# HPC Lab 4: Report

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

September 23, 2016

## Part I

## Hardware Details:

- CPU model: AMD A10-5750M APU with Radeon(tm) HD Graphics
- memory information:
  1. L1d cache: 16K
  2. L1i cache: 64K
  3. L2 cache: 2048K
- No. of cores: The CPU has 4 physical cores with 1 thread per core. This means, there are 4 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. Calculating pi using random numbers

### 1 Context:

- Brief description of the problem.

  We have to use monte-carlo method to find the ratio of generated random numbers lying inside a circle of diametre 2 units to square of side 2 units.

$$pi/4 = no.ofpointsinsidecircle/no.ofpointsinsidesquare$$

Table 1: Serial Profiling

| index | % time | self | children | called | name |
|---|---|---|---|---|---|
| | | 2.77 | 0.00 | 1/1 | main [2] |
| [1] | 100.0 | 2.77 | 0.00 | 1 | pi_rand [1] |
| | | | | | > |
| [2] | 100.0 | 0.00 | 2.77 | | main [2] |
| | | 2.77 | 0.00 | 1/1 | pi_rand [1] |

- Complexity of the algorithm (serial).

  The algorithm works for the number of points(N) which are generated randomly to find the value of pi. Hence the serial running time is of O($n$)

- Possible speedup (theoretical).

  We run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999980

  By the definition of random numbers, each generated random number is independent of the previously generated numbers and the numbers are evenly distributed throughout the range.

  In C, the rand() function actually generates pseudo-random numbers. i.e. each group of numbers has a certain seed which is assigned to them and numbers generated from that seed are repeated after certain intervals of time.

  Still, there is no dependency on any previous or future iterations and hence whole code can be parallelized.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

- Profiling information (e.g. gprof). Serial time.

  granularity: each sample hit covers 2 byte(s) for 0.36% of 2.77 seconds

  With reference to Table 1, Clearly there is only one function which is doing all the work. The major work includes generating random numbers which is independent of the other iterations. Hence we need to parallelize this pi_rand() function to parallelize almost the whole code.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. Simply parallelizing the code by using pragma omp parallel for.
  2. the count variable to count the number of points lying inside the circle can be used with reduction

3. Since simple parallelization gave us speed-down, we thought of using another function rand_r which takes in the value of seed in order to determine random numbers.

- Problems faced in parallelization and possible solutions.

  The serial code was very easy but the major problem was how to parallelize it. First we tried the simple way just by using the pragma. Here instead of getting a speed-up we got speed-down! This is because the random numbers in C are actually pseudo random numbers. These numbers are divided in certain number of huge groups each having a seed. Also after each number generation, the groups go into a particular state.

  When we try to parallelize it using simple pragma, the seed for the program is fixed and hence all the threads use the same seed. Therefore all the threads access the same group during rand() call. Since each group changes to a particular state after rand() call, that state must be reflected in all threads. Hence there must be mutual exclusion in the threads otherwise, same value will be generated by all the threads, which is the same point on the circle or square. This is why rand() is not thread-safe. Due to mutual exclusion, the time of execution for all the threads combined increases even above the serial code. Hence we get speed-down

  The solution is to make the seed of each thread different so that the group accessed by each thread is different and the numbers generated are different. This eliminates the possibility of repetition and hence mutual exclusion is no more needed. The threads can work independently

## 2 Hardware details:

We have already put all the hardware details at the beginning.

Here the cache size does not matter much since there is no data access. The whole problem works solely based on generating random numbers and hence no memory access is done.

## 3 Input. Output

The input for both the serial and parallel codes is the number of random points we want to take as data set over which we will apply the above mentioned formula to find pi.

The output should be approximate pi value. Here we are getting pi which is correct up to 4 decimal places.

