CPDS: OpenMP cheatsheet

Directives syntax

In C/C++, OpenMP is used through compiler directives. The syntax is ignored if the compiler does not know OpenMP.

```c
#pragma omp construct [clauses]
```

Memory model

OpenMP defines a relaxed memory model. Threads can see different values for the same variable (Variables can be shared or private to each thread). Memory consistency is only guaranteed at specific points.

Constructs

The `parallel` construct

```c
#pragma omp parallel [clauses]
    // structured block...
```

In that sense, directives work just like `if` statements. If the conditional block of the `if` is just one block, it can be written just like this:

```c
if (something) doIt();
```

This syntax can be used for *joining* different blocks even though they are not in the same line, as shown below. The `for` loop will only run if the condition `something` is true, even though we didn't enclose the conditional block between curly braces ({ }).

```c
if (something)
for (;;) {
    // do something...
}
```

The same goes for OpenMP directives. The following two pieces of code are equivalent.

```c
#pragma omp parallel
{
    for (;;) {
        // do something...
    }
}
```

```c
#pragma omp parallel
for (;;) {
    // do something...
}
//
```

Number of threads

The `nthreads-var` ICV (internal variable) is used to determine the number of threads to be used for parallel regions. It's a list of positive integer values. For each occurrence of `parallel`, the first element is popped from the list.

The value of the variable can be set with the environment variable `OMP_NUM_THREADS`, through the function `omp_set_num_threads`, or by defining the directive `num_threads`.

```c
unsigned int N = ...;
omp_set_num_threads(N);
#pragma omp parallel num_threads(N)
```

The `if` clause

It can be used to conditionally run regions parallelly. When the condition evaluates to false, the region is run on one only thread.

```c
int iterations = ...;
#pragma omp parallel if (iterations > 10)
```

Example: amount of threads

```c
void main() {
    #pragma omp parallel
    // ...
    omp_set_num_threads(2);
    #pragma omp parallel
    //...
    #pragma omp parallel num_threads(random() % 4 + 1)
    ↪   if(0)
    //...
}
```

How many threads are used in each parallel region? The first region uses the default amount of threads. The second region will use 2 threads. The third region will not run parallelly, as the `if` will always evaluate to false.

Data-sharing attributes

Shared

When a variable is marked as `shared`, all threads use the same variable, this is, access the same location in memory. By default, variables are implicitly shared.

Private

When a variable is marked as `private`, it means that each thread has a different variable with an originally undefined value that can be accessed without any kind of synchronization.

Firstprivate

Whan a variable is marked as `firstprivate`, it means that each thread has a different variable initialized to the original value that can be accessed without any kind of synchronization.

Example: Data-sharing

```c
int x = 1;
#pragma omp parallel XXXXXX num_threads(2)
{
    x++;
    printf("%d\n", x);
}
printf("%d\n", x);
```

What gets printed on screen if `XXXXXX` is `shared(x)`, `private(x)` or `firstprivate(x)`? Note that the final `printf` falls out of the `parallel` construct.

When `x` is shared, it is difficult to know. A race condition will occur. One of the threads will run `x++` before the other. So, the two first lines will be a 2 and a 3 (with undetermined order), followed by a 3.

When `x` is private, the result is undefined, as the variables that the threads will be able to access are undefined.

When `x` is firstprivate, it will print 2 twice, because each thread modifies a private variable initialized to 1. The original variable remains untouched, so it's just where it was initialized.

Try it yourself

Compile the source `data-sharing.c` with `make data-sharing` and run it with `./data-sharing`. Because of the compilation options, the uninitialized variables will be set to 0 by default. Thus, the example with private variables will print 1 three times.

Example: Computation of π

An approximation of π can be calculated with the following sequential code.

```c
1  static long num_steps = 100000;
2  double step;
3
4  void main() {
5      int i;
6      double x, pi, sum = 0.0;
7
8      step = 1.0 / (double) num_steps;
9
10     for (i = 1; i <= num_steps; i++) {
11         x = (i - 0.5) * step;
12         sum = sum + 4.0 / (1.0 + x * x);
13     }
14     pi = step * sum;
15 }
```

Say the goal is to parallelize the code. The `for` loop can be parallelized. How would the `#pragma` construct affect the data sharing? Variables `i`, `x`, and `sum`, are accessed and written in the loop, and, by default, these variables are shared. Not having a proper data sharing design would alter the result.

