HPC Project Report: Parallel Implementation of Logical Regression

Rajdeep Pinge 201401103, Aditya Joglekar 201401086

November 16, 2016

Part I

Hardware Details:

We have tested our codes on two machines having following two different architectures. This is done because certain parallel implementations of code give better performance for some specific number of processors.

1.
   - **CPU model:** GPU Cluster: GenuineIntel, 1200 MHz
   - **Memory Information:**
     (a) L1d cache: 32K
     (b) L1i cache: 32K
     (c) L2 cache: 256K
     (d) L3 cache: 20480K
   - **No. of cores:** The CPU has 16 physical cores with 2 thread per core. This means, there are 32 logical cores.
   - **Compiler:** gcc version 4.4.7
   - **Precision used:** We have used double precision wherever possible so as to get more accuracy in the result.
   - **cache line size:**
     This problem doesn't have any data access. Hence we don't specifically need the information about cache line size. **Cache line size** of the system can be obtained by typing in the following commands: getconf LEVEL1_DCACHE_LINESIZE, getconf LEVEL1_ICACHE_LINESIZE, getconf LEVEL2_CACHE_LINESIZE, getconf LEVEL3_CACHE_LINESIZE
     For all the four level of cache mentioned above, the cache line length is 64 bytes.

Part II

1 Context:

- **Brief description of the problem.**
  have to calculate the value of pi using the trapezoidal method for the function,

$$f(x) = 4.0/(1 + x^2)$$

  The above function when integrated over the interval [0,1] gives pi i.e. the area under the curve for the above function in the interval [0,1] is pi.

- **Description and Complexity of Algorithm (Serial):**

  The serial code written is to numerically calculate the integral of a function using the trapezoidal rule. The key idea is to recognize that integration is essentially the area under the function for a given interval. This can be approximated by taking small enough trapezoids underneath the curve, calculating their area and summing them up. Since the trapezoids sometimes leave out some area while in some cases overshoot it, the answer obtained is approximate. The approximation improves and the error decreases as we go on increasing the number of trapezoids. Hence : Our problem size is a function of number of trapezoids(n). For more details please see the code attached with the report.

  The problem boils down to calculating and summing the area of all trapezoids and hence the problem has a complexity of the order of number of trapezoids (n) i.e. O(n)

- **Possible speedup (theoretical):**

  For problem size = 1e9

  Total time taken to run serial code: 19.860216

  Time taken to run the parallelizable code: 19.860161

  Hence p(fraction of code which can be parallelized)= 0.9999

  So, we can expect a very high speedup as p is very much close to 1. The actual speedup we can expect is bounded by the computational power available(eg. number of cores, latency etc). Thus Amdahl's law is strictly theoretical.

  Theoretically as there is not much data dependency and since the problem is embarrassingly parallel the speedup should be p.

- **Profiling Information:**

  The profiling data shows that the program spends about 80% time in the trap_area function which is the main function to calculate pi. Although this function calls function f(x) which uses 20% time, it is a computational single command function hence cannot be parallelized. This suggests that the efforts of parallelizing the code should be focused on trap_area function.

- **Optimization strategy:**

  The optimization strategies using OpenMP have already been discussed in one of the previous labs. Here we will discuss the optimization strategies used in MPI implementation.

  MPI can be used for dividing the work between several machines. It can also be used for prallelizing on a single machine using the cores available similar to openMP.

  It will be interesting to compare the performance of code using these two paradigms(OpenMP and MPI). MPI apparently creates processes and not threads on each computational unit. So, overhead will be more. Also, communicating overhead between machines will have more latency. Usually, MPI is more useful when problems need communication and when a cluster of computers (> 16 cores usually) has to be made.

  We can also write the MPI_Reduce() function to apply the reduction operationality of OpenMP in MPI.

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

  MPI is a bit different from OpenMP. Here we have to **Initialize the MPI** environment and then do the required pre-processing. After this we are ready to run the code.

  In actual code also, the message passing using **MPI_Send() and MPI_Recv() must happen in a synchronised manner.**

  The code must work for **variable number of processors**. For this we have a **'for loop'** in the code. This loop must receive all the partial sums for which each message must have a **unique tag**.

  We faced problems in all the above mentioned parts. Basically all these come under the synchronization problem. It was difficult to remove them considering we had to write a generalized code for variable number of processors.