## 4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  Number of Points = 1e9

  Serial code: Time taken = 33.497898 s

  Parallel code (1 thread): Time taken = 33.591260 s

  Overhead = 33.591260 - 33.497898 = 0.093362 s ( 0.28 % of serial time)

- CASE II:

  Number of Points = 1e8

  Both serial and parallel code give the same warped image

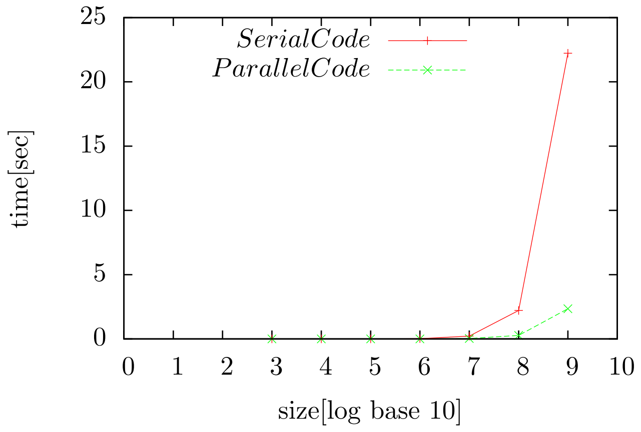
  Serial code: Time taken = 3.354567 s

  Parallel code (1 thread): Time taken = 3.366928 s

  Overhead = 3.366928 - 3.354567 = 0.012361 s ( 0.36 % of serial time)

## 5 Problem Size vs Time (Serial, parallel) curve:

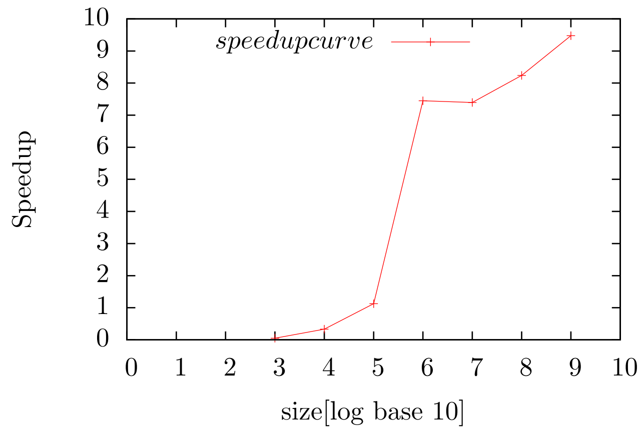
Graph to plot the variation in time compared to the number of points in log scale



The above graph shows that when the problem size is same, the amount of time taken to generate random numbers 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, the processing part increases and the communication decreases comparatively. Hence the time taken by the parallel code in comparison to the serial code decreases

## 6 Problem size vs. speedup curve.

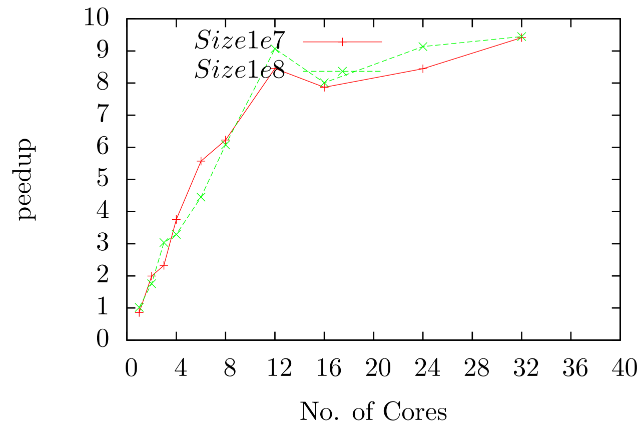
Graph to plot the variation in speedup compared to the number of points in log scale



A point to be noted. Here the speedup is increasing with increase in problem size. As the number of points increase, the amount of processing increases but the communication overhead does not increase with the same rate. Hence granularity and speedup increase. The speedup continues to increase with the problem size at the end indicating the scalability of the problem.

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

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



Here, the speedup increases almost linearly till about 6 cores. After this as the overhead increases while the problems size and hence the computation is same, the efficiency starts decreasing. We observe that there is a dip in both the graphs for 16 cores indicating that the problem distribution is chunksizes for 16 cores is worse than lesser core while the communication overhead is much more. The graph seems to flatten out at the end indicating that the overhead is matching the computation and the granularity is decreasing.

