HPC Lab 5: Report

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

October 28, 2016

Part I

Hardware Details:

- **CPU model:** Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- **Memory Information:**
  1. L1d cache: 32K
  2. L1i cache: 32K
  3. L2 cache: 256K
  4. L3 cache: 15360K
- **No. of cores:** The CPU has 12 physical cores with 1 thread per core. This means, there are 12 logical cores.
- **Compiler:** gcc version 4.8.4
- **precision used:** We have used double precision wherever possible so as to get more accuracy in the result.

Part II

Q1. Performing prefix scan

1 Context:

- **Brief description of the problem.**

  Prefix Scan till a point in an array is the cumulative addition of all the numbers till that point in the array. The problem asks for prefix scan of whole array. So, position 'i' contains sum of all the elements from 0 to i. And the final position contains the sum of whole array

- **Complexity of the algorithm (serial).**

  The algorithm works for the whole array of size N and adds each element to its next element. Hence the serial running time is of O(N)

- **Possible speedup (theoretical).**

  Method 1: We can arrange the prefix scan to work in a tree and inverse tree structure. Assuming we theoretically have infinite number of processors, the forward tree can be computed in (log N) time. Similarly the inverse tree will also take nearly (log N) time. Therefore, total time taken by the code should be (2 log N). This will result in a speedup of N / (2 log N).

  Method 2: We can perform simple work division among threads, calculate the prefix scan for partial arrays and then add the last element of the previous array to all the elements of next array. This has a lot of data dependency. Hence we would not be able to get linear speedup. We will traverse the array twice, hence the speedup should be (2*N/p) for p processors.

- **Optimization strategy:**

  We have used the following optimization strategies to parallelize and optimize the processing time.

  1. Changing the structure of code completely and using reduction tree and inverse tree structure to perform scan
  2. Using intermediate scan which is much closer to simple work division among threads but with added processing of an intermediate array and the addition of intermediate sums back to the original array to get the full scan. Here the first scan after work division corresponds to the reduction tree which intermediate array processing and adding back to the original array corresponds to the inverse tree.

- **Problems faced in parallelization and possible solutions.**

  Since we **don't have a large number of processors**, which is the basic requirement for the tree structure to work at its best, the **tree implementation** only gives a **speedup of 1**.

  Load Imbalance is also a problem in the tree structure which reduces the effective speedup. It increases the idle time in the total code and hence increases the total overhead. Therefore the efficiency decreases. The cost of execution increases. Hence tree structure is **not cost efficient**.

  In **Intermediate scan** also, we need to take care of the data access both by the intermediate array and the original array which must not go out of bounds and must be added to proper divided part of array among each thread.

2 Input. Output

The input for the serial method, parallel tree method and the intermediate scan method is the prefix scan of the full array. The correctness can be checked merely by checking if the last element contains the sum of whole array.

3 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

Number of Points = 1e8

Serial code: Time taken = 0.2941 s

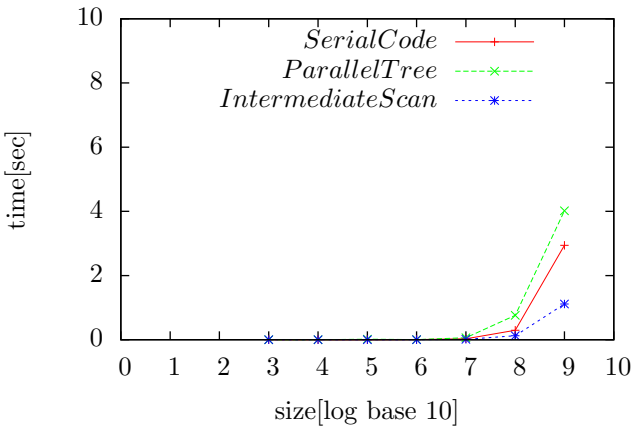
Parallel code (1 thread): Time taken = 0.4535 s

Overhead = 0.4535 - 0.2941 = 0.1594 s (54% of serial time!)

As per the theory, the speedup should be half the serial code. We are getting better than half because there is no intermediate scan array for 1 thread and the code is almost similar to the serial code. But since there is a lot of added overhead and since the code traverses the array the second time in parallel code, the time increases drastically.

