PAR Laboratory Assignment

Lab 2: Brief tutorial on OpenMP programming model

Aleix Boné
Alex Herrero
par2109

2020-03-27

1 OpenMP questionnaire

A) Parallel regions

1.hello.c:

1. 24 since the boada server has 24 cores so *OpenMP* defaults to 24 threads.
2. If we specify the number of threads that *OpenMP* can use with the environment variable `OMP_NUM_THREADS=4` we will only see 4 `"Hello World!"` messages.

2.hello.c:

1. Since the variable `id` is shared by default, there are times when the number displayed are not correct. (There are pairs of Hello that share the same number or numbers that don't appear). To fix this we have to add the clause `private(id)` to our directive.
2. The lines not always are printed in the same order and sometimes they appear inter-mixed since the execution is parallel and some threads might end before the others.

3.howmany.c:

1. There are printed 20 `"Hello world ..."` lines. The first Hello world is printed 8 times since there are 8 threads defined with the environment variable `OMP_NUM_THREADS`. The second Hello world runs 2 times for the first iteration of the loop where we set the number of threads with the function `omp_set_num_threads(i)` and 3 for the second iteration. The third parallel runs 4 times since we specify 4 threads on the pragma. The last parallel region prints 3 times since the call `omp_set_num_threads(3)` set the threads to 3. This makes a total of: $8 + 2 + 3 + 4 + 3 = 20$.
2. Inside the parallel region it returns the number of threads executed in parallel, outside it returns 1 since there is only on thread.

4.datasharing.c:

1. With the attribute `shared` we got that `x` is usually 120 but sometimes it gets lower values because we have a race condition since += reads the value and then writes to it. With `private` attribute `x` is always 5 since it's private each thread has its own value of x that is independent from the `x` of the main thread, so `x` is never updated. With `firstprivate` it's also 5 since its the same as `private` but with the only difference that the variable `x` of each thread is initialized with the value on the main thread and with `private` it's not. Finally, with `reduction` it's always 125. Which is $5 + 1 + 2 + \cdots + 16$.

B) Loop parallelism

1.schedule.c:

`static`: The threads get assigned iterations is chunks of 3 ($12/4 = 3$) in order: the first thread gets iterations 0 to 2, the seconds gets 3 to 5 an so on.

`static, 2`: The threads get assigned iterations in chunks of 2 in order: the first thread gets iterations 0 and 1, the second 2 and 3 and so on. Since there are not enough threads to run all the 12 iterations in parallel in chunks of 2, the first and second thread also get assigned the iterations 8,9 and 10,11 respectively.

`dynamic, 2`: The threads get assigned iterations in chunks of 2 without any other. The order will depend on which thread is available to process the chunks as they are needed.

`guided, 2`: With guided scheduling the threads get assigned iterations initially in chunks of 2 but the size of the chunks increases throughout the loop.

2.nowait.c:

1. Any thread can get any of the iterations because the scheduling is dynamic so there is no order guaranteed, the only thing we can know is that the first thread will be assigned to the first iteration of the loop.
2. If we remove the `nowait`, the iterations must finish before the loop continues, so Loop 1 will always output before Loop 2.
3. If we change `dynamic` to `static`, the order of assignment of the threads will be the same, even with the `nowait` clause. And only 2 threads will be used since the for loops have 2 iterations. The output will always be:

```
Loop 1:  thread (0) gets iteration 0
Loop 1:  thread (1) gets iteration 1
Loop 2:  thread (0) gets iteration 2
Loop 2:  thread (1) gets iteration 3
```

3.collapse.c:

1. As shown in the next table, the thread 0 execute the first four iterations, the thread 1 executes the next three consecutive iterations and so on until the thread number 7 executes the three last iterations of the loop.
2. Without `collapse` it's not correct, not all indices of the matrix are shown, because `j` is declared before the pragma and therefore it's shared among the threads which. We should add the `private(j)` clause so that the different threads don't interfere with one another.

C) Synchronization

1.datarace.c:

1. We tried 100 executions of the program with:

   `for i in {1..100}; do ./1.datarace; done | grep Sorry -c`

   and we got 100 wrong 0, correct. The program almost always will produce the wrong output. This happens because there is a data race when reading and updating the value of the shared variable `x`.

2. To make it correct we must make sure that the reads and writes to `x` of the threads are properly synchronized. We can do that by replacing the clause `shared(x)` by `reduction(+: x)`. Another option is to add the directive `#pragma omp atomic` or `#pragma omp critical` before the instruction `x++`.