Also, we could not get sufficient improvement in MPI execution time even when we wrote the MPI_Reduce() function. It may be because we are using a for loop to receive the messages and also we are storing the received partial sums in an array local to cpu : 0. Hence there is no race condition and also we don't need any critical section.

The Same code was run on HPC cluster and GPU cluster. But the results were conflicting. The HPC cluster had a problem of matching frequencies.

On the cluster,even for a large enough problem size (10^9) and large number of cores, we were getting a pathetic speedup of x2.

The same code when run on a GPU gave almost perfect results!

This shows how important it is to keep the overhead in check while using a cluster. And this was for an embarassingly parallel problem, imagine what will happen for probelems with other bottlenecks?

The cluster probably has machines which have very different frequencies, this will cause load imbalance as the slowest machine will hold up the rest of the cluster. The cluster aws warning us about this issue.

Also, the cluster has 4 machines each with 12 cores. So if we try running the problem for 16 cores, the cluster will have to allocate the problem amongst the machines. This problem can't be solved on a single machine. Then, we will also have to deal with the communcation overhead between the machines. 12 core speedup 4 communication overhead

2 Input. Output

Input params: Interval start, Interval end, number of steps

Output: value of function (approximate value of pi in this case)

The answers of serial and parallel(OpenMP and MPI) are almost exactly the same. On occasions when they differ the difference is negligible.

3 Parallel overhead time. (MPI version on 1 core vs serial:

CASE I:

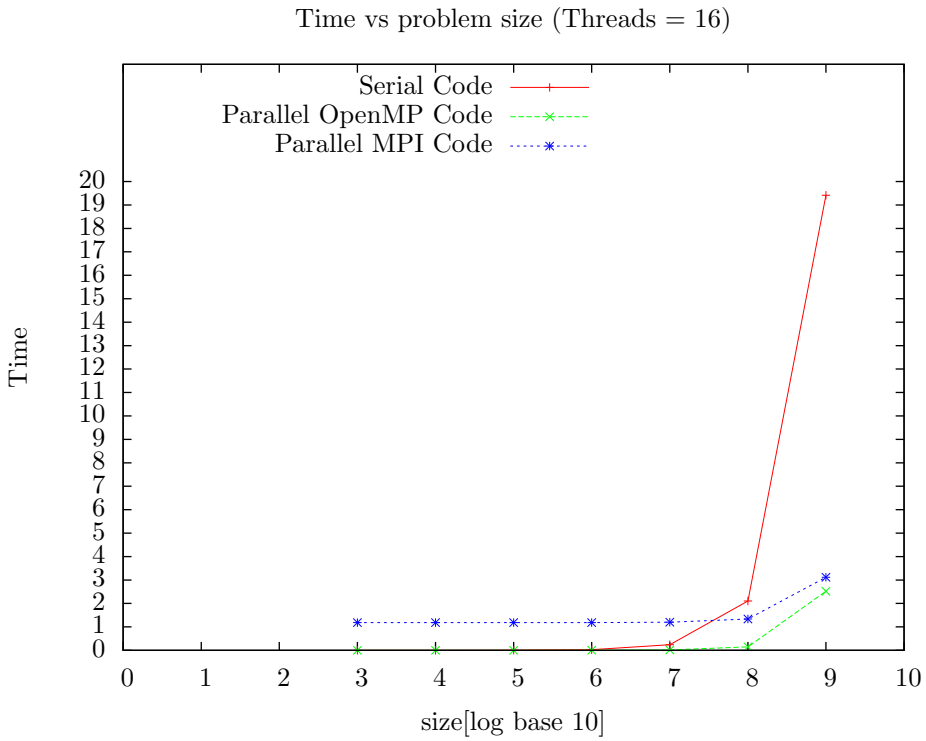
Number of steps(h) = 1e9

Serial code: Time taken = 19.42 s

Parallel code (1 thread): Time taken = 19.62 s

Hence overhead = 19.62 - 19.42 = 0.20 s (1.02% of serial time)

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



Naturally, the serial code takes the most time to run. The red graph agrees with the theoretical observation that the serial comlpexity is O(n). The log scale graph looks piece wise linear as the scale on x-axis is compressed.

