Khurram Hashmi 12210382

Assignment 1

Approach:

In the code attached , there are two **gspa_taskloop** function **gspa_tasks** function which uses openmp construct taskloop and tasks constructs. I used block sizes 128, 256 and 512. I found out that taskloop gives better speed when I use a block size of 128.

Taskloop: I used the outer loops that populate the S matrix by applying a taskloop construct. I parallelize the initialization of boundary conditions by separately using the omp parallel for directive. The grain_size parameter helps to distribute the tasks. To keep track of the work done in parallel, use a reduction operation. This helps combine the visited count in all the parallel tasks.

Tasks: In task function both the outer and inner loops are done by using tasks. To make sure tasks execute in the correct order,explicitly specify dependencies between them using **depend** clauses.Here, I take a finer approach, creating tasks for individual elements of the matrix. Used an atomic operation that allows the visited count to be updated safely and avoids conflicts.

The taskloop approach looks at the large details, while the tasks approach looks at the small details, making sure the parallelization strategy is finely tuned. The decision between them depends on what exactly needs to be done and how fast we want it to be.

Granularity and the total number of tasks:

In the taskloop approach, I organize the work into tasks using "grain_size." This grain_size decides how many times we group together the steps in the big task that is the outer loop, to make a smaller task. The number of tasks in each loop is determined by the grain_size. For instance, if the grain_size is set to 2, it means two steps together in each task. So, if there are 100 steps in the outer loop, there will be 50 tasks, each handling 2 steps. The grain_size parameter helps how many steps to put in each group. The granularity of the task is controlled by the grain_size parameter. Each task processes a chunk of iterations, and the number of iterations in each chunk is determined by the grain_size.

In the tasks approach.The number of tasks depends on how many times to go through the outer and inner loops. So, every single element in the matrix becomes its own task. omp task doesn't allow a reduction clause like taskloop,uses a temporary variable called **visit_loc** for each task to avoid problems when many tasks are working with **visited**.These tasks are then put in a pool, and threads use them to do the calculations. To make sure the works are connected and don't mess up, I set data scoping to **default(none) in #pragma omp task. X, Y, S, visited**, and **block_size** are then shared, meaning they're accessible to all tasks. **visit_loc**, **rowi**, and **colj** are firstprivate, so each task has its own private copy to avoid problems when lots of threads are doing different tasks all at once.

Dependences

In **gspa_taskloop** , the taskloop construct is used, making sure tasks happen in the right order. The tasks are created based on the steps in the outer loops. So, if Task 2 comes after Task 1 in the loop order, it automatically depends on the result of Task 1. This means Task 2 waits for Task 1 to finish before it starts. The taskloop construct ensures that tasks go in a sequence. I used the anti-diagonal iteration method which addresses data dependency issues in the matrix **S**. The variable **'visited'** encounters a data race condition, causing it to return inconsistent values with each code run. To resolve this issue, a reduction clause has been added into taskloop. This involves creating a private copy of each list variable for every thread. Upon completing the reduction, the reduction variable is applied to all the private copies of the shared variable, and the ultimate result is then written to the global shared variable.

In the **gspa_tasks** function, use the depend clauses. This clause defines the data dependencies between tasks, stating which data is needed before a task can start execution. I made the code better by jumping levels in the anti diagonal fashion. This change in granularity helps the code run faster. However, as the program progresses through iterations, it encounters many dependencies. This happens because each task encompasses multiple cells that are no longer computed independently. Furthermore, each task follows dependency rules, involving two dependencies of type **depend(in)** and one of type **depend(out)**, depend(in) ensures that a task initiating its first cell waits for dependent cells to complete its calculations, while each task must complete its final cell to satisfy the depend(out) condition.

The clauses **depend(in : S[rowi + block_size - 1][colj - 1])** and **depend(in : S[rowi - 1][colj + block_size - 1])** indicate that a task depends on the values of specific elements in the S matrix. An atomic operation (#pragma omp atomic) is used to ensure

the correct accumulation of the visited count between all tasks and thus preventing race conditions.

In **gpsa_taskloop**, dependencies are managed implicitly based on the natural order of loop iterations, while in **gpsa_tasks**, dependencies are explicitly specified using **depend** clauses. The explicit specification of dependencies in gpsa_tasks provides more fine-grained control and is useful when dealing with irregular dependencies.

