Parallel Romberg Integration

Iresh Agrawal & Swayambhu Nath Ray

Department of Computational and Data Science
Indian Institute of Science.

April 20, 2017

Motivation

- Making integration efficient and stable using **Romberg integration**.
- Making large range integration faster using parallel systems.
- Achieving a relative error of the order of $10^{-10}\%$.
- Performing numerical integration using various parallel paradims.
- Comparing speed-up with reference(parallel Simpson's method), Matlab and sequential runtime.

Introduction

- Romberg Integration is a highly efficient method for definite numerical integration of the following form:

$$S(b) = \int_a^b f(x)\,dx$$

- But needs huge computation to get the required level of accuarcy. Computation increases exponentially with the accuracy required.
- Parallelising the computations, the required accuarcy can be achieved. Thus using parallel resources efficiently the integration can be made faster and accurate.
- A parallel Romberg integration for large range integration has been implemented in CUDA, MPI and openMP-CUDA hybrid.

Problem Statement

- Computing numerical integrations of highly oscillating functions within a large range.
- Computing integration faster and with reasonable accuracy.
- Achieving speed-up over various existing efficient methods.

Methodology

Romberg's method is used to estimate the definite integral by applying Richardson extrapolation repeatedly on the trapezoidal rule. By trapezoidal composite method:

$$S(b) = \frac{h}{2}[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)] + k_1 h^2 + k_2 h^4 + k_3 h^6 + \cdots$$

$$\Rightarrow S(b) = R_n + k_1 h^2 + k_2 h^4 + k_3 h^6 + \cdots \ldots.(i)$$

By doubling the number of panels:

$$S(b) = R_{2n} + k_1(\frac{h}{2})^2 + k_2(\frac{h}{2})^4 + k_3(\frac{h}{2})^6 + \cdots \ldots\ldots.(ii)$$

4*(ii) - (i) gives:

$$3S(b) = 4R_{2n} - R_n - k_2\frac{3h^4}{4} - k_3\frac{15h^6}{16} - \cdots$$

Order of accuracy becomes $O(h^4)$.

Methodology

Doubling the panels and doing some mathemetical manipulations the least order of error can be removed at each step. Thus the romberg integration can be generalized as:

$$R_{m,j} = \frac{4^j R_{m,j-1} - R_{m-1,j-1}}{4^j - 1} + O(\frac{h}{2m})^{2j+2}$$

where m = number of panel doublings and j = number of error removals. A typical Romberg triangle goes as follows:

m \ j	0	1	2	3
0	$R_{0,0}$			
1	$R_{1,0}$	$R_{1,1}$		
2	$R_{2,0}$	$R_{2,1}$	$R_{2,2}$	
3	$R_{3,0}$	$R_{3,1}$	$R_{3,2}$	$R_{3,3}$

Methodology

- Column 0 of Romberg table requires $2^{rowsize}$ function evaluations at regular interval which is crucial for the accuracy required.
- Hence the main bottle-neck of the process lies in this large number of function evaluations for the first coloumn.
- Also the distribution of the function evaluations is highly skewed towards the bottom of the column 0.
- Thus distributing the computated values among the rows is also very crucial.
- All the above bottle-necks has been well handled by our algorithm.

Algorithm

- The range of integration is divided into small ranges. For each range a romberg table has been computed.
- For each block function evaluations are done in parallel using **2 different algorithm**.
- For the **first version** the function evaluations are divided between different threads and stored in **shared memory**. Bank conflicts has been minimized in this case.
- For the **second version** the function evaluations are divided between different threads. Each thread calculates its own part and the results are stored in the **register memory** efficiently by studying the pattern of the data distribution in the column, giving better speed-up than the first one due to the use of register memory.
- After the computation of the first column the rest of the computations are done in **register memory.**

Algorithm

- The final result is just the summation of the results from each block, which is done in CPU.
- The second algorithm has also been implemented in MPI and openMP-CUDA hybrid for experimentation .
- Sequential implementation has been done by us, using the second algorithm.
- For MPI each processor has been given a romberg table. The final result is obtained by reducing the result (MPI_Reduce : bottleneck) from each processor.
- In case of hybrid implementation,
  1. First, unprocessed first column of 1/4 of total romberg tables is calculated in GPU global memory.
  2. Using two CUDA streams, (i) calculation of 3/4 of romberg tables in GPU and (ii) transfer of unprocessed columns to CPU is done concurently.
  3. unprocessed columns are processed and final values are added to final result.

