Tasks

Task abstraction

represents any contained sequence of instructions in the code, logically defining a finite work/function/assignment

Asynchronous + Interleaved execution + dependencies

Dependency graph among task.
Must be acyclic.
Related to data dependencies.

Data abstraction

Concurrent access

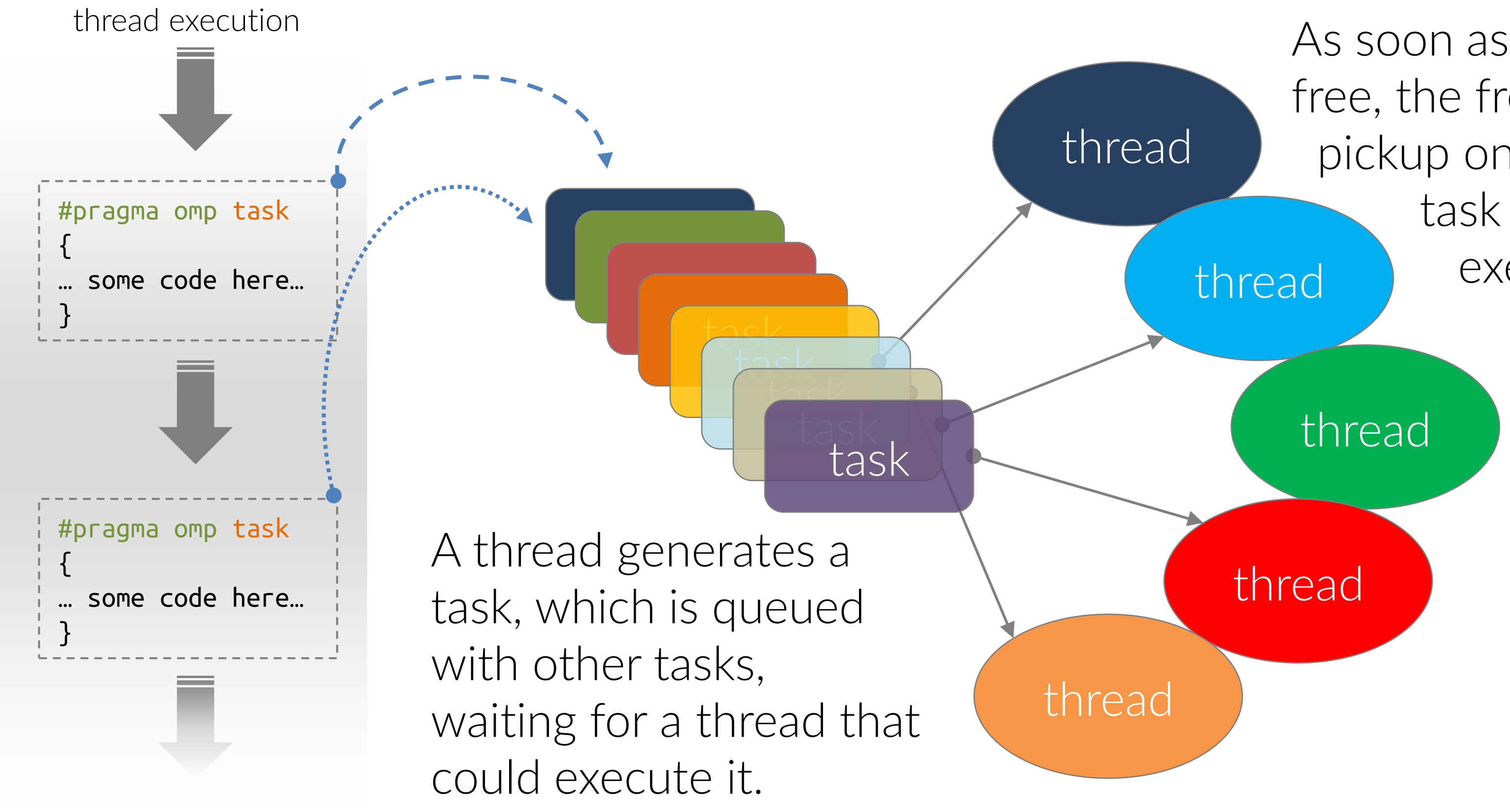
represents any piece of logically uniform "information", that may be accessed by several threads; out-of-order access needs to be managed

As we have seen in the previous example `(02_sections_nested_irrecular.c)`, it is sometimes possible to parallelize a workflow which is irregular or runtime-dependent using OpenMP sections.
However, often the solution is quite ugly and convoluted.