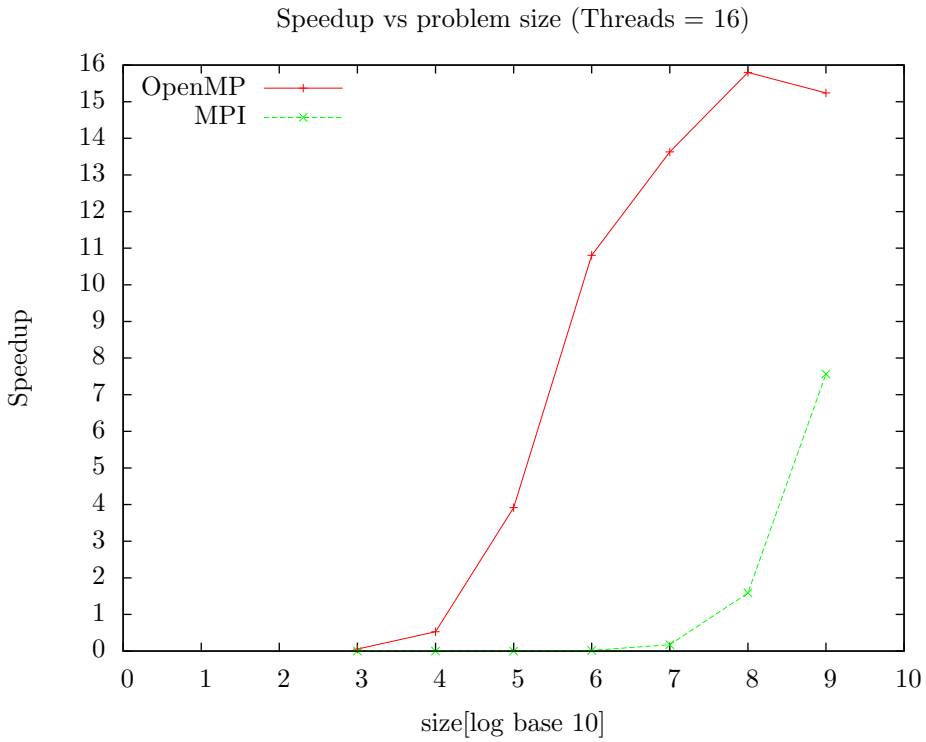
The OpenMP implementation is taking less time to execute than the MPI version. This can be attributed to the fact that MPI's true use is in distributed systems ie when a program is run simultaneously over several machines.

Here, we have used MPI on several **cores** of a **single** machine. Also, note that unlike OpenMP whose basic computational unit is a thread, MPI creates processes on each core. We know that overhead for a process is much more than for a thread.

We know that parallelization becomes effective only when the overhead is overshadowed by the gains in computational time. Thus, problem size needs to be sufficiently large.

An **important observation for time taken by MPI code:** The time was remaining constant upto 10^8, then it was increasing with size as is the expected behaviour. This is because till size 10^8, The **major component of time taken is the overhead time (approx 1.1 sec)**. The **computational time is negligible compared to the overhead time** for 16 threads. Therefore the MPI graph is constant at 1 sec. It is only for very large problem sizes that the computational time crosses the overhead time. Thus we expect to observe speedup only when this breaking even happens. This seemingly happens for problem sizes $>$ 1e8

5 Problem size vs. speedup curve.



The goal of the assignment is to compare the performance of MPI with OpenMP. We must keep in mind that the **overhead for MPI is larger than OpenMP as MPI uses processes instead of threads.**

