Assignment 2: Programming with OpenMP

Github repo for our assignment: https://github.com/smithc36-tcd/hpc-group

Exercise 1 - OpenMP Hello World, get familiar with OpenMP Environment

1. Write an OpenMP C code with each thread printing Hello World from Thread X! where X is the thread ID.

```c
#include "omp.h"
#include<stdio.h>
int main()
{
omp_set_num_threads(4)
#pragma omp parallel
  {
    int ID = omp_get_thread_num();
    int np = omp_get_num_threads();
    printf("Hello World from Thread %d!\n",ID);
  }
  return 0;
}
```

2. How do you compile the code in question 1? Which compiler and flags have you used?

`cc -fopenmp  hello_world.c`(gray)

OpenMP flag:`-fopenmp`

3. How do you run the OpenMP code on Dardel? What flags did you set?

```
export OMP_NUM_THREADS=4
srun -n 1 ./hello_world.out
```

4. How many different ways can the number of threads in OpenMP be changed? Which are they?

Here are three ways.

Changing the enviroment variable `OMP_NUM_THREADS`:

`export OMP_NUM_THREADS=<number of threads to use>`(bash shell)

`setenv OMP_NUM_THREADS <number of threads to use>`(csh or tcsh shell)

Using the runtime library functions:

`omp_set_num_threads(int num_threads)`

As a clause as part of the directive:

`#pragma omp parallel num_threads(int num_threads)`

Exercise 2 - STREAM benchmark and the importance of threads

1. Run the STREAM benchmark five times and record the average bandwidth values and its standard deviation for the copy kernel. Prepare a plot (with error bars) comparing the bandwidth using 1,32,64, and 128 threads.

	1	2	3	4	5	average	standard deviation
1	11200.5	12608.3	12235.2	11996.6	12730.7	12154.0	608.2
32	33298.0	33375.9	33311.3	34379.5	33999.8	33673.0	491.3
64	42701.0	43847.7	42172.4	42172.4	42116.8	42602.0	735.9
128	48619.0	45990.2	45977.6	47662.5	45977.6	46845.0	1229.9

