DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING

[INQ0091623] AA 2021-22

Lab 10

Parallelization using OpenMP

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

OpenMP www.openmp.org

(Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran.

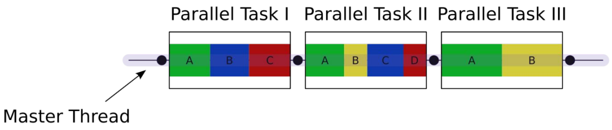
Multithreading REPLAY:

OpenMP is an implementation of multithreading: a primary thread (the main process) creates a specified number of sub-threads and the system divides a task among them. The threads run concurrently:

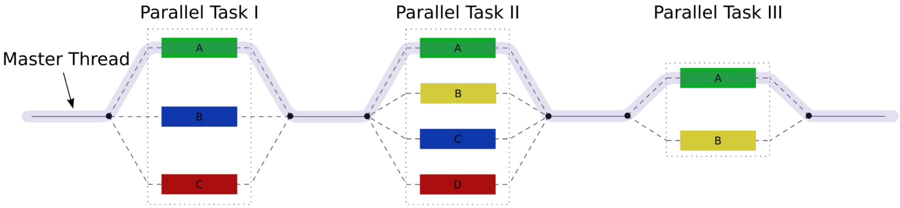
- the system runtime environment can **allocate threads to different hardware** processors (CPU cores that can share the same memory segments),
- it can **interleave threads in time** with a proper preemption of the process execution even in a single core,
- or it can **do both** distributing threads among different cores and in time.

OpenMP

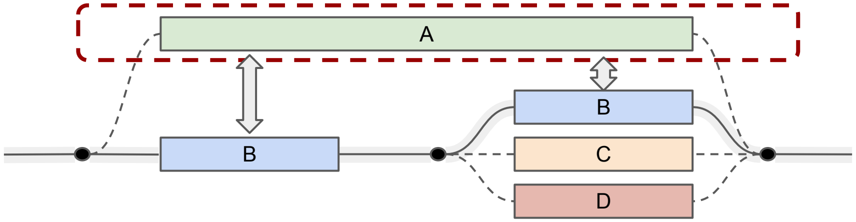
The typical organization of a parallel process can be meant with two different purposes:



1. Parallelization of independent tasks to speed up the overall computation



2. Parallel execution of tasks where each (or some) of them is specialized with a specific function to insulate a time constrained process from the rest of the system. This is the typical RealTime processing organization



OpenMP: Independent execution of parallel tasks

The OpenMP library is specifically designed for the first case, the parallel execution that distribute a similar computation among cores to speed up the process computation time.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code.

Both ***task parallelism*** and ***data parallelism*** can be achieved using OpenMP in this way.

OpenMP C interface

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

In C (and C++), OpenMP uses **#pragmas**.

NOTE: pragmas are specific instructions that can be added to code and are directed to the compiler tuning the compilation itself. So OpenMP must be something that plays with the compiler !

Indeed .. OpenMP appears as an internal tool that is added to most of the high level compilers such as GCC, LLVM (clang), Intel compiler, etc..

```
$ gcc -fopenmp hello.c -o hello
```

OpenMP directives

The typical omp #pragma directive is added to the code as the following line:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

omp acts as a namespace to identify the OpenMP infrastructure, while the directive is one of the declared directives in the API interface.

For instance we will see that we can add a pragma to directly parallelize a for loop with the following preprocessor directive:

```
#pragma omp for private(a)
```

In this case the for loop accepts a ***clause*** to specify additional information that is useful to configure the parallelization. In the example the clause “private(a)” is used to tell the preprocessor to create a private copy of the variable *a* within each threads.

Thread creation: the *parallel* construct

The very basic omp directive si the **omp parallel** that is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as master thread with thread ID 0.

Example (C program): Display "Hello, world." using multiple threads.

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

int main(void)
{
    #pragma omp parallel num_threads(2)
    printf("Hello, world.\n");
    return 0;
}
```

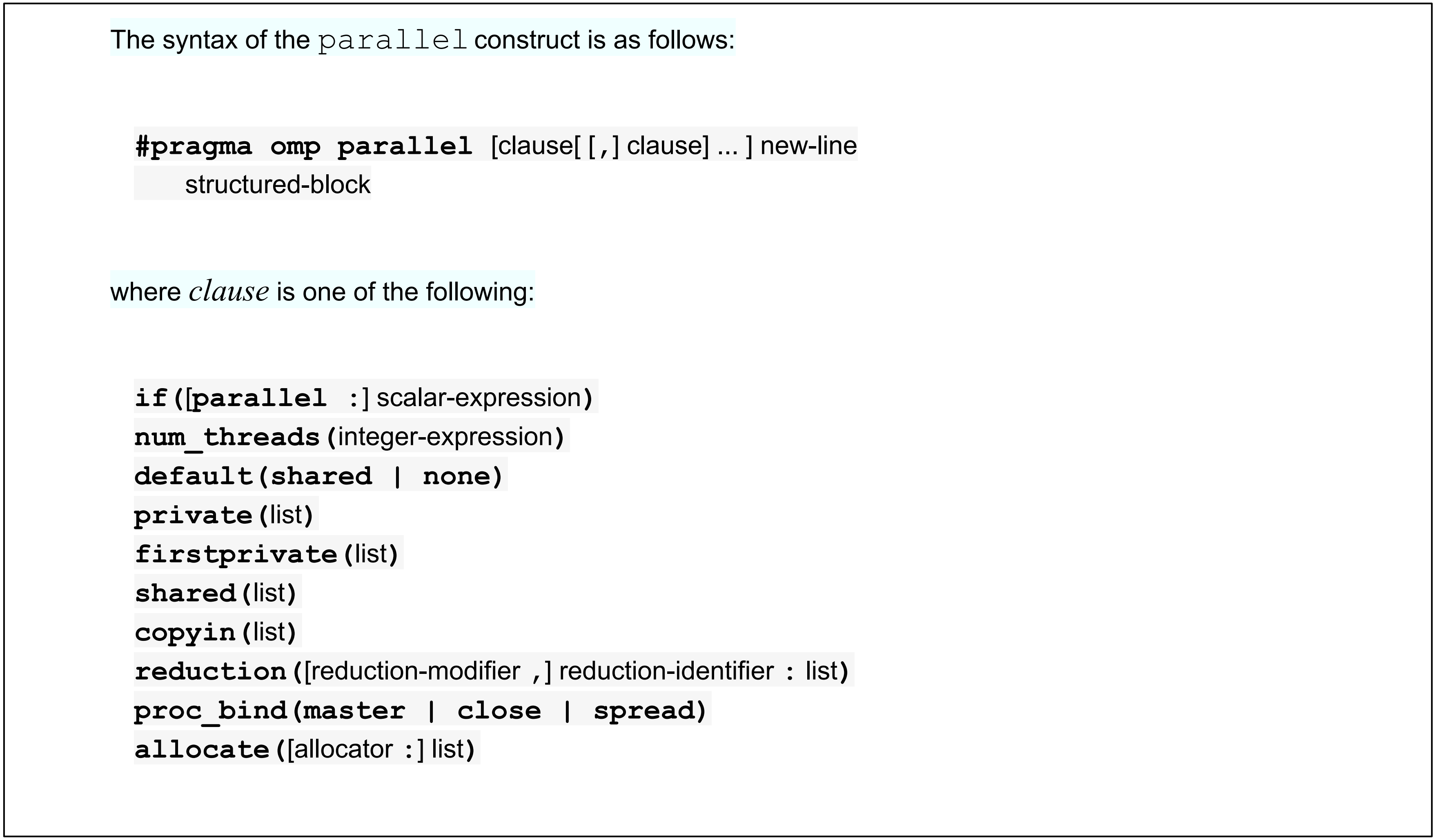
output

