Assignment II: Programming with OpenMP
DD2356 VT24 Methods in High Performance Computing

Long Ma, Yining Hou

April 12, 2024

1 OpenMP Hello World, get familiar with OpenMP Environment

1.1 Question 1

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

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

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello␣World␣from␣Thread␣%d!\n", id);
    }
    return 0;
}
```

1.2 Question 2

Compile the code with `Cray` compiler and use flag `-openmp`. Use flag `-o` to specify output file name.

```
cc -openmp hello_world.c -o hello_world.out
```

1.3 Question 3

1) Set the number of threads through environment variable. `OMP_PLACES=cores` is also necessary to ensure the correct placement of the threads.

```
export OMP_NUM_THREADS=4
OMP_PLACES=cores
```

2) Set flags.

- –nodes: Specify the number of nodes to be allocated.
- -t: Specify the duration of allocation.
- -A: Specify the allocation code which can be found with projinfo command.
- –ntasks-per-node: Specify the tasks to run per node.
- –cpus-per-task: Set the CPUs allocated per task. Notice that we use four cores per tasks because it needs to be set as $2 \times OMP\_NUM\_THREADS$.
- -p: Shared to use the shared partition, where multiple users can share the same node.

```
salloc --nodes=1 -t 00:05:00 -A edu24.DD2356 -p shared --ntasks-per-node=1 --
    cpus-per-task=8
```

3) Run the OpenMP code on Dardel.

```
srun -n 1 ./hello_world.out
```

1.4 Question 4

3 ways to change the number of threads in OpenMP.
1) Set environment variable

```
export OMP_NUM_THREADS=4
```

2) Apply runtime function in the code

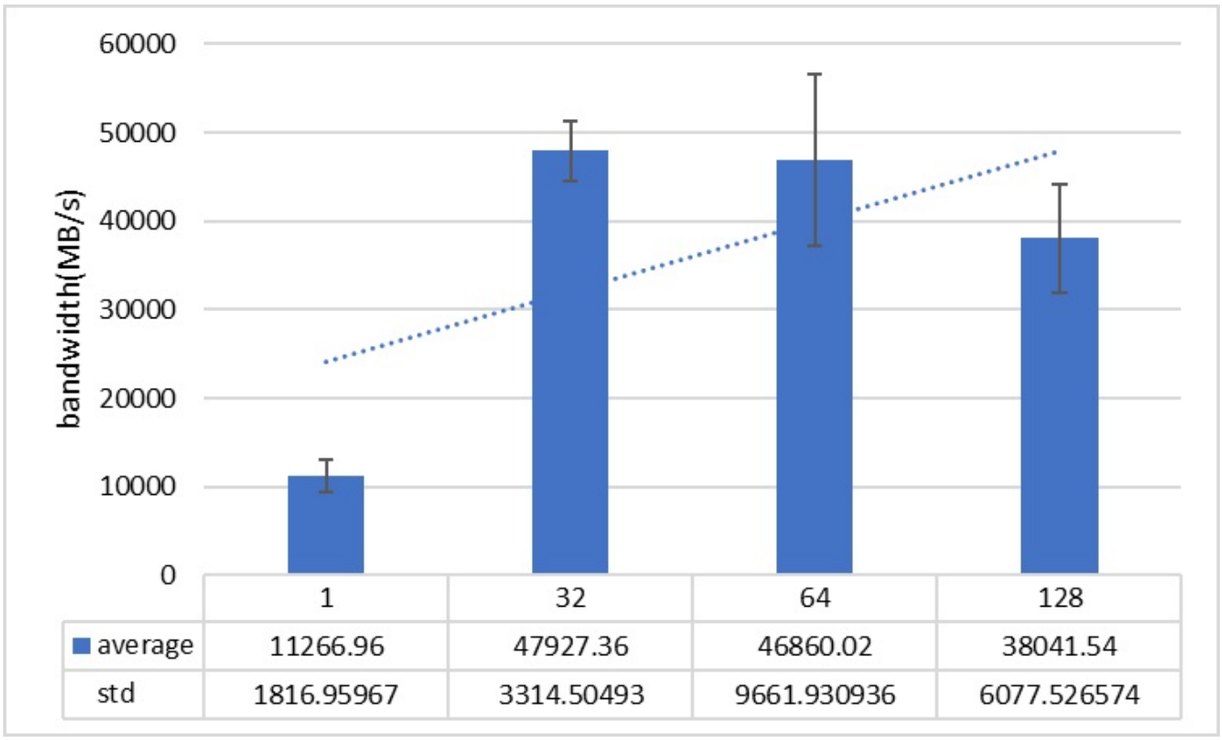
```
omp_set_num_threads(4);
```

3) Use compiler directive in the parallel region