Since version 3.0, OpenMP offers a new elegant construct designed for this class of problems: the OpenMP task.

What happens under the hood is that OpenMP creates a “bunch of work” along with the data and the local variables it needs, and *schedules* it for execution at some point in the future.

OpenMP tasks



As soon as they are free, the free threads pickup one waiting task each, and execute it

A thread generates a task, which is queued with other tasks, waiting for a thread that could execute it.

OpenMP `tasks`

As almost everything else in OpenMP, a task must be generated *inside* a parallel region and it is linked to a specific block of code.
If its execution is not properly "protected", it might be executed by *more* than one thread (i.e. by all threads that encounter the task definition), which is not in general what we want.

To guarantee that each task is executed only once, every task must be generated within a `single` or `master` region.

The `single` region is preferable because of its implied barrier that makes all tasks to be completed before passing. In case you use a `master` region, pay attention to the execution flow.
Moreover, the `master` has often the heavier burden so it's best to user a `single` region, possibly with the `nowait` clause.

OpenMP tasks

A classical example: traversing a linked list

```
#pragma parallel region
{
    …;
    #pragma omp single nowait
    {
        while( !end_of_list(node) ) {
          if( node_is_to_be_processesed(node) )
              #pragma omp task
              process_node ( node );
          node = next_node( node );
    }
    …;
}
```

A task is generated for each node that must be processed

The calling thread continues traversing the linked list

Due to the `nowait` clause, all the threads skip the implied barrier at the end of the `single` region and wait here for being assigned a task

OpenMP tasks

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

```
#pragma omp task shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

Both first and last, which are two shared variables, are key variables for the task execution.

What if they are keep changing?
At the moment of execution, their value could be different than at the moment of task creation, and then the processing would be totally different than the original intention.

OpenMP tasks

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

The values of variables that are susceptible to change and that enter in the execution of the task must be protected to ensure the correctness of the task itself.

With the `firstprivate` clause, we are creating private local variables that will be referred to at the moment of the execution and will still have the correct value.

```
#pragma omp task firstprivate(first, last) shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

A second key point to account for when dealing with the asynchronous execution is the *data* environment.
A task is a confined code section that performs some operations on a data set, that is referred at the moment of the task creation.

You are in charge of ensuring that that reference will still be valid *at the moment of execution*, which is somewhere in the future.

With the `untied` clause, you are signalling that this task – if ever suspended – can be resumed by *any* free thread. The default is the opposite, a task to be `tied` to the thread that initially starts it.
If untied, you must take care of the data environment, of course: for instance, no `threadprivate` variables can be used, nor the thread number, and so on.

```
#pragma omp task firstprivate(first, last) shared(result) untied
{
  double myresult    = 0;
  for( int ii = first; ii < last; ii++)
    myresult += heavy_work_0(array[ii]);
  #pragma omp atomic
  result += myresult;
}
```

OpenMP tasks synchronization

A third key point to catch with asynchronous execution, is about the *timing*, i.e. when a task is executed and how to synchronize them.

At the moment of creation, a task may be *deferred* or not, i.e. its execution may be scheduled for the future or immediately taken while the task region that has generated it is frozen.

There are some constructs that enforce synchronization:

barrier	Implicit or explicit barrier
taskwait	Wait on the completion of all child tasks of the current task
taskgroup	Wait on the completion of all child tasks of the current task **and** of their descendant

OpenMP tasks synchronization

```c
#pragma omp parallel shared(result)
 {
   double result1, result2, result3;

   #pragma omp single nowait
   {

     #pragma omp task  shared(result1)
     {
       double myresult = 0;
       for( int jj = 0; jj < N; jj++ )
         myresult += heavy_work_0( array[jj] );
      #pragma omp atomic update
       result1 += myresult;
     }

     #pragma omp  task shared(result2)
     {
       double myresult = 0;
       for( int jj = 0; jj < N; jj++ )
         myresult += heavy_work_1( array[jj] );
      #pragma omp atomic update
       result2 += myresult;
     }

     #pragma omp task  shared(result3)
     {
       double myresult = 0;
       for( int jj = 0; jj < N; jj++ )
         myresult += heavy_work_2(array[jj] );
      #pragma omp atomic update
       result3 += myresult;
     }

   }

   #pragma omp taskwait

   #pragma omp atomic update
   result += result1;
   #pragma omp atomic update
   result += result2;
   #pragma omp atomic update
   result += result3;
 }
```