```
Hello, wHello, woorld.
rld.
```

Do you remember why? we saw it in lab9 with valgrind complaining about a printf out of the critical section… printf is not thread safe because it shares the I/O resource without locking.

Thread creation: the *parallel* construct

The parallel construct creates a *team* of OpenMP threads that execute the region.



The syntax of the `parallel` construct is as follows:

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

```
if([parallel :] scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction([reduction-modifier ,] reduction-identifier : list)
proc_bind(master | close | spread)
allocate([allocator :] list)
```

Worksharing-loop constructs: **parallel for**

Example:
initialize the value of a large array in parallel, using each thread to do part of the work

```
int main(int argc, char **argv)
{
    int a[100000];
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 100000; i++)
            a[i] = 2 * i;
    }
    return 0;
}
```

or

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++)
        a[i] = 2 * i;

    return 0;
}
```

This is a clear example of independent execution with **data parallelization.** The OpenMP parallel for flag tells the OpenMP system to split this task among its working threads. The threads will each receive a unique and private version of the index ***i*** variable.

The pragma tells the compiler to split the execution with a tempered for loop for each forked thread where the index spans on a subset of the overall declared range.

For instance, with two worker threads, one thread might be handed a version of i that runs from 0 to 49999 while the second gets a version running from 50000 to 99999.

Worksharing-loop constructs: **parallel for**

The syntax of the worksharing-loop construct is as follows:

```
#pragma omp for [clause[ [,] clause] ... ] new-line
      for-loops
```

where clause is one of the following:

```
private(list)
firstprivate(list)
lastprivate([ lastprivate-modifier:] list)
linear(list[ : linear-step])
reduction([ reduction-modifier,]reduction-identifier : list)
schedule([modifier [,  modifier]:]kind[, chunk_size])
collapse(n)
ordered[(n)]
nowait
allocate([allocator :]list)
order(concurrent)
```

The *if* / *private* / *shared* clauses

```
#pragma omp parallel if (n > threshold) \
       shared(n,x,y) private(i)
{
      #pragma omp for
      for (i=0; i<n; i++)
            x[i] += y[i];
}
```

if (scalar expression)

- Only execute in parallel if expression evaluates to true
- Otherwise, execute serially

private (list)

- No storage association with original object
- All references are to the local object
- Values are undefined on entry and exit

shared (list)

- Data is accessible by all threads in the team
- All threads access the same address space
- May need to be protected on write

parallel for example2.c

CODING EXAMPLE CODING EXAMPLE

```c
#define _GNU_SOURCE // sched_getcpu(3) is glibc-specific (see the man page)

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

int main(int argc, char **argv)
{
   int a[100000];
   int printed_once = 0;

   #pragma omp parallel for private(printed_once)
   for (int i = 0; i < 100000; i++) {
       if (!printed_once) {
           int thread_num = omp_get_thread_num();
           int cpu_num = sched_getcpu();
           printf("cpu: %d - thread: %d - i=%d \n", cpu_num, thread_num, i);
           printed_once = 1;
       }
       a[i] = 2 * i;
   }
   return 0;
}
```

```
$ make && ./example2
cc  -fopenmp   example2.c   -o example2

cpu: 2 - thread: 3 - i=37500
cpu: 7 - thread: 0 - i=0
cpu: 6 - thread: 6 - i=75000
cpu: 3 - thread: 7 - i=87500
cpu: 5 - thread: 1 - i=12500
cpu: 1 - thread: 5 - i=62500
cpu: 4 - thread: 4 - i=50000
cpu: 6 - thread: 2 - i=25000
```

```
cpu: 4 - thread: 6 - i=75000
cpu: 3 - thread: 0 - i=0
cpu: 7 - thread: 3 - i=37500
cpu: 2 - thread: 4 - i=50000
cpu: 6 - thread: 5 - i=62500
cpu: 0 - thread: 2 - i=25000
cpu: 5 - thread: 7 - i=87500
cpu: 2 - thread: 1 - i=12500
```

private variables: the ***first*** / ***last*** private clauses

REMEMBER !

- Private variables are undefined on entry and exit of the parallel region.
- The value of the original variable (before the parallel region) is undefined after the parallel region !
- A private variable within a parallel region has no storage association with the same variable outside of the region.
- Use the first/last private clause to override this behaviour.
- We will illustrate these concepts with an example.

the firstprivate and lastprivate clauses come to help !

firstprivate (list)

All variables in the list are initialized with the value the original object had before entering the parallel construct

lastprivate (list)

The thread that executes the sequentially last iteration or section updates the value of the objects in the list.

