Submission for High Performance Computing Exercise 2

Authors: Danylo Kravchenko (kravch0000), Lukas Daugaard Schröder (schroe0006)

April 5, 2023

1 Matrix Multiply in OpenMP

In the naive version, we decided to optimize the code, vectorize the most inner loop, and add `OpenMP` directives. We need to specify `shared(a,b,c) private(i,j,k)` to tell the OpenMP how to use `i,j,k` variables. Without these constructs, `i,j,k` would be updated in each thread and the result would be incorrect. We use `collapse(2)` to parallelize 2 most outer loops and `simd reduction(+:sum)` directive to accumulate the values of the most inner loop into the `sum` variable.

```
#pragma omp parallel for collapse(2) shared(a,b,c) private(i,j,k)
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += a[i*n+k] * b[k*n+j];
        }
        c[i*n+j] = sum;
    }
}
```

For the block version (cache aware), we optimized the code, vectorized the innermost loop and used `simd reduction(+:sum)` directive to accumulate the values into the `sum` variable.

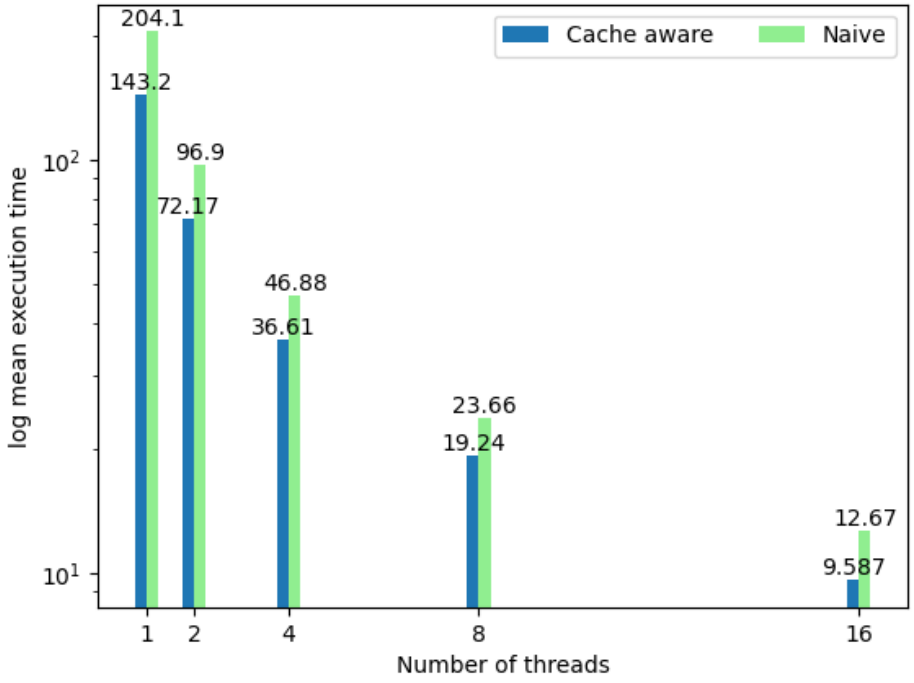
```
#pragma omp parallel for collapse(2) shared(a,b,c) private(i,j,k,ii,jj)
    for (ii = 0; ii < n; ii +=bn)
        for (jj = 0; jj < n; jj += bn)
            for (i = ii; i < MIN(ii+bn,n); i++)
                for (j = jj; j < MIN(jj+bn,n); j++) {
                    double sum = 0.0;
                    #pragma omp simd reduction(+:sum)
                    for (k = 0; k < n; k++) {
                        sum += a[i*n+k] * b[k*n+j];
                    }
                    c[i*n+j] = sum;
                }
```

