BOISE STATE UNIVERSITY

(BSU, USA)

Name: Brian KYANJO
Course: Parallel Scientific Computing

Project Number: Final
Date: May 6, 2021

Problem description

Monte Carlo integration to estimate the integral:

$$\int_0^\infty e^{-\lambda x} \cos x dx$$

for $\lambda > 0$ by using an exponential distribution $p(x) = \lambda e^{-\lambda x}$ for $x \geq 0$, which has a variance $var\{p(x)\} = \frac{1}{\lambda^2}$. Where X is a random value obtained from a uniform distribution on the interval $[0, 1]$, with random variable Y drawn from $p(x)$ can be obtained by transformation:

$$Y = -\frac{1}{\lambda} \ln X$$

The exact value of the integral is given by;

$$\int_0^\infty e^{-\lambda x} \cos x dx = \frac{\lambda}{1 + \lambda^2}$$

Task 1 - Monte Carlo integration with MPI

The problem has been programed using MPI. And the code takes N samples as an input parameter. The entire program has been measured using MPI_Wtime function.

1. I used $\lambda = 1$, and the number of samples ($N = 10^{10}$), required to achieve an error tolerance $\epsilon = 10^{-5}$ with 99% confidence in the code named **mc_task1_2.c**. This code was run with the determined N, and the solution was achieved as shown in figure 1 below.

```
[bkyanjo@r2-login final_project]$ cat output.o328675
nproc = 64       uexact = 0.500000       umc = 0.500019  error = -1.889905e-05
```

Figure 1: Verification of the solution against the exact solution.

Note: I performed many simulations on the problem with different the number of processors, but the best confidence error from the exact solution, depended on the Number of samples rather than the number of processors. This led me to use just 64 processors, since its even considered as the upper bound in next parts of the question.

2. The computational time required to obtain a solution for $N = 10^q$ samples, where $q = 3, 4, 5, 6, 7, 8, 9, 10$ was measured in the code named **mc_task1_1.c**, and results for Time against N was measured and plotted as shown in figure 2.

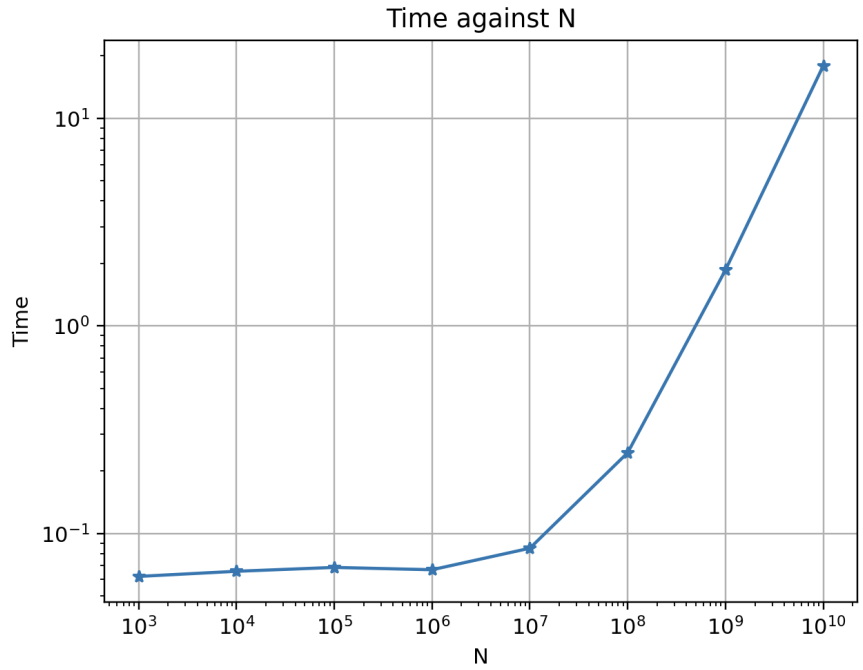


Figure 2: Time evaluation plot for the MPI code

Figure 2, depicts that computational time is small for number of samples between 10^3 and 10^7, but after that increases with increase in N. This is because large number of samples require more resouces than smaller one, so much time is spent on the loops that generate random samples for these bigger problem size. Making computational time expensive. However, large samples have proved to produce the most effective results in accordance to smaller ones.