4 Problem Size vs Time (Serial, parallel) curve:

Graph to plot the variation in time compared to the number of points in log scale



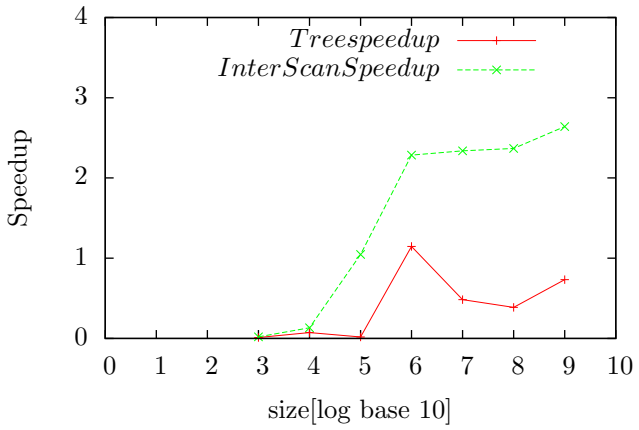
The above graph shows that when the problem size is small, the amount of time taken to perform scan is close in both cases since the overhead time of the parallel code is comparable to the total time of serial code.

As the size increases, because of lack of sufficient processors to perform the tree method efficiently, the tree method almost behaves like a serial code but with added overhead of dividing work into threads. Hence the time taken by the tree method is even more than the serial code for any input.

The intermediate scan is the simple work division but with a clever use of an intermediate array to accomplish the prefix scan. Hence the original array is traversed twice. But with p processors. Hence we require less time.

5 Problem size vs. speedup curve.

Graph to plot the variation in speedup compared to the number of points in log scale



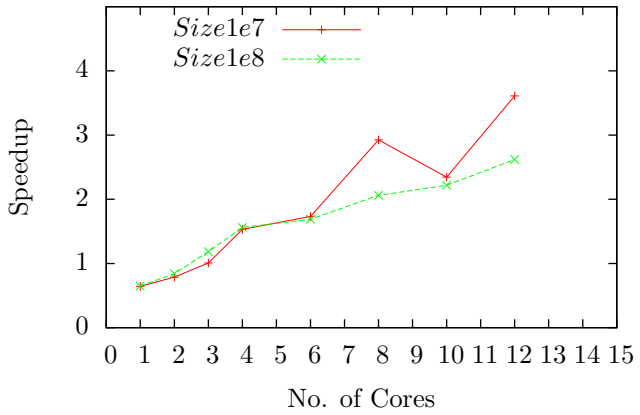
As usual, for small problem size ¡ 1e5, the serial code is much faster, we reinforces the fact that the parallel code must be run for a large input.

For the intermediate scan method after a particular problem size, as the problem size increases, the speedup increases. Hence it is a weakly scalable code.

For Tree method, we see an interesting trend. Here initially, we see more than 1 speedup for problem size 1e6. But as we gradually increase the problem size keeping the number of processors same, The amount of computation by each processor increases. Also, since there is dependency between the forward and inverse tree computations, the time is not reduced at the same rate. Therefore speedup decreases.

6 No. of cores vs. speedup curve for a couple of problem sizes.

Graph to plot the variation speedup compared to the number of cores



For the intermediate scan method, we expect a speedup of nearly (p/2) for p processors. The trend is seen in the red graph. Here, for 4 and 8 processors, the graph nearly achieves desired speedup. Hence we infer that the machine that we used to run this code had a lot of load at that time, hence it could not give the required speedup to the accurate degree but it has given the correct trend.

7 Measure performance in MFLOPS/sec.

For serial code, we have total 1 Floaing Point Operation per cycle of for loop and we have N array elements meaning N number of cycles. Therefore we have total N FLOP. We measure the time taken to execute those many operations.

In parallel implementation with intermediate scan we have total 1 Floaing Point Operation per cycle of for loop to generate array for intermediate scan. We have number of thread of operations for intermediate scan array and we have total 1 Floaing Point Operation per cycle of for loop to add the intermediate scna back into the original array. We have N array elements meaning N number of cycles. Therefore we have total 2*N+(no. of threads) FLOP. We measure the time taken to execute those many operations.

$$performance = No.of MFLOP/time\_taken(sec)$$

For parallel code, performance = 383.467260 MFLOPS

For serial code, performance = 159.222369 MFLOPS

The performance is nearly twice, which is comparable with the theory. It is just more than twice because the number of operations performed are more than twice in half time.

8 Conclusion

This is a special type of question where there is no straight-forward parallel implementation. There are **various methods to parallelize the code**. **Depending on the availability of resources, some of the parallel implementations work better than others.**

One of the parallel solutions is the **implementation using tree structure**. But tree structure **works the best when number of available processors are equal to the problem size**. Here we have a **limited number of processors**, hence tree structure does not work to its full potential as seen earlier and gives **speedup less than 2**.