Now let's define a `T1.job`. We calculated that the correct cache block size for the miniHPC node would be $36 = \lfloor\sqrt{\frac{32000}{(3*8)}}\rfloor$. In addition, we have increased the timeout of the job to be able to see results, and we have changed the order of the program calls with different thread number. The job specification:

```
#!/bin/sh
#SBATCH --job-name=Exercise2Task1
```

```
#SBATCH --time=12:00:00
#SBATCH --nodes=1
#SBATCH --partition=xeon
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks-per-socket=1
#SBATCH --cpus-per-task=20
#SBATCH --hint=nomultithread
#SBATCH --output=T1-result

ml intel/2019a

export OMP_PROC_BIND=close
export OMP_PLACES="cores"
export KMP_CPU_SPEED=2600
export LD_LIBRARY_PATH=/storage/shared/projects/dls-openmp/exerciseScheduleHPC/libomp/intel

# block_size = 36 => floor(sqrt(32K/(3*8)))
export OMP_NUM_THREADS=16
icc -O3 -fopenmp -lstdc++ T1.c -o T1.o
./T1.o 4000 0
./T1.o 4000 36

export OMP_NUM_THREADS=8
icc -O3 -fopenmp -lstdc++ T1.c -o T1.o
./T1.o 4000 0
./T1.o 4000 36

export OMP_NUM_THREADS=4
icc -O3 -fopenmp -lstdc++ T1.c -o T1.o
./T1.o 4000 0
./T1.o 4000 36

export OMP_NUM_THREADS=2
icc -O3 -fopenmp -lstdc++ T1.c -o T1.o
./T1.o 4000 0
./T1.o 4000 36

export OMP_NUM_THREADS=1
icc -O3 -fopenmp -lstdc++ T1.c -o T1.o
./T1.o 4000 0
./T1.o 4000 36
```

Strong scalability plot:



2 Scheduling in OpenMP

Firstly, we have implemented the function, to fill the upper triangle of the matrix:

```
/**
 * @brief fill only the upper triangle of the matrix,
 * the lower part should be zeros.
 *
 * @param n size of the matrix
 * @param matrix matrix to fill
 */
void fillMatrix_upper(int n, double * matrix) {
    int i, j;
    for (i = 0; i < n*n; i++)
    {
        for (j=i; j < n; j++)
        {
            matrix[i*n+j] = (rand()%10) - 5; //between -5 and 4
        }
    }
}
```

Since the lower triangular part of the matrix consists of zeros, OpenMP runtime should figure out how to schedule the matrix multiplication correctly and not take into consideration the lower triangular part of the matrix, because multiplication with this part would result in 0.

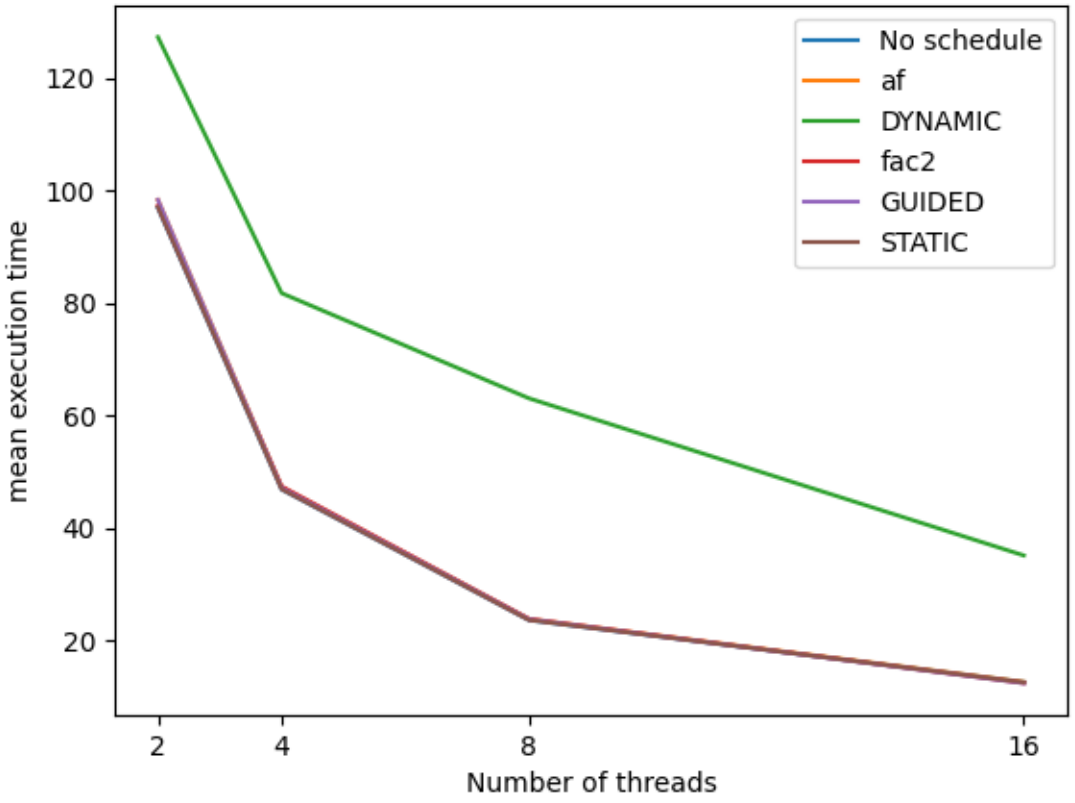
To distinguish, b and c parts of the task, we created different jobs for each part, and for c part different jobs for each schedule type.