According to figure 3, the red dotted horizontal line straches up to the y-axis, which gives us the estimate of the time reqired to obtain a tolerance of 10^{-5} with $N = 10^{10}$, this estimated time is 15.667943 seconds.

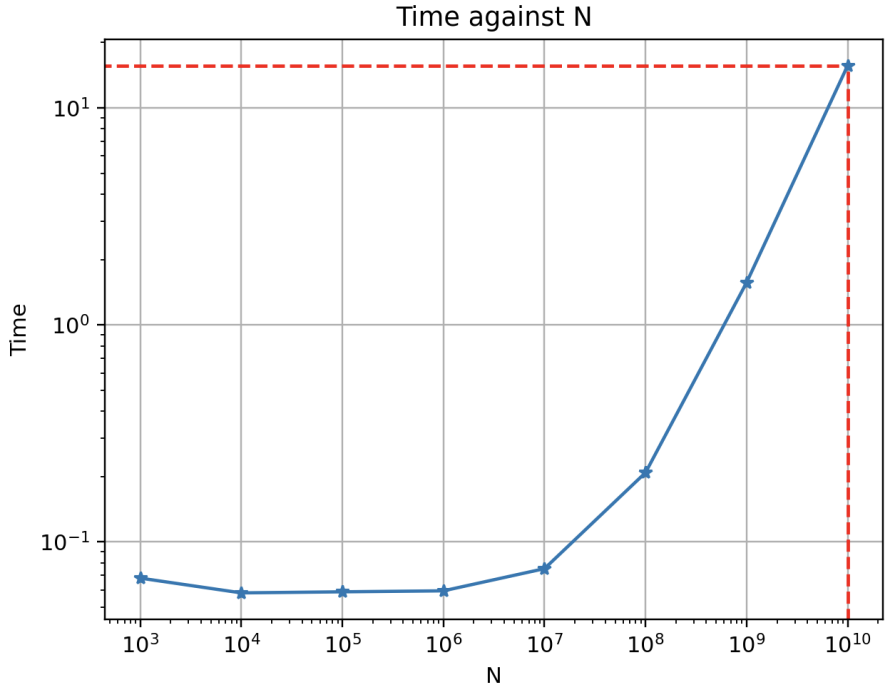
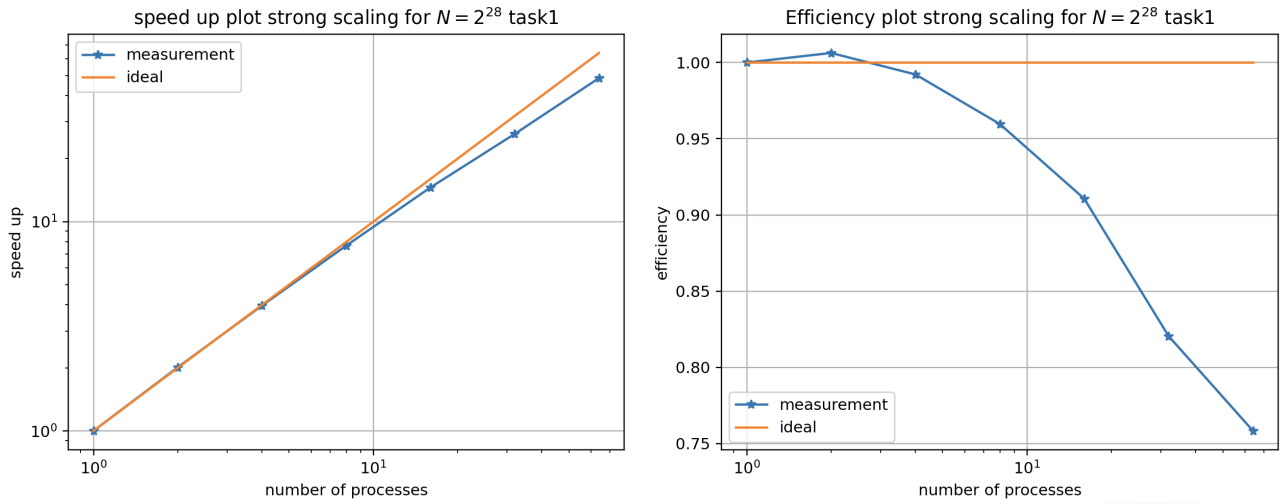


Figure 3: Estimated time to reach a tolerance of 10^{-5}

3. Strong scaling plots for $p = 1, 2, 4, 8, 16, 32, 64$ processors have been created using a problem size $N = 2^{28}$ which is upper bound for the $p = 64$ processors in the weak scaling simulations. The speedup and efficiency plots for the strong sclaing simulation have been plotted and displayed in figures 4a and 4b respectively.

