Second deliverable

Par4113: Jordi Bosch, Dean Zhu

April 2018

Contents

1 OpenMP **2**
1.1 Basics 2
1.1.1 `hello.c` 2
1.1.2 `hello.c` 2
1.1.3 `how_many.c` 2
1.1.4 `data_sharing.c` 2
1.1.5 `parallel.c` 2
1.1.6 `datarace.c` 3
1.1.7 `barrier.c` 3
1.2 Worksharing 3
1.2.1 `for.c` 3
1.2.2 `schedule.c` 3
1.2.3 `nowait.c` 3
1.2.4 `collapse.c` 3
1.2.5 `ordered.c` 4
1.2.6 `doacross.c` 4
1.3 Tasks 4
1.3.1 `serial.c` 4
1.3.2 `parallel.c` 4
1.3.3 `taskloop.c` 5

2 Parallelization Overheads **5**
2.1 Overhead associated with a parallel region (fork and join) 5
2.2 Overhead associated with the creation of a task and its synchronization . . 6
2.3 Overhead comparing task and taskloop 7
2.4 Overhead in Critical regions 7
2.5 Overhead in Atomic regions 8
2.6 Overhead in False Data sharing 8

1 OpenMP

1.1 Basics

1.1.1 `hello.c`

1. 24, As many threads as we have
2. `export OMP_NUM_THREADS=96`

1.1.2 `hello.c`

1. No, there is a datarace condition so we should add private(x)
2. They aren't always in the same order, and might be intermixed

1.1.3 `how_many.c`

1. 16
2. between 16 and 19

1.1.4 `data_sharing.c`

1. When using the shared directive the value is indetermined, when using private is 5 and using firstprivate is 71.
2. We should use the reduction directive to add all x values.

1.1.5 `parallel.c`

1. 26, all iterations between the thread id and 8
2. We should increase i by the number of threads.

```
int main()
{
    #pragma omp parallel num_threads(NUM_THREADS)
    {
    int id=omp_get_thread_num();
        for (int i=id; i < N; i=i+NUM_THREADS) {
        printf("Thread ID %d Iter %d\n",id,i);
        }
    }
    return 0;
}
```

1.1.6 `datarace.c`

1. No
2. We can add the atomic or critical directive before updating x.

```
for (i=id; i < N; i+=NUM_THREADS) {
    #pragma omp atomic
    x++;
}
```

```
for (i=id; i < N; i+=NUM_THREADS) {
    #pragma omp critical
    x++;
}
```

1.1.7 `barrier.c`

1. No, but due to the barrier directive we know that all threads will exit the barrier only when all threads have done all the previous work. And due to the sleeptime it is very probable that the order to wake up is 0 1 2 3.

1.2 Worksharing

1.2.1 `for.c`

1. The default implementation is using a static scheduling
2. We should add a *#pragma omp single* directive.

1.2.2 `schedule.c`

1. When using static, the ith loop executes the threads in the [N/numthread * i, N/numthread * (i+1) - 1] range. When using static with chunks of length 2 we assign them cyclicacly to each of the available threads.

1.2.3 `nowait.c`

1. After taking out the *nowait* directive we guarantee that the first loop is finished for all the threads before executing the second loop. That is, all the printf's from the first loop will execute before the printf commands in the second one.
2. As the *nowait* directive is taking the implicit barrier away, if we remove the nowait clause in the second for nothing will change as we do not execute any other commands after the second for.

1.2.4 `collapse.c`

1. The iterations are splitted in blocks of (N*N)/8 , and then splitted sequentially as ordered pairs.
2. No, as the variable j is shared in all threads. We should add a private(j) clause.

1.2.5 `ordered.c`

1. The printf before the ordered gets printed in a random order, but the printf command with the ordered directive gets printed in order of ascending iteration number.

2. We have to specify at the schedule directive to use chunks of size 2.

1.2.6 `doacross.c`

1. The order from the outside messages is random, whereas the inside message from the i-th iteration will only be printed if the one from the i-2 th iteration has already been printed.

2. You only can execute the i,j iteration if and only if you have executed the iterations i-1,j and i,j-1. Due to the sleep directive it makes the program execute in northeastward-diagonals, although it might not do each diagonal in order.

3. Now the program only executes obeying to the rule that you can only execute the current iteration if i-1,j and i,j-1 have already been executed which means the order becomes more randomized.

1.3 Tasks

1.3.1 `serial.c`

1. Yes, it is calculating the fibonacci numbers correctly. Although it is not running in parallel.

2. We should specify the parallel region with the pragma. To avoid various threads creating the same task again we should also add a single directive fter declaring the parallel region.

1.3.2 `parallel.c`

1. No, it seems to be calculating the value once one each thread and adding it together. Thus it is running in parallel although not correctly

2. We should add a omp single directive.

3. Nothing happens as p is first private by default.

4. If p is not firstprivate, it may cause a segfault if we access p->foo after the last value $p = p\text{-}{>}next$ making p NULL. (But it is first private by default).