To add load balance to the `omp for` directive, we need to specify a new clause `schedule(runtime)` to tell OpenMP to take a schedule of the runtime. We have tried the following schedules: `dynamic, static, guided, af, fac`.

Naive schedules execution time:

number of threads	16	8	4	2
no schedule	12.631609	23.654623	46.875944	96.881314
af	12.661289	23.822520	47.279767	98.278620
dynamic	35.133957	63.040021	81.725453	127.20787
fac2	12.479275	23.862324	47.380583	97.218500
guided	12.450985	23.780721	47.110249	98.290247
static	12.591200	23.659383	46.899315	96.992544

Strong scalability plot:

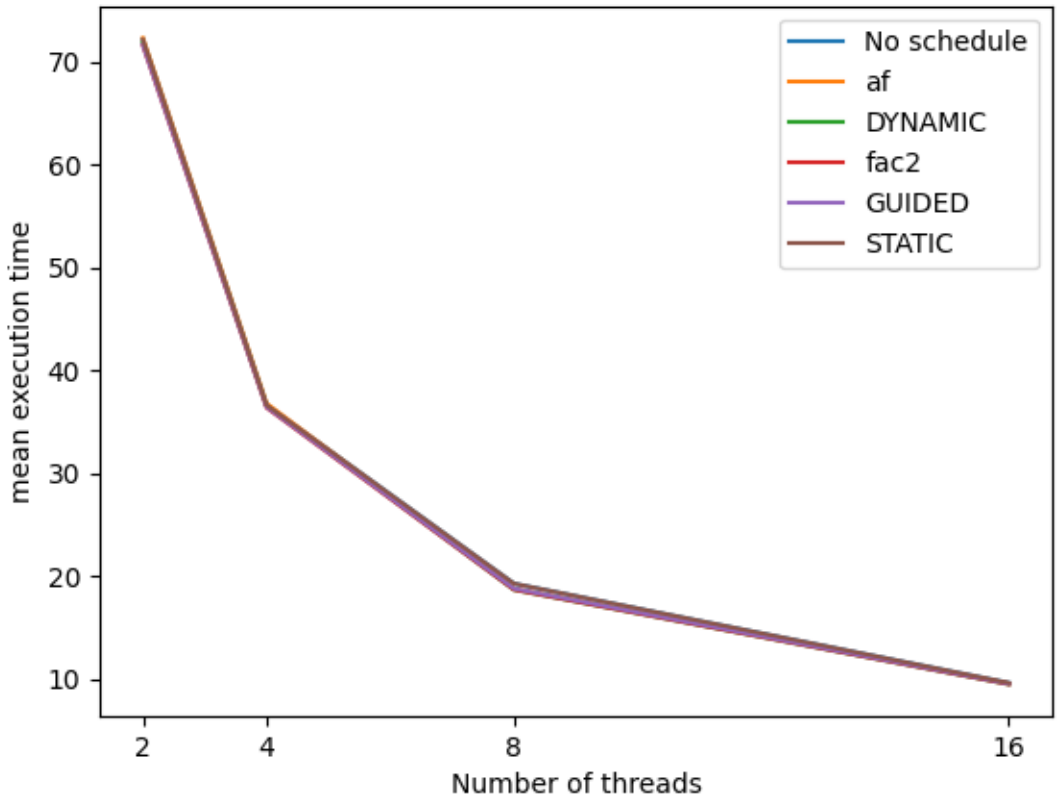


From the result, we could see that the `dynamic` performs the worst for the Naive version of matrix multiplication, since each thread executes a chunk of iterations in random order and then takes another chunk until there are no more chunks available. Potentially, `dynamic` schedule is not aware that the lower triangle part of the matrix is zero and what to do with batches of data. For the Block version of matrix multiplication, all schedules perform very similar.