The ***first*** / ***last*** private clauses

CODING EXAMPLE

CODING EXAMPLE

```
main()
{
int i,n,B,C,A = 10;
#pragma omp parallel
{
   #pragma omp for private(i,A,B)
   for (i=0; i<n; i++)
   {
       B = A + i;  /*-- A undefined, unless declared
                        firstprivate --*/
   }
   C = B;      /*-- B undefined, unless declared
                    lastprivate --*/
}
} /*-- End of OpenMP parallel region --*/
```

NO !

```
main()
{
int i,n,B,C,A = 10;
#pragma omp parallel
{
   #pragma omp for private(i) firstprivate(A) lastprivate(B)

   for (i=0; i<n; i++)
   {
       B = A + i;  /*-- A = 10 --*/
   }
   C = B;      /*-- B = sum_i(A+i) --*/
}
} /*-- End of OpenMP parallel region --*/
```

YES !

What about **n** ???

if no specifically set the default behaviour si **shared** ! so **n** is shared among threads and it is safe because we only access it for read.

The *nowait* clause

CODING EXAMPLE CODING EXAMPLE

```c
#define _GNU_SOURCE // sched_getcpu(3) is glibc-specific (see the man page)
#include <stdio.h>
#include <sched.h>
#include <omp.h>

int main(int argc, char **argv)
{
   int a[100000];
   int printed_once = 0;
   printf("Threads available: %d\n", omp_get_max_threads());
   #pragma omp parallel private(printed_once)
   {
       #pragma omp for
       // parallelize this for loop with all avaliable threads
       for (int i = 0; i < 100000; i++) {
           if (!printed_once) {
               int thread_num = omp_get_thread_num();
               int team_num = omp_get_team_num();
               int cpu_num = sched_getcpu();
               printf("cpu: %d - team: %d - thread: %d - i=%d\n",
                      cpu_num, team_num, thread_num, i);
               printed_once = 1;
           }
           a[i] = 2 * i;
       }
       // after the for has finished one of the threads executes this code.
       #pragma omp single
       {
           int thread_num = omp_get_thread_num();
           int team_num = omp_get_team_num();
           int cpu_num = sched_getcpu();
           printf("cpu: %d - team: %d - thread: %d - single process\n",
                  cpu_num, thread_num);
       }
   }
   return 0;
}
```

```c
#define _GNU_SOURCE // sched_getcpu(3) is glibc-specific (see the man page)
#include <stdio.h>
#include <sched.h>
#include <omp.h>

int main(int argc, char **argv)
{
   int a[100000];
   int printed_once = 0;
   printf("Threads available: %d\n", omp_get_max_threads());
   #pragma omp parallel private(printed_once)
   {

       #pragma omp for nowait
       // parallelize this for loop with all avaliable threads
       for (int i = 0; i < 100000; i++) {
           if (!printed_once) {
               int thread_num = omp_get_thread_num();
               int team_num = omp_get_team_num();
               int cpu_num = sched_getcpu();
               printf("cpu: %d - team: %d - thread: %d - i=%d\n",
                      cpu_num, team_num, thread_num, i);
               printed_once = 1;
           }
           a[i] = 2 * i;
       }
       // immediately after one thread has finished ...
       #pragma omp single
       {
           int thread_num = omp_get_thread_num();
           int team_num = omp_get_team_num();
           int cpu_num = sched_getcpu();
           printf("cpu: %d - team: %d - thread: %d - single process\n",
                  cpu_num, thread_num);
       }
   }
   return 0;
}
```

```
Threads available: 8
cpu: 4 - team: 0 - thread: 6 - i=75000
cpu: 7 - team: 0 - thread: 1 - i=12500
cpu: 2 - team: 0 - thread: 3 - i=37500
cpu: 0 - team: 0 - thread: 2 - i=25000
cpu: 6 - team: 0 - thread: 5 - i=62500
cpu: 3 - team: 0 - thread: 7 - i=87500
cpu: 6 - team: 0 - thread: 4 - i=50000
cpu: 3 - team: 3 - thread: 0 - single process
```

```
Threads available: 8
cpu: 4 - team: 0 - thread: 7 - i=87500
cpu: 6 - team: 0 - thread: 6 - i=75000
cpu: 2 - team: 0 - thread: 2 - i=25000
cpu: 2 - team: 0 - thread: 3 - i=37500
cpu: 4 - team: 7 - thread: 0 - single process
cpu: 7 - team: 0 - thread: 4 - i=50000
cpu: 0 - team: 0 - thread: 5 - i=62500
cpu: 0 - team: 0 - thread: 1 - i=12500
```

