Allen Tung (U)
Ming Tai Ha (G)
10/272016
ECE 451-566

Project 1 - Pthreads

1) Parallelizing Quicksort and Bitonic Sort

Parallelizing Quicksort: The quicksort algorithm first partitions the array into two subarrays based on the pivot element and then performs quicksort on the left and right subarrays. Since the subarrays which quicksort recursively calls upon are disjoint, this step can be parallelized. The partitioning step cannot be parallelized since this step requires a traversal across the array. Thus, the parallel implementation of quicksort is a sequential partitioning step to identify the pivot, and two new threads running quicksort on the disjoint subarrays which are split by the pivot. The parent thread waits for the children quicksort threads to join.

Parallelizing Bitonic Sort: The bitonic sorting algorithm takes an array and recursively calls bitonic sort on disjoint parts of the array. One part of the array is sorted in ascending order and the other part is sorted in descending order. Then, the bitonic sorting algorithm will call the merging algorithm to merge the two parts of the array so that the subarrays follow one order. The merging algorithm will swap elements in the given array to follow the order imposed. The merging algorithm will then recursively call itself to swap elements from the left and right subarrays, which are disjoint. So, the bitonic sorting algorithm can be parallelized by creating children threads to recursively run bitonic sort on the subarrays, then wait for the children threads executing bitonic sorts on the subarrays to join back to the parent thread before calling the merging algorithm. The merging algorithm will create children threads to recursively run the merging algorithm on the disjoint subarrays and wait for the children threads to join before returning back to the bitonic sorting algorithm.

Implementation Notes: The sorting arrays used to test the parallel and sequential sorting algorithms are have a length which is a power of 2 for exponential growth. Also, the user can only possible to specify 2^n - 1 number of threads to be created in order to simplify the implementation. This way of creating the number of threads in the implementation does not require any synchronization aside from the ones required to parallelize the sorting algorithm.

Performance Comparison: For all number of threads used. The quicksort algorithm performs better than the bitonic sort algorithm in terms for runtime. This is because quicksort has a worst-case runtime of O(n*log(n)) and bitonic sort has a worst-case runtime of n*log(n)^2).

Bitonic sort experienced a greater speedup performance when going from 1 thread to 3 threads than did quicksort. This is because the quicksort algorithm takes the last element as the pivot to partition the array into subarrays; there is no guarantee that the partitioning step will produce subarrays with similar sizes. Bitonic sort, however, will does partition the subarrays evenly. Thus, bitonic sort can more evenly split the work among threads than can quicksort.

As the number of threads used in both sorting algorithms doubled, the time required to sort the algorithms increased by a factor close to two. This is because the time spent sorting the algorithm is dominated the time it takes to create pthreads. Since the processor only supports 4 threads, we expected there to be performance speedups when using less that 4 threads (going from 1 thread to 3 threads) and performance losses when using more than 4 threads because the algorithm will create more pthreads than the CPU can execute at any point in time.

<u>Timings Results:</u> The algorithms were run on the Acer Aspire V15 Nitro Black Edition laptop, which has an Intel i7-5500U processor and 8GB RAM. The Intel i7-5500U processor supports 4 threads of execution. The number of threads spawned never exceeds the size of the array. Otherwise, the extra threads would not be used for sorting at all. The timings of the sorting algorithms are given below. We observed fluctuations in the time required to create pthreads, and so each timing below is the median of 11 runs.

Quicksort - All timings are reported in seconds

	Array Size		
# Threads (2^n - 1)	1024 (2^10)	16384 (2^14)	131072 (2^20)
1 (sequential)	0.000133	0.001906	0.018344
3	0.000099	0.000815	0.015214
7	0.000324	0.001755	0.016428
15	0.001470	0.003296	0.015171
31	0.012764	0.018772	0.020782
63	0.038023	0.038676	0.041100
127	0.078646	0.078543	0.090476
255	0.143776	0.174067	0.209922
511	0.193874	0.363594	0.347269
1023	0.242774	0.739782	0.652137
2047	N/A	1.095714	1.243457
4095	N/A	1.740549	1.848415
8191	N/A	2.805303	3.551824

16383	N/A	3.708458	4.413022

Bitonic Sort - All timings are reported in seconds

	Array Size		
# Threads (2^n - 1)	1024 (2^10)	16384 (2^14)	131072 (2^20)
1	0.000444	0.010395	0.113517
3	0.000193	0.007119	0.109569
7	0.000656	0.005133	0.117570
15	0.006606	0.016214	0.147339
31	0.042740	0.040716	0.163533
63	0.152433	0.151396	0.247572
127	0.352414	0.364094	0.430554
255	0.873200	0.932445	1.022036
511	2.287788	2.351477	2.345626
1023	4.937112	4.994215	5.144061
2047	N/A	11.402923	12.037989
4095	N/A	22.787390	24.271001
8191	N/A	39.986085	41.686996
16383	N/A	76.744795	78.027316

2) Parallelizing Gaussian Elimination

Implementation details:

When parallelizing gaussian elimination, one may decompose the problem into square submatrices, in order to produce a upper right trigonal matrix. After processing each submatrix, we may disregard the leftmost and uppermost column and row, as no more operations will be performed on them. In this way, we obtain the next submatrix to operate on.