## 8 Measure performance in MFLOPS/sec.

According to the problem, we have total 6 Floaing Point Operations per cycle of for loop and we have N number of points meaning N number of cycles. Therefore we have total 6*N FLOP. We measure the time taken to execute those many operations.

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

For parallel code, performance = 679.435081 MFLOPS

For serial code, performance = 179.635983 MFLOPS

The performance of the parallel code is more than thrice the serial code. But theoretically it should have been 4 times! The reason that the performance is 3 times is because of the pipeline stalls. Since the operations contain conditional executions for incrementing counter based on the position of the point. Also, the rand_r function may take variable amount of time based on its state. Also since only execute phase of the pipeline is used, there is no optimal use of pipeline and hence the performance optimization is limited.

## 9 Conclusion

The logic of the problem (the Montecarlo derivation for the approximate value of pi) was pretty simple and took a very short amount of time to code.

However, the lack of thread safety of the rand() function was the key point of this question. It was to remind us that while parallelizing a built in function we make overlook an important point which might lead to undesirable results. Hence it is important to make sure that the functions which we use are thread safe.

Once, the problem that each thread had to wait to update the state of the Random Number Generator was resolved, we could get a satisfactory speedup.

Code Balance and Machine Balance: Here we need not calculate these two since there is no memory access. We are directly calculating random number and hence no effect of memory bandwidth.

Table 2: $ProfilingTable$

| index | % time | self | children | called | name |
|---|---|---|---|---|---|
| [1] | 100.0 | 6.08 | 4.23 | | main [1] |
| | | 4.23 | 0.00 | 1/1 | reduction_serial [2] |
| | | | | | |
| | | 4.23 | 0.00 | 1/1 | main [1] |
| [2] | 41.0 | 4.23 | 0.00 | 1 | reduction_serial [2] |

# Part III

# Q2. Implementing Reduction

## 1 Context:

- Brief description of the problem.

  Reduction is an important feature of openMP. It works on associative operations and can act as private as well as shared variable simultaneously. It is important to understand internal working of the reduction in the context of multi-core architecture. Therefore we have to implement reduction functionality on our own and then compare it with the original reduction tag

- Complexity of the algorithm (serial).

  We have taken the operation as sum. The code can easily be extended to other reduction compatible operations. The serial code is essentially adding up all numbers in the array into a single variable.

- Possible speedup (theoretical).

  For the sake of completeness, we run the code to find the fraction of parallelizable code

  Fraction of parallelizable code (p) = 0.999994

  The simplest and most effective way to perform parallel reduction is to simply split the array into chunks equal to the number of threads. Each thread will add up its chunk serially and thus we would get a speedup close to number of processors.

  Therefore by Amdahl's Law the Theoretical speedup that we should get = No. of processors = 4 in this case.

- Profiling information (e.g. gprof). Serial time.

  Again for the sake of completeness we are taking the profiling information.

  granularity: each sample hit covers 2 byte(s) for 0.10% of 10.32 seconds

In Table 2, We see that almost 59% time is spent in main function but in main we only initialize the array. We don't take initialization into account. Therefore major time is take in logic of reduction. For serial code this just calculates the sum of all elements of array. We must parallelize the 'for' loop inside the function.

- Optimization strategy.

  We will use the following opimization strategies to parallelize and optimize the processing time.

  1. We used 2 different strategies to parallalize this problem. The first was the simple yet effective method of dividing the problem in chunks = number of processors as described above. We expect the problem to give very good speedup with this strategy.
  2. The other method was to use a tree stucture in which treads would reduce the problem level wise by sequentially running through the problem at each level and reducing the problem by a factor of 2.
  3. We suspect that this method will suffer from less speedup as it requires writing the computed values to an array instead of private variables.This may lead to false sharing. Problems of cache invalidation might arise and it will not exploit cache reuse.
  4. Note that the first method will have a lot of cache reuse as problem has been divided into large chunks.

- Problems faced in parallelization and possible solutions.

  As we suspected, the speedup up obtained in the second method is quite poor. We think, this may be because of frequent cache invalidation and lack of reuse.