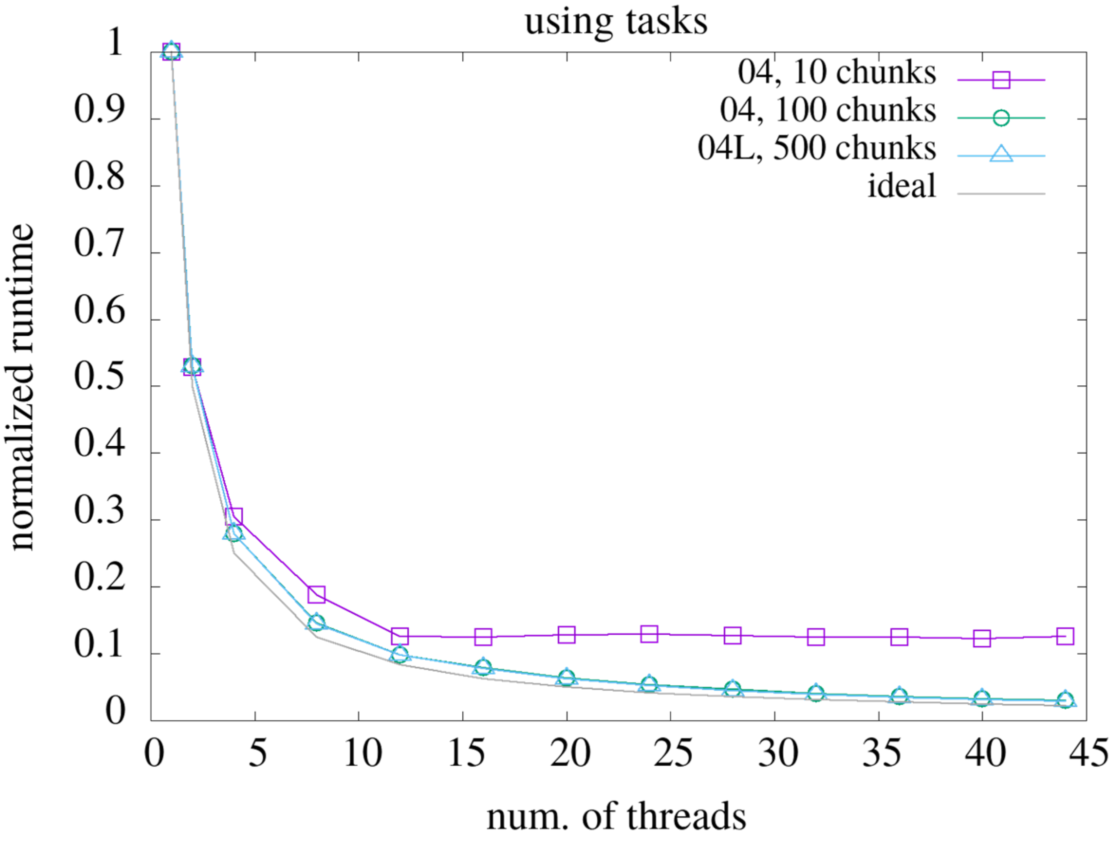
- You need the tasks to be completed *before* to arrive at this point where the final results are accumulated.
- Due to the `nowait` clause, the implied barrier at the end of the `single` region is no respected and the threads are flowing freely beyond that region.
- Without the `taskwait` directive, the threads that are not in the `single` region would execute the updates of `result` with meaningless data.

parallel_tasks/
03_tasks_wrong.c

parallel_tasks/
04_tasks.c

finer implementation

Tasks performance



Tasks priorities

Even if you want your tasks to run concurrently, sometimes it is advisable that some tasks run earlier than others.
For instance, it may be good that the tasks that are receiving data have an higher *priority* than the tasks that post-process them.

