OpenMP - PAR

API extension for C, C++. Can create threads, tasks and synchronize them.

Use a fork-join model: a master thread deploy a team of threads that joins a the end of a parallel region. The memory model only guaranteed consistency in certain points.

The compiler defines _OPENMP to perform conditional compilation/coding.

API Calls

```
//number of threads current team
int omp_get_num_threads();
//id of the thread [0-numThreads-1]
int omp_get_thread_num();
//sets the number of threads to be used in next parallel region
void omp_set_num_threads();
//returns the number of threads that could be used in a paralle region
int omp_get_max_threads();
//number of seconds since arbitrary point
double omp_get_wtime();
```

Part 1 : Basic constructs

Creating threads & accesing data: parallel

```
#pragma omp parallel [clauses]
```

A internal control variable (ICV) is used to determine the num of threads. By default all variables are marked as shared.

- Clauses:
  - num_threads(exp): Ignore ICV and sets the num of threads.
  - if(exp): if exp evaluates false -> the parallel only use 1 thread.
  - shared(var-list): All threads view the same var (not necessarily the same value).
  - private(var-list): All threads have a different var, initialy with undefined value.
  - firstprivate(var-list): Same as private, but initialized to the original value.
  - reduction(op:var-list): Defines the kind of join for a private var that will perform at the end of the parallel region. (op: +,-,*,|,||,&,&&,^, min,max).

Synchronization mechanisms: barrier

```
#pragma omp barrier
//all foo ocurrences after all bar occurrences
#pragma omp parallel
{
    foo();
    #pragma omp barrier
    bar();
}
```

Threads cannot proceed past a barrier point until all threads reach the barrier and all previously work is completed. parallel construct have an implicit barrier.

Synchronization mechanisms: critical

```
#pragma omp critical [(name)]
int x=1, y=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp critical(y)
    y++;
    #pragma omp critical(x)
    x++;
}
```

Provides a region of mutual exclusion: Only one thread can be working at any given time.

Synchronization mechanisms: atomic

```
#pragma omp atomic
update -> x+= 1, x= x - foo(), x[index[i]]++;
read -> value = *p;
write -> *p = value;
```

Ensures that a specific location is accessed atomically, avoiding simultaneous reading and writing threads. More efficient than critical.

Only protects if the expression is one of the specfied in the example.

Synchronization mechanisms: Locks

Through this API, we can use low-level synchornization

```
omp_lock_t lock;
void omp_init_lock(&lock); //initialize a lock
void omp_set_lock(&lock);
void omp_unset_lock(&lock);
void omp_test_lock(&lock); //won't block
void omp_destroy_lock(&lock);
```

Memory consistency: flush

OpenMP don't ensure all time consistency memory model (is a relaxed one). To force a var to read/write from/to the memory; otherwise it can read/write in the thread's temporary view.

```
#pragma omp flush(list)
```

Synchronization constructs have an associated flush operation.

Part 2: Worksharing

A worksharing construct defines a team of threads that execute a certain region of code. Is a better way than split the code with the threads ID.

Is less flexible than tasks, but have lower overhead.

Worksharing: Loop worksharing

```
#pragma omp for[clauses]
    for(int i = 0; i<exp; ++i);
#pragma omp parallel for[clauses]
```

The loop iterations are divided among the threads of the team. The loop iterations must be independet and the number must be knowed.

- Clauses:
  - private, firstprivate, reduction: same as parallel
  - schedule: determines wich it's are executed by each thread.
    - static -> the it space is broken in chunks with size N/num_threads.
    - static, N ->chunks of size N.
    - dynamic, N -> Threads grab chunks of N it's until all iterations have been executed (if no N -> N=1).
    - guided ->Variant of dynamic. The size of the chunk deceases as the thread grab iterations, but is al least of size N.

    The dynamic schedule have higher overhead but can solve the imbalance problems of the static schedule.
  - nowait: The implicit barrier at the end of the loop disapears.
  - collapse(n): Allows to distribute work from a set of 'n' nested loops.