```
#pragma omp parallel num_threads(4)
```

2 STREAM benchmark and the importance of threads

2.1 Question 1



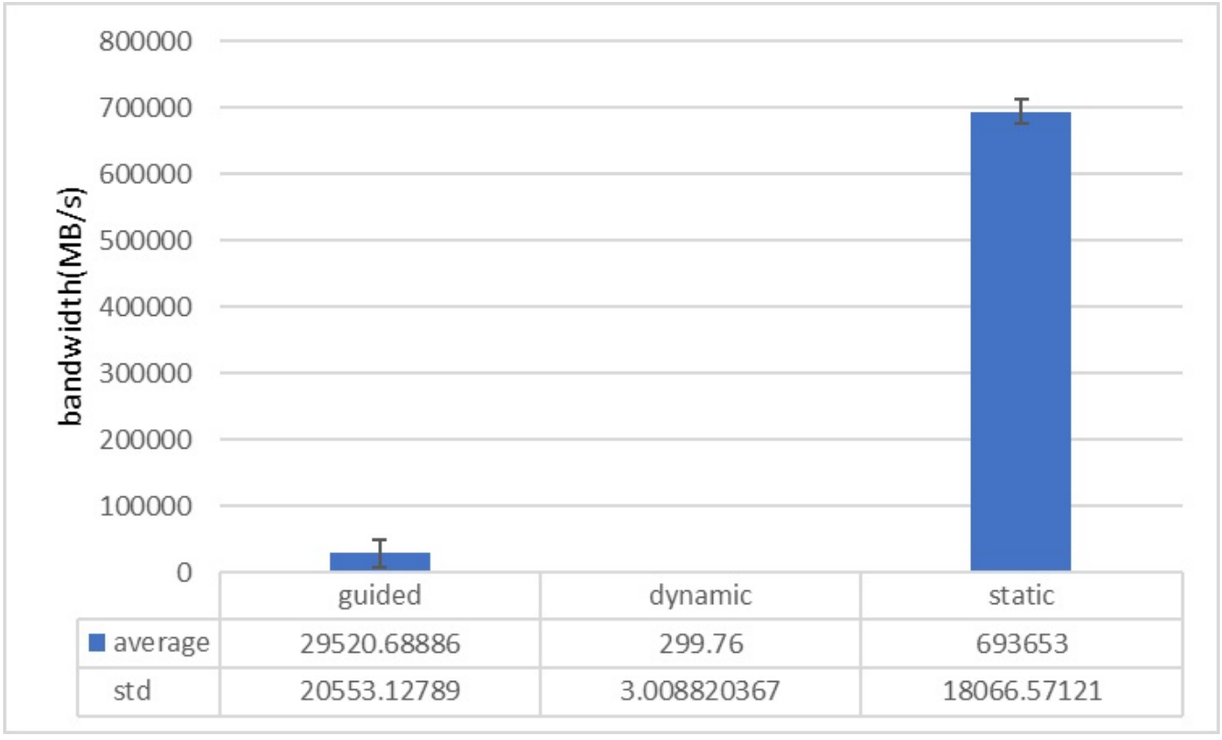
	1	32	64	128
average	11266.96	47927.36	46860.02	38041.54
std	1816.95967	3314.50493	9661.930936	6077.526574

Figure 1: Average bandwidth values and its std for the copy kernel using different number of threads

2.2 Question 2

As the number of threads increases, the average bandwidth first increases and then decreases. When the number of threads increases from 1 to 32, the average bandwidth increases. This indicates that with the increase in the number of threads, the bandwidth improves. However, when the number of threads continues to increase to 64 and 128, the average bandwidth starts to decrease. This phenomenon may be due to factors such as resource contention and cache efficiency.

2.3 Question 3



	guided	dynamic	static
average	29520.68886	299.76	693653
std	20553.12789	3.008820367	18066.57121

Figure 2: Average bandwidth values and its std for the copy kernel using different schedules

2.4 Question 4

The code initially uses a *guided* schedule. You only need to change the *guided* in the code to *static* or *dynamic* to change the schedule.

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

