CS410: Parallel Computing

Assignment 1

1. Introduction

In this assignment, we implemented a parallel merge sort using Open Cilk and conducted various experiments related to parallelism whose observations can be seen in this report.

2. Pseudo Code

ParallelMergeSort(A, A_start, tmp, tmp_start, size, depth, cutoff):

```
1. if depth > cutoff:
2.     tmp[tmp_start...tmp_start+size]= A[A_start...A_start + size]
3.     SerialSort(tmp.begin()+A_start, tmp.begin()+A_start+size)
4. depth++
5. B = vector(size)
6. half = size/2
7. cilk_spawn ParallelMergeSort(A, 0, B, 0, half, depth, cutoff)
8. ParallelMergeSort(A, half, B, half, half, depth, cutoff)
9. cilk_sync
10. ParallelMerge()
11. Copy from tmp back to A
```

ParallelMerge(vec, A_start, A_end, B_start, B_end, tmp, tmp_start, depth, cutoff):

```
1. if depth > cutoff:
2.     SerialMerge(A, A_start, B, B_start, tmp, tmp_start)
3. depth++
4. median_A = A[A_start + (A_end - A_start)/2]
5. // Find position of Median of A in B
6. j = BinarySearch(B, B_start, B_end, median_A)
7. cilk_spawn ParallelMerge(vec, A_start, A_start + (A_end - A_start)/2, B_start, j-1, tmp, tmpStart, depth, cutoff)
```

```
8. ParallelMerge(vec, A_start + (A_end - A_start)/2 + 1, A_end, j,
   B_end, tmp, tmp_start + (A_end - A_start)/2 + j - B_start + 1,
   depth, cutoff);
9. cilk_sync;
```

3. Exploitation of Parallelism

In order to parallelize the merge sort, we exploit the recursive decomposition technique of designing parallel algorithms. Each recursive call to the Mergesort function happens in parallel. In other words, we sort the first half of the array in parallel to the second half of the array.

We also parallelize the merging of the two sorted arrays. Let's say we want to merge A and B, which are independently sorted. Without loss of generality, let A be bigger in size than B. We find the median of A in B. Let the position of the median of A in A and B be i and j respectively. Then we can recursively combine A[1...i] and B[1...j] and A[i+1...A_end] and B[j+1...B_end] in parallel.

In order to tradeoff speedup and overhead due to extra parallelism, we introduce cutoff and depth parameters. We can think of these parameters as controlling the granularity of parallelism. Whenever the depth of the above recursions increases beyond the cutoff value, we execute our program serially. This way we ensure that only an adequate amount of parallelism is created.

We also parallelized the `validate()` function by utilizing `cilk_for`. In order to avoid race conditions, we introduced a `reducer_opadd<int64_t>` for keeping a count of mistakes.

We also parallelized the `Init()` function by utilizing `cilk_for`.

Another way to exploit parallelism could be to use data decomposition techniques for designing parallel algorithms. We could first divide the input array to be sorted into chunks of equal sizes where each chunk could be sorted in parallel. Once, we had each of the chunks sorted then we could merge them using the `ParallelMerge` algorithm described in section 2. We chose to use the recursive decomposition technique because merge sort by its nature is a recursive algorithm and we could easily parallelize each of the recursive calls.

