The difference between tasks and sections is in the time frame in which the code will execute. Sections are enclosed within the sections construct and (unless the nowait clause was specified) threads will not leave it until all sections have been executed:

[ sections ]

Thread 0: -------< section 1 >---->\*------

Thread 1: -------< section 2 >\*------

Thread 2: ------------------------>\*------

... \*

Thread N-1: ---------------------->\*------

Here N threads encounter a sections construct with two sections, the second taking more time than the first. The first two threads execute one section each. The other N-2 threads simply wait at the implicit barrier at the end of the sections construct (show here as \*).

Tasks are queued and executed whenever possible at the so-called task scheduling points. Under some conditions, the runtime could be allowed to move task between threads, even in the mid of their lifetime. Such tasks are called untied and an untied task might start executing in one thread, then at some scheduling point it might be migrated by the runtime to another thread.

Still, tasks and sections are in many ways similar. For example, the following two code

// sections

...

#pragma omp sections

{

#pragma omp section

foo();

#pragma omp section

bar();

}

...

// tasks

...

#pragma omp single nowait

{

#pragma omp task

foo();

#pragma omp task

bar();

}

#pragma omp taskwait

...

taskwait works very like barrier but for tasks - it ensures that current execution flow will get paused until all queued tasks have been executed. It is a scheduling point, i.e. it allows threads to process tasks. The single construct is needed so that tasks will be created by one thread only. If there was no single construct, each task would get created num\_threads times, which might not be what one wants. The nowait clause in the single construct instructs the other threads to not wait until the single construct was executed (i.e. removes the implicit barrier at the end of the single construct). So they hit the taskwait immediately and start processing tasks.

taskwait is an explicit scheduling point shown here for clarity. There are also implicit scheduling points, most notably inside the barrier synchronisation, no matter if explicit or implicit. Therefore, the above code could also be written simply as:

/ tasks

...

#pragma omp single

{

#pragma omp task

foo();

#pragma omp task

bar();

}

...

Here is one possible scenario of what might happen if there are three threads:

+--+-->[ task queue ]--+

| | |

| | +-----------+

| | |

Thread 0: --< single >-| v |-----

Thread 1: -------->|< foo() >|-----

Thread 2: -------->|< bar() >|-----

Show here within the | ... | is the action of the scheduling point (either the taskwait directive or the implicit barrier). Basically thread 1 and 2 suspend what they are doing at that point and start processing tasks from the queue. Once all tasks have been processed, threads resume their normal execution flow. Note that threads 1 and 2 might reach the scheduling point before thread 0 has exited the single construct, so the left |s need not necessary be aligned (this is represented on the diagram above).

It might also happen that thread 1 is able to finish processing the foo() task and request another one even before the other threads are able to request tasks. So both foo() and bar() might get executed by the same thread:

+--+-->[ task queue ]--+

| | |

| | +------------+

| | |

Thread 0: --< single >-| v |---

Thread 1: --------->|< foo() >< bar() >|---

Thread 2: --------------------->| |---

It is also possible that the singled out thread might execute the second task if thread 2

+--+-->[ task queue ]--+

| | |

| | +------------+

| | |

Thread 0: --< single >-| v < bar() >|---

Thread 1: --------->|< foo() > |---

Thread 2: ----------------->| |---

In some cases the compiler or the OpenMP runtime might even bypass the task queue completely and execute the tasks serially:

Thread 0: --< single: foo(); bar() >\*---

Thread 1: ------------------------->\*---

Thread 2: ------------------------->\*---

If no task scheduling points are present inside the region's code, the OpenMP runtime might start the tasks whenever it deems appropriate. For example it is possible that all tasks are deferred until the barrier at the end of the parallel region is reached.