   Figure 4a depict linear scaling for number of processors:$1, 2, 4$, and 8 after which scaling drops. This means that the program was perfectly scaled up to 8 processors then attains typical sucess for the remaining processors. Which implies that the upper limit for the scaled speedup is attained at 8 processes, and that's when each process is contributing 100% of its computational power.

   Figure 4b shows that as the number of processors increase with fixed problem size, parallel efficiency exceeds 100%, for the first resourse, this is due to minimized load imbalance and overhead, and maximized concurency. After that as the number of processors increase with a fixed problem size, parallel efficiency drops gradually. This is because according to Amdahl's law as the number of processors increases, with a fixed problem size, and running the program on a single processor(serial time), since some part of the code where not parallelized, the time taken by the parallelized fraction of the code, for what ever number of processors used, cannot be less than the execution time by the serial part, hence as the processes increases, parallel efficiency decreases, however, it sustains the efficiency good enough that it only droped to below 80%.



(a) (b)

Figure 4: (a) and (b) respectively show speedup and Efficiency plots for strong scaling Task1.

A weak scaling plot for $p = 1, 2, 4, 8, 16, 32, 64$ processors has been created with doubling the problem size starting at $N = 2^{22}$ to $N = 2^{28}$ which is the problem size used in performing strong scaling simulations. This means that both figures 4b and 5 have the same end point on the plot. The efficiency plot for the weak scaling simulation has been ploted as shown in figure 5.

Figure 5 depicts that the code was 100% parallel efficiencient for the first three simulations as the problem size doubles, after which it drops gradually with increasing number of processors, as the problem size doubles. This implies that this program has been made cost optimal, hence the system sustained efficiency, good enough almost all the runs it was better than strong scaling. This is much possible due to the right allocation of available resources efficitiently to the right problem size hence good efficiency satisfying Gustafson's law.

