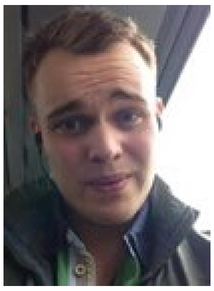
Technical University of Denmark

F19
02346
DTU DIPLOM
DISTRIBUTED AND PARALLEL SYSTEMS

MANDATORY ASSIGNMENT 3

GRUPPE 17

7. APRIL 2019



JOACHIM S. MORTENSEN
s175179

MIKAEL KRISTENSEN
s094766

NICOLAI KAMMERSGÅRD
s143780

Problem 1 (group): Basic OpenMP

Handed-in C File: ***pipara.c***

Report on your work with Databar Exercises 5.3. Be sure to address the questions in the exercises.

The integration method used in pi.c needs the dx value to be defined before starting the calculation of pi or summarizing of the integral intervals. This means that we have a fixed size of intervals, in order to expand or give pi further resolution, we would have to calculate it all over with a higher interval size. The trapezoid method on the other half, lets each summarizationg have a margin of error, and then for each sum or "interval"the precision of pi gets higher.

The reason the printed value of "true pi"disagrees with the PI25DT constant macro is because doubles are only precise to the 15th decimal, and the macro has 25 decimals of precision.
In order to get the wanted precision a more precise floating point data type would be needed.

When changing the number of intervals we see the more decimals matching the "true"pi value. With 51 million intervals we are just above 1 second and have 13 decimals of precision. Same thing goes for a run of 50 million intervals which took just below 1 second. So its safe to say 1 second of runtime (with the given code) equals to a 13 decimal precision for pi. The 13 decimal precision stays the same for a runtime of 10 seconds.

Number of intervals	51000000	50000000
Computed Pi	3.14159265358968	3.14159265358956
The true Pi	3.14159265358979	3.14159265358979
Error	0.00000000000010	0.00000000000023
Elapsed time (s)	1.049444	0.996866

When the pi program is used with the OpenMP, quite a speedup is seen from 2 threads and up. we ran into some problems going higher than 8 threads (Queue time for HPC being quite long), but the speedup kept rising with the number of threads used. Efficiency wise we saw the biggest leep when going to 4 or 8 threads. numbers below.

Threads	Intervals	Runtime(s)	Speedup	Efficiency
1	50000000	1.026036	0.97	1.03
2	50000000	0.530795	1.88	1.06
4	50000000	0.311586	3.20	1.25
8	50000000	0.266358	3.74	2.14

Problem 2 (group): OpenMP Implicit work division

Handed-in C File: ***filter.c***

Report on your work with Databar Exercises 6.1 and 6.2 The program works by looking at each pixel in the original image, averaging all the colours over a WINDOW_SIZE*WINDOW_SIZE region with the pixel as the center, and writing the average of these colours to the processed image. The filtering technique is simple, but it removes a lot of noise through blurring.

Verified and run multiple times on the HPC cluster.
See appendix A for the graph over the parallel efficiency.

Thread #	Speedup (S)
1 Thread	0.842426035
2 Threads	1.662490019
3 Threads	2.492293865
4 Threads	3.322738955
5 Threads	4.121660863
6 Threads	4.962164580
7 Threads	5.626631011
8 Threads	6.250057390
9 Threads	6.984072199
10 Threads	7.744741741
11 Threads	5.858011429
12 Threads	4.287254221
13 Threads	4.627403313
14 Threads	10.71601345
15 Threads	11.54548083

The efficiency is 0.842426035 when run on a single thread, meaning the parallelized program becomes slower when run on only a single thread.

The reason for this is the extra overhead copying memory and scheduling the threads.

The form of the efficiency curve closes in on $Time_{serial}/Thread_{count}$

Adding the scheduling clause `schedule(static,500)` approximately the same, just a tiny bit slower.

Doing this for the different scheduling types yields these results:

scheduling	(2 Threads)
default	1.080851
st500	1.082710
st1	1.283460
dynamic	1.474654
guided	1.208014

scheduling	(4 Threads)
default	0.540790
st500	0.550006
st1	0.642453
dynamic	0.742451
guided	0.540765

scheduling	(6 Threads)
default	0.362121
st500	0.362588
st1	0.428174
dynamic	0.515186
guided	0.367818

They all seem to make the process slower, as most likely the workload is not big enough for this scheduling to really matter. The more threads are added the more guided seems to keep up.

Problem 3 (Individual): Pthreads AllReduce Monitor

Handed-in C File: ***allReduce.c***
s094766

Description of solution:
all_reduce (int * k); starts by using Pthread_mutex_lock to ensure that there is only one thread at a time that writes in the variable sum.

After this, the bar_pass() monitor barrier with rounds is used, to ensure that all threads have written in sum, before sum is written to *k.

bar_pass() is then used again to ensure that all threads have written sum to * k before it is set to 0.

bar_pass() is used one last time to ensure that all threads have finished setting sum to 0.