As opposed to the other variables, `sum` is only read just before writing it by means of an addition. Additions are commutative $(3+2=2+3)$ and associative $((2+3)+4=2+(3+4))$, so `sum` can be shared. If `sum` was to be private, the initial value would be undefined, this is, the sum would not start from 0. If it was to be firstprivate, the results of each thread would need to be collected somehow after the fact.

Meanwhile, different values of `i` and `x` can be read in crucial moments of the calculation. Specifically, the addition in line 12 may access a different value of `x` than the one calculated by the same thread a line before.

The solution would be to make `i` and `x` private, as shown in the following code snippet:

```c
#pragma omp parallel private(i, x)
for (i = 1; i <= num_steps; i++) {
    x = (i - 0.5) * step;
    sum = sum + 4.0 / (1.0 * x * x);
```

```
}
pi = step * sum;
```

Recall that only the `for` loop will be parallelized. The last assignment (`pi = step * sum`) is run sequentially.

Some API calls

- `int omp_get_num_threads()` returns the number of threads in the current team.
- `int omp_get_thread_num()` Returns the id of the thread in the current team. Goes from 0 to `omp_get_thread_num() - 1`
-

Thread synchronization

OpenMP follows a shared memory model. Threads communicate by sharing variables. Unintended sharing of data may cause race conditions. Threads need to synchronize to impose some ordering in their sequence of actions.

Thread barrier

```
#pragma omp barrier
```

Threads cannot proceed past a barrier point until all the parallel threads reach the barrier. Some constructs, such as `parallel`, have an implicit barrier at the end.

```
1 #pragma omp parallel
2 {
3     foo();
4     #pragma omp barrier
5     bar();
6 }
```

The explicitly defined barrier in line 4 forces all threads to finish running `foo()` before running `bar()`. At the same time, the end of the parallel region at line 6 implicitly means that all threads must finish running `bar()` before the code keeps running sequentially.

Exclusive access: critical construct

```
#pragma omp critical [(name)]
    // structured block
```

Makes a parallel region accessible to only one thread at any given time, this is, mutual exclusion. Unless explicitly named, all critical regions are the same.

```
#pragma omp parallel
{
    foo();
    #pragma omp critical
    bar();
    #pragma omp critical
    baz();
}
```

In the example above, there are two different critical regions. Nevertheless, as none of them are named, only one thread can run either one of the regions. The two of them will never be run at the same time. This can be fixed by naming either of the regions, as shown below.

```
#pragma omp parallel
{
    foo();
    #pragma omp critical part1
    bar();
    #pragma omp critical part2
    baz();
}
```

Exclusive access: atomic construct

```
#pragma omp atomic [ update | read | write ]
    // expression
```

The construct ensures that a specific storage location is accessed in a mutually exclusive way, avoiding the possibility of multiple, simultaneous reading and writing threads. It is usually more efficient than a `critical` construct. There are three types of atomic accesses:

- Updates: `x++, x -= foo()`. An operation that reads from and writes in the same memory space. (These are those operations that can be represented as `+=`, `--`, `<<=` or `/=`, among others)
- Reads: `value = *p`. In this case, the value of `p` is being directly read.
- Writes: `*p = value`. In this case, the value of `p` is being directly written on.

Reduction clause

```
#pragma omp parallel [...] reduction(operator:variable)
    // block...
```

Reduction is a pattern where all threads accumulate values into a single variable. Valid operators are `+`, `-`, `*`, `|` (bitwise OR), `||` (logical OR), `&` (bitwise AND), `&&` (logical AND) and `^` (bitwise XOR). The compiler implicitly creates a properly initialized private copy of the variable for each thread and, at the end of the region, it takes care of safely updating it with the partial solutions.

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

In the example above, each thread will have a private copy of `sum`, probably initialized to 0. Threads will calculate the sum of the respective iterations they run, depending on their ID. As additions are commutative and associative, the results can be aggregated after the barrier without any loss of information.

Locks

OpenMP provides lock primitives for low-level synchronization. Locks work much like critical regions. They are acquired before entering a mutual exclusion region and must be released afterwards.

```
1  #include <omp.h>
2  void foo() {
3      omp_lock_t lock;
4      omp_init_lock(&lock);
5      #pragma omp parallel
6      {
7          omp_set_lock(&lock);
8          // mutual exclusion region
9          omp_unset_lock(&lock);
10     }
11     omp_destroy_lock(&lock);
12 }
```

Note, in the example above, that locks need initialization and destruction, which is taken care of with the functions on lines 4 and 11, respectively.

