Lab 1 – OpenMP

Name 1: Alejandro González _____NIA: 252658

Name 2: Luca Franceschi _____NIA: 253885

Cholesky:

1. **Expose your parallelization strategy to divide the work in the Cholesky algorithm and in the matrix multiplication. Justify the selection of the scheduler and chunk size and compare different schedulers with different chunk sizes and show the results.**

To parallelize the Cholesky algorithm, the option that worked best for us was to parallelize the two inner for loops separately. Since each one of them is operating over the two "triangles" of a matrix (the columns before and after the diagonal), we chose the dynamic scheduling with chunk size 1 (default), so that it optimizes the threads as much as possible.

```
start = omp_get_wtime();
for(i=0; i<n; i++) {
    // Calculate diagonal elements
    tmp = 0.0;
    #pragma omp parallel for schedule(dynamic) reduction(+:tmp)
    for(k=0;k<i;k++) {
        tmp += U[k][i]*U[k][i];
    }
    U[i][i] = sqrt(A[i][i]-tmp);
    // Calculate non-diagonal elements
    #pragma omp parallel for schedule(dynamic)
    for(j=i+1;j<n;j++) {
        tmp = 0.0;
        for(k=0; k<i; k++) {
            tmp += U[k][j]*U[k][i];
        }
        tmp = (A[j][i] - tmp) / U[i][i];
        U[i][j] = tmp;
    }
}
end = omp_get_wtime();
printf("Cholesky: %f\n", end-start);
```

For the matrix multiplication, we parallelized the algorithm in such a way that we create a group of threads and divide the loop iterations between them. We also swapped the two inner for loops to maximize the spatial locality of the cache. We stayed with the default scheduler (static) since the threads have equally-distributed jobs.

```
start = omp_get_wtime();
// TODO B=LU
#pragma omp parallel for private(i, j, k) shared(B, L, U)
for(i=0; i<n; i++) {
    for(k=0; k<n; k++) {
        for(j=0; j<n; j++) {
            B[i][j] += L[i][k] * U[k][j];
        }
    }
}
```

For 4 CPU cores we tried with different schedulers (static, dynamic, guided) and chunk sizes (1, 2, 4). The results were the following:

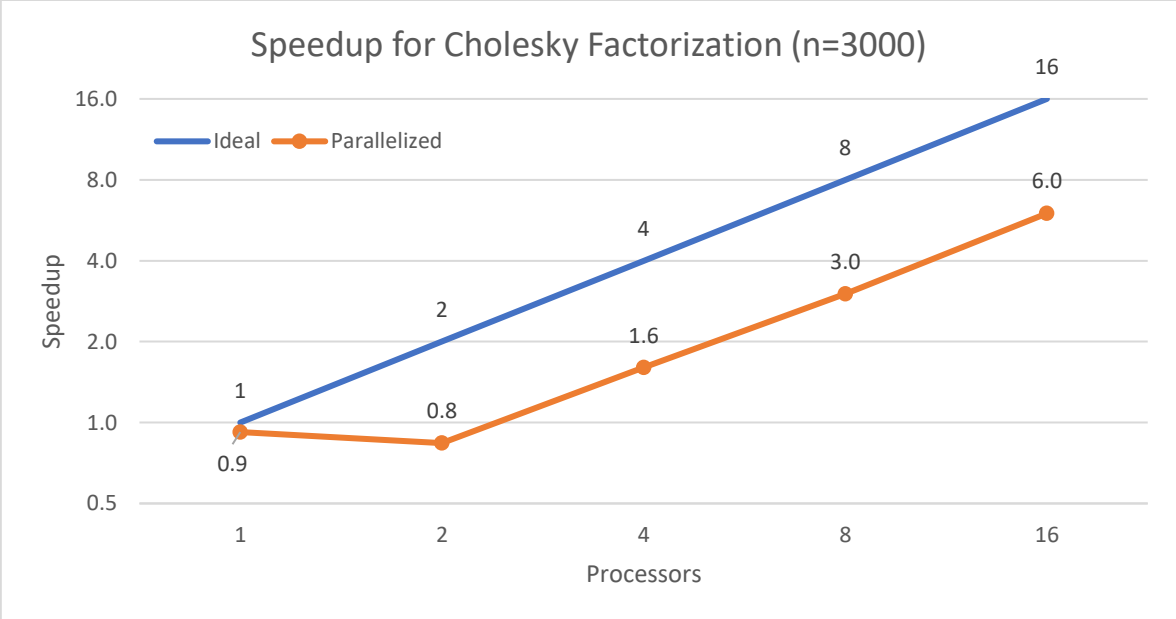
Time (s) for Cholesky	Chunk_Size=1	Chunk_Size=2	Chunk_Size=4
Static	26.8092	27.6963	29.5471
Dynamic	26.7502	27.5932	27.2788
Guided	29.3458	30.3381	29.5222