You can suggest this to the OpenMP scheduler by using the `priority(p)` clause.
The higher the value of `p`, the sooner the corresponding task will be scheduled for execution.

```
#pragma omp parallel
#pragma #omp single
{
   …;
  #pragma omp task priority(100)
   read_data(…);
  #pragma omp task priority(50)
   process_and_save_data(…);
  #pragma omp task priority(10)
   postprocess_and_send_data(…);
}
```

Tasks dependencies

Often, there are **dependencies** among different tasks:

a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

Tasks dependencies

Often, there are **dependencies** among different tasks: a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write
"flow dependence"

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );

#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

Tasks dependencies

Often, there are **dependencies** among different tasks: a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read
"anti-dependence"

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Tasks dependencies

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

WaW
Write after Write
"output depend."

Both task 0 and task 1 write the same memory region;

```
#pragma omp task depend(OUT:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Tasks dependencies

Often, there are **dependencies** among different tasks:
a given tasks may have to use the results of another one, or in any case to wait for its operations to terminate

RaW
Read after Write

The task 1 read a memory region written by task 0

```
#pragma omp task depend(OUT:the_answer)
 function_wise( *the_answer );
#pragma omp task depend(IN:the_answer)
 function_curious( *the_answer );
```

WaR
Write after Read

The task 0 reads a memory region written by task 1

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

WaW
Write after Write

Both task 0 and task 1 write the same memory region;

```
#pragma omp task depend(OUT:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

RaR
Read after Read

Both task 0 and task 1 read the same memory region; no particular order is needed

```
#pragma omp task depend(IN:the_question)
 function_sage( *the_question );
#pragma omp task depend(OUT:the_question)
 function_curious( *the_question );
```

Tasks dependencies

dependency types:

- **`IN`**: the task will be dependent on a previously generated task if that task has an `out`, `inout` or `mutexinoutset` dependence on the same memory region.
- **`OUT, INOUT`** : the task will be dependent on a previously generated task if that task has an `in`, `out`, `inout` or `mutexinoutset` dependence on the same memory region.
- **`MUTEXINOUTSET`**: the task will be dependent on a previously generated task if that task has an `in`, `out`, `inout` or dependence on the same memory region;
  it will be *mutually exclusive* with another `mutexinoutset` sibling task.

Tasks dependencies

```
Flow-dependence: will write "x=2"

int x = 1;
...;
#pragma omp task shared(x) depend(out:x)
 x = 2;

#pragma omp task depend(in:x)
 printf("x = %d\n", x);
```

```
Anti-dependence: will write "x=1"

int x = 1;
...;
#pragma omp task shared(x) depend(in:x)
 printf("x = %d\n", x);

#pragma omp task shared(x) depend(out:x)
 x = 2;
```

```
output-dependence: will write "x=3", the
dep is enforced by the generation order

#pragma omp single
{
  #pragma omp task shared(x) depend(out:x)
   x = 2;
  #pragma omp task shared(x) depend(out:x)
   x = 3;
  #pragma omp taskwait
   printf("x = %d\n", x);
}
```

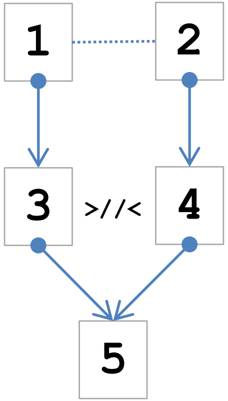
```
No dependence: output is variable, the
printing tasks are independent off each
other
#pragma omp single
{
  #pragma omp task shared(x) depend(out:x)
   x = 2;
  #pragma omp task shared(x) depend(in:x)
   printf("x + 1 = %d\n", x+1);
  #pragma omp task shared(x) depend(in:x)
   printf("x + 2 = %d\n", x+2);
}
```

examples taken from openmp.org

Tasks dependencies

Mutually exclusive dependency

```
...;
#pragma omp task shared(x) depend(out:x)
 x = get_x();                         // task 1

#pragma omp task shared(y) depend(out:y)
 y = get_y();                         // task 2

#pragma omp task shared(z,x) depend(in:x) depend(mutexinoutset:z)
 z *= x;                              // task 3

#pragma omp task shared(z,y) depend(in:y) depend(mutexinoutset:z)
 z *= y;                              // task 4

#pragma omp task shared(a,z) depend(in:z) depend(out_a)
 a = z;                              // task 5
```