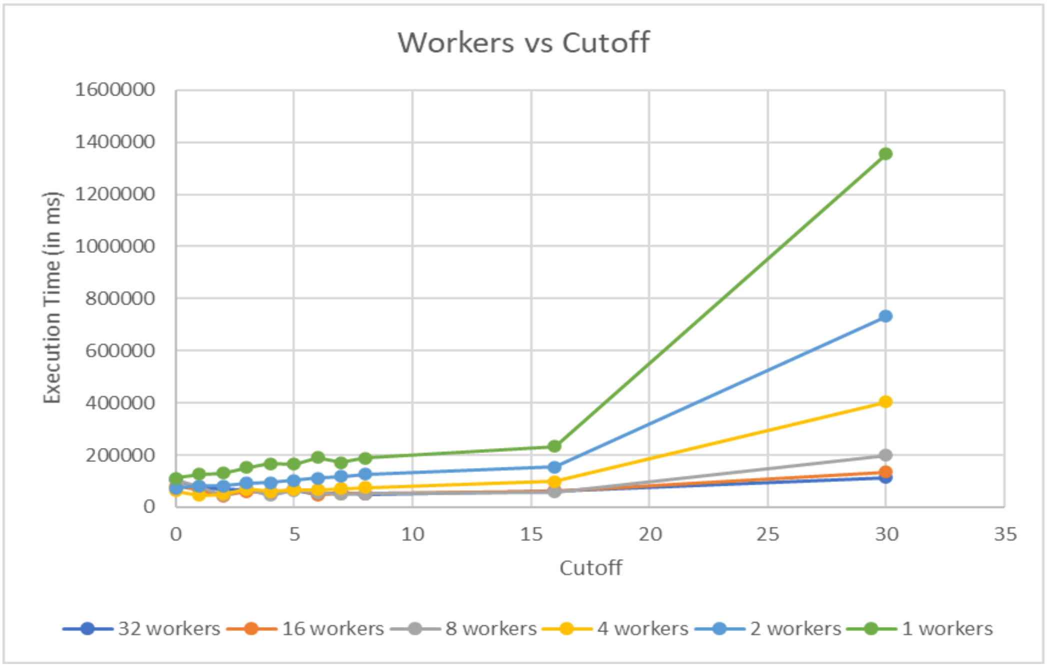
4. Synchronization of Parallel Work

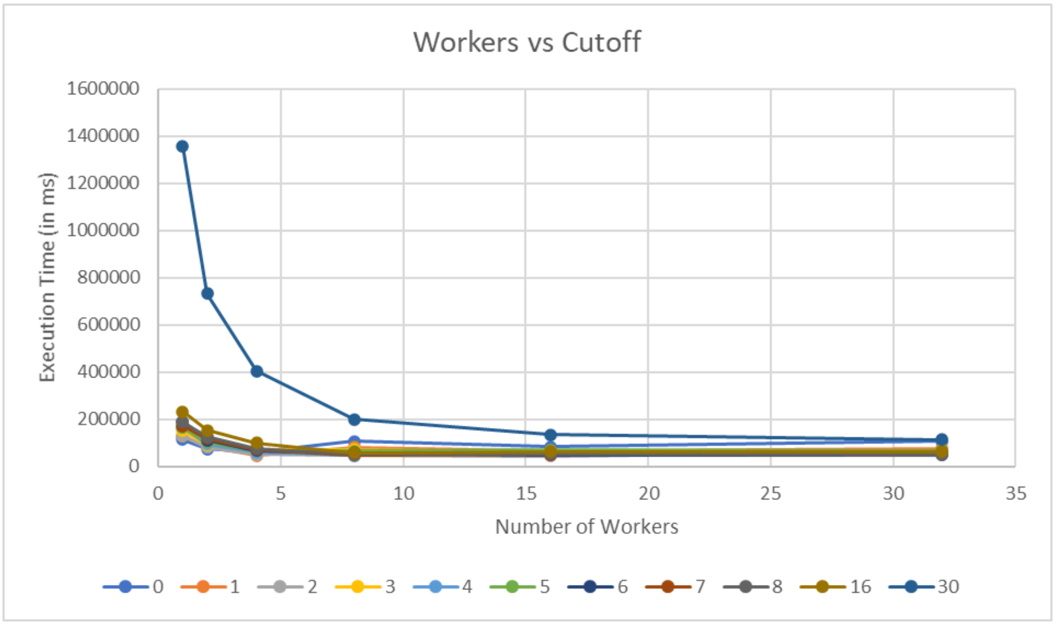
In order to synchronize the merge sort and parallel merge execution, we introduced `cilk_sync` after the recursive calls to the function (as shown in the pseudo-code for `ParallelMergeSort` and `ParallelMerge`). This way we ensure the parent waits for all the children to finish their work before progressing beyond `cilk_sync`.

Moreover, in order to avoid race conditions, we introduce a new array C, before recursively calling the `ParallelMergeSort` function (as shown in the pseudo-code for `ParallelMergeSort`).

5. Running Time

The following graph shows the variation of the execution time as a function of cutoff and number of workers. We ran the experiments with 1, 2, 4, 8, 16 and 32 workers and 0, 1, 2, 3, 4, 5, 6, 7, 8, 16 and 30 cutoffs.