Memory consistency

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

It enforces consistency between the temporary view and memory for those variables in the list. Synchronization constructs (implicit or explicit) have an associated flush operation.

Loop parallelism

The worksharing concept

Worksharing constructs divide the execution of a code region among the members of a team. Threads cooperate to do some work. It is a better way to split work than thread IDs, and has a lower overhead than tasks, even though it's less flexible.

The `for` construct

```
#pragma omp for [clauses]
    for (init-expr; test-expr; inc-expr)
```

The iterations of the loops will be divided among the threads. Loop iterations must be independent and it must follow a shape that allows induction of amount of iterations. Valid types for inductions are integers, pointers and random access iterators (C++). This inducted variable is private.

```
void main() {
    int i, id;
    double x, pi, sum;

    step = 1.0 / (double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i = 1; i <= num_steps; i++) {
        x = (i - 0.5) * step;
        sum  = sum + 4.0 / (1.0 + x * x);
    }
    pi = sum * step;
}
```

The `for` construct automatically detects `i` as the variable for iteration control. OpenMP makes it private and then distributes iterations among the threads. `x` still must be marked as private, race conditions can still occur. `sum` will have a different copy for each of the threads, initialized to 0. Thanks to the `reduct` clause, all the local results of `sum` per thread will be added after the synchronization.

The `schedule` clause

This clause determines which iterations are executed by each thread. If no clause is present, the implementation should take care of this. There are several options:

- `static`. The iteration space is broken in chunks of size $\frac{N}{\text{num\_threads}}$. This is, the iterations are evenly divided across the threads. Chunks are assigned to threads in Round-Robin fashion.
- `static,N` (interleaved). The iteration space is broken in chunks of size N. Then this chunks are scheduled to threads in Round-Robin fashion.

The overall characteristics of static scheduling are low overhead, (usually) good locality and the possibility of load imbalance problems.

- `dynamic,N`. Threads dynamically grab chunks of N iterations until all iterations have been executed. $N = 1$ by default.
- `guided,N`. The size of the chunks decreases as the threads grab iterations, but it is at least of size N, $N = 1$ by default.

These dynamic schedules result in higher overhead, not very good locality (usually) but they can solve imbalance problems.

The `nowait` clause

The `nowait` clause removes the implicit `barrier` from the end of a parallel region. This allows to overlap the execution of non-dependent loops/tasks/worksharings.

```
#pragma omp for nowait
for (i = 0; i < n; i++)
    v[i] = 0;
#pragma omp for
for (i = 0; i < n; i++)
    a[i] = 0;
```

In the example above, the work of the second loop is independent from the work of the first loop, hence there is no need for the threads to wait for synchronization after running the first loop. The `nowait` construct allows this.

```
#pragma omp for schedule(static, 2) nowait
for (i = 0; i < n; i++)
    v[i] = 0;
#pragma omp for schedule(static, 2)
for (i = 0; i < n; i++)
    a[i] = v[i] * v[i];
```

In the example above, a static scheduling policy is defined. 2 different chunks will be created with the halves of the iterations of each `for` loop. So, even though the second loop is directly dependent on the results of the first loop, each of the threads will be dependent only of the elements calculated in the iterations run by themselves. The `nowait` construct allows the threads to continue running when they finish with the first loop.

The `collapse` clause

The `collapse` clause allows to distribute work from a set of n nested loops. The loops must be perfectly nested. The nest must traverse a rectangular iteration space.

```
#pragma omp for collapse(2)
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        foo(i, j);
```

In the example above, the loops of i and j are folded and iterations distributed among all threads, and both variables are privatized.

The `single` construct

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

The construct makes the structured block to be run in only one thread. The clauses can be `private`, `firstprivate` and `nowait`. There is an implicit `barrier` at the end.

Task parallelism

Task parallelism model

Tasks are work units whose execution may be deferred, but they can also be executed immediately. Threads, separated in teams, cooperate to execute them.

Task creation

Parallel regions create tasks. One implicit task is created and assigned to each thread. Each thread that encounters a `task` construct packages the code and data and creates a new explicit task.