The specific operations performed are as follows:

The top row, A_0 of size n is divided by its first element, $A_0[0]$.
Each following row A_1, A_2, A_3 … A_{n-1} is similarly divided by their respective first element
A_0 is then subtracted from each of the specified rows A_1, A_2, A_3 … A_{n-1}

In each submatrix, the size of the matrix is n*n, with n decreasing as we progress. With the problem specified as such, we realize that each subproblem must be solved sequentially - there is no way to obtain the matrix for the subproblem without having computed each previous sub problem. Therefore, the parallelization must occur within each sub problem.

In order to reduce cache hits, we hand each thread a consecutive block of rows, storing our matrix in row-major form. We will have each thread processing several rows of the sub matrix at a time, and allow the user to specify the number of threads used. When we reach a point at which the number of threads is greater than the number of rows, we assign one thread to each row, and leave the remaining threads to idle.

Since the rows are passed to each thread as merely the pointer to the beginning of the sub-matrix, there is little overhead with copying arrays and retrieving results- each thread adjusts the matrix directly. Since the matrix is accessed by each thread directly, however, we must ensure that no concurrency issues are present. To this end, we require a barrier to block all threads upon finishing their assigned work, until every thread has completed. Only then can we begin editing the matrix values in the main thread, and assigning the new rows to each thread to begin their new work. The specific implementation uses pthread_barrier to synchronize threads.

What didn't work:

In a previous implementation, pthread_join was used to synchronize the threads, which led to a very large number of threads being created- In each subproblem, the user-specified number of threads were created to work, then deleted. This caused major performance issues, as the thread creation overhead was staggering. To increase performance, which is the goal of parallelization, we instead create the required number of threads only once, and simply block while waiting for the main thread to supply new arguments.

Timing details:

Array creation time for large matrixes takes a rather long time, and so the start time is recorded once the array is created. Upon the completion of all assigned tasks, each program prints the output to the console.

Performance comparison:
The current implementation of the parallel implementation seems to perform slightly more poorly than the sequential version with any number of threads, which may be attributed to a number of reasons.

In creating the parallel version of Gaussian Elimination, there are additional calculations to determine what is passed to each thread, as well as additional memory allocation. The initialization and waiting on pthread_barrier also consumes a relatively large amount of time.

There may be some reason which limits the number of cores used to one, which would mean that there is only pipelining. In this case, no two threads would compute at the same time, resulting in no improvement over sequential elimination. The overhead in creating threads, and the overhead of synchronization would add up to make the parallel version take slightly more time.

The time measured is the number of clock cycles used, converted to seconds and milliseconds, so pipelining would not affect the measured performance. Since the computation time of the sequential and parallel versions of the program are similar, the claim that there is only one core processing information is further substantiated, with the additional time taken by the parallel version attributable to the difference.

Increasing the number of threads worsens the performance, substantiating the claim that the thread overhead as well as synchronization overhead is detrimental to the performance of the parallel program. If we presume that each thread is running on one core, this claim would be reasonable.

The variation between instances of running the program on the same number of threads and array size is small for small matrices, but increases with the size of the matrix and the number of threads. This may be attributed to the synchronization primitives, taking slightly more or less time for the threads to synchronize and proceed, which occurs n times. The synchronization also must have all threads at the barrier, so the more threads there are, and the larger n is, the more variability there will be between runs.

For either edition of Gaussian Elimination, the number of operations required increases by a factorial of the size of the array, which causes the absolute time taken to lengthen considerably.

We test very large numbers of threads for a 1000x1000 matrix, and we see how the thread overhead affects the performance of the program significantly. Even with each thread doing very little work, the overhead of creation and waiting in the synchronization periods adds entire seconds to the computational time. Even if there are threads doing absolutely nothing, the overhead is still significant.

Sequential performance

N parameter of matrix	Seconds	Milliseconds
9	0	0
90	0	0
900	0	900
1000	1	250
1500	4	230
2000	10	90
2500	19	660
9000	912	580

Parallel performance

N parameter of matrix	Number of threads	Seconds	Milliseconds
9	4	0	0
90	4	0	0
900	4	0	890
1000	4	1	10
1500	4	1	220
2000	4	10	750

2500	4	20	220
9	8	0	0
90	8	0	0
1000	8	1	580
1500	8	2	180
2000	8	17	470
2500	8	34	540
1000	16	2	250
1000	16	2	240
1000	16	2	280
2500	16	35	960
2500	16	36	0
2500	16	35	890
1000	32	2	320
1000	32	2	300
1000	32	2	310
2500	32	35	780
2500	32	35	680
2500	32	35	830
1000	250	3	250
1000	500	4	360
1000	750	6	90
1000	1001	7	300
1000	2000	14	250