OpenMP Pragma Cheat-Sheet

Run the code: `g++ -fopenmp prog.cpp -o prog`

`#include <omp.h> // This is the header`

Pragma	Purpose
`#pragma omp parallel`	Starts a parallel region where multiple threads execute the enclosed code concurrently.
`#pragma omp for`	Distributes loop iterations among threads within a parallel region.
`#pragma omp parallel for`	Combines parallel and for to create a parallel loop directly.
`#pragma omp single`	Specifies a block of code to be executed by only one thread.
`#pragma omp critical`	Defines a critical section where only one thread can execute the block at a time.
`#pragma omp barrier`	Synchronizes all threads, making each wait until all have reached the barrier.
`#pragma omp master`	Specifies a block of code to be executed only by the master thread.
`#pragma omp sections`	Divides work into separate sections to be executed by different threads.
`#pragma omp section`	Defines a single section within a sections block.
`#pragma omp task`	Creates a task that can be executed by any thread in the team.
`#pragma omp atomic`	Ensures atomic updates to a variable, preventing race conditions with minimal overhead.
`#pragma omp reduction`	Performs a reduction operation (e.g., sum, product) across all threads.

Common Clauses in OpenMP Pragmas

OpenMP pragmas can include various clauses to control the behavior of the parallel regions and loops. Some common clauses include:

- **`private(variable)`**: Each thread has its own instance of the variable.
- **`shared(variable)`**: The variable is shared among all threads.
- **`default(none)`**: Requires explicit specification of the data-sharing attributes for all variables.
- **`firstprivate(variable)`**: Each thread has its own instance of the variable, initialized with the value before the parallel region.
- **`lastprivate(variable)`**: The value of the variable from the last iteration is copied back to the original variable after the parallel region.
- **`schedule(kind, chunk_size)`**: Controls how loop iterations are divided among threads. Common kind values include static, dynamic, guided, and auto.

  You can schedule in two ways, one is clause-way:

  ```
  #pragma omp parallel for schedule(static, chunk_size)
  #pragma omp parallel for schedule(dynamic, chunk_size)
  #pragma omp parallel for schedule(guided, chunk_size)
  ```

  Other way is like mentioning as function:

  ```
  omp_sched_t sched_type // is the parameter
  (takes values omp_sched_static/omp_sched_dynamic/..)

  omp_set_schedule(sched_type, chunk_size)
  ```

  - **`static`**: Iterations are divided into chunks of chunk_size and assigned to threads in a round-robin fashion. (`omp_sched_static`)
  - **`dynamic`**: Threads request new chunks as they finish their current ones. (`omp_sched_dynamic`)
  - **`guided`**: Similar to dynamic but with decreasing chunk sizes. (`omp_sched_guided`)
  - **`auto`**: The scheduling decision is left to the compiler/runtime. (`omp_sched_auto`)
- **`collapse(n)`**: Merges n nested loops into a single loop for parallelization.
- **`reduction(operator:variable)`**: Specifies a reduction operation on a variable.

How to use Pragma for recursion: (Sample program: nth fibbonacci)

```
#include <omp.h>
#include <iostream>
```

```
int fib(int n) {
    int left, right;
    if (n < 2) {
       return n;
    } else {
        #pragma omp task shared(left)
        left = fib(n - 1);

        #pragma omp task shared(right)
        right = fib(n - 2);

        #pragma omp taskwait
        return left + right;
   }

}

// Main

#pragma omp parallel
{
   #pragma omp single
   result = fib(n)
}
```

Some of the Pragma functions:

```
int omp_get_num_threads(); // to get number of threads (N)
int omp_get_thread_num(); // to get ID of the thread (0 to N – 1)
void omp_set_num_threads(int N); // to set number of threads to run
```

Plotting graphs using GNUPlot:

- Save the metrics using file handling operations to a textfile/csv file.
- Time the code using `#include <ctime>` header.

```
std::ofstream fp("metrics.txt");

double start_time, end_time;
start_time = omp_get_wtime();
#pragma omp parallel for
{
      // stmts
}
end_time = omp_get_wtime();
fp << n_value << " " << (end_time – start_time) << "\n";
fp.close();
```

GNUPlot code:
<Example: Plotting 2 curves (size vs serial, parallel time) using GNUPlot>

```
set terminal png size 800,600
set output 'performance_plot.png'
set title "Performance: Size vs Time"
set xlabel "Size (N)"
set ylabel "Time Taken (s)"
set gridplot 'metrics.txt' using 1:2 with linespoints title 'Serial Execution Time' \
                           using 1:3 with linespoints title 'Parallel Execution Time'
```

To get graph, type `gnuplot -p plot.gp` in terminal

Sample code (Dot product parallelization):

```
#include <omp.h>
#include <iostream>
#include <vector>

int main() {
    const int N = 100000;  // Size of the vectors
    std::vector<double> a(N, 1.0);  // Vector a initialized with 1.0
    std::vector<double> b(N, 2.0);  // Vector b initialized with 2.0

    double dot_product = 0.0;


    #pragma omp parallel for reduction(+:dot_product)
    for (int i = 0; i < N; i++) {
       dot_product += a[i] * b[i];
    }

    std::cout << "Dot product: " << dot_product << std::endl;
    return 0;
}
```