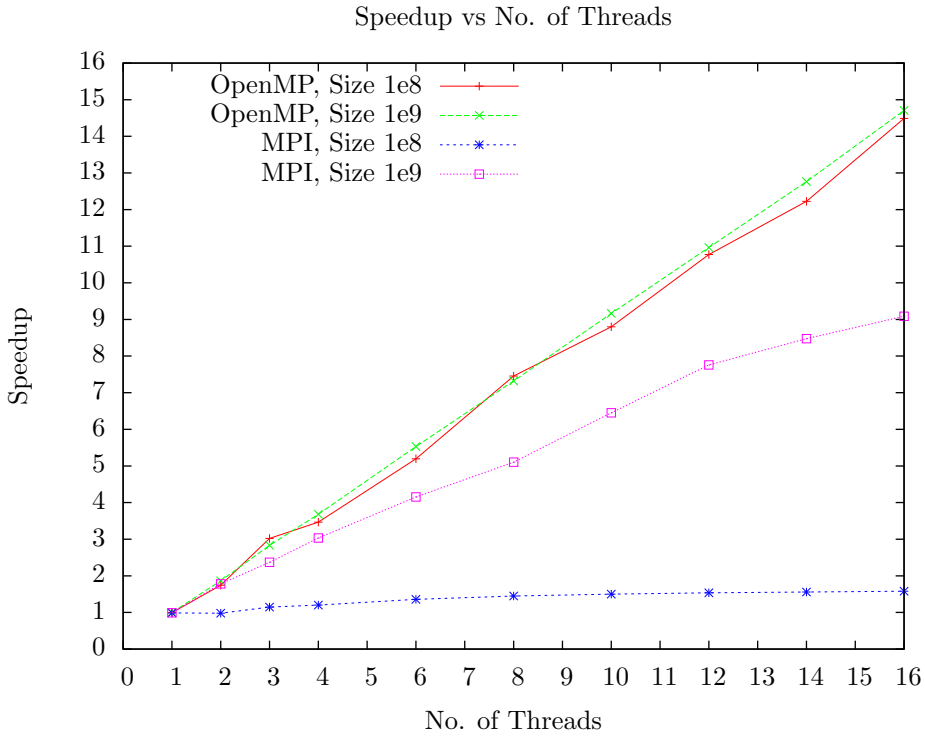
Straightaway, we observe that OpenMP is given much better speedup, compared to MPI. In fact, for **sufficiently large problem size it is given almost perfect speedup(16 for 16 cores)! because the problem is embarrassingly parallel.** The **decrease at the last data point must be an aberration** as there is no reason for the speedup to go down with increase in problem size.

The **less Speedup in MPI** is due to the **larger overhead MPI has to incurr**. The fact that **MPI needs a larger problem size** 10^8 **to starting achieveing speedup as compared to** 10^4 **for OpenMP** means the overhead for MPI is definitely larger.

Both graphs follow expected behaviour. The speedup goes on increasing for larger problem sizes.

But, **MPI is not able to reach perfect speedup because of its overhead issues.**

6 No. of cores vs. speedup curve



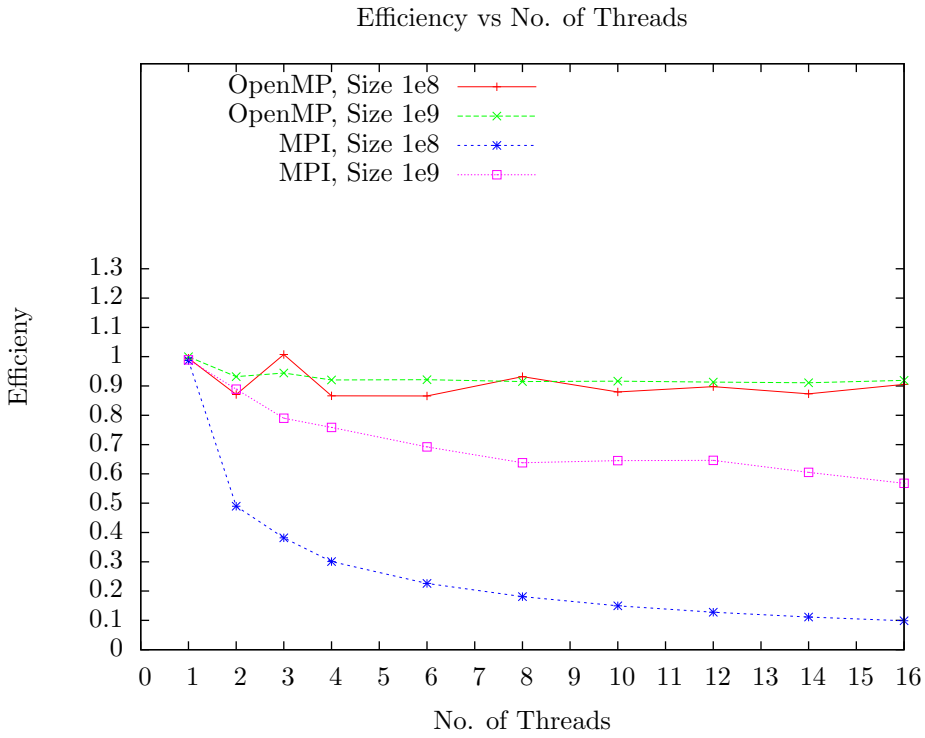
The curves more or less show expected behaviour. For a fixed problem size, there is an **optimal number of processors** beyond which the speedup starts to decrease.