- ordered(n): Specify sequential ordering in the execution of a block of statements in a set of 'n' nested loops. Also if the nested loop have cross-iterations dependences we need to use the clauses:
  - depend(sink:i-1) defines the wait point for the completion of the i-N
  - depend(source) defines the completion of the iteration i

Worksharing: Single construct

```
#pragma omp single [clauses]
    block
```

- clauses:
  - private, firstprivate, nowait

Only one thread of the team executes the block. There is an implicit barrier at the end.

Part 3: Task parallelism

A task is a work unit whose execution may be deferred. All the tasks in thre program waits to be executed in a task pool. One thread assigns tasks to the threads in a team.

Task creation

Each thread that encounters a task: packages the code and data and puts the task in the task pool. We need to use the parallel and the single construct. The threads (also the single one when finishes the task generation) cooperate to execute them.

```
#pragma omp task[clauses]
    block
```

- clauses:
  - shared, private, firstprivate.
  - if(exp): if the exp evaluates false, the task is suspended
  - final(exp): if he exp evaluates true, the task is immediately executed(named included task). All his child tasks also will be final.
  - mergeable: When be apply mergeable to a final construct, won't create a task and context
  - depend(in:vars): The task will be a dependent task of all previously generated that reference at least one of the vars in his inout or out list.
  - depend(out:vars) && depend(inout:vars) : The task will be dependent task of all previous task with at least one of the list item in inout, out or in list.

By default; Global vars are shared, Vars declared in the scope of a task are private.

Task synchronization: taskwait

Suspends the current task waiting on the completion of child tasks of the current task. This constructor is a stand-alone directive

```
#pragma omp task{}      //t1
#pragma omp task        //t2
{
    #pragma omp task{}  //t3
}
#pragma omp task{}      //t4
#pragma omp taskwait// Only t1, t2 && t4 are guaranteed to have finish.
```

Task synchronization: taskgroup

Suspends the current task at the end of structured block waiting on completion of child tasks of the current task and their descendent tasks.

```
#pragma omp task {}     // T1
#pragma omp taskgoup
{
    #pragma omp task    // T2
    {
        #pragma omp task {} // T3
    }
    #pragma omp task {} // T4
} //Only T2, T3 && T4 are guaranteed to have finish.
```

Taskloop construct

This constructor convert one o more associated loop into OpenMP tasks. There is a implicit taskgroup for synchronization.

```
#pragma omp taskloop [clauses]
    for(int i = 0; i < A; ++i)
```

- Clauses:
  - shared, private, firstprivate, if, final, mergeable
  - grainsize(n)
  - num_tasks(n)
  - collapse(n)
  - nogroup: Override the implicit taskgroup construct.
- No existeix una opcio per fer el 'reduction', cal fer-ho a ma.

Examples

Example: Pi v1

In this example we have a 'Data race' with sum variable (by default shared). We can use atomic or critical to avoid it. Another option is to use reduction + for the variable.

```
#include <omp.h>
#define NUM_THREADS 2
double step;
static long num_steps = 100000;

void main(){
    int i, id;
```

```
    double x, pi, sum =0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x,i,id)
    {
        id = omp_get_thread_num();
        for(i=id+1; i<= num_steps; i=i+NUM_THREADS){
            x = (i-0.5)*step;
            #pragma omp critical
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

Also, we can use in the parallel version the 'reduction(+:sum)' to avoid use the omp critical.

Example: Pi v2

In this version, we use the for worksharing constructor

```
#include <omp.h>
#define NUM_THREADS 2
double step;
static long num_steps = 100000;

void main(){
    int i, id;
    double x, pi, sum =0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for private reduction(+:sum)
    {
        id = omp_get_thread_num();
        for(i=id+1; i<= num_steps; i=i+NUM_THREADS){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```

Example: Pi v3

In this example we use the task parallelism to define tasks to the diferent elements of a list. The tasks didn't have dependences.

```
int main(){
    List l;
    #pragma omp parallel
    #pragma omp single
    travese_list(l);
}
void travese_list(List l){
    Element e;
    for(e = l->first; e; e=e->next){
```

```
        #pragma omp task
        process(e);
    }
}
```