  The second method will score over the first, when we have a large number of processors. A possible compromise between the two methods will be to use the first method to reduce the problem to the number of processors and then using the treelike reduction.

## 2 Hardware details:

Since there is possibility of false sharing due to the updates in close locations in the array several times, it is important to get to know the cache line size of the system. It can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE

For level1 cache it is = 64 bytes, for level2 cache also it is = 64 bytes.

## 3 Input parameters. Output.

The input parameters are an array which needs to be reduced and its size.

The output is the reduced value which is the sum in our case.

## 4 Parallel overhead time:

(openmp version on 1 core vs serial without openmp)

- CASE I:

  problem size = 1e8

  Serial code: Time taken = 0.433423 s

  Parallel code (1 thread): Time taken = 0.411706 s

  Overhead = 0.411706 - 0.433423 = -0.021717 s

- CASE II:

  problem size = 1e7

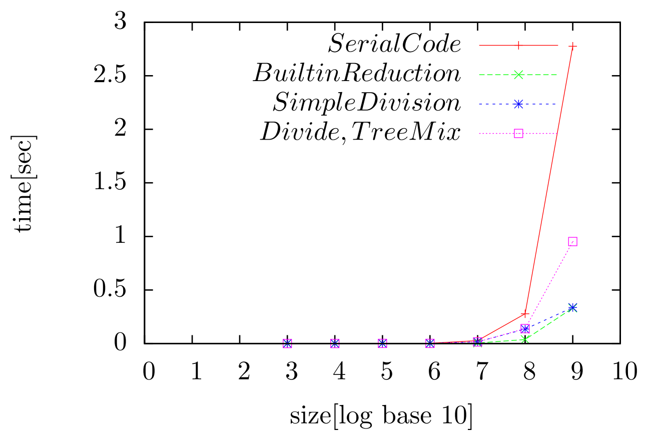
  Serial code: Time taken = 0.043470 s

  Parallel code (1 thread): Time taken = 0.041617 s

  Overhead = 0.041617 - 0.043470 = -0.001853 s

## 5 Problem Size vs Time (Serial, parallel) curve:

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

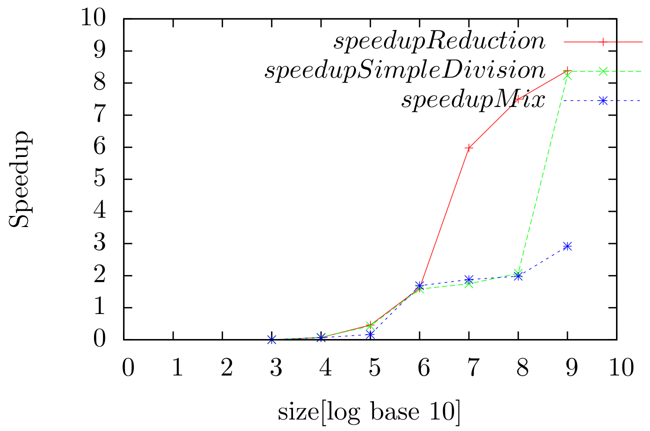


Notice that the built in reduction performs the best. The simple division is almost same as the built in implementation but starts to diverge a bit after $10^8$.

The serial code is naturally the worst. Our tree- reduction and simple division mixture is bogged down by the poor performance of the tree based code.

## 6 Problem size vs. speedup curve.

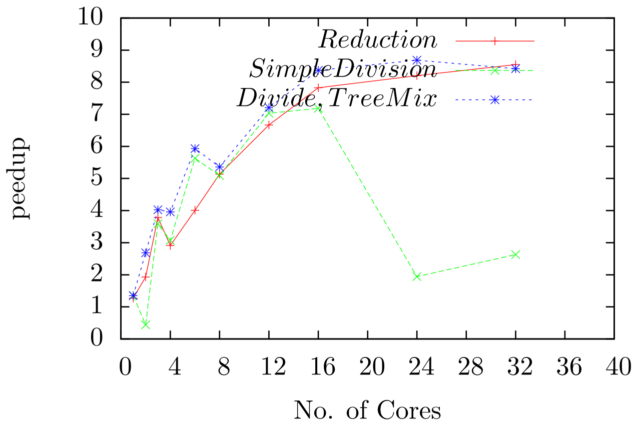
Graph to plot the variation in speedup compared to the number of points in log scale