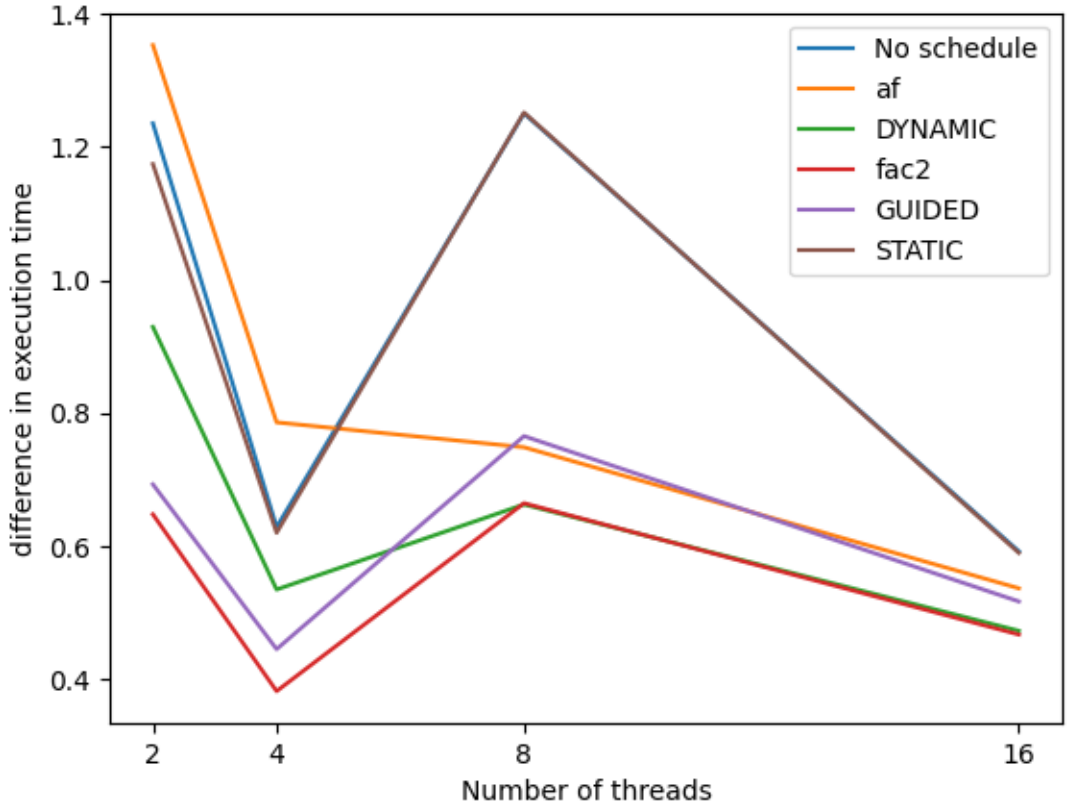
Blocked schedules execution time:

number of threads	16	8	4	2
no schedule	9.592557	19.250339	36.628906	72.235406
af	9.536976	18.748834	36.786263	72.353021
dynamic	9.473077	18.662862	36.535113	71.929608
fac2	9.467514	18.664774	36.382445	71.648097
guided	9.517252	18.765597	36.445530	71.693121
static	9.590226	19.251630	36.620387	72.174246

Strong scalability plot:



Difference between schedules plot:



This is made by minusing the lowest integer value of each number of threads runtime. Eg. minus -71 of all the runtimes of 2 threads. This could not be done to the naive implementation because the dynamic schedule was an outlier.

As we can see from the graph above, the fac2 schedule, made by the HPC team of University of Basel, outperforms or does almost equally the same at our 4 different thread counts. Overall, for this type of matrix multiplication, the fac2 schedule seems to be the best suited schedule.

3 Task programming in OpenMP

For this exercise, we need to use `taskloop` construct. Tasks are used to perform operations in parallel that take different amount of time. It's a good idea to use `taskloop` when processing a large amount of data in each loop, so managing tasks would have a smaller overhead comparing to the work done inside the task.

So for our particular implementation, we need to replace `parallel for` with `taskloop` constructs. An important note is that we need to use `#pragma parallel #pragma single` to let OpenMP know that only one thread creates tasks.

```
void multiply(int n, double * a, double * b, double * c) {
    int i, j, k;

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop collapse(2) num_tasks(omp_get_max_threads()) shared(a, b, c) priva
    for (i = 0; i < n; i++) {
```