The *barrier* clause

CODING EXAMPLE

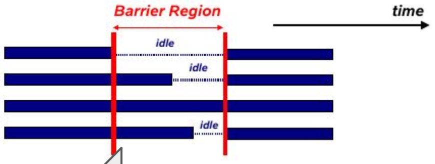
CODING EXAMPLE

```c
#define _GNU_SOURCE // sched_getcpu(3) is glibc-specific (see the man page)
#include <stdio.h>
#include <sched.h>
#include <omp.h>

int main(int argc, char **argv)
{
   int a[100000];
   int printed_once = 0;
   printf("Threads available: %d\n", omp_get_max_threads());
   #pragma omp parallel private(printed_once)
   {
       #pragma omp for
       // parallelize this for loop with all avaliable threads
       for (int i = 0; i < 100000; i++) {
           if (!printed_once) {
               int thread_num = omp_get_thread_num();
               int team_num = omp_get_team_num();
               int cpu_num = sched_getcpu();
               printf("cpu: %d - team: %d - thread: %d - i=%d\n",
                      cpu_num, team_num, thread_num, i);
               printed_once = 1;
           }
           a[i] = 2 * i;
       }
       // after the for has finished one of the threads executes this code.
       #pragma omp single
       {
           int thread_num = omp_get_thread_num();
           int team_num = omp_get_team_num();
           int cpu_num = sched_getcpu();
           printf("cpu: %d - team: %d - thread: %d - single process\n",
                  cpu_num, thread_num);
       }
   }
   return 0;
}
```

```
Threads available: 8
cpu: 4 - team: 0 - thread: 6 - i=75000
cpu: 7 - team: 0 - thread: 1 - i=12500
cpu: 2 - team: 0 - thread: 3 - i=37500
cpu: 0 - team: 0 - thread: 2 - i=25000
cpu: 6 - team: 0 - thread: 5 - i=62500
cpu: 3 - team: 0 - thread: 7 - i=87500
cpu: 6 - team: 0 - thread: 4 - i=50000
cpu: 3 - team: 3 - thread: 0 - single process
```

```c
#define _GNU_SOURCE // sched_getcpu(3) is glibc-specific (see the man page)
#include <stdio.h>
#include <sched.h>
#include <omp.h>

int main(int argc, char **argv)
{
   int a[100000];
   int printed_once = 0;
   printf("Threads available: %d\n", omp_get_max_threads());
   #pragma omp parallel private(printed_once)
   {
       #pragma omp for nowait
       // parallelize this for loop with all avaliable threads
       for (int i = 0; i < 100000; i++) {
           if (!printed_once) {
               int thread_num = omp_get_thread_num();
               int team_num = omp_get_team_num();
               int cpu_num = sched_getcpu();
               printf("cpu: %d - team: %d - thread: %d - i=%d\n",
                      cpu_num, team_num, thread_num, i);
               printed_once = 1;
           }
           a[i] = 2 * i;
       }
       #pragma omp barrier
       #pragma omp single
       {
           int thread_num = omp_get_thread_num();
           int team_num = omp_get_team_num();
           int cpu_num = sched_getcpu();
           printf("cpu: %d - team: %d - thread: %d - single process\n",
                  cpu_num, thread_num);
       }
   }
   return 0;
}
```

```
Threads available: 8
cpu: 4 - team: 0 - thread: 6 - i=75000
cpu: 7 - team: 0 - thread: 1 - i=12500
cpu: 2 - team: 0 - thread: 3 - i=37500
cpu: 0 - team: 0 - thread: 2 - i=25000
cpu: 6 - team: 0 - thread: 5 - i=62500
cpu: 3 - team: 0 - thread: 7 - i=87500
cpu: 6 - team: 0 - thread: 4 - i=50000
cpu: 3 - team: 3 - thread: 0 - single process
```



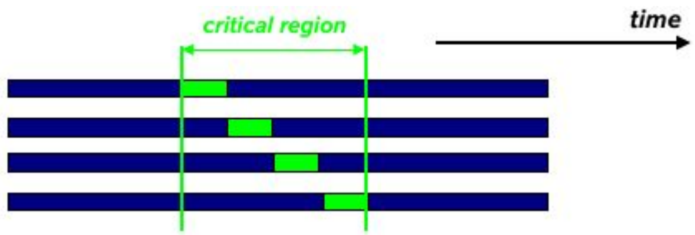
Critical sections !

Sometime we want to access in edit to a shared resource. For instance if sum is a shared variable, this loop can not be run in parallel:

```
#pragma omp parallel for
for (i=0; i < N; i++){
 sum += a[i];
}
```

WARNING !!!

RACE CONDITION on sum



```
#pragma omp parallel for
for (i=0; i < N; i++){
 // do something in parallel
 #pragma omp critical(sum)
 sum += a[i];
 // do something else in parallel
}
```

or

```
#pragma omp parallel for
for (i=0; i < N; i++){
 // do something in parallel
 #pragma omp atomic
 sum += a[i];
 // do something else in parallel
}
```

OK !

The schedule clause

schedule (static | dynamic | guided [, chunk])
schedule (runtime)

static [, chunk]

- Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
- In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads

dynamic [, chunk]

- Fixed portions of work; size is controlled by the value of chunk
- When a thread finishes, it starts on the next portion of work

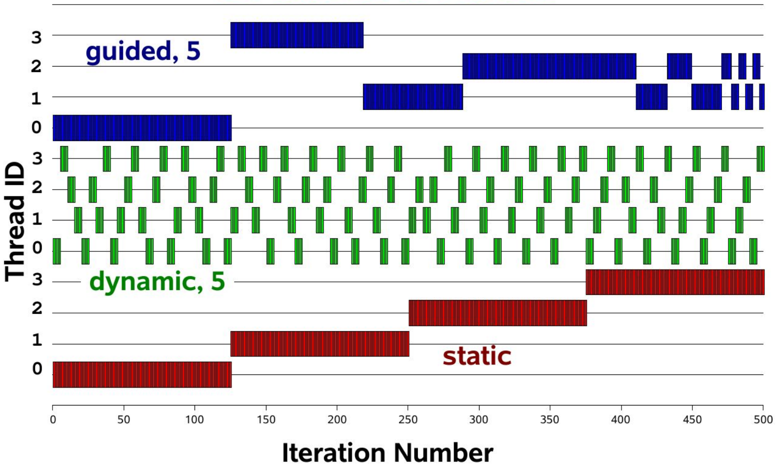
guided [, chunk]

- Same dynamic behaviour as "dynamic" , but size of the portion of work decreases exponentially

runtime

- Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE

500 iterations on 4 threads



Schedule clause **static**

CODING EXAMPLE CODING EXAMPLE

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16

int main()
{
   int i;

   #pragma omp parallel for schedule(static) num_threads(THREADS)
   for (i = 0; i < N; i++)
   {
       /* wait for i seconds */
       sleep(i);
       printf("Thread %d has completed iteration %d.\n",
              omp_get_thread_num(), i);
   }
   /* all threads done */
   printf("All done!\n");
   return 0;
}
```

This program specifies static scheduling, in the parallel for directive. This is actually the default, so is not really needed.

How long does this program take?

How long could it take in the best case?