mutually independent

mutually exclusive

examples taken from openmp.org

Tasks dependencies

You can enforce to wait for some particular dependence

```
int x = 1, y = 2;

// task 1
#pragma omp task shared(x) depend(inout:x)
 x += 1;

// task 2
#pragma omp task shared(y)
 y *= 2;

#pragma omp taskwait depend(in:x)   // wait for task 1 only

printf("x = %d\n", x);              // this print is safe
printf("y = %d\n", y);              // this print is unsafe

#pragma omp taskwait

printf("y = %d\n", y);              // *now* this print is
                                    // safe too
```

examples taken from openmp.org

Tasks dependencies

You can enforce to wait for some particular dependence

At this point, the `in:x` dependence is fullfilled and the generating thread can prosecute to the `printf` instructions, without waiting for the `task 2` which is not modifying `x`.
What would you modify to make both prints safe and eliminate the last taskwait ?

```
int x = 1, y = 2;

// task 1
#pragma omp task shared(x) depend(inout:x)
 x += 1;

// task 2
#pragma omp task shared(x, y) depend(in:x) depend(inout:y)
 y *= x;

#pragma omp taskwait depend(in:x)   // wait for task 1 only

printf("x = %d\n", x);              // this print is safe
printf("y = %d\n", y);              // this print is unsafe

#pragma omp taskwait

printf("y = %d\n", y);              // *now* this print is
                                    // safe too
```

examples taken from openmp.org

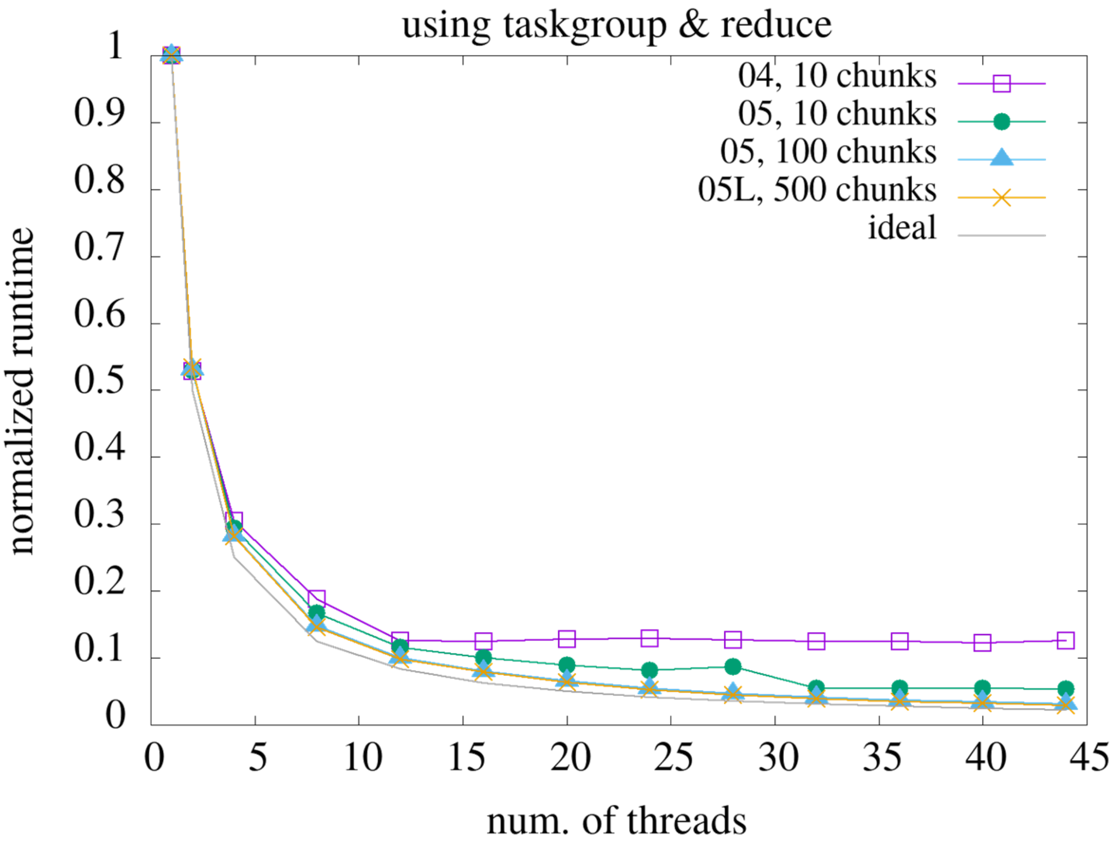
OpenMP taskgroup

```c
#pragma omp parallel proc_bind(close)
  {

    #pragma omp single nowait
    {
      #pragma omp taskgroup task_reduction(+:result)
      {
        int idx   = 0;
        int first = 0;
        int last  = chunk;

        while( first < N )
          {
            last = (last >= N)?N:last;
            for( int kk = first; kk < last; kk++, idx++ )
              array[idx] = min_value + lrand48() % max_value;

            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }
            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }
            #pragma omp task in_reduction(+:result) firstprivate(first, last) untied
            {
              ...
            }

            first += chunk;
            last  += chunk;
          }
      }
    }

    #pragma omp taskwait
  } // close parallel region
```