2.barrier.c:

1. We can predict that all threads will output first their going to sleep messages and since the time it takes to display the message is much less than the time the threads sleep and we have enough threads they fill most certainly output the message of waking up after all the threads have printed their sleep messages, although there is no guarantee. Once they *all* threads print their wake up messages, the barrier will unlock and they will display the *We are all awake!* message, this time there is a barrier so its guaranteed that no thread will reach the *We are all awake!* message before all threads have reached the barrier.

   It seems there is no specific order in which the threads exit the barrier although in our experiments, most of the time, the last thread that outputs the *wakes up and enters barrier* message is the first to exit it.

3.ordered.c:

1. The order of the *Outside* messages is non-deterministic since it depends on the dynamic scheduling of the threads. In the other hand, the order of the *Inside* messages is always the same relative to themselves and they follow the order that the loop would have if executed sequentially (0, 1, 2, ..., N).

2. If we modify the clause `schedule` to include a chunk size of 2 like so: `schedule( dynamic, 2)` the tasks will get assigned 2 consecutive iteration of the loop.

D) Tasks

1.single.c:

1. Since we have the `nowait` clause on our `single` directive, all four threads are assigned the single clauses of the loop without waiting for them to finish. They appear to be executed in burst since although we have `nowait`, we only have 4 threads so they all run 4 tasks, sleep for 1 second, they finish roughly at the same time and get assigned to the 4 next iterations.

2.fibtasks.c:

1. Because there is no `#pragma omp parallel` directive and therefore no threads are created.
2. To execute it correctly in parallel we have to add `#pragma omp parallel` and `#pragma omp single` and the clause `firstprivate(p)` As shown below:

```
66  #pragma omp parallel
67  #pragma omp single
68  while (p != NULL) {
69      printf("Thread %d creating task that will compute %d\n",
        ↪  omp_get_thread_num(), p->data);
70      #pragma omp task firstprivate(p)
71          processwork(p);
72      p = p->next;
73  }
```

3.synchtasks.c:

1. As we can see in Figure ?? `foo1` and `foo2` must be done before `foo4` and at the same time `foo4` before `foo5`.

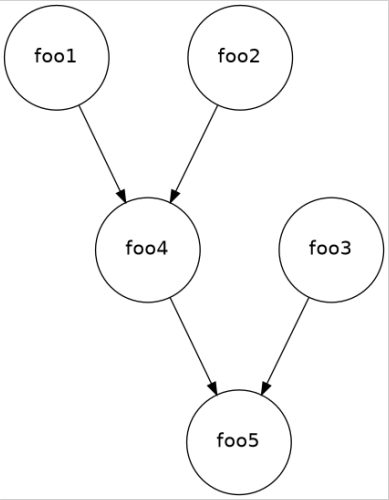


Figure 1: Task dependence graph of `3.synchtasks.c`

2. To make the program execute properly using only `taskwait`, we have to put a one `taskwait` before every task that has a dependency, in this case we put them before `foo4` and `foo5`:

```
41 int main(int argc, char *argv[]) {
42     #pragma omp parallel
43     #pragma omp single
44     {
45             printf("Creating task foo1\n");
46             //#pragma omp task depend(out:a)
47         #pragma omp task
48             foo1();
49             printf("Creating task foo2\n");
50             //#pragma omp task depend(out:b)
51         #pragma omp task
52             foo2();
53             printf("Creating task foo3\n");
54             //#pragma omp task depend(out:c)
55         #pragma omp task
56             foo3();
57             printf("Creating task foo4\n");
58             //#pragma omp task depend(in: a, b) depend(out:d)
59         #pragma omp taskwait
60         #pragma omp task
61             foo4();
62             printf("Creating task foo5\n");
63             //#pragma omp task depend(in: c, d)
64         #pragma omp taskwait
65         #pragma omp task
66             foo5();
67     }
68     return 0;
69 }
```

4.taskloop.c:

1. With `grainsize` each task gets 6 iterations. With `num_tasks` each task gets 3 iterations, this is due to the fact that there are 4 threads (which is less than 5) and 12 iterations, so each task gets $\frac{12}{3} = 4$ iterations of the loop.
2. If we add the `nogroup` clause, the taskloop does not create an implicit `taskgroup` and therefore the program does not wait for the execution of the first loop to finish before moving to the next step.

2 Observing overheads

2.1 Synchronisation overheads

	1 thread	4 threads	8 threads
`Critical`	4.358014s	37.993402s	35.054946s
`Atomic`	1.820260s	6.537028s	6.638720s
`Reduction`	1.838303s	0.475096s	0.251802s
`Sumlocal`	1.844493s	0.482096s	0.249314s
Sequential version	1.793754s		

Table 1: Execution times of 100.000.000 iterations

- `critical`: The `critical` region protects access to the sum ensuring exclusive access to it. This means that it can only be accessed by one thread at a time and that we have to perform a synchronization for every iteration of the loop. This introduces massive overhead while giving no real benefit since most of the code is affected by this region an thus is not parallel. In the table we can appreciate how much more time it takes (even in the 1 thread version) compared to the sequential version and the rest of the parallel versions.