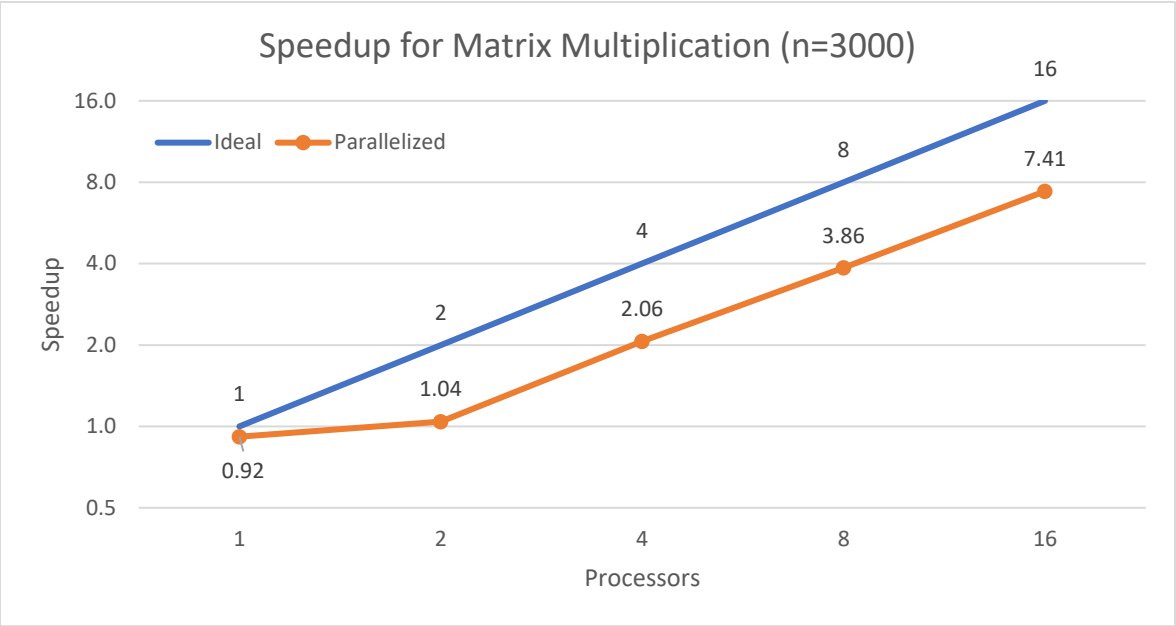
For the Cholesky algorithm we can see a clear winner in dynamic scheduling with chunk size 1.

Time (s) for MatMult	Chunk_Size=1	Chunk_Size=2	Chunk_Size=4
Static	14.4860	14.2863	14.3473
Dynamic	14.3978	14.4005	13.7548
Guided	14.4646	14.2923	13.9145

However, for matrix multiplication there is not a clear winner, and we think that the dynamic with chunk size 4 winner is a fluctuation that deviates from the usual running time, so we stick with default static.

2. **Make two plots: one for the speedup of the Cholesky factorization and another for the matrix multiplication for n=3000. Use 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results.**





First of all, when we see a speedup smaller than 1, it means that the program was slowed down due to the overheads of the parallelization, mainly. Since the structure for a parallelized execution is created, but it is run by one or two processors, that creates overheads that can make the program having up to 0.8 speedup.

For what we can see, the speedup of our parallelization will never reach the ideal one, indeed, it seems to be "half the speedup of the ideal" in each plot. We can see that our parallelization seems to follow an exponential trend, but lags behind the ideal one.

Even though it does not appear in these plots, we can argue that for a higher number of processors the speedup would stabilize at a certain rate, resulting in two plots more similar to the ones seen in the lectures.