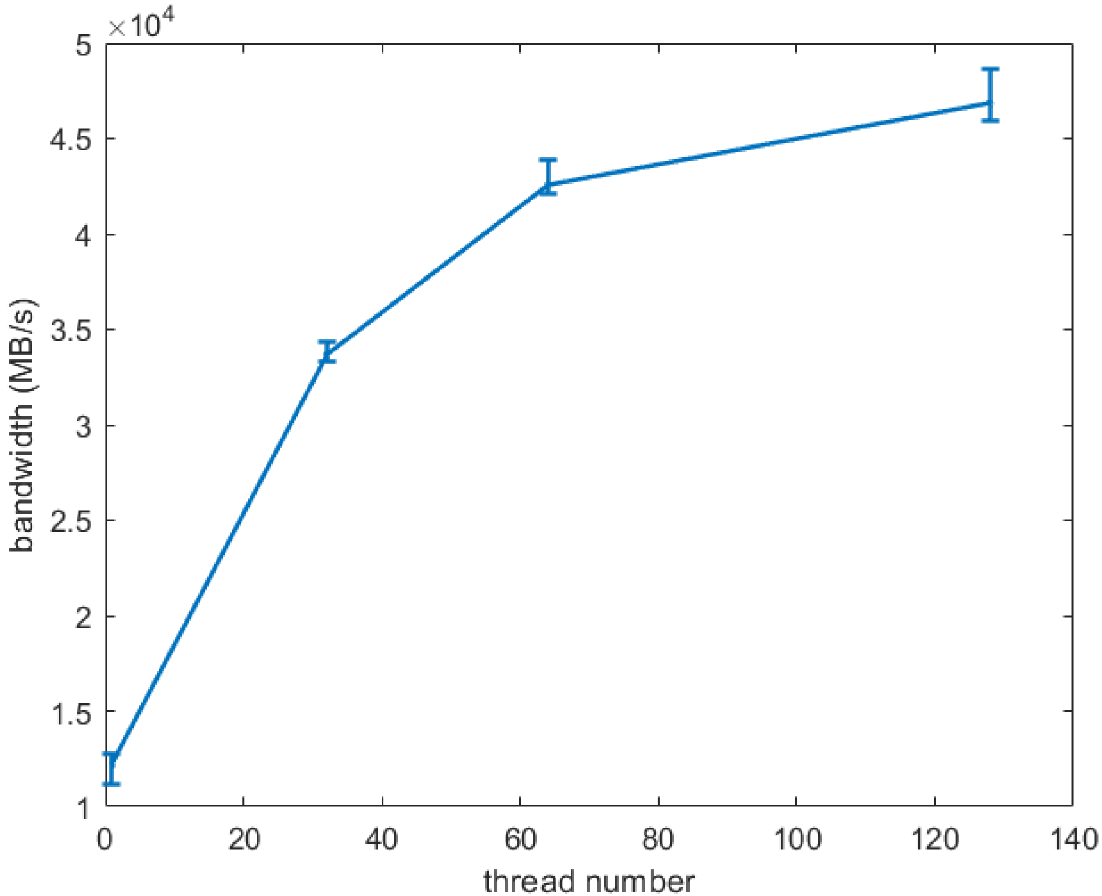

Figure 1: graph.png

2. How does the measured bandwidth with the copy kernel depend on the number of threads?

As the number of threads increases, the measured bandwidth gradually increases, but the increase tends to slow down.

3. Prepare another plot comparing the bandwidth measured with copy kernel with static, dynamic, and guided schedules using 128 threads.

	1	2	3	4	5	average	standard deviation
static	44894.9	45883.3	46949.0	47689.6	46808.2	46445.0	1078.6
dynamic	21019.5	20332.9	20418.9	20610.8	21450.8	20767.0	465.1
guided	32894.9	31904.9	30922.9	31110.7	29514.0	31269.0	1252.3

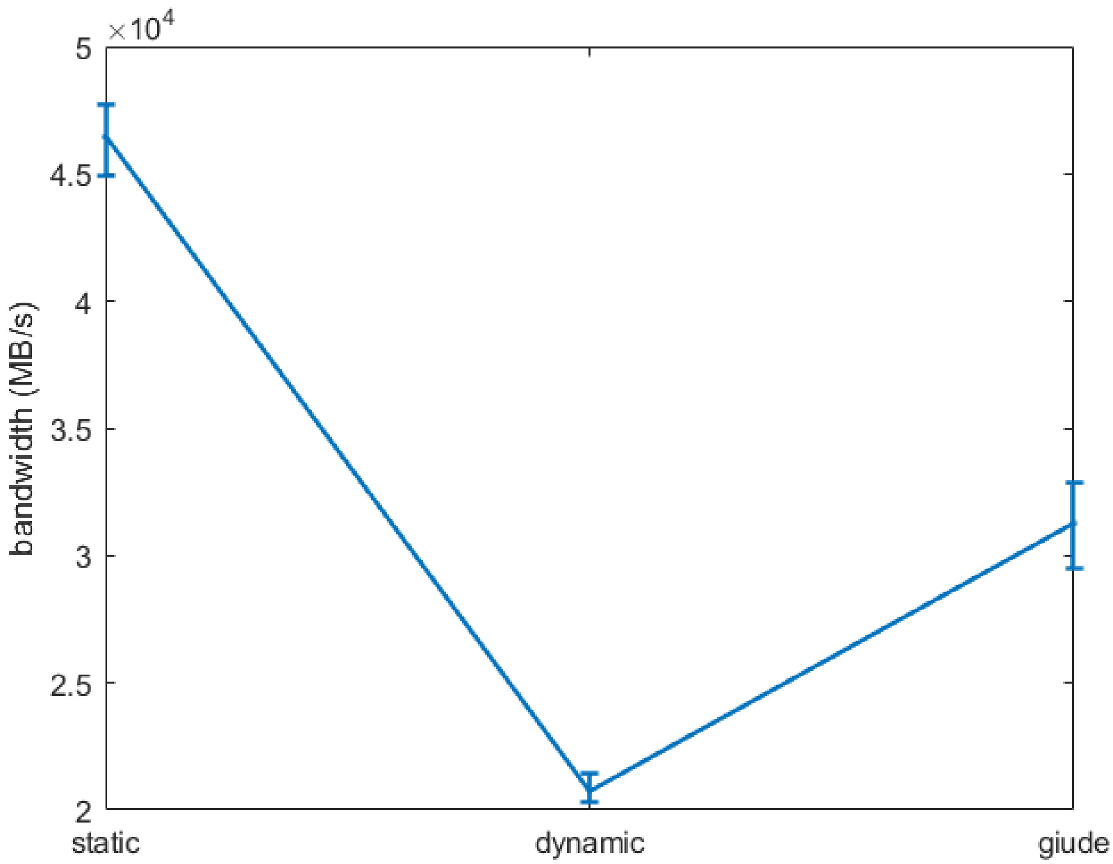

Figure 2: graph.png

4. How do you set the schedule in the STREAM code? What is the fastest schedule, and why do you think it is so?

If using static schedule, we don't need to modify the STREAM code., since the defualt schedule is static in openMP.