```
Thread 0 has completed iteration 0.
Thread 0 has completed iteration 1.
Thread 0 has completed iteration 2.
Thread 1 has completed iteration 4.
Thread 0 has completed iteration 3.
Thread 2 has completed iteration 8.
Thread 1 has completed iteration 5.
Thread 3 has completed iteration 12.
```

```
Thread 1 has completed iteration 6.
Thread 2 has completed iteration 9.
Thread 1 has completed iteration 7.
Thread 3 has completed iteration 13.
Thread 2 has completed iteration 10.
Thread 2 has completed iteration 11.
Thread 3 has completed iteration 14.
Thread 3 has completed iteration 15.
All done!  … real    0m54.009s !! :-o
```

Schedule clause **dynamic**

CODING EXAMPLE CODING EXAMPLE

```c
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16

int main()
{
   int i;
#pragma omp parallel for schedule(dynamic) num_threads(THREADS)
   for (i = 0; i < N; i++)
   {
       /* wait for i seconds */
       sleep(i);
       printf("Thread %d has completed iteration %d.\n",
              omp_get_thread_num(), i);
   }
   /* all threads done */
   printf("All done!\n");
   return 0;
}
```

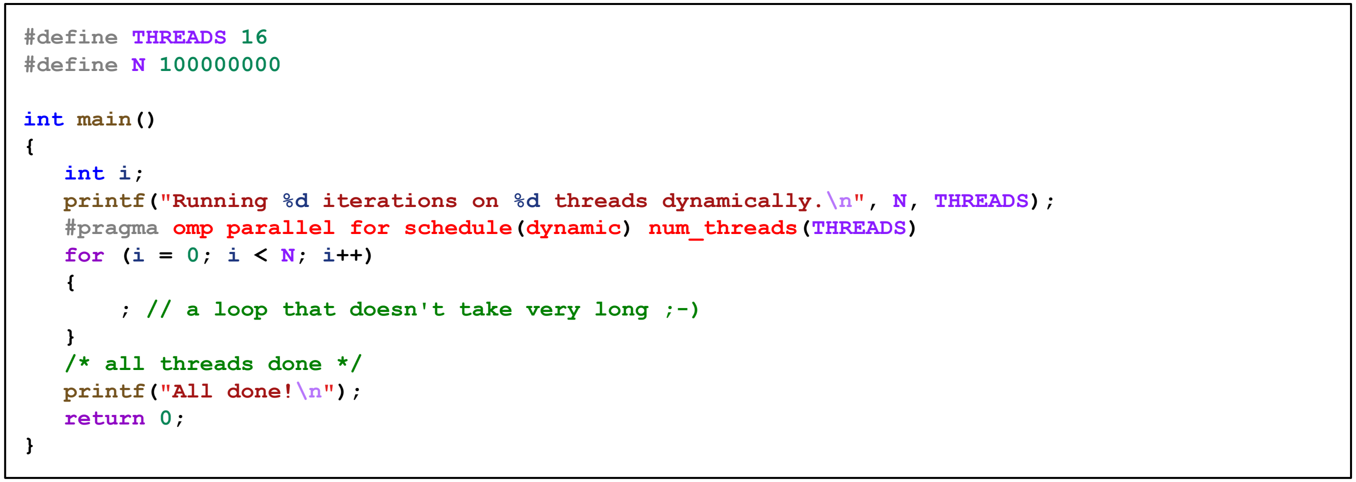
```
Thread 0 has completed iteration 0.
Thread 3 has completed iteration 1.
Thread 2 has completed iteration 2.
Thread 1 has completed iteration 3.
Thread 0 has completed iteration 4.
Thread 3 has completed iteration 5.
Thread 2 has completed iteration 6.
Thread 1 has completed iteration 7.
```

```
Thread 0 has completed iteration 8.
Thread 3 has completed iteration 9.
Thread 2 has completed iteration 10.
Thread 1 has completed iteration 11.
Thread 0 has completed iteration 12.
Thread 3 has completed iteration 13.
Thread 2 has completed iteration 14.
Thread 1 has completed iteration 15.
All done!

real   0m36.010s
```

Scheduling dynamic (use with care)

Dynamic scheduling seems to be a optimal solution, the selection of a new thread is done using a round robin policy on the free threads. This is for sure better when we do not know a priori the time of each task, and the iterations take very different amounts of time each other.

However, there is some overhead to dynamic scheduling ! After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration, and perform a context switch.



```c
#define THREADS 16
#define N 100000000

int main()
{
   int i;
   printf("Running %d iterations on %d threads dynamically.\n", N, THREADS);
   #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
   for (i = 0; i < N; i++)
   {
       ; // a loop that doesn't take very long ;-)
   }
   /* all threads done */
   printf("All done!\n");
   return 0;
}
```

```
Running 100000000 iterations on 16 threads dynamically.
All done!

real    0m1.828s
user    0m21.045s
```

```
Running 100000000 iterations on 16 threads static.
All done!

real    0m0.023s
user    0m0.142s
```

Schedule clause **guided** (best?)

CODING EXAMPLE

CODING EXAMPLE

```
#define THREADS 4
#define N 16

int main()
{
   int i;

   #pragma omp parallel for schedule(static)
num_threads(THREADS)
   for (i = 0; i < N; i++)
   {
       /* wait for i seconds */
       sleep(i);
       printf("Thread %d has completed iteration %d.\n",
              omp_get_thread_num(), i);
   }
   /* all threads done */
   printf("All done!\n");
   return 0;
}
```

```
#define THREADS 16
#define N 100000000

int main()
{
   int i;
   printf("Running %d iterations on %d threads
dynamically.\n", N, THREADS);
   #pragma omp parallel for schedule(guided)
num_threads(THREADS)
   for (i = 0; i < N; i++)
   {
       ; // a loop that doesn't take very long ;-)
   }
   /* all threads done */
   printf("All done!\n");
   return 0;
}
```

```
Thread 0 has completed iteration 0.
Thread 0 has completed iteration 1.
Thread 0 has completed iteration 2.
Thread 3 has completed iteration 4.
Thread 0 has completed iteration 3.
Thread 1 has completed iteration 7.
Thread 3 has completed iteration 5.
Thread 2 has completed iteration 10.
Thread 3 has completed iteration 6.
Thread 1 has completed iteration 8.
Thread 0 has completed iteration 12.
Thread 2 has completed iteration 11.
Thread 1 has completed iteration 9.
Thread 3 has completed iteration 13.
Thread 0 has completed iteration 14.
Thread 2 has completed iteration 15.
All done!

real    0m36.004s
user    0m0.080s
```

```
All done!