The formula we used for the speedup for i number of processors:

$$S_i = \frac{T_{seq}(i)}{T_{par}(i)}$$

Where $T_{seq}(i), T_{par}(i)$ are sequential and parallel times for i number of processors, respectively.

Sudoku:

1. Briefly explain how the code works.

The code solves a standard sudoku recursively calling the solve function.

The solve function does the following: First, it finds the first unassigned number of the grid. If all of the numbers are assigned, then the function returns 1. Then, for that slot, tries all possible combinations of possible numbers. Then, for each valid combination, calls recursively the solve function over that combination. If at any iteration the sudoku is solved, it prints the solved grid.

Basically it's a recursive brute force algorithm that tries all possible valid combinations until the sudoku is solved.

2. Explain how the parallelized code distributes the workload among threads.

First of all, we create a parallel region that contains a master thread (the solve function at level 1) that will be the one calling the recursive tasks.

```
#pragma omp parallel
{
    #pragma omp single
    solve(sudoku,1);
}
```

Next, for each possible valid combination a new task is created. After doing its job, each task waits in parallel outside the 'for loop'. This way, each recursive call of the 'solve' function is a task that it is executed in parallel. This way, the workload should be very distributed among threads, which is reflected in the running time of the execution.

```
int solve(int grid[SIZE][SIZE], int level) {

    int row = 0;
    int col = 0;
    if (!find_unassigned(grid, &row, &col)) return 1;

    for (int num = 1; num <= SIZE; num++) {
        if (is_safe_num(grid, row, col, num)) {
            #pragma omp task firstprivate(grid, row, col, num, level) final(level>1)
            {
                int copy_grid[SIZE][SIZE];
                memcpy(copy_grid,grid,SIZE*SIZE*sizeof(int));
                copy_grid[row][col] = num;
                if(solve(copy_grid, level+1)) {
                    print_grid(copy_grid);
                    double end = omp_get_wtime();
                    double time_spent = end - start;
                    printf("\nFound in %f s\n",time_spent);
                }
            }
        }
    }
    #pragma omp taskwait
    return 0;
}
```

3. Add the clause 'final(level>1)' to your task-generating pragma. What does it do?

The improvement of using the final clause is very noticeable. The running time for the sequential program is 3.46 seconds. For the parallelization without the final clause is 4.55 seconds. The final parallelization including the final clause is 0.27 seconds. From a 0.76 (worse) to a 12.82 speedup (significantly better).

When we include the clause "final(condition)" in the task construct, if the condition evaluates to 1/true, the task will be generated as a final task, and if there are more task constructs in the

execution of this final task (which is the case in this exercise), they would be also included/final tasks. What's special about these tasks is that they will be executed immediately by the encountering threads, thus reducing the overhead of placing tasks in the conceptual pool.

Histogram:

1. Explain how have you solved each of the parallelizations.

For the critical sections' implementation, we select the pieces of code that will only be accessed one thread at a time. That way, all threads will be synchronized.