If using dynamic schedule, we set the code like

```
#pragma omp parallel for schedule(dynamic)
      for (j=0; j<STREAM_ARRAY_SIZE; j++)
        a[j] = b[j]+scalar*c[j];
#end
...
```

If using guided schedule, we set the code like

```
#pragma omp parallel for schedule(guided)
      for (j=0; j<STREAM_ARRAY_SIZE; j++)
```

```
        a[j] = b[j]+scalar*c[j];
#end
...
```

Static schedule is fastest because the copy bandwidth is largest, which means the computing time is lowest. Static schedule when loop limits known, work per iteration constant.

Exercise 3 - Parallel Sum

1. Measure the performance of the serial code (average + standard deviation).

Out of 5 measurements with 10ˆ7 elements we obtained:

- Average: 0.032559 s
- Standard deviation: 3.74e-5 s

2. Implement a parallel version of the serial_sum called omp_sum and use the omp parallel for construct to parallelize the program. Run the code with 32 threads and measure the execution time (average + standard deviation). Is and should the code be working correctly? If not, why not?

Using `omp parallel for` and 32 threads we get:

- Average: 0.052364 s
- Standard deviation: 0.00201

The code doesn't run correctly as multiple threads try to access and modify the sum variable concurrently, making it susceptible to race conditions.

3. Implement a new version called 'omp_critical_sum' and use the omp critical to protect the code region that might be updated by multiple threads concurrently. Measure the execution time for the code in questions 2 and 3 by varying the number of threads: 1, 2, 4, 8, 16, 20, 24, 28, and 32. How does the performance compare to the program in questions 1 and 2? What is the reason for the performance gain/loss?

	1	2	4	8	16	20	24	28	32
omp_sum	0.033876	0.078455	0.056979	0.054617	0.053033	0.049611	0.054719	0.050377	0.051933
omp_critical_sum	0.069273	0.161543	0.173142	0.193622	0.197640	0.195231	0.191220	0.193787	0.193857

Performance is lost when using critical to protect the code as the code is essentially serialised as threads only access the sum one by one (nonetheless giving a correct result).

4. Try to avoid the use of a critical section. Implement a new version called omp_local_sum. Let each thread find the local sum in its own data, then combine their local result to get the final result. For instance, we can use temporary arrays indexed by their thread number to hold the values found by each thread, like the code below.

- `double local_sum[MAX_THREADS];`
- Measure the performance of the new implementation, varying the number of threads to 1,32,64, and 128 threads. Does the performance increase as expected? If not, why not?

As expected, this method gives the correct sum. We get the following performance depending on the amount of threads:

	1	32	64	128
omp_local_sum	0.044071	0.027366	0.015175	0.014685

The performance increases with number of threads, and is better than the serial sum.

5. Write a new version of the code in question 4 called opt_local_sum using a technique to remove false sharing with padding. Measure the performance of the code by varying the number of threads to 1, 32, 64, and 128.

Using a 256 byte struct (double + 248 char padding) the following results are obtained

	1	32	64	128
opt_local_sum	0.052609	0.041504	0.027234	0.029865

There is an improvement in performance over the serial sum, but omp local sum seems to perform better overall despite not accounting for false sharing. This could be because we are not aware of the cache line size of the system. Another way to remove false sharing is to create a local variable in each thread to store the local sum. This gives slightly better performance.

	1	32	64	128
opt_local_sum_nopad	0.033760	0.006540	0.007709	0.012429

Removing false sharing halves the quickest time compared to `omp_local_sum` for the optimal number of threads!

Exercise 4 - DFTW, The Fastest DFT in the West

1. Our method for parallelisation is the following:

```
// DFT/IDFT routine
// idft: 1 direct DFT, -1 inverse IDFT (Inverse DFT)
int DFT(int idft, double *xr, double *xi, double *Xr_o, double *Xi_o, int N) {
#pragma omp parallel for reduction(+ : Xr_o[:N]) reduction( +:Xi_o[:N])
  for (int k = 0; k < N; k++) {
    for (int n = 0; n < N; n++) {
      Xr_o[k] += xr[n] * cos( n * k * PI2 / N) + idft * xi[n] * sin( n * k * PI2 / N);
      Xi_o[k] += -idft * xr[n] * sin( n * k * PI2 / N) + xi[n] * cos( n * k * PI2 / N);
    }
  }

  // normalize if you are doing IDFT
  if (idft == -1) {
#pragma omp parallel for
    for (int n = 0; n < N; n++) {
      Xr_o[n] /= N;
      Xi_o[n] /= N;
    }
  }
  return 1;
}
```

Making use of the OpenMP reduction statement.