parallel_tasks/
05_task_taskgroup.c

A `taskgroup` region is declared: at its end, the completion of all tasks generated within it, and of their descendant, is explicitly ensured.

This task are participating to the reduction


using taskgroup & reduce
normalized runtime
num. of threads
04, 10 chunks
05, 10 chunks
05, 100 chunks
05L, 500 chunks
ideal
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
25
30
35
40
45

OpenMP taskloop

```c
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
     //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
    #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
         array[ii] = min_value + lrand48() % max_value;
         result += heavy_work_0(array[ii]) +
           heavy_work_1(array[ii]) +
           heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

parallel_tasks/
06_task_taskloop.c

OpenMP taskloop

```
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
    //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
   #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
        array[ii] = min_value + lrand48() % max_value;
        result += heavy_work_0(array[ii]) +
          heavy_work_1(array[ii]) +
          heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

A `taskloop` region is declared: it blends the flexibility of tasking with the easeof loops

Tasks are created for each iteration

parallel_tasks/
06_task_taskloop.c

OpenMP taskloop

```
#pragma omp parallel proc_bind(close)
 {

  #pragma omp single nowait
   {
     //#pragma omp taskloop grainsize(N/1000) reduction(+:result)
    #pragma omp taskloop num_tasks(N/10) reduction(+:result)
     for( int ii = 0; ii < N; ii++ )
       {
        array[ii] = min_value + lrand48() % max_value;
        result += heavy_work_0(array[ii]) +
          heavy_work_1(array[ii]) +
          heavy_work_2(array[ii]);
       }
   }
   PRINTF("* initializer thread: initialization lasted %g seconds\n", CPU_TIME_th - tstart );

 } // close parallel region


  double tend = CPU_TIME;
#endif
```

To limit overhead, you can control the task generation by using of `num_tasks` and `grainsize` clauses

~~Tasks are created for each iteration~~ Tasks are created accordingly to clauses

parallel_tasks/
06_task_taskloop.c

Hybrid codes
MPI + OpenMP

A handy definition of hybrid

Hybrid programming:

Having more than one *programming models (paradigms)* or *programming systems* in the same code

The abstract definition of a programming concept

`Message-Passing`

`Shared-Memory`

An implementation, possibly with a software stack that realizes it (compiler, libraries,..), of a programming model

`MPI`

MPI has a partial internal implementation of memory sharing

`OpenMP`

A handy definition of hybrid

At the time MPI was designed, threads were of course already existent and used.
At odds with other message-passing implementation, MPI was conceived to be *thread-safe*$^{(*)}$ by design, purposely to encourage hybrid programming.
For instance, that includes not to have a concept like the "current" message, but considering each message as an object itself.

However, the programmer still have to ensure that accesses to data that are shared must be properly implemented/protected

Even if the threads model inside message-passing enables the concurrency within the distributed parallelism, the messages are always exchanged at process-level and not at thread-level: in other words, a thread can perform an MPI call on behalf of its father process, but it will always address another MPI process and not a precise thread inside that process.