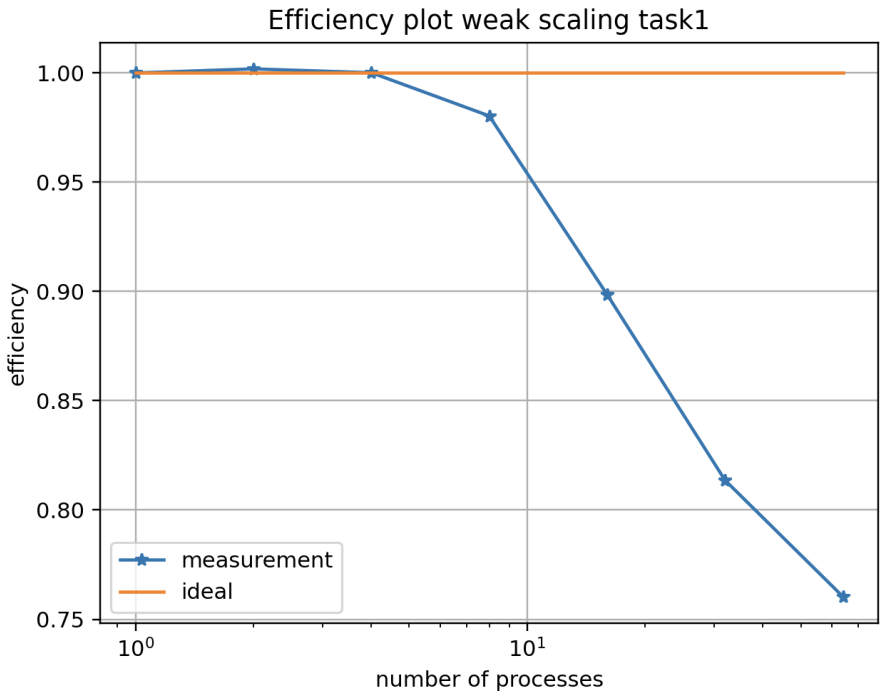


Figure 5: Weak scaling parallel efficiency

Task 2 - Monte Carlo integration with CUDA

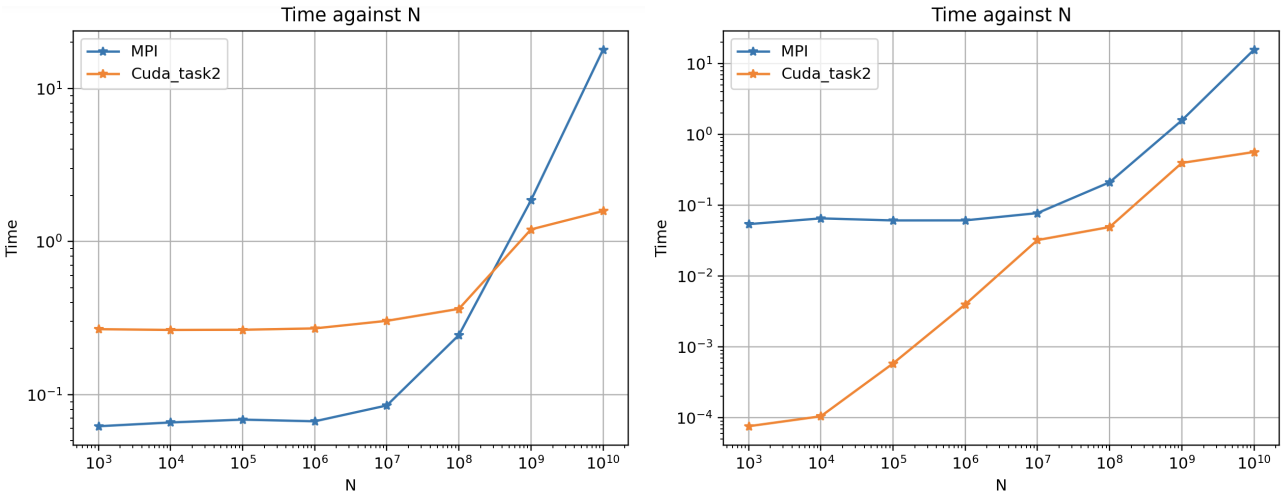
The MPI-Monte Carlo integration problem in the previous has been implemented using CUDA. In a similar way like in Task1, the program named **mc_task2.cu** has been implemented and was tested using $\lambda = 1$ with $N = 10^{10}$, which gave a tolerance of 10^{-5} with confidence 99%, as shown in figure 6 below.

```
[bkyanjo@r2-login final_project]$ cat output.o329456
N = 10000000000          I = 0.499987    Error = 1.257658e-05    Time = 0.453513 s
```

Figure 6: Results obtained using $\lambda = 1$ with $N = 10^{10}$

The time required to compute a solution for $N = 10^q$ samples has been measured and ploted versus N. The results has been compared to MPI results as shown in figure 7a. For $N \leq 10^8$, MPI performs better than Cuda. However this is because of the cost of setup time in the cuda code according to figure 7b for $N \leq 10^8$ than the MPI code. According to Cuda tool kit ([2]), its written that cudaMalloc is expensive, hence may affect performance comparisions, so therefore excluding setup time makes comparision reasonable and exposes cudas power over MPI (figure 7b). Hence, Cuda performs much better than MPI, and in that very figure, generally its also observed that computational time evolution with increasing problem size trend is well exhibited.

Due to the fact that the choice of threads per block influences occupancy and matching the problem size, the restriction of 65535 blocks per grid dimesion which are not large enough to create enough threads to cover large N values. So observing that threads per block should be a multiple of 32, since the kernel gives instructions in wraps (32 threads) [1]. I decided to use 128 threads per block so that i can be able to fit necessary blocks in my streaming multiprocessors before hitting the maximum 65535 blocks. In this way atleast maximizing occupancy, eventhough not enough threads may be created to cover lage N values.



(a) (b)

Figure 7: (a) and (b) respectively show the comparision of Cuda results against MPI with and without allocation of memory taken into account.