real    0m0.018s
user    0m0.144s
sys     0m0.001s
```

Scheduling CONCLUSION

OpenMP for automatically splits for loop iterations for us.

But, depending on our program, the default behavior may not be ideal.

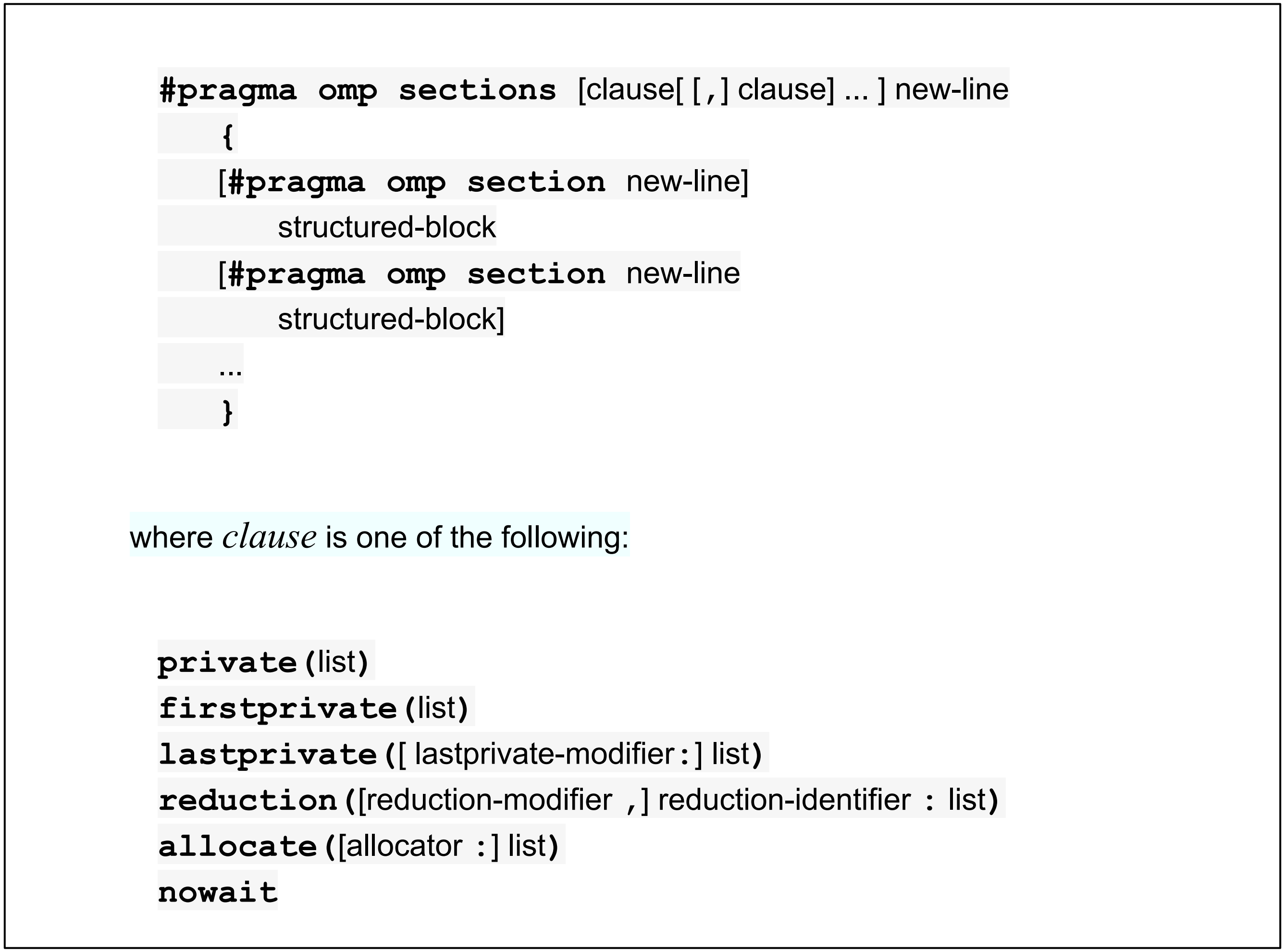
1. For loops where each **iteration takes roughly equal time**, **static** schedules work best, as they have little overhead.

2. For loops where each **iteration can take very different amounts of time**, **dynamic** schedules, work best as the work will be split more evenly across threads.

Specifying chunks, or using a guided schedule provide a trade-off between the two. Choosing the best schedule depends on understanding your loop.

The *sections* directive

The individual code blocks are distributed over the threads



```
#pragma omp sections [clause[ [,] clause] ... ] new-line
    {
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block]
    ...
    }
```

where *clause* is one of the following:

```
private (list)
firstprivate (list)
lastprivate ([ lastprivate-modifier :] list)
reduction ([reduction-modifier ,] reduction-identifier : list)
allocate ([allocator :] list)
nowait
```

“Orphaning”

The OpenMP standard does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be “orphaned”

... that is, they can be written outside the lexical extent of a parallel region.

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

int main(void)
{

   function1(); // sequential call
   #pragma omp parallel num_threads(2)
   {
       function1();
       function2();
   }
   return 0;
}
```

```c
void function1() {
   printf("function one\n");
   int i,n=10;
   #pragma omp for nowait
   for(i=0; i<n; i++)
       printf("[%d]  ",i);
   #pragma omp barrier
   printf("\n");
}
```

```c
void function2() {
   #pragma omp single
   printf("function two\n");
}
```

```
$ cc  -fopenmp   example4.c   -o example4
function one
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
function one
[0] [1] [2] [3] [4] function one
[5] [6] [7] [8] [9]
function two
```

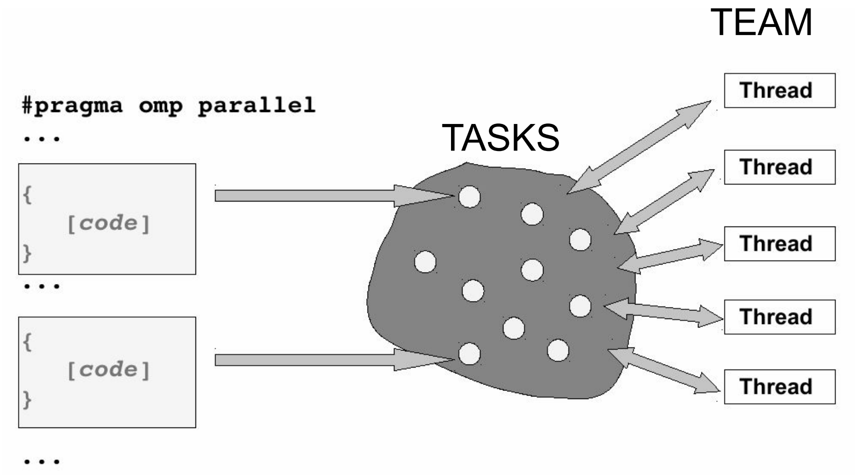
Task scheduling

Tasks

The idea proposed is to switch from a thread-centric model to a task centric-model: a model in which we identify independent unit of works and rely on the system to schedule them.

