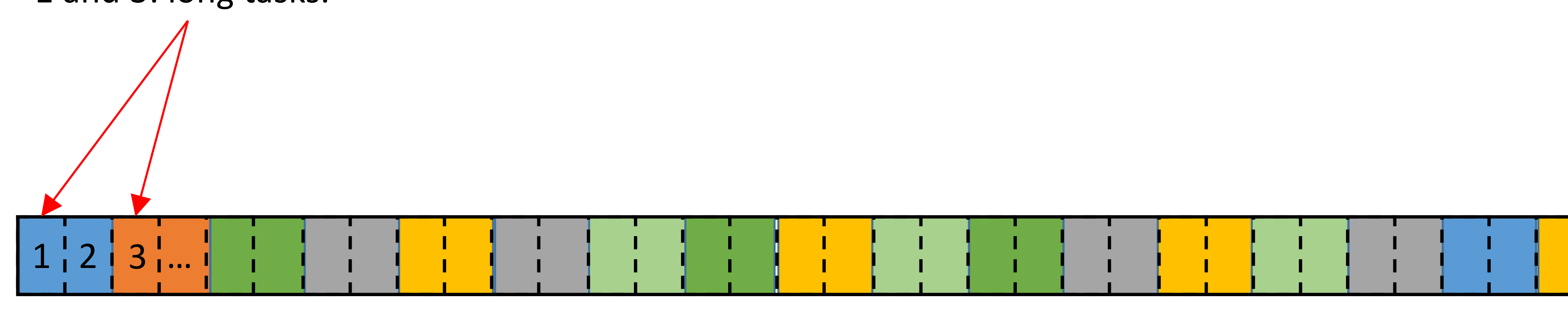
Advantage of large chunks: less overhead, cache friendly

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.



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=102858413O
Thread #3, r=105705637
$ ./srand.exe
Thread #1, r=1644198542
Thread #0, r=105705637
Thread #3, r=1341262422
Thread #2, r=102858413O
$ ./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`)

a[0]	a[1]	…	**a[b]**	…	a[N]

```
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 !

2
3
3
1
3
4
0
2
4
1
0

```
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

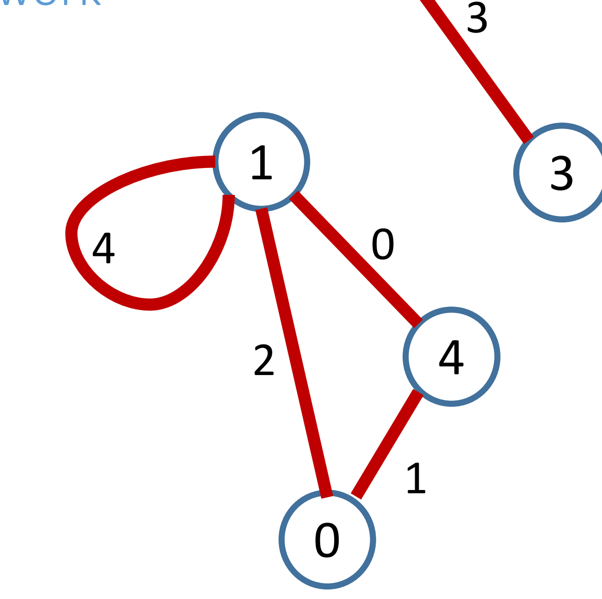
Applications: computing the number of neighbors of a given node in a network

```
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

```c
#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));
      }
   }
}
```

```c
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];
}
```