The speedups match with previous observations, the simple division code however seems to have slowed down for size of order of 6 to 8 and then suddenly rushed back at 9 to meet the builtin$_r eduction.Thisisnotsupportedbytheoryandcanbeattributedtoaberrationsduet$

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

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



The logic of the previous plot should apply to the speedup as well. After all speed up is proportional to the time of execution. Here, surprisingly we see the simpledivision apparently performing much worse. However, this seems to be an aberation and could be accounted to the strain of resoruces on the cluster while running a long script.

## 8 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.

## 9 Conclusion

The way in which reduction should be handled depends on the number of processors available to us.

On normal PC's where we have number of processors ranging from 1-32, using the first method will prove to be more advantageous as we need get the answer of the chunks in a few private variables which are then simply added. If number of processors is less and equal to p, we can achieve close to p speedup because we have the ratio (N-1)/ (N/p-1) which is roughly equal to p. This holds if N is large and p is pretty small.

If we have a many core system (lots of processors), then adding p private variables will turn out to be slow.

Here, the tree method would be better and level wise reduction will work better.

Since we have a limited number of cores we expected the performance of the tree like reduction code to be bad. When we ran this code we got some terrible performances from the tree code

To implement the partial additions at each level, we used two arrays, one additional to fill the next level values obtained(this can be done in place as we later found out...). For every addition operation computed we need to write the value to a completely different array than the one from which we are accessing the input values. This leads to a horrific case of new values having to be loaded in the cache every time. There is no cache reuse whatsoever. Hence we obtain an abysmal performance with this version of the code. e.g for input size of 10000, time taken was a shameful 4.3 s

In comparison, the simple division into chunks gives the advantage of cache reuse and no communication whatsoever until the very end when the values are being merged to give the final sum. In the tree case however at every level, the traversal to perform the additions have to be synchronized, leading to a loss of pace.

Sir had also mentioned another strategy which is a mixture between the previous two. We first compute the partial sums serially and store it in a array of size equal to number of processors. Then we apply the tree method on this array in the hopes, that as the tree method has to do a tiny part of the work(here just add 4 array elements corresponding to threads), the speed will be comparable to the fast method.

To our surprise even if the simple division method almost the entire work, still the minuscule tree portion because of the need to write to memory still manages to slow it down. e.g - simple division for input size 1000000, time:0.0011 mixture for same size time:0.3

This is a stark reminder of the fact that data access is the primary bottleneck in HPC.

Memory Balance in simple division: Since we are looking at arrays of sizes of order of 1e9, The arrays must be stored in RAM therefore, the memory bandwidth is approx 10e9. Also typical bandwidth of HPC networks is 1e9. Since we have run our code on HPC cluster, The memory bandwidth is taken as 1e9 while the No. of FLOPS = 0.8e9. Therefor Machine Balance comes out to be 1.25. this is the lower bound.

Code balance in simple division: In simple division serial code, the main task is to add the new array value to the partial sum. Hence, for any iteration there are 3 data accesses(1 each for reading from and writing to the partial sum variable and 1 for the reading of array element). But, the is only 1 FLOP being done. Hence roughly the code balance is 3 which shows that data access will be troubling the entire problem.

From the above two, the light speed = Bm / Bc = 1.25 / 3 = 0.42. We get pretty less light speed. This may be mainly attributed to the fact that we have limited number of cores and hence we cannot implement tree structure completely. If that was possible, we could increase the floating oint operations in each cycle, which would reduce the code balance and increase the light speed which is our aim.

For the tree based code, the code balance in the loop is roughly the same which is 3. However, this code is bogged down by the accesses to arrays and the threads having to wait for each other after the iteration to every thread. Also, threads do not get to compute on contiguous blocks, so there is no scope for cache reuse.