```
#pragma omp parallel for schedule(static)
    ...
```

```
#pragma omp parallel for schedule(dynamic)
    ...
```

Among the three scheduling methods, static scheduling is the fastest. Because its load-balancing method evenly distributes work to threads at compile time, reducing the overhead of thread synchronization and management. And the execution time of the work is similar, the dynamic load balancing of *dynamic* and *guided* schedule only increases the overhead without improving performance.

3 Parallel Sum

3.1 Question 1

Implement a simple C code to find the sum of an array with at least 10^7 elements. Use the function `omp_get_wtime()` to get the time for measurement. Measure the performance of the serial code (average + standard deviation).

```
// Performance measurement
for (int i = 0; i < num_trials; i++) {
    generate_random(input, size);

    start_time = omp_get_wtime();
    serial_sum(input, size);
    end_time = omp_get_wtime();

    total_time = end_time - start_time;
    times[i] = total_time;
    mean_time += total_time;
}

mean_time /= num_trials;

// Calculating standard deviation
for (int i = 0; i < num_trials; i++) {
    std_dev += (times[i] - mean_time) * (times[i] - mean_time);
}
std_dev = sqrt(std_dev / num_trials);
```

Version	Average Time(second)	Standard Deviation(second)
serial_sum	0.033201	0.000409

3.2 Question 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).

```
#pragma omp parallel for
for (size_t i = 0; i < size; i++) {
    sum_val += x[i];
}
```

Version	Average Time(second)	Standard Deviation(second)
omp_sum	0.114308	0.001007

The code will not work correctly due to a race condition. Multiple threads will modify the shared variable `sum_val` concurrently without synchronization and some thread updates may be lost.

3.3 Question 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.

```
#pragma omp parallel for
for (size_t i = 0; i < size; i++) {
    #pragma omp critical
    {
        sum_val += x[i];
    }
}
```

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.

	omp_critical_sum		omp_sum	
Threads	Average Time(second)	Standard Deviation(second)	Average Time(second)	Standard Deviation(second)
1	0.108767	0.000865	0.032554	0.000032
2	0.201272	0.003106	0.054492	0.000800
4	0.309074	0.001256	0.049400	0.000573
8	0.441249	0.001394	0.045172	0.002106
16	0.598792	0.001649	0.051368	0.000489
20	0.649764	0.002999	0.088299	0.000548
24	0.830469	0.008741	0.098817	0.000476
28	0.919232	0.007240	0.105722	0.000720
32	0.948559	0.004797	0.114308	0.001007

- Compare the performance to the program in questions 1, the performance of `omp_critical_sum` is worse than `serial_sum` due to the overhead from initializing and managing threads introduced by `omp parallel for` and the critical section forces threads to wait for its turn to execute the critical section, leading to increased waiting times.
- Compare the performance to the program in questions 2, the performance of `omp_critical_sum` is worse than `omp_sum` because the critical directive introduces more overhead due to thread synchronization.

3.4 Question 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 omp_local_sum(double *x, size_t size)
{
    double local_sum[MAX_THREADS] = {0.0};
    int num_threads;

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        num_threads = omp_get_num_threads();

        #pragma omp for
        for (size_t i = 0; i < size; i++) {
            local_sum[id] += x[i];
        }
    }

    double sum_val = 0.0;
    for (int i = 0; i < num_threads; i++) {
        sum_val += local_sum[i];
    }
    return sum_val;
}
```

Measure the performance of the new implementation, varying the number of threads to 1,32,64 and 128 threads.

Threads	Average Time(second)	Standard Deviation(second)
1	0.033350	0.000072
32	0.012099	0.000918
64	0.008402	0.001417
128	0.008816	0.002639

The performance of `omp_local_sum` is much better than `omp_critical_sum` because it eliminates the need for threads to wait for each other. Each thread works on its local data and writes to its own memory location, which avoids the overhead associated with locking.
As the number of threads increases, the performance increases but slightly decreases at 128, which is likely due to exceeding the optimal number of threads that the hardware can efficiently manage. Besides, the overhead associated with managing these threads can outweigh the performance gains from parallel execution.