5. Because the creation of tasks was sequential and not parallel.

1.3.3 `taskloop.c`

1. First of all a thread creates the first level tasks, that is Task T1 and T2 and goes to taskwait immediately after. 2 different threads perform the tasks in the pool, One will wait for 5 seconds and the other will create tasks T3 and T4. Then 2 other threads will pick up this tasks. one will now sleep for 10 seconds and the other will create new tasks from the double for loop. And the other threads will then execute them whenever possible.

2 Parallelization Overheads

2.1 Overhead associated with a parallel region (fork and join)

Nthr	Overhead	Overhead per thread
2	1.8711	0.9355
3	1.7511	0.5837
4	2.1216	0.5304
5	2.9226	0.5845
6	2.9834	0.4972
7	2.7065	0.3866
8	3.1083	0.3885
9	3.5937	0.3993
10	3.4123	0.3412
11	4.2939	0.3904
12	3.6636	0.3053
13	4.1018	0.3155
14	4.4929	0.3209
15	4.0894	0.2726
16	4.7880	0.2993
17	4.8280	0.2840
18	4.2563	0.2365
19	4.5779	0.2409
20	4.5766	0.2288
21	4.9388	0.2352
22	5.2615	0.2392
23	5.2372	0.2277
24	5.4667	0.2278

We see how every new node increases the time of overhead. The overhead per thread, although, decreases and seems to decrease towards 0.22. So we can induce that the overhead in function of the number of of threads, when the number of threads tends to infinity will be linear. It is on the order of microseconds.

2.2 Overhead associated with the creation of a task and its synchronization

Ntasks	Overhead per task
2	0.0969
4	0.1277
6	0.1257
8	0.1268
10	0.1250
12	0.1249
14	0.1245
16	0.1255
18	0.1273
20	0.1261
22	0.1254
24	0.1247
26	0.1240
28	0.1233
30	0.1233
32	0.1229
34	0.1240
36	0.1236
38	0.1235
40	0.1232
42	0.1232
44	0.1226
46	0.1218
48	0.1217
50	0.1223
52	0.1221
54	0.1219
56	0.1221
58	0.1219
60	0.1218
62	0.1211
64	0.1212

The order is in microseconds. We see the overhead / number tasks is almost constant for each number of tasks. So that means the overhead will be 0.12*number of tasks.

2.3 Overhead comparing task and taskloop

Ntasks	Overhead per task
2	0.0356
4	0.0048
6	0.0069
8	-0.0006
10	0.0081
12	0.0050
14	0.0040
16	0.0034
18	-0.0003
20	-0.0016
22	-0.0014
24	-0.0017
26	-0.0020
28	-0.0037
30	-0.0036
32	-0.0040
34	-0.0047
36	-0.0046
38	-0.0065
40	-0.0061
42	-0.0064
44	-0.0066
46	-0.0061
48	-0.0066
50	-0.0074
52	-0.0068
54	-0.0075
56	-0.0074
58	-0.0031
60	-0.0097
62	-0.0080
64	-0.0187

The order is in microseconds. We see that when we have around 18 tasks it becomes better to use the taskloop instead of task. If we have only few tasks it is better to use task as the overhead is smaller when we use a large amount of fork and join calls and taskloops are optimized for running forloops in tasks.

2.4 Overhead in Critical regions

When comparing the results of pi_omp_critical, pi_omp and pi_seq we obtain that it takes 39.03s, 0.147 and 0.79s respectively. That is, the overhead from our critical is 4.78s while from the normal parallel execution is -0.08s. We observe that pi_omp_critical takes much more time than the other programs as it has a much larger overhead. It is mainly split between wait, lock, execution and unlock time of the critical region. The increase in overhead when increasing the number of processors is due to similar factors, including: No threads can execute the for-loop when any single one of them is in the critical region, there is more time needed for forks and joins and lastly there is a natural overload on the processor when more threads are used.

2.5 Overhead in Atomic regions

When executing with the atomic directive we see it takes 6.27s, giving as still an overhead of 0.685s. As in the previous case, we see that the parallel executions where we block a commonly accessed variable makes the program slower. But we see that in practise, the atomic directive is quite faster than critical. That is because atomic is optimized for cases like this in which we only want to avoid datasharing on one variable. Then again when we up the number of threads, more tasks have to wait to execute the command after the atomic, there is an increase in time to fork and join. And there is an overhead to synchronize our program.

2.6 Overhead in False Data sharing

When submitting the vector sum version and the padding version we observe that the time required is 0.567s and 0.116s respectively. The increase in time from the vector sum version is due to the false data sharing which is happening in our program. Different threads invalidate their value saved in cache whenever we modify a value in another thread. Thus making more misses in the cache than there really is. For this to work faster we use Padding in such a way that different data associate with different cache entries and thus stopping the false data sharing which causes cache misses.