Experiments Performed

- Both the algorithms, implemented in the 3 mentioned parallel paradigm, has been compared with the reference method, matlab and sequential implementation for the following type of integration using various values for x:

$$S(x) = \int_0^x f(x)\, dx$$

- The experiments are performed keeping the worst case relative error of the order of $10^{-10}\%$ for all the methods.

EXPERIMENTAL RESULTS

f(x) = x*sin(x)

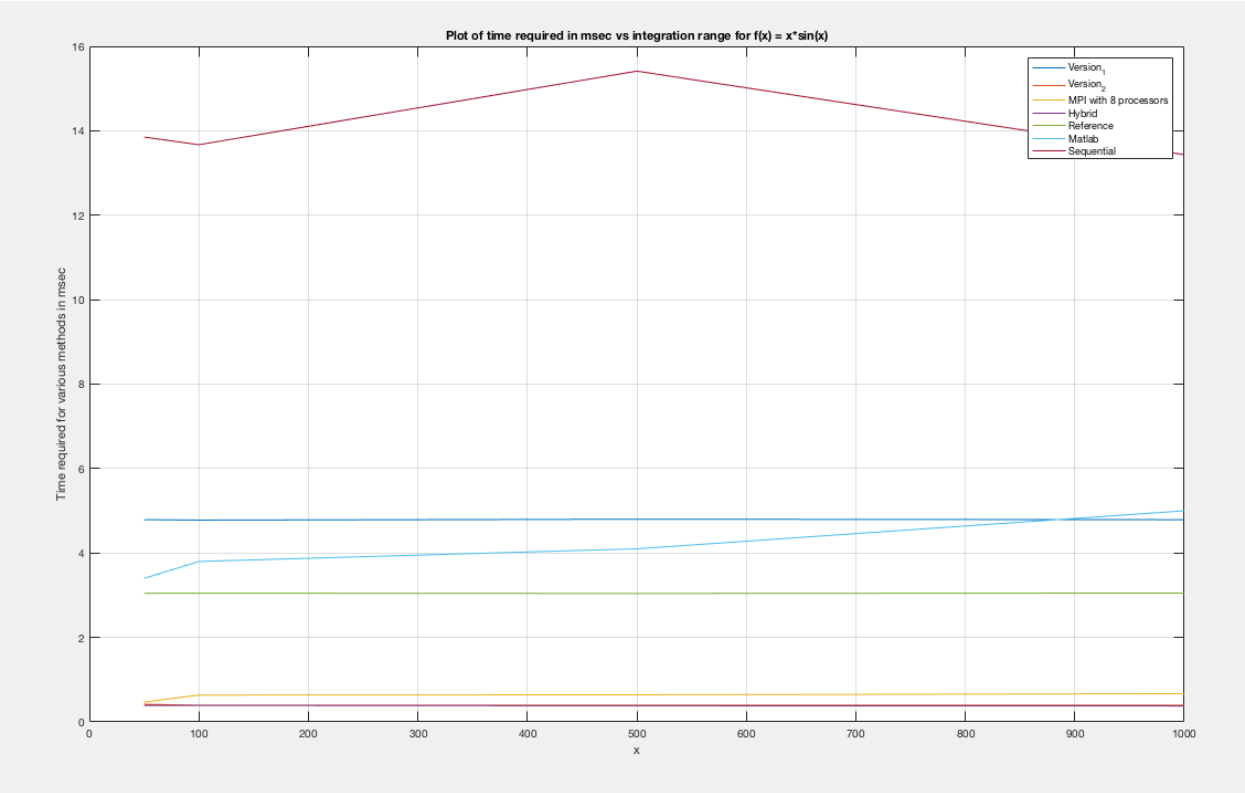

Figure: Time required for various methods

EXPERIMENTAL RESULTS

f(x) = x*sin(x)