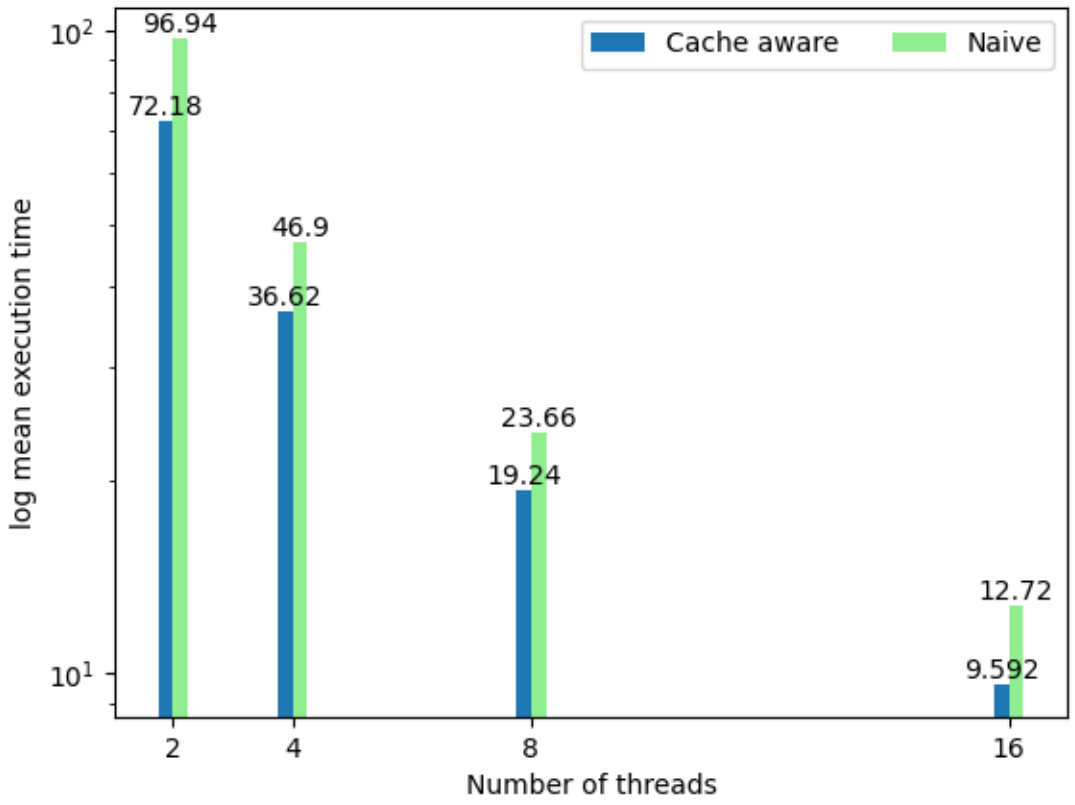
```
        for (j = 0; j < n; j++) {
            double sum = 0.0;
            for (k = 0; k < n; k++) {
                sum += a[i*n+k] * b[k*n+j];
            }
            c[i*n + j] = sum;
        }
    }
}

void blockMultiply(int n, int bn, double * a, double * b, double * c) {

    int i, j, k, ii, jj;

    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop collapse(2) num_tasks(omp_get_max_threads()) shared(a, b, c) priva
    for (ii = 0; ii < n; ii +=bn)
        for (jj = 0; jj < n; jj += bn)
            for (i = ii; i < MIN(ii+bn,n); i++)
                for (j = jj; j < MIN(jj+bn,n); j++) {
                    double sum = 0.0;

                    for (k = 0; k < n; k++)
                        sum += a[i*n+k] * b[k*n+j];

                    c[i*n + j] = sum;
                }
}
```

It turned out that for the task implementation the `#pragma omp taskloop simd reduction(+:sum)` directive made the computation slower, therefore it has not been added to the code, like its corresponding `#pragma omp simd reduction(+:sum)` directive from the previous exercises.

Executions times of naive and cache aware with respect to number of threads:

Number of threads	16	8	4	2
Naive	12.719382	23.663764	46.898695	96.937390
Cache aware	9.592155	19.237361	36.623906	72.176554

Strong scalability plot:



As can be seen in the table and plot the execution times is nearly identical to the execution times of the one from the first exercise, even though we do not use a `simd reduction` directive.

c. Discuss parallelization through tasks benefits

The parallelization of the matrix multiplication can benefit from task programming due to the control the user gains with tasks over `worksharing` constructs. More specifically, we can determine how many tasks should be made for each block of code, done with the `num_tasks` clause for the `taskloop` construct.
Furthermore, for loops we also have control over the division of the iterations into respective tasks of the problem space (called blocking), which is particularly relevant when having for loops that have different amounts of operations dependent on what iteration is currently being executed, which is the case multiplication of sparse, upper or lower matrices.
As an example, in the second task we could define the blocking of the tasks so, that instead of evenly spread iterations, we look at how many values need to be multiplied (that aren't 0, since they don't need to be multiplied) and space them equally out into the individual tasks so that each task can perform and equal amount of multiplications and therefore they would theoretically have a more equal runtime than just partitioning them into equal amounts of iterations.