The number of blocks per each problem size N, was computed using the equation (1);

$$B = \frac{N + T - 1}{T} \tag{1}$$

where B is the number of blocks, N is the problem size, T is the threads per block. However B is restricted not exceed 65535 blocks, so for any N that makes $B \geq 65535$, a default value of $B = 65535$ is used in the simulation.

Task 3 - Reduction operation in CUDA

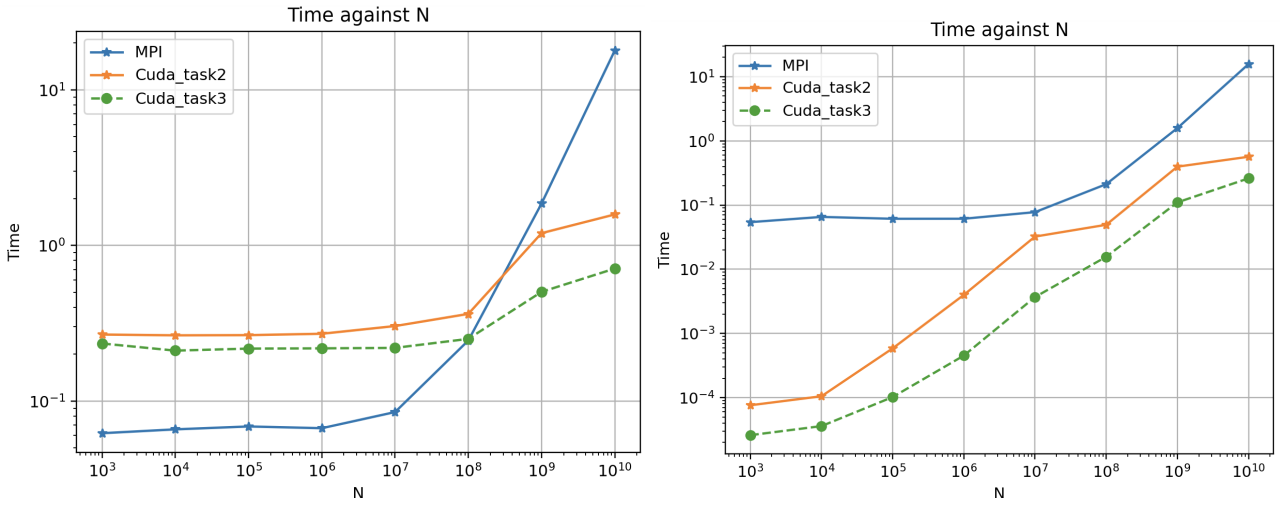
The reduction kernel was implemented in the Cuda code named **mc_task3.cu**, and the computation time was measured and ploted against previous results as shown in figure 8a with memory allocation taken into account.

Figures 8a and 8b represent time evolution with increasing problem size N, for MPI, task2 and 3 Cuda results with and without allocation of memory respectively. In figure 8a, the time evolution for $N \leq 10^8$ depicts a very infinitesimal change for the two simulations: task2 and 3, making MPI to win with cheap execution time. This kind of behaviour is due to the fact that setup time is expensive, and since for $N \leq 10^8$, the computational time is very small, then the setup time dominats and we can't really see the difference, however when these two graphs are plotted independently, without MPI, we see a potential time change. For values $N > 10^8$, Cuda simulations performs better than MPI.

After implementation of the reduction kernel to replace atomicAdd, we see an overal improvemnet in the computational time, and the tolence too was improved for task3 compared to task2. This is because the butterfly sumation implemented uses parallel reduction which is, way faster and converges to a better solution than the serial atomicAdd.

However, observing figure 8b, where setup time is not taken into account, the performance for the three simulations is well exhibitted, and the time evolution for all of them is well exposed as N shoots up. In this case the strength of the implemented reduction kernel to replace the atomicAdd, is well felt, as we can see that task3 simulation (green) wins out for the entire

simulations. I therefore conclude that implemtation of the reduction kernel improved the device performace.



(a) (b)

Figure 8: (a) and (b) respectively show the comparision of Cuda results (task2 and 3), against MPI with and without allocation of memory taken into account.

References

[1] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[2] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.