2. Measure the performance on Dardel 32 cores reporting the average values and standard deviation for DFTW using an input size equal to 10000 (N=10000).

```
DFTW calculation with N = 10000
Mean running time across 20 runs: 0.339029 seconds
Standard deviation of running time for 20 runs: 0.016917 seconds
```

3. Prepare a speed-up plot varying the number of threads: 1,32,64, and 128.

Table comparing execution speed against thread count using 32 cores, varying the number of threads.

Threads	1	2	4	8	16	32	64	128
Seconds	7.080946	3.567994	1.799715	0.924253	0.47055	0.345014	1.003984	2.834436

4. Which performance optimizations (think about what you learned in the previous module) would be suitable for DFT other than parallelization with OpenMP? Explain, no need to implement the optimizations.

There are several suitable optimizations

- Pre-computation and lookup tables. We could precompute the values for the trigonometric functions and store them in a lookup table.
- Common expression elimination. If lookup tables couldnt be used we could implement common expression elimination. The term `n * k * PI2/N`, computing the expressions `sin(n * k * PI2/N)` and `cos(n * k * PI2/N)` once for each loop would save computation.
- Cache friendly memory access. We could a cache friendly memory access with strip mining and loop reordering to implement blocking.

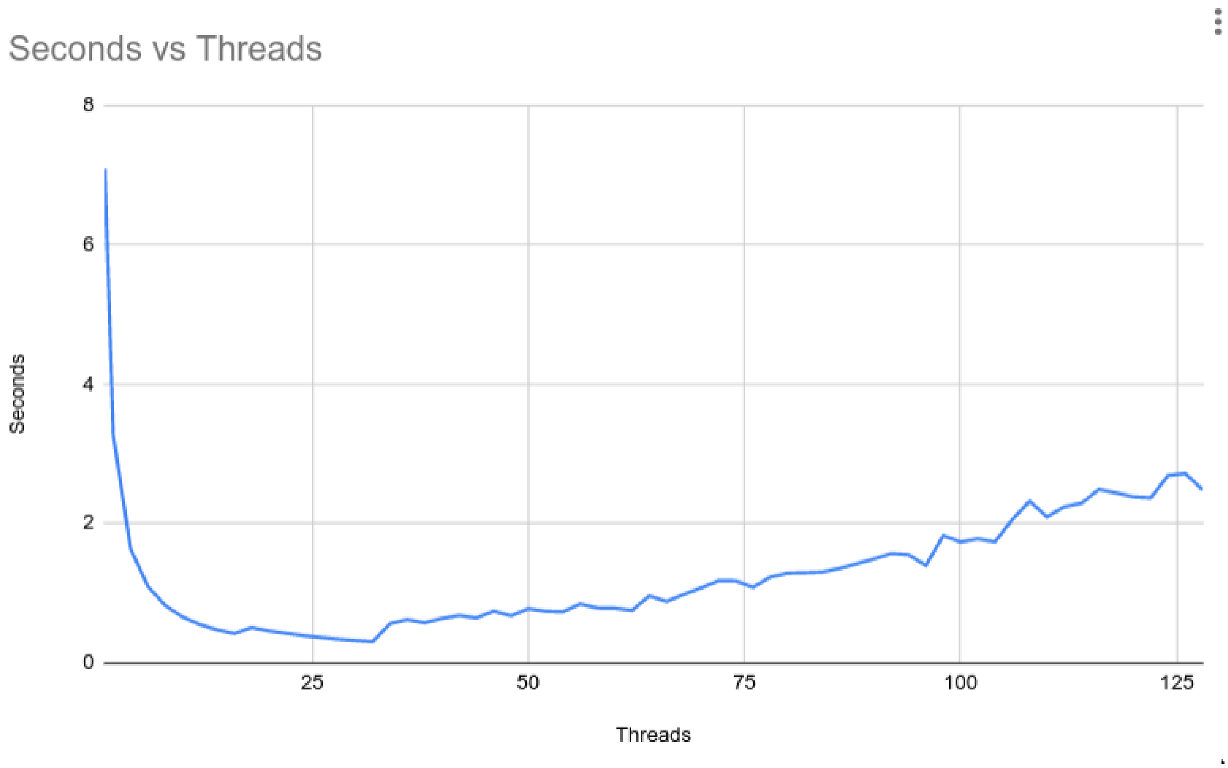

Figure 3: Graph of execution time against thread count on Dardel

- Vectorization: We could implement vectorisation with SIMD instructions, which is even possible with OpenMP using the `#pragma omp simd` directive.
- Algorithm: This Algorithm runs in `O(N^2)` time while the FFT Algorithm runs in `O(N logN)`. Changing implementation would have a significant effect on execution time for large N.