Explicit task creation

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

Where some possible clauses are `shared`, `private`, `firstprivate`, `if(expr)`, `final(expr)` and `mergeable`.

```
void traverse_list(List l) {
    Element e;
    for (e = l->first; e; e = e -> next) {
        #pragma omp task
        process(e); // e is firstprivate by default
    }
}
```

The code above defines a task block, the `process` function. But tasks are useless if they are not defined within a parallel region. Let's complete the code.

```
List l;

#pragma omp parallel
traverse_list(l);

void traverse_list(List l) {
    // ...
}
```

In the code above the `traverse_list` function is going to be run by as many threads as defined. All of the threads will run all of the tasks, so the traversal of the list will be calculated by all threads. Execution is not actually parallelized.

```
List l;

#pragma omp parallel
#pragma single
traverse_list(l);

void traverse_list(List l) {
    // ...
}
```

With the addition of the construct `single`, only one of the threads will proper run the `traverse_list` function. This thread will create the tasks. The rest of threads (and the first thread, once the task generation is complete) will run the tasks in parallel.

Default task data-sharing attributes

When no data clauses are specified, global variables are shared, variables declared within the scope of a task are private, and the rest are firstprivate, except when a `shared` attribute is inherited.

```
int a;
void foo() {
    int b, c;
    #pragma omp parallel shared(b)
    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
        }
    }
}
```

In the code above, the variables are:

- `a` is global, and so shared by default.
- `b` is firstprivate.
- `c` is shared because, in the context of the task, it is effectively global.
- `d` is firstprivate, because it is declared within the parallel region and it its attribute is not explicitly defined.
- `e` is private, because it is declared withing the scope of the task.

Immediate task execution

The `if` clause

When an `if` clause is present on a task construct, and its expression evaluates to `false`, an undeferred task is generated. The encountering thread must suspend the current task region, the execution of which cannot be resumed until the generated task is completed. This allows implementations to optimize task creation.

The `final` clause

If the expression of a `final` clause evaluates to `true`, then the generated task and its children will be final and included. Execution of included tasks is done immediately after the generating task. So, all tasks created within a `final` region will be run sequentially and immediately, and if more tasks are created, they will also be final and have the same final task generating capabilities.
When a `mergeable` clause is present on a task construct, and the generated task is an included task, the implementation may generate a merged task instead (i.e. no task and context creation for it).

Task synchronization

There are two types of task barriers:

- `taskwait`: Suspends the current task waiting on the completion of child tasks of the current task. This construct is stand-alone.
- `taskgroup`: Suspends the current task at the end of a structured block waiting on completion of child tasks of the current task and their descendent tasks.

`taskwait`

```
#pragma omp task {}     // T1
#pragma omp task        // T2
{
    #pragma omp task {} // T3
}
#pragma omp task {}     // T4

#pragma omp taskwait
```

With the code above, only tasks 1, 2 and 4 are guaranteed to have finished after `taskwait`, as it only guarantees the completion of child tasks of the current task.

```
int fib(int n) {
    int i, j;

    if (n < 2) return n;

    #pragma omp task shared(i) final(n <= THOLD) mergeable
    i = fib(n - 1);
    #pragma omp task shared(j) final(n <= THOLD) mergeable
    j = fib(n - 2);

    #pragma omp taskwait
    return i + j;
}
```

`taskgroup`

```
#pragma omp task {}         // T1
#pragma omp taskgroup {
    #pragma omp task        // T2
    {
        #pragma omp task {} // T3
    }
    #pragma omp task {}     // T4
}
```

With the code above, only tasks 2 to 4 are guaranteed to have finished after the `taskgroup` clause, as it guarantees the completion of all child tasks, and their child tasks recursively.

Data sharing inside tasks

In addition one can use `critical` and `atomic` to synchronize the access to shared data inside tasks.

```
void process(Element e) {
    // ...
    #pragma omp atomic
    solutions_found++;
    // ...
}
```

Task dependencies

Dependence between sibling tasks can be expressed as follows:

```
#pragma omp task    [depend (in : var_list)]
                    [depend (out : var_list)]
                    [depend (inout : var_list)]
```

The exact dependence between tasks is inferred from the dependence type and the items in the variable list. This list may include array sections.

- Tasks with the `in` dependence-type will be dependent of all previously generated sibling tasks that reference, at least, one of the items in the variable list in an `out` or `inout` dependence-type list.
- Tasks with the `out` or `inout` dependence-types will be dependent on all the previously generated tasks mentioning, at least, one of the items in the variable list.

```
#pragma omp parallel private(i, j)
#pragma omp single
{
    for (i = 1; i < n; i++) {
        for (j = 1; j < n; j++) {
            #pragma omp task
                    depend(in : block[i - 1],
                    ↪   block[i][j - 1])
                    depend(out : block[i][j]
            foo(i, j);
        }
    }
}
```