The task is actually the OpenMP internal parallelization unit.

When a work-sharing construct is used a set of task is created and enqueued in an execution list that will be consumed by a team of threads.



```
#pragma omp parallel
{
    {   // a set of N tasks to print A
        int id = omp_get_team_num();
        printf(" A%d ", id);
    }

    {   // a set of N tasks to print B
        int id = omp_get_team_num();
        printf(" B%d ", id);
    }
}
```

With 8 cores a team of default 8 threads is created and the tasks are scheduled for execution:

[**A0 A0 A0 A0 B0 A0 A0 B0 B0 A0 B0 B0 B0 A0 B0 B0**]

The ***task*** construct

The tasks work units can be also programmatically defined with a specific construct. Each time we call this construct a new task is created and spooled to be executed by the team.

```
#pragma omp task [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

```
if([ task :] scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
in_reduction(reduction-identifier : list)
depend([depend-modifier,] dependence-type : locator-list)
priority(priority-value)
allocate([allocator :] list)
affinity([aff-modifier :] locator-list)
detach(event-handle)
```

The *team* construct

The teams construct creates a league of initial teams and the initial thread in each team executes the region.

```
#pragma omp teams [clause[ [,] clause] ... ] new-line
        structured-block
```

where *clause* is one of the following:

```
num_teams(integer-expression)
thread_limit(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
reduction([default ,] reduction-identifier : list)
allocate([allocator :] list)
```

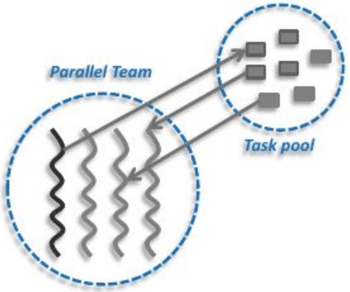
The ***task*** construct

Tasks, whose execution may be deferred or immediately executed, are created ...

- ***when reaching a parallel region*** implicit tasks are created (per thread)
- ***when encountering a task construct*** explicit task is created

and also (but we won't see)...

- ***when encountering a taskloop construct*** explicit tasks per chunk are created
- ***when encountering a target construct*** target task is created



WHY USING EXPLICIT TASK creation ??

Mainly because it supports the “unstructured parallelism”:

- unbounded loops

```
while ( <expr> ) {
  … [ task ]
}
```

- recursive functions

```
void myfunc( <args> ) {
  [ task ] ...;  myfunc( <newargs> ); ...; [ task ]
}
```

EXAMPLE

```
#pragma omp parallel
#pragma omp master
while (elem != NULL) {
   #pragma omp task
   compute(elem);
   elem = elem->next;
}
```

Task scheduling: tied vs untied

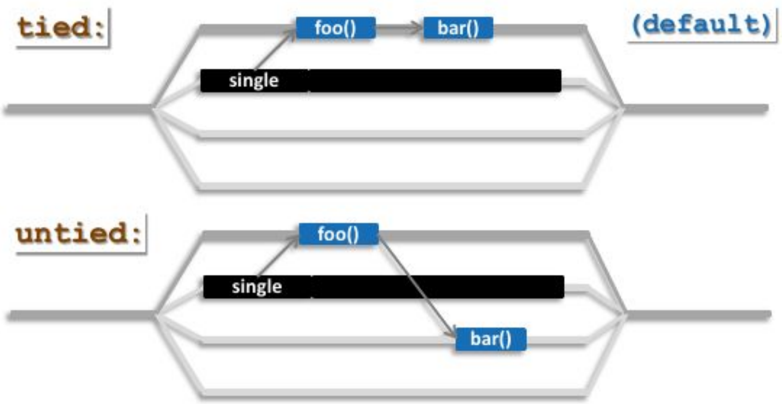
When not explicitly set all tasks are ***tied***

- tied tasks are executed always by the same thread (not necessarily creator)
- **WARNING:** tied tasks may run into performance problems

They can be then scheduled ***untied***

- can potentially switch to any thread (of the team)
- **WARNING:** bad mix with thread features: thread-id, threadprivate, critical regions...
- gives the runtime more flexibility to schedule tasks

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task untied
    {
    foo();
    #pragma omp taskyield
    bar()
    }
}
```

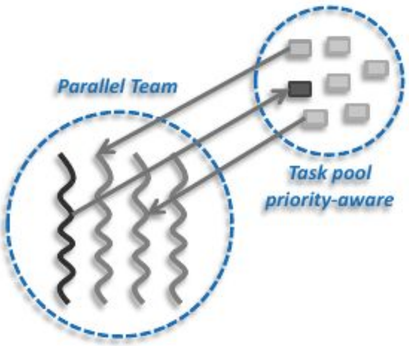


Task scheduling: priority

Another important feature of task scheduling is the possibility to assign the task a priority of execution.

The tasks are entered then in a priority queue where the idle threads get the next task to execute from the highest priority (front of the queue).

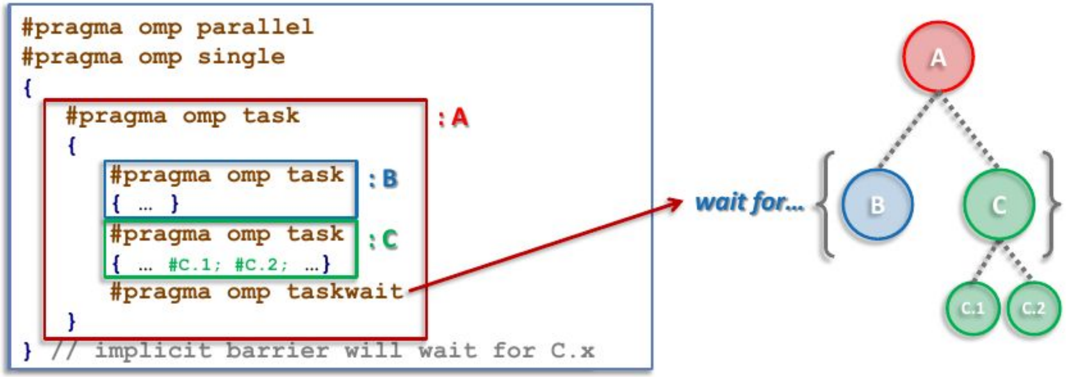
```
#pragma omp parallel
#pragma omp single
{
for ( i = 0; i < SIZE; i++) {
       #pragma omp task priority(1)
       { code_A; }
   }
   #pragma omp task priority(100)
   { code_B; }
...
}
```



