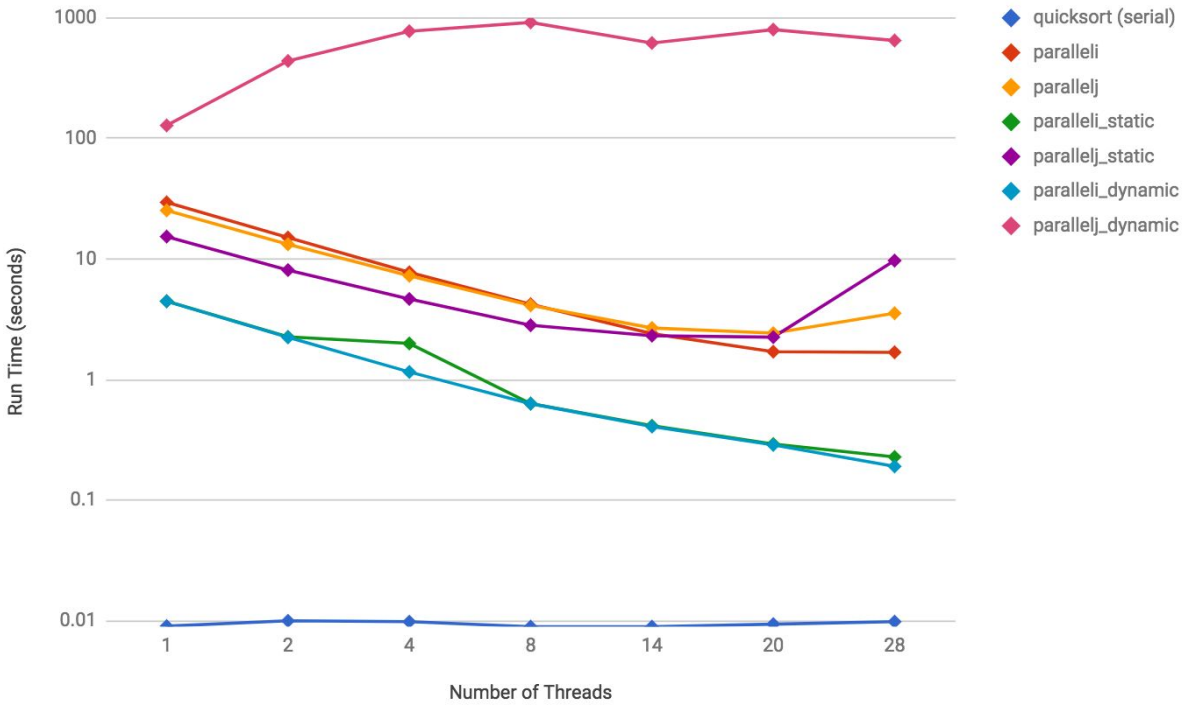
1.

Data for this problem is found in under cse415-mcgreg45/homework/4 at gitlab.msu.edu. The following displays show data originally recorded in hw4-1i.txt, each version of the student sorting algorithm described in HW4.pdf problem 1 was run for N=100000 on 1, 2, 4, 8, 14, 20, and 24 parallel threads, the sort was also performed using quicksort for comparison.

Number of Threads / Algorithm	quicksort (serial)	paralleli	parallelj	paralleli_static	parallelj_static	paralleli_dynamic	parallelj_dynamic
1	8.99E-03	2.95E+01	2.53E+01	4.4703	1.53E+01	4.4572	1.28E+02
2	9.96E-03	1.51E+01	1.33E+01	2.258	8.1016	2.2385	4.40E+02
4	9.79E-03	7.7621	7.2619	1.992	4.662	1.156	7.77E+02
8	8.91E-03	4.2143	4.126	6.31E-01	2.822	6.31E-01	9.19E+02
14	8.90E-03	2.4028	2.6779	4.14E-01	2.3098	4.07E-01	6.20E+02
20	9.34E-03	1.7011	2.4263	2.92E-01	2.2482	2.87E-01	8.01E+02
28	9.78E-03	1.6822	3.5507	2.28E-01	9.7059	1.90E-01	6.50E+02

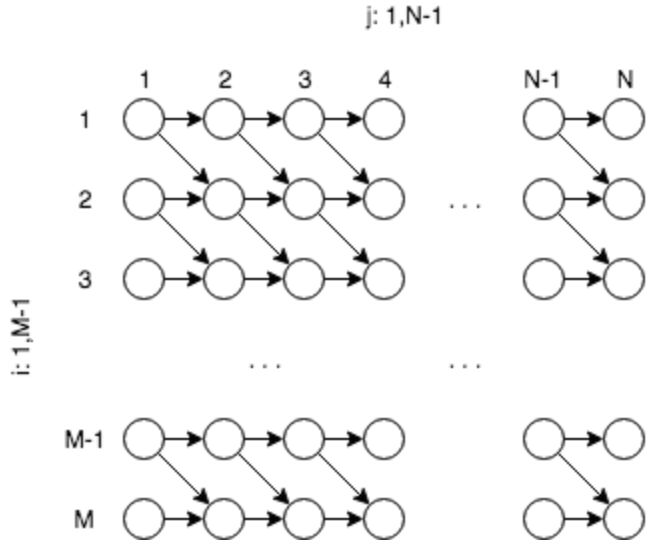


These results show the effectiveness of static or dynamic omp for scheduling for the outer loop (i) and the high cost of overhead for the same omp for implementations on the inner loop (j), specifically for the dynamically scheduled version. This slowdown occurs because of the build up and break down of omp threads required at each iteration of i to be used in parallelizing j.

Seeing that the some versions of the sort performed worse at 24 threads, 20 threads was chosen to test the effects of chunk size for static and dynamic scheduling. The i and j loops were parallelized with static and dynamic scheduling for chunk sizes 1, 2, 5, 10, 100, 1000. The results of this test can be found in hw4-1iii.txt, they show that a smaller chunk size than the default loop_count/numberOfThreads (5000 in this case) yields moderate increase in performance for i static, i dynamic, j static, and particularly j dynamic runs. A moderate chunk size is optimal as it exploits the load balancing of omp for and minimization of overhead, too small and the dividing up of tasks is cumbersome, too large and one or a few threads may get stuck with much more work than others and the rest will idle while waiting for the few to finish.

Quicksort out performs all versions by at least an order of magnitude.This is because quicksort rearranges items in place by rearranging pointers rather than copying to an extra temporary array like the summercamp algorithm. It’s advantage comes from much better use of spatial locality in memory.

2.



It is possible to parallelize this loop since for a given j there are no dependencies over iterations of i. The code can be refactored to take advantage of this by switching the inner and outer loops, though this change could have memory performance implications that should be explored in more detail.

The parallelized version is as folllows:

```
for(int j = 1; j <= N; j++)
        #pragma omp parallel for
        for(int i = 1; i <= M; i++)
                data[i][j] = data[i][j-1] + data[i-1][j-1]
```

3.

```
#include <mpi.h>
main (int arg, char *argv[])
{
        int npes, my rank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        int color = npes / 4;
        MPI_Comm ROWS_COMM;
        MPI_Split(MPI_COMM_WORLD, color, &myrank, &ROWS);
        MPI_Comm_free(&ROWS_COMM);
}
```