Another crucial method which works best for our situation is the **intermediate scan method**. This is an innovative method which is **built on the simple work division method** used in earlier assignments. It **can be parallelized for any number of threads but for a large number of threads it would have very high overhead.**

Also we **cannot get linear speedup in this problem** due to the **hardware constraints** and the **data dependency present in the algorithm**. At best we can get a **speedup of (2*N/p)**.

Part III

Q2. Implementing and efficient Filter

1 Context:

- **Brief description of the problem.**

  The main functioning in this problem is the filter which, given an array and a threshold element for that array, would filter out all the elements greater than or equal to the threshold and copy them in another array in the same order. Here we have to take array size greater that 1e6 with array elements ranging from 0 to 100. For convenience purpose we will keep the threshold at 10, meaning we will take all the elements greater than or equal to 10 during filtering.

- **Complexity of the algorithm (serial).**

  For serial code, before making the filter array, we must determine its size which can be found out by traversing the array and counting the number of elements greater than or equal to threshold. This takes time O(N) for array of size N. Once the size is determined, we again have to traverse the original array to actually filter it which again takes O(N) time. So overall we need O(2*N) time to complete the filtering operation serially.

- **Possible speedup (theoretical).**

  Here we have used a flag array and the prefix scan method to perform filtering. To generate the flag array, we must traverse through the original array which takes N/p time with p processors. Now we perform prefix scan which takes (2*N/p) time and finally we traverse the flag array to filter the elements which again takes (N/p) time. So in total theoretically the time required would be O(4*N/p). Therefore, speedup would be

  $$2 * N/(4 * N/p) = p/2$$

- **Optimization strategy.**

  We have used the following optimization strategies to parallelize and optimize the processing time.

  1. First we have simply used #pragmas to parallelize the serial code. Since there is some scope for parallel work while counting the elements grater than the threshold.
  2. Next we have used a completely different method from the serial code which generates a flag array and makes use of the prefix scan to identify the locations of elements greater than threshold. This method is quite innovative and difficult to get hold of the first time.

- **Problems faced in parallelization and possible solutions.**

  While simply parallelizing the serial code, we encountered a problem because there is loop dependency in the code which cannot be removed and hence the desirable speedup cannot be achieved.

  In the filter with prefix scan, we must be careful while identifying which sections of code are embarrassingly parallel and where is loop dependency. Also while actually parallelizing the code, we have written 3 different parallel sections of code. If we miss even one of them, the speedup decreases by a significant amount.

2 Input parameters. Output.

The input parameters are the input array which needs to be filtered, the output array where the filtered elements are stored and the size of the original array.

The output is the filtered array. We have ensured that while using any of the above methods, we should get accurate output array of filtered elements.

3 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)
For Filtering using prefix scan
problem size = 1e8

Serial code: Time taken = 8.2380 s

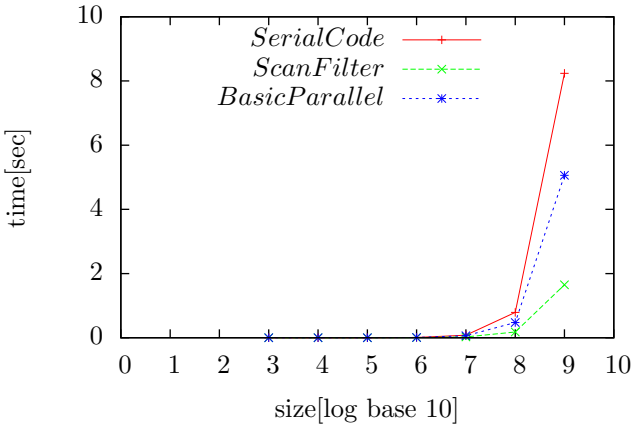
Parallel code (1 thread): Time taken = 13.9580 s

Overhead = 13.9580 - 8.2380 = 5.7800 s (69 % of the serial time!)

The **large overhead in the parallel code with 1 core** can be attributed to the fact that we are creating a whole **new array of the same size** as the input array and processing it on 1 thread. Therefore, we are making O(3N) iterations, N on original array to generate flag array, N on the flag array to perform prefix scan and N on the flag array for filtering. So we are making N iterations more that serial code and with the 1 processor only. Hence the overhead.