Data scoping

gpsa_taskloop:

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int d = 1; d <= rows + cols - 1; d += block_size) {
            #pragma omp taskloop reduction(+:visited) grainsize(grain_size)
            for (int rowi = std::min(rows - block_size + 1, d); rowi >= std::max(1, d - cols + 1); rowi -= block_size) {
                int colj = d - rowi + 1;
                for (int i = rowi; i < std::min(rowi + block_size, rows); i++) {
                    for (int j = colj; j < std::min(colj + block_size, cols); j++) {
                        char newX = X[i - 1];
                        char newY = Y[j - 1];
                        try {
```

- The **#pragma omp parallel** and **#pragma omp single** directives create a parallel region with a single master thread. The master thread executes the following loop, dividing the work among other threads using tasks.
- Inside the loop, the **#pragma omp taskloop** construct is used to parallelize the iterations of the outer loops. The reduction(+:visited) clause ensures that the variable visited is updated correctly across parallel tasks.
- In this case, data scoping is managed implicitly, and the variables used within the loop, such as **rowi, d,** and **visited,** are shared among the tasks by default.

gpsa_tasks:

```
// Main part
#pragma omp parallel
{
    #pragma omp single
    {
        for (int d = 1; d <= rows + cols - 1; d += block_size) {
            int rowi;
            for (rowi = std::min(rows - block_size + 1, d); rowi >= std::max(1, d - cols + 1); rowi -= block_size) {
                int colj = d - rowi + 1;
                unsigned long visit_loc = 0;
                #pragma omp task default(none) \
                    shared(SUB, cmap, X, Y, S, visited, d, block_size) \
                    firstprivate(visit_loc, rowi, colj) \
                    depend(in : S[rowi + block_size - 1][colj - 1]) \
                    depend(in : S[rowi - 1][colj + block_size - 1]) \
                    depend(out : S[rowi + block_size - 1][colj + block_size - 1])
                {

                for (int i = rowi; i < std::min(rowi + block_size, rows); i++) {
                    for (int j = colj; j < std::min(colj + block_size, cols); j++) {
                        char newX = X[i - 1];
                        char newY = Y[j - 1];
```

The #pragma omp parallel and #pragma omp single directives create a parallel region with a single master thread, similar to the gpsa_taskloop function. Inside the loop, the #pragma omp task construct is used to parallelize the inner loops. The default(none) clause ensures that variables used inside the task must be explicitly declared. The shared clause specifies that variables like **SUB, cmap, X, Y, S, visited, d, and block_size** are shared among all tasks. The firstprivate clause declares variables (**visit_loc, rowi, colj)** that are private to each task and initialized with the values they have in the enclosing context. The **depend** clauses specify dependencies between tasks, ensuring that tasks are executed in the correct order.

Speedup graph(s)and numbers with respect to the sequential version

Block Size 128 , Gransize 1

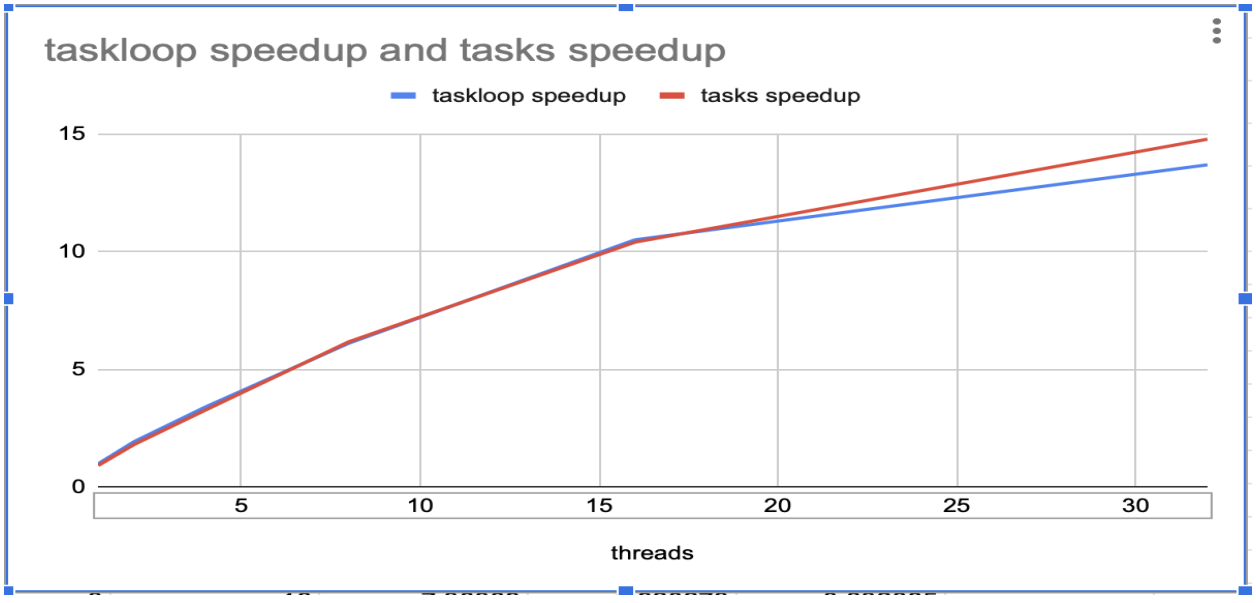
The above graph is calculated when I used block size of 128 and grain size 1. The above graph signifies that Tasks version performs better than taskloop version. At 32 threads ,tasks version achieve speed up of 14.98 and taskloop version 13.67.