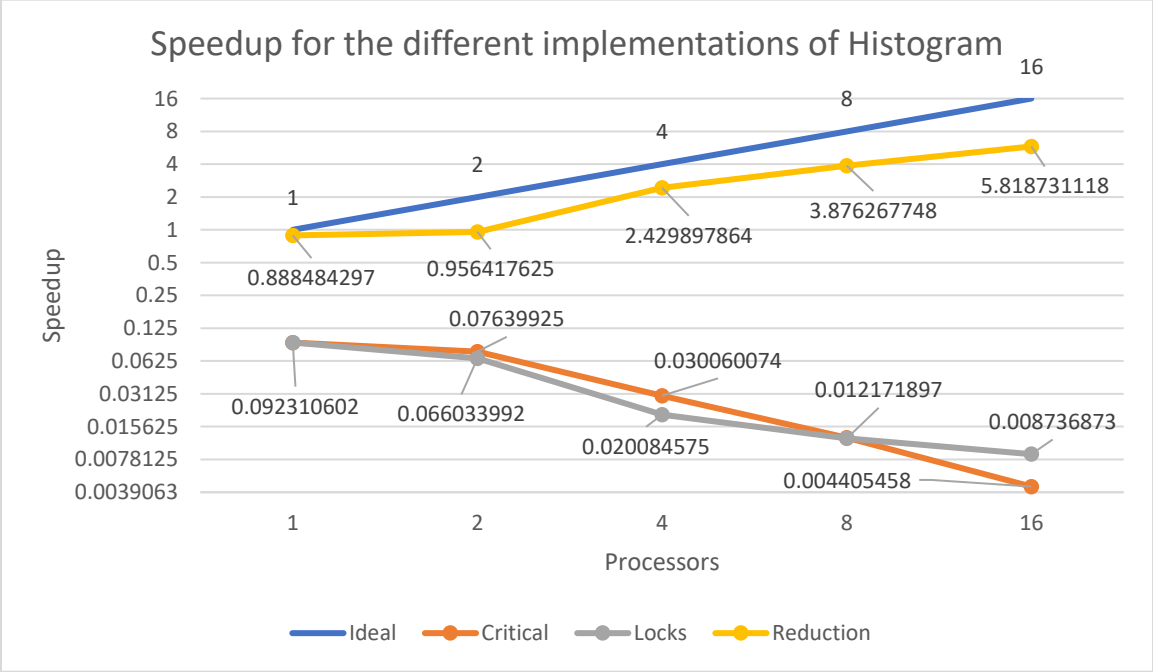
For the locks implementation, very similarly to the critical sections, we create a lock, and set the lock when we are modifying a global variable, and unset it when we exit that dangerous area. We also have to destroy the locks when finished.

For the reduction implementation, we create the following reduction: reduction(+:hist). Since hist is an array that we can have a reduction on, the compiler has the job to synchronize that part of the code in the best way it can. Since the compiler is much smarter than us, it is the most efficient method.

2. Explain the time differences between different parallel methods if there are any.

As shown in the plot below, there is a significant improvement when looking at the reduction implementation, however, with the other two we have not gotten any good results. For this problem, our implementations of critical sections and locks do not scale well.

3. Make a speedup plot for the different parallelization methods for 1, 2, 4, 8, and 16 cores. Discuss the results.



From this plot, we can see that the only method that works well in this scenario is the Reduction. It is the only method that follows that exponential trend, that would be stabilized with more CPU cores. For low processor count, it is a little bit slower due to the overheads created, but with more than 4 cores it takes advantage over the other methods.

For the other methods, we did not expect to have such a slow performance on critical sections method and lock methods, but it seems that in this scenario it does not work well at all.

Producer / consumer:

1. Explain how have you synchronized the threads.

To synchronize the threads, we first checked if the number of threads was equal to 2, and if not we printed a message to console and finished the code with return code 1 (error).

```
if(Nthreads != 2) {
    printf("Error: incorrect number of threads, %d\n", Nthreads);
    return 1;
}
```

Then, after checking the number of threads was correct, we open a parallel region for the different sections of the program, with the directive "#pragma omp parallel sections". Once we have defined this parallel region, we split in two sections the code, one will be for the filling of the array and the other section will be for the consuming/sum of the elements in the array.

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        fill_rand(N, A);        // Producer: fill an array of data
        #pragma omp flush
        flag = 1;
        #pragma omp flush(flag)
    }

    #pragma omp section
    {
        while(!flag) {
            #pragma omp flush(flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);  // Consumer: sum the array
    }
}
```

Notice the “#pragma omp flush(flag)” commands and the use of the flag. The flush commands are used to ensure that all threads have the same view of memory for all shared objects (in this case just the flag variable as we specified it inside the flush command, if we didn’t specify it would flush all shared variables).

In the “producer” section, the array is filled with random numbers. Then, we use the flush directive to make sure that the flag is written after the array is filled. We then write the flag variable to 1. Finally, we flush the variable to make sure that the flag variable is written correctly to memory.

In the “consumer” section, we read the flag variable in a while loop using the flush directive, basically we wait for the variable to be written by the “producer” thread to 1. That means that the array is filled and that we can proceed to sum the array. Once it is read, we have to ensure that the array is consistent with the memory, so we use the flush directive again to synchronize all shared variables. Finally, we can proceed to sum the array.

To ensure completely the execution of the program, we could also use atomic writes and reads to operate over the flag variable, but we think that is out of the scope for this lab.