4 Problem Size vs Time (Serial, parallel) curve:

Graph to plot the variation in time compared to the number of points in log scale



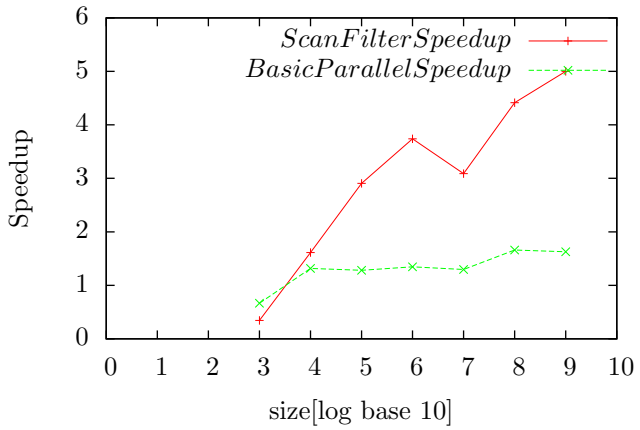
As we have seen, the actual difference in execution time is visible only when the problem size is large. Here we have used 12 threads for both the parallel code and the code which uses scan.

Simple Parallelization method represented by the blue plot takes too much time. This is as mentioned earlier, because of the loop dependency in the method which restricts the scope of parallelization.

The other method which uses **prefix scan** to filter out the elements seems to be working very well as the time taken by this method remains less even when the input size increases by a fair amount.

5 Problem size vs. speedup curve.

Graph to plot the variation in speedup compared to the number of points in log scale



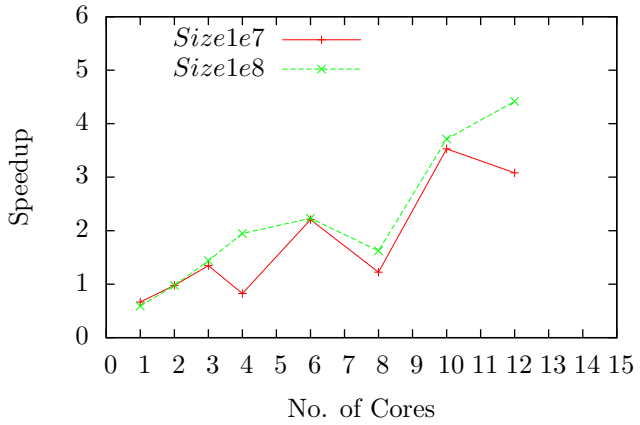
Both the code using scan and the basic parallel code have used 12 threads. As the problem size increases from 1e3 to 1e9, we see stark contrast between the speedup obtained by the two methods.

The method which uses **prefix scan** to implement filter works better and better as the problem size increases. This is because as the problem size increases, the problem becomes coarse granular. The flag array on which the prefix scan is to be performed becomes larger hence the prefix scan also contributes to the speedup. The amount of computation among the processors increases while the communication among them largely remains the same.

On the other hand the method of **simple parallelization** does not perform well even for large problem sizes. The speedup is almost between 1 and 2. It seems that there is a lot of serial part in the code which is the bottleneck in parallelization. This can be attributed to the huge data dependency that is present when transferring the filtered elements to the second array.

6 No. of cores vs. speedup curve for a couple of problem sizes.

Graph to plot the variation int speedup compared to the number of cores for filtering using scan



For Filtering using scan method: Here we expected a speedup of p/2 for p processing elements. The graph closely matches the theoretical prediction. For 2, 3 and 4 processors, the corresponding speedup is almost 1, 1.5 and 2 respectively. But as the number of cores increase, the only bottleneck in the code which is the overhead in the prefix scan of the flag array, increases. This might be one of the reasons why the obtained speedup is somewhat less than (p/2).

7 Measure performance in MFLOPS/sec

According to the problem, we have only 1 Floaing Point Operations per cycle of for loop and we have N cycles (size of array). Therefore we have total N FLOP. We measure the time taken to execute those many operations.

$$performance = No.of MFLOP/time\_taken(sec)$$

For parallel code, performance = 830.346694 MFLOPS
For serial code, performance = 229.084619 MFLOPS

Here we have found out FLOPS for only the simple division method. This is because we have already observed that since we have limited number of cores, the simple division is faster while the tree method would take much more time. Since we are simply dividing the work among 4 threads for the majority of the part, the expected speedup is almost 4. Here we are getting slightly less than 4. This can be explained by the overhead caused during parallelization, the data access and the critical section present in the code.

8 Conclusion

In this problem, the simple method of parallelization using pragmas does not give required speedup since there is **dependency in the code** where filtered elements are copied into the output array. This dependency cannot be removed and hence the corresponding block of code cannot be parallelized.

Another method is the complex one using the prefix scan. Here we are **performing the prefix scan using the intermediate array scan method.** This method **removes the loop dependency of the previous method** at the expense of using **extra space to store the flag array**. There are three blocks in the code which can be parallelized separately. **Two of the blocks give embarrassingly parallel** performance but the main **bottleneck is the prefix scan** which takes most of the time because it is **not work efficient**. The **prefix scan performs 2*N operations instead of N** and hence the **total speedup obtained is not linear.**