Task synchronization: taskwait and taskgroup

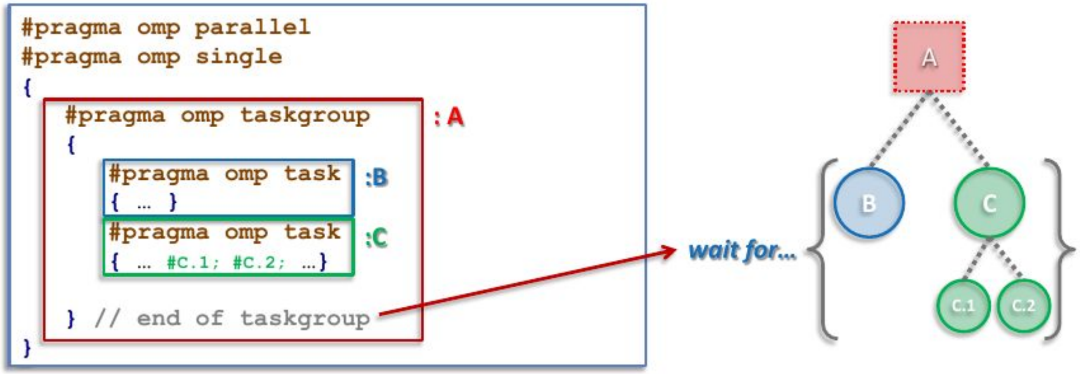
When tasks are created they start concurrently, to create a synchronization we can use the *taskwait* and *taskgroup:*

- **taskwait** construct (***shallow synchronization***) wait the internal operations but not the nested tasks



```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task                      : A
  {
     #pragma omp task                   : B
     { … }
     #pragma omp task                   : C
     { … #C.1; #C.2; …}
     #pragma omp taskwait
  }
} // implicit barrier will wait for C.x
```

- **taskgroup** construct (***deep synchronization***) wait all nested tasks end



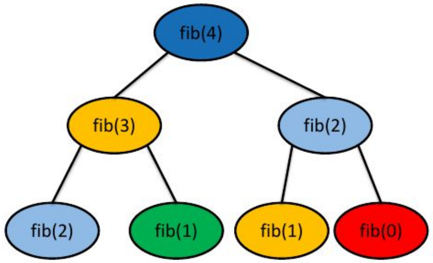
```
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskgroup                 : A
  {
     #pragma omp task                   :B
     { … }
     #pragma omp task                   :C
     { … #C.1; #C.2; …}

  } // end of taskgroup
}
```

EXAMPLES

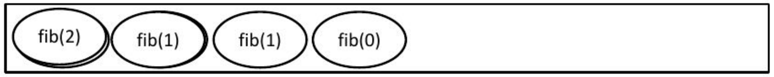
EXAMPLE: fibonacci

```
int fibonacci(int x) {
    if(x<2) { usleep(500); return 1; }
    int n1,n2;
    #pragma omp task shared(n1)
    n1 = fibonacci(x-1);
    #pragma omp task shared(n2)
    n2 = fibonacci(x-2);
    #pragma omp taskwait
    return n1+n2;
}


int main(void) {
    #pragma omp parallel
    #pragma omp single
    {
        printf("fibonacci(10) = %d", fibonacci(10));
    }
}
```

- **T1 enters fib(4)**
- **T1 creates tasks for fib(3) and fib(2)**
- **T1 and T2 execute tasks from the queue**
- **T1 and T2 create 4 new tasks**
- **T1 - T4 execute tasks**



Task Queue



EXAMPLE: producer and consumer with OMP

CODING EXAMPLE CODING EXAMPLE

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <omp.h>

#define BUFFER_SIZE 5
#define NITER 30

// Shared by the producer and consumer thread
int buffer[BUFFER_SIZE];
int in = 0;     //next free position
int out = 0;    //first full position
int finished = 0;

// Forward declarations
void* producer();
void* consumer();
```

```c
int main(int argc, char**argv){
  printf("\n");

  int nConsumers;
  if(argc != 2)
  {
      printf("Usage: prodcons <numConsumers >\n");
      exit(0);
  }
  sscanf(argv[1], "%d", &nConsumers);

  #pragma omp parallel
  #pragma omp single
  {
     #pragma omp task
     producer();

     for(int i=0; i<nConsumers; ++i) {
       #pragma omp task
       consumer();
     }
  }
  printf("\n");

  return 0;
}
```

EXAMPLE: producer and consumer with OMP

CODING EXAMPLE CODING EXAMPLE

```
void *producer(){
  for (int i=0; i< NITER; i++) {

     // Do nothing until a slot is available
     while (((in + 1) % BUFFER_SIZE) == out)
        usleep(100);

     #pragma omp critical
     {
       buffer[in] = i;
       in = (in + 1) % BUFFER_SIZE;
     }
     printf("+");
     fflush(stdout);
  }
  finished = 1;
  return NULL;
}
```

```
void *consumer(){
  int next_consumed;

  while( 1 ){
     // Do nothing if the buffer is empty and the producer did not finish
     while (!finished && in == out)
        usleep(100);

     // If we finished and all elements has been consumed then exit
     if(finished && in==out) {
        return NULL;
     }

     #pragma omp critical
     {
       next_consumed = buffer[out];
       out = (out + 1) % BUFFER_SIZE;
     }
     printf(" %d ",next_consumed); fflush(stdout);

  }
  return NULL;
}
```

```
$ make && ./prodcons_omp 3
cc  -fopenmp   prodcons_omp.c   -o prodcons_omp
+++++ 0  1  2  3  4 ++++ 5  6  7  8 ++++ 9 + 11  13  10 ++++ 12  14  15  16  17 ++++ 18  19  20  21 ++++ 22  23  24  25 ++++ 26  27  28  29
```