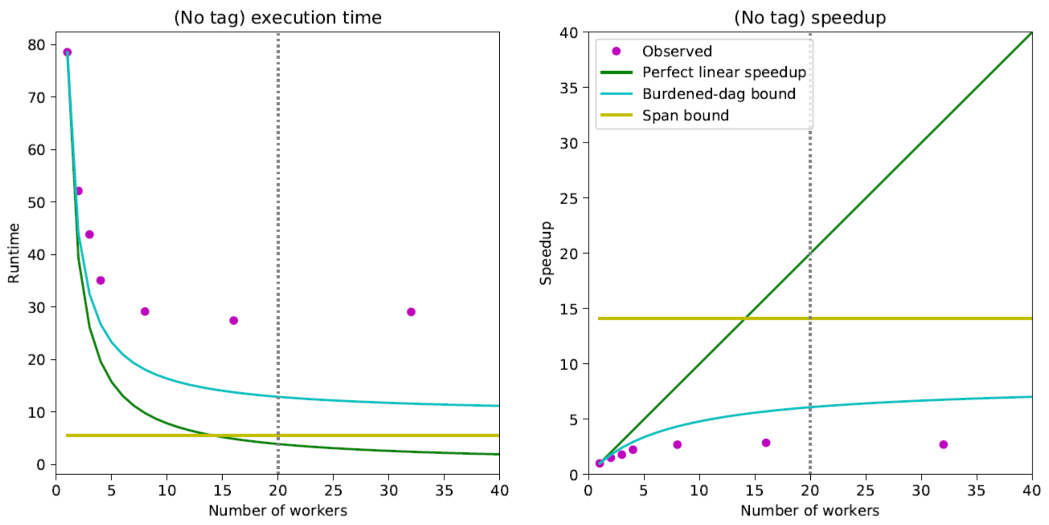
We can see that as we increase the cutoff, the execution time increases significantly. This can be attributed to the fact that when we increase the cutoff, we are doing a fine-grained parallelism. This leads to high-level overheads caused by excessive parallelism. We also observe that as the number of workers decreases, the execution time increases. This is because there are not enough workers for executing the work in parallel and thereby, the program runs almost as if it were running serially. From our experiments, we conclude that the cutoff between 5 and 7 works best.

6. CilkScale

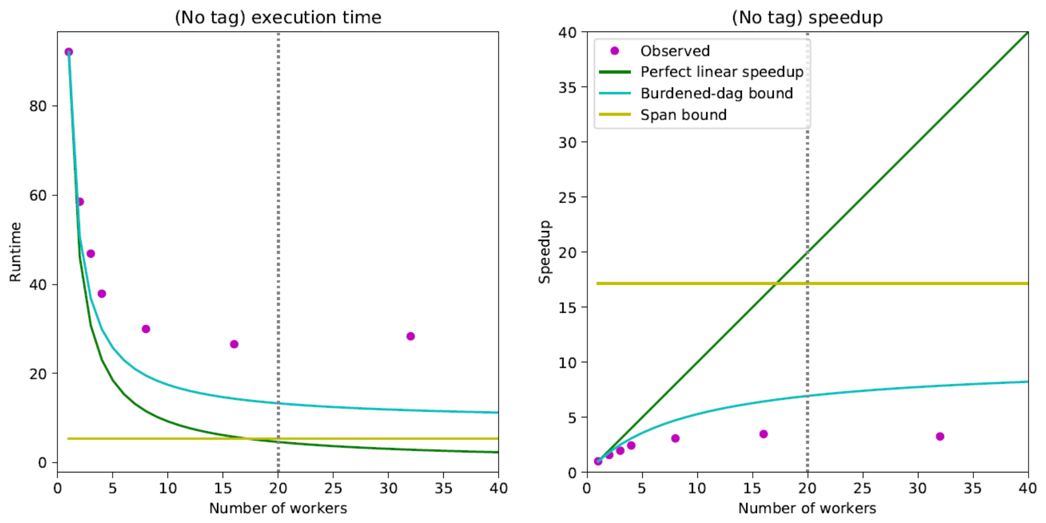
For running the CilkScale, we tried cutoff values of 2, 4, 8, 16 and 24. The results of the experiments are listed in the graph above and the table below.