- `atomic`: This is similar to the `critical` region since it protects the same variable, the main difference is that the synchronization is performed at the memory access at a hardware level which allows for a smaller sequential region to synchronize. We can see in the table that is much better than its `critical` counterpart but it still introduces too much overhead when having to synchronize multiple threads.

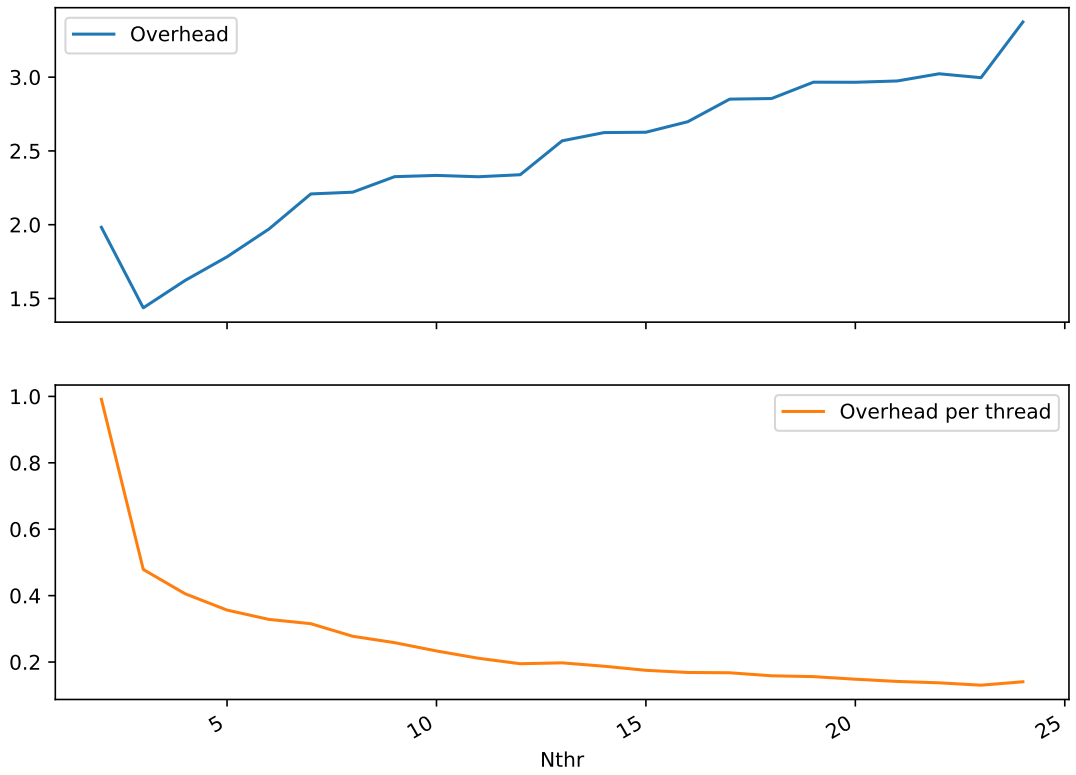
- `reduction`: In this version the synchronization only occurs once all the tasks generated by the for loop finish and then their values for *sum* which where independent for each thread are added together. This allows the parallel execution of all the tasks generated by the `for` loop since there is no dependence between the iterations. This is reflected on the results shown on the table in which we achieve an Speedup of 7.12 with 8 threads.

- `sumlocal`: This version of the code performs the same operations that `reduction`, but instead of using the built-in directive of *OpenMP* the code is modified to perform the reduction manually. The results obtained are very similar to the `reduction`, this shows that both versions are equivalent.

2.2 Thread creation and termination

As we can see in figure **??** the overhead of creating/terminating threads increases with the number of threads in a mostly linear way, the main exception is the execution with 2 threads where there is more overhead than with 3 to 5 threads. We can also see that the overhead per thread decreases and seems to stabilize at around 0.14 seconds.

Figure 2: Overhead plots for `pi_omp_parallel 1 24`



2.3 Task creation and synchronization

Figure ?? shows the overhead of task creation and synchronization. There is clearly a linear trend between the number of tasks created and the overhead of the program, which is further reinforced by the fact that the overhead per task is nearly the same throughout the different number of tasks (the y axis ranges from 0.122 to 0.130. As with the previous section, the overhead per task is more noticeable at the very first few values.

Figure 3: Overhead plots for `pi_omp_tasks 10 1`