- Non-blocking communications
    - Overlapping of computation and communication
    - Hiding of communication latency

- Enables exploiting SMP

A handy definition of hybrid

[*] *thread-safe* by design means that multiple threads from an MPI process can perform MPI calls without interfering with each other; that is possible because MPI is designed so that all the information relative to a message is encapsulated in that same message objects and does not reside anywhere in the library common space.

However, that is not enough.
The thread library [*] must have the ability to yield the execution from one thread to another (for instance when a thread is executing a blocking operation of any kind – MPI, system call, I/O,...).
The programmer must be aware of this and correctly implement the use of threads.

Beware: all the caveats that we encountered about OpenMP programming, still hold in hybrid programming (memory races, false sharing, threads overhead, threads placing, ...)

[*] *POSIX pthread, the most widely used in *nix systems; it is not the only possibility (Java, C++11 also have their native implementation)*

Few things to remember

- **System calls**

  Remember that explicitly calling system calls may lead to portability issues due to different behaviour of user and kernel threads on different systems. Using only MPI calls is instead portable.

- **Competing for closing the same call**

  MPI does not allow two threads to compete for closing the same non-blocking MPI call. It is instead permitted that a thread inits a call that is subsequently closed by a different one.

Initialize MPI for multithreading

```
MPI_Init_thread ( int *argc, char ***argv, int required, int *provided )
```

This function initializes the MPI library with the required level of support, and give back the granted support level.
The allowed levels are the following:

MPI_THREAD_SINGLE	Only the main thread will be running
MPI_THREAD_FUNNELED	Many user threads, only the main one calls MPI
MPI_THREAD_SERIALIZED	Many user threads, only one at a time makes MPI calls
MPI_THREAD_MULTIPLE	Many user threads, any number of threads can make MPI calls at a time

By checking the returned support, you may choose the right MPI library to link with.
`MPI_Init()` actually is a shortcut for `MPI_Init_thread()` with `MPI_THREAD_SINGLE`; hence, whichever you use, you call `MPI_Finalize()`.

Who calls who

If a routine is called, it may need to understand what is

1. The threading support level
   that is achieved by calling `MPI_Query_thread(int *provided)`, which returns the provided level

2. Whether the calling thread is the main thread
   (i.e. the thread that `called MPI_Init_thread()`)
   that is achieved by calling `MPI_Is_thread_main(int*flag)`

A very important clarification

The support level that you require, which may perhaps change at run-time in different runs of your code, IS NOT a requirement neither on the MPI nor on the OpenMP standards.

Then, if you require `MPI_THREAD_FUNNELED` or `MPI_THREAD_SERIALIZED`, that does not mean in any way that **neither MPI nor OpenMP are instrumenting your code so that the MPI calls are made accordingly to the required level**.
That requirement is only a notice to the MPI library which will optimize its internal behaviour accordingly.

You are still in charge of ensuring the correctness of your hybrid code.

TIPS

The MPI standard does NOT require the environmental variables to be propagated to every process by the MPI itself (although it is a quite common case that a specific implementation does it anyway).

If you use a threading library that allows to specify the number of threads to be created by usage of env vars, like OpenMP, you should explicitly take care of this by retrieving the env vars values with Process 0 and then propagating them to the other Processes (alternatively, you do not use env vars and require a given number of threads in a different way).

Probing messages

If you need to probe a message by using `MPI_Prob/MPI_Iprobe` routines with a support level ≥ `MPI_THREAD_SERIALIZED`, you must use instead the thread-safe routine that has been introduced starting from MPI 3.0, and the related "m" routines:

```
MPI_Mprobe( int source, in tag, MPI_Comm comm,
            MPI_Message *message, MPI_Status *status )

MPI_Improbe( int source, in tag, MPI_Comm comm, int *flag,
             MPI_Message *message,MPI_Status *status )
```

Once a message has been m-probed, it can not be matched by other probe o receive operation; it must instead be matched by either `MPI_Mrecv()` or `MPI_Imrecv()`.

Not I, not any one else can travel that road for you,

You must travel it for yourself.

It is not far, it is within reach,

Perhaps you have been on it since you were born and did not know,

Perhaps it is everywhere on water and on land.

W. Whitman,
Song of myself

Whatever your travel is, may it be gorgeous.
Good luck.
And thanks for all the fish.

that’s all, have fun