We can observe that as we increase the cutoff, the work that is the number of instructions executed without parallel overhead increases. This is because when the cutoff increases, there is an increase in the number of recursive calls that are being made and thereby increasing the number of instructions. We don't observe much change in the span i.e. the number of instructions being executed on the critical path without parallel overhead, as the cutoff is increased. This indicates that the increase in the depth of the DAG due to an increase in the cutoff is balanced by the shorter execution time of each individual task. An interesting point is that the burdened span which is the number of instructions executed on the critical path with parallel overhead also remains the same as the cutoff is increased. This implies that the increase in the execution time of the program as the cutoff is increased is majorly due to the increase in the work.

Cutoff 2



Cutoff 4

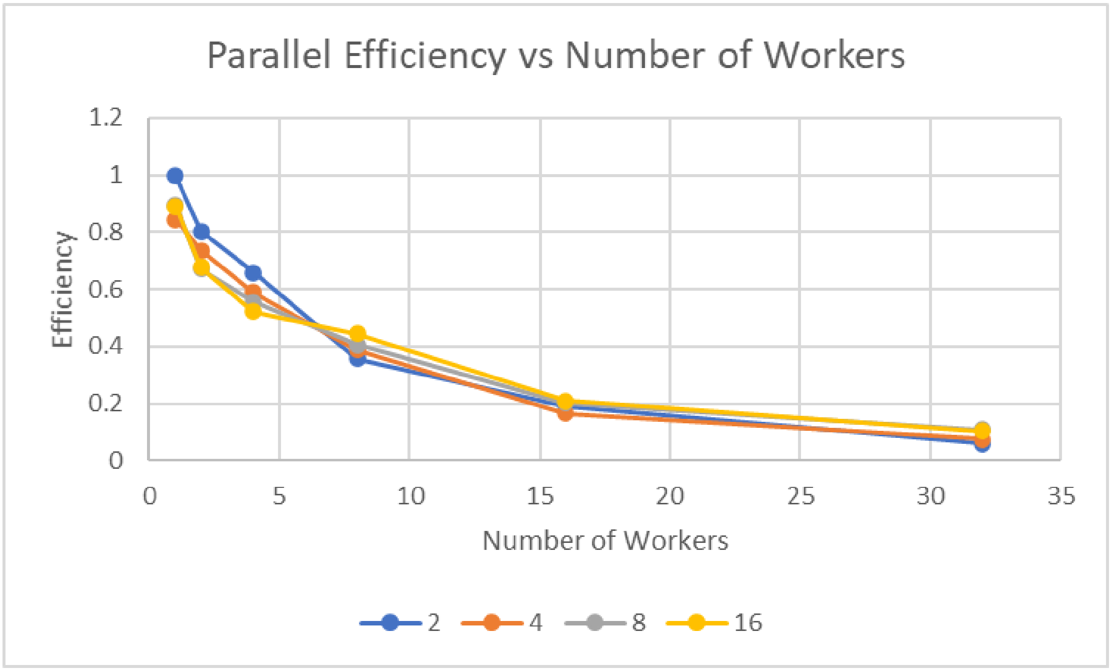


Cutoff	work (seconds)	span (seconds)	parallelism	burdened _span (seconds)	burdened_ parallelism	1c time (seconds)	2c time (seconds)	3c time (seconds)	4c time (seconds)	8c time (seconds)	16c time (seconds)	32c time (seconds)
2	404.505	28.6317	14.1279	28.6672	14.1104	78.5416	52.1274	43.8278	35.0844	29.1563	27.4461	29.0892
4	454.91	26.4889	17.1736	26.526	17.1496	92.1404	58.5217	46.8712	37.8862	29.9642	26.538	28.3168
8	712.234	30.9493	23.0129	30.9859	22.9857	114.388	70.627	54.2769	44.0205	32.5851	28.6159	27.9506
16	923.803	32.3995	28.5129	32.4353	28.4814	129.185	80.7565	59.8651	49.7506	35.3336	31.3161	34.005
24	1972.08	30.6884	64.2614	30.7277	64.1792	194.042	145.921	96.078	74.7811	44.6764	35.4922	34.8433

7. Parallel Efficiency

The following graph shows parallel efficiency as a function of the number of workers and cutoff values. We varied the value of workers between 1, 2, 4, 8, 16 and 32, and cutoff between 2, 4, 8 and 16.

We can observe that as the number of workers increases, the parallel efficiency decreases. This is because when we increase the number of workers, there is an increase in the overhead caused by sharing and stealing of work between workers. This causes an increase in the execution time of the program.


Parallel Efficiency vs Number of Workers
Efficiency
Number of Workers