3.5 Question 5

Write a new version of the code in question 4 called `opt_local_sum` using a technique to remove false sharing with padding. `getconf LEVEL1_DCACHE_LINESIZE` returns the size of the cache line.

```
#define CACHE_LINE_SIZE 64

typedef struct {
    double sum;
```

```
    char pad[CACHE_LINE_SIZE - sizeof(double)];
} padded_sum;

double opt_local_sum(double *x, size_t size)
{
    padded_sum local_sum[MAX_THREADS] = {0.0};
    int num_threads;

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        num_threads = omp_get_num_threads();

        #pragma omp for
        for (size_t i = 0; i < size; i++) {
            local_sum[id].sum += x[i];
        }
    }

    double sum_val = 0.0;
    for (int i = 0; i < num_threads; i++) {
        sum_val += local_sum[i].sum;
    }
    return sum_val;
}
```

Measure the performance of the code by varying the number of threads to 1, 32, 64, and 128.

Threads	Average Time(second)	Standard Deviation(second)
1	0.032591	0.000198
32	0.003972	0.000655
64	0.003836	0.002096
128	0.004451	0.002158

4 DFTW, The Fastest DFT in the West

4.1 Question 1

For each *for* loop, we add a *#pragma omp parallel for schedule(static)* to use multiple threads to execute.

```
int DFT(int idft, double *xr, double *xi, double *Xr_o, double *Xi_o, int N) {
  #pragma omp parallel for schedule(static)
  for (int k = 0 ; k < N ; k++) {
    #pragma omp parallel for schedule(static)
    for (int n = 0 ; n < N ; n++) {
      // Real part of X[k]
      Xr_o[k] += xr[n] * cos(n * k * PI2 / N) + idft*xi[n]*sin(n * k * PI2 / N)
          ;
      // Imaginary part of X[k]
      Xi_o[k] += -idft*xr[n] * sin(n * k * PI2 / N) + xi[n] * cos(n * k * PI2 /
          N);
    }
  }
  ...
}
```

For the serial version, we got the following result after calculating

Threads	Average Time(second)	Standard Deviation(second)
1	23.2626418	0.144319005

We get this result when we use the OpenMP version with only 1 thread, which gives us a similar result as the serial version.

Threads	Average Time(second)	Standard Deviation(second)
1	23.0004904	0.597892777

4.2 Question 2

After we executed 5 times, we got the following result after calculating

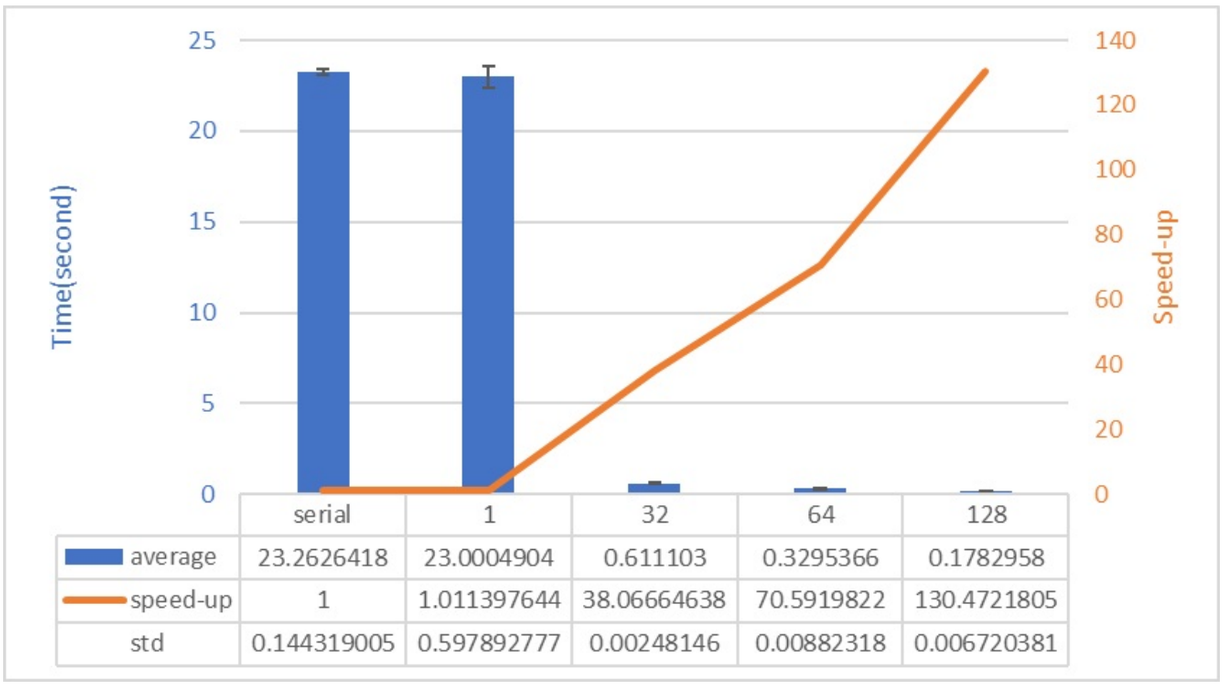
Threads	Average Time(second)	Standard Deviation(second)
32	0.611103	0.00248146