Consider, the MPI curves, for a larger problem size, we get more speedup(Amdalh's effect). For the smaller problem size (blue), it starts facing the problem of increase in overhead and hence the **graph shows a saturating trend** (it will probably decrease for threads > 16). For **larger problem size (pink) however, the speedup is still increasing, it is still to reach the optimal point for this problem size.**

Table 1: Karp Flatt Metric for trapezoids = 1e9

Cores	2	3	4	6	8	10	12	14	16
OpenMP Karp_Flatt	0.0727,	0.0295	0.0288	0.0170	0.0132	0.0101	0.0085	0.0074	0.0058
MPI Karp_Flatt	0.1243	0.1328	0.1060	0.0888	0.0810	0.0611	0.0497	0.0501	0.0506

7 Efficiency graph



Naturally, efficiency curves show a declining trend as the number of processors increase and the speedup does not. For larger problem sizes, the efficiency obtained is better.

Interestingly, the **openMP implementation manages to maintain a almost 95% (constant) efficiency throughout, this is because the speedup increase manages to keep pace with the increase in resources.**

We conclude that the **OpenMP implementation is scalable while the MPI version suffers from overhead issues.**

Also, the **MPI code might be able to improve the efficiency if the problem becomes coarse granular.**

8 Peformance metric calculations

As seen in Table 1, The **Karp Flatt metric is decreasing as speed up increases with larger number of cores. This is intuitive but contradictory to the theory which says that the metric must remain constant.**

It is **more for MPI as due to the larger overhead**

This metric is a **measure of the non parallelizable serial fraction in the code**. As this is an **embarassingly parallel problem, the metric is low** as expected.

The **OpenMP code is almost linearly scalable because metric is ¡ 0.1**.

9 Measure performance in MFLOPS/sec.

According to the problem, we have total 8 Floaing Point Operations per trapezoid area calculation in serial code and OpenMP parallel code while 9 FLOP for MPI parallel code and we have 1e9 such trapezoids. Therefore we have total 8*1e9 FLOP. We measure the time taken to execute those many operations.

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

For OpenMP parallel code, performance = 3174.6032 MFLOPS
For MPI parallel code, performance = 2885.137853 MFLOPS
For serial code, performance = 412.1590 MFLOPS

The important observation here is that the **MPI parallel code gives less performance than the OpenMP parallel code.** This must be because, the **MPI code creates processes instead of threads and the process overhead is much more.** Also the **synchronization between the processes in MPI takes time.**

9.1 Code and Machine balance:

This problem simply has repeated calculation of function values over different intervals and summing them up. It **does not have any data access, which we have learnt is one of the main bottlenecks in HPC. Thus, it is a very ideal problem for parallelization**

Machine balance is a hardware constant, it depends on the machine's bandwidth and Floating point operations executed in a unit time.

Device's maximum memory-bandwidth = 25.6 GB/s. Peak possible performance = 76.8 GFLOPS (1.2 GHz * 16 cores * 4 FLOP/cycle) as discussed above,

Machine balance(Bm) = 0.333

This shows that **in this system, the memory would be the bottleneck as established by the CPU-DRAM gap trends.**

Code balance(Bc) is ratio of memory access to computations performed. Here, code balance is 0 as there is no memory acccess. Light speed is inversely proportional to code balance, so light speed will be high. So we expect good speedup.

Since code balance is 0, **performance fraction Bm/Bc comes out to be infinite.** But we know that the **performance fraction can't be $>$ 1, because performance is bounded by the hardware capability.**

10 Conclusion

The problem of using the trapezoidal rule was nothing new, the goal was to compare the performance of the two frameworks.

We found that the **problem is embarrassingly parallel so implementations which can keep the overhead in check can expect almost perfect speedups.**

MPI is a framework which is actually more useful when communication in distributed systems is needed to solve a problem. This problem has almost no data access. Also The is no need for communication between the decomposed parts whatsoever. Hence no specific advantage of distributed systems. Also, MPI uses processes as its basic computational unit instead of threads. Thus the overhead for MPI is more.

Thus, for the given problem, OpenMP beats MPI handsdown. It achieves almost perfect speedup for optimal problem size and cores. MPI however, suffers from overhead issues and gives far lesser speedup.

We found running the problem on **GPU yields better results than on the cluster**. There seems to be some **frequency matching issues on the cluster.**