The test program:
The test program creates the variable num and stores a random number between -50 and 50 in it, then prints it out and calls the all_reduce function with &num as the argument. After the function call to all_reduce it prints out the new value of num.

Problem 4 (individual): Non-associativity of addition

Handed-in C File: ***piassoc.c***
s175179

Pacheco 5.5a
Final output is:
sum = 1010.0
since:

Value is 1010.0
mantissa is rounded to 101
exponent is 10^1

Pacheco 5.5b
Thread 1:
Value is 7.0 mantissa is rounded to 7
exponent is 10^0
Thread 2:
Value is 1003.0
mantissa is rounded to 100
exponent is 10^1

Final reduction:
Value is 1007.0
mantissa is rounded to 100
exponent is 10^1

Final output is: sum = 1000.0

assocpi.c

in the `piassoc.c` file the same procedure is run twice, except one of the times it's run in reverse.

Number of intervals:	10000
Computed Pi	3.141592654423134067798173
Reversed Pi	3.141392644424374491762819
The true Pi	3.141592653589793115997963
Pi vs Reverse Pi	-0.000200009998759576035354

There's a clear difference in the results, and this is caused by rounding errors, and this makes floating point arithmetic non-associative.

Problem 8 (individual): Work division reporting

Handed-in C File: ***scheduling.c***
s143780

To output the different threads iteration interval, we need to know what interval each thread got. To get this, we have a thread specific boolean, which we on each threads first iteration, get the thread specific start value, and sets the thread specifc counter (end) on only first iteration.

```
if(first) {
    first = 0;
    start = i;
    end = i;
```

Now each thread can just increment "end"value with 1, which gives us the iteration interval [start,end] at the end of the loop execution.

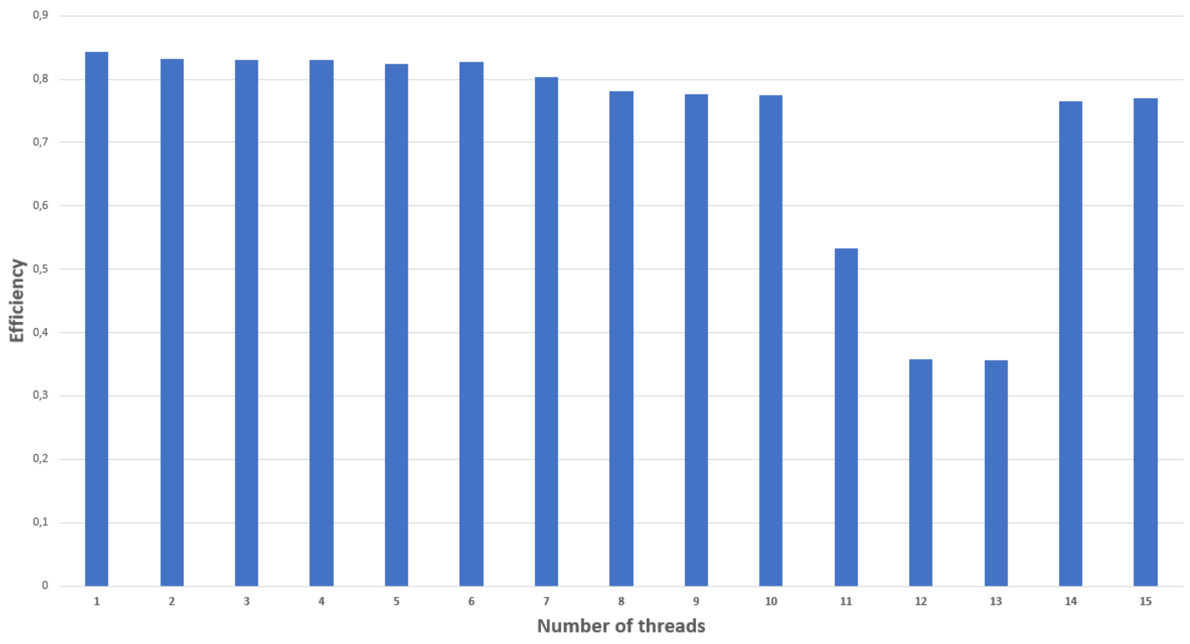
To ensure each thread prints their thread specific iteration, not only the for loop is parallelized but a scope around the for loop containing the thread specific variables initialization and the print after the for loop.

```
#pragma omp parallel
{
    int i;
    int first = 1;
    int tid = omp_get_thread_num();
    int start, end;

    #pragma omp for
    for (i=0 ; i<iterations ; i++) {
        if(first) {
            first = 0;
            start = i;
            end = i;
        } else {
            end++;
        }
    }

    printf("Thread %d: Iterations %d - %d\n", tid, start, end);
}
```

Conclusion

Through this mandatory assignment, the group expanded their knowledge in the OpenMP field, which gave an insight into the differences in code architecture when chosing a parallelization framework such as OpenMP and MPI. Further an understanding of PThreads was also achieved.

Appendix A


0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Efficiency
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Number of threads