4.3 Question 3

We calculate Speed-up using the following formula,

$$\text{Speed-up} = \frac{T_{serial}}{T_{parallel}} \tag{1}$$

T_{serial} is the execution time of the serial version, $T_{parallel}$ is the execution time of the parallel version. Speed-up represents the multiple execution speed improvement of the parallel version relative to the serial version.



	serial	1	32	64	128
average	23.2626418	23.0004904	0.611103	0.3295366	0.1782958
speed-up	1	1.011397644	38.06664638	70.5919822	130.4721805
std	0.144319005	0.597892777	0.00248146	0.00882318	0.006720381

Figure 3: Average time and speed-up plot using different number of threads

4.4 Question 4

In the function, n*k*PI2/N is used multiple times, so it can be defined as a local variable. Moreover, cos() and sin() are also calculated twice, and these two variables can also be defined as local variables.

```
int DFT(int idft, double *xr, double *xi, double *Xr_o, double *Xi_o, int N) {
  #pragma omp parallel for schedule(static)
  for (int k = 0; k < N; k++) {
    #pragma omp parallel for schedule(static)
    for (int n = 0; n < N; n++) {
      double discrete_frequency = n * k * PI2 / N;
      double cos_v = cos(discrete_frequency);
      double sin_v = sin(discrete_frequency);
      // Real part of X[k]
      Xr_o[k] += xr[n] * cos_v + idft * xi[n] * sin_v;
      // Imaginary part of X[k]
      Xi_o[k] += -idft * xr[n] * sin_v + xi[n] * cos_v;
    }
  }
```

This can reduce the time of repeated calculations and improve the cache hit rate.

```
longm@uan01:~/Private/DD2356/Assignment-II> srun -n 1 perf stat -e L1-dcache-
   load-misses,L1-dcache-loads ./DFTW_4
DFTW calculation with N = 10000
DFTW computation in 0.107124 seconds
Xre[0] = 10000.000000
...
          61362950      L1-dcache-load-misses:u   #    0.24% of all L1-dcache
             hits
       25442022258      L1-dcache-loads:u
...
longm@uan01:~/Private/DD2356/Assignment-II> srun -n 1 perf stat -e L1-dcache-
   load-misses,L1-dcache-loads ./DFTW_2
DFTW calculation with N = 10000
DFTW computation in 0.173312 seconds
Xre[0] = 10000.000000
...
          54699264      L1-dcache-load-misses:u   #    0.12% of all L1-dcache
             hits
       44473196948      L1-dcache-loads:u
...
```

Since xi[n] and xr[n] will be used twice in each inner loop, and Xr_o[k] and Xi_o[k] will only be used once in each inner loop. Therefore, changing the *for* loop order of k and n can also increase the cache hit rate and shorten the running time. However, a **rounding error problem** occurred during the test, possibly due to the accumulation of errors in the representation of floating-point numbers. This causes the error between the result x and IDFT(DFT(x)) beyond the specified 0.01.

We can also split the loop into smaller blocks. This can also increase the cache hit rate and shorten the running time.