```
[configuration]: OMP_NUM_THREADS=32 ./gpsa
Loaded X and Y sequences with sizes 16383 and 16383
Matrix S size: [16384x16384]

== Sequential version completed in 7.36146 seconds.
   Entries visited: 268435456
   Score: 36224, Similarity Score: 10291, Identity Score: 8508, Gaps: 2684, Length (with gaps): 17725

== Taskloop version completed in 0.538422 seconds.
   Entries visited: 268435456
   Score: 36224, Similarity Score: 10291, Identity Score: 8508, Gaps: 2684, Length (with gaps): 17725
   Checking results: OK

== Explicit Tasks version completed in 0.491316 seconds.
   Entries visited: 268435456
   Score: 36224, Similarity Score: 10291, Identity Score: 8508, Gaps: 2684, Length (with gaps): 17725
   Checking results: OK
```

threads	taskloop speedup	tasks speedup
1	0.94	0.9
2	1.82	1.75
4	3.32	3.22
8	6.22	6.16
16	8.93	10.35
32	13.67	14.98

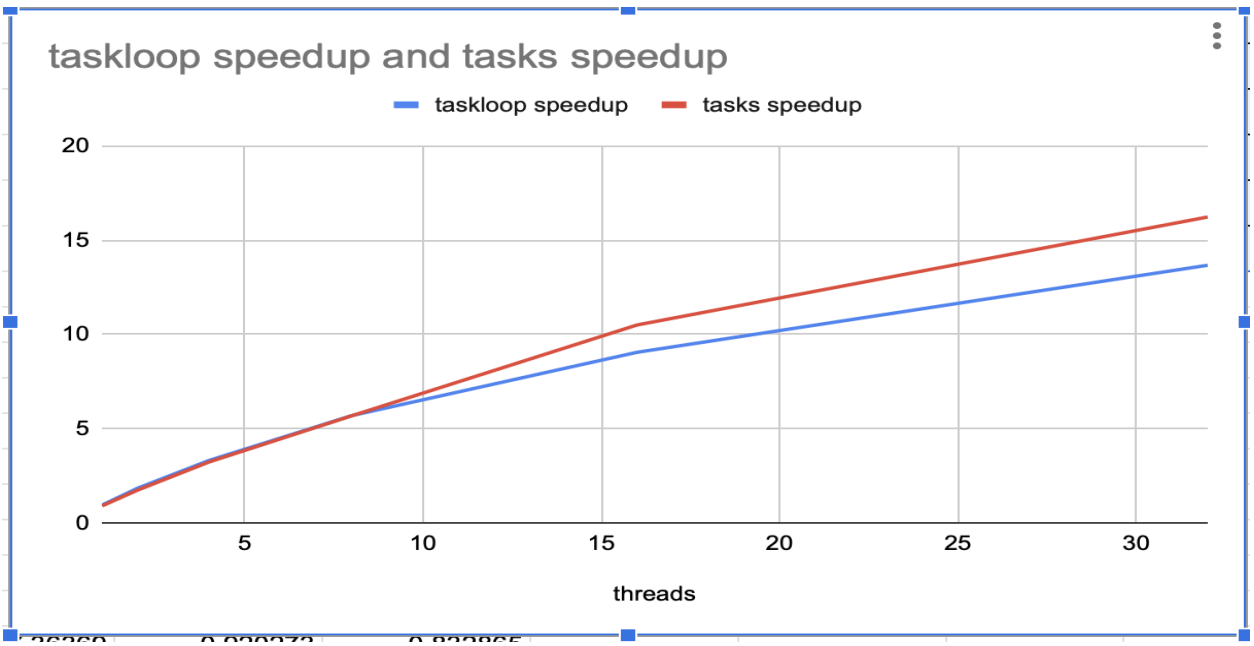
Taskloop version speed up performs little better when grain size is increased to 2 as shown in below graph

Block Size 128 , Gransize 2



When I used a block size of 256 , there was more improvement in the speed of the tasks version but not much difference in the taskloop version as shown in below graph

Block Size 256 , Gransize 1



threads	taskloop speedup	tasks speedup
1	0.94	0.9
2	1.84	1.74
4	3.31	3.23
8	5.69	5.68
16	9.05	10.5
32	13.67	16.23

Bottleneck

Using omp atomic instead of in_reduction to avoid data races is a bottleneck. Another bottleneck is caused by dependencies, which prevent certain tasks from completing until their preceding entries are finished. The results obtained from running the three versions with various Block size configurations are 128,256 512 for different granularities.

The **depend** clauses in the gpsa_tasks function helps to manage dependencies between tasks explicitly. This reduces idle time and helps to improve the overall speed.The use of the **grainsize** parameter in the **#pragma omp taskloop** in the **gpsa_taskloop** function allows for control over the granularity of tasks. Adjusting the granularity has an impact on the workload distribution among threads and affects performance. Explicitly using clause default(none) and shared in the gpsa_tasks function ensures proper handling of data between threads and tasks.

External Sources

https://en.wikipedia.org/wiki/Granularity_(parallel_computing)
https://www.ibm.com/docs/en/zos/2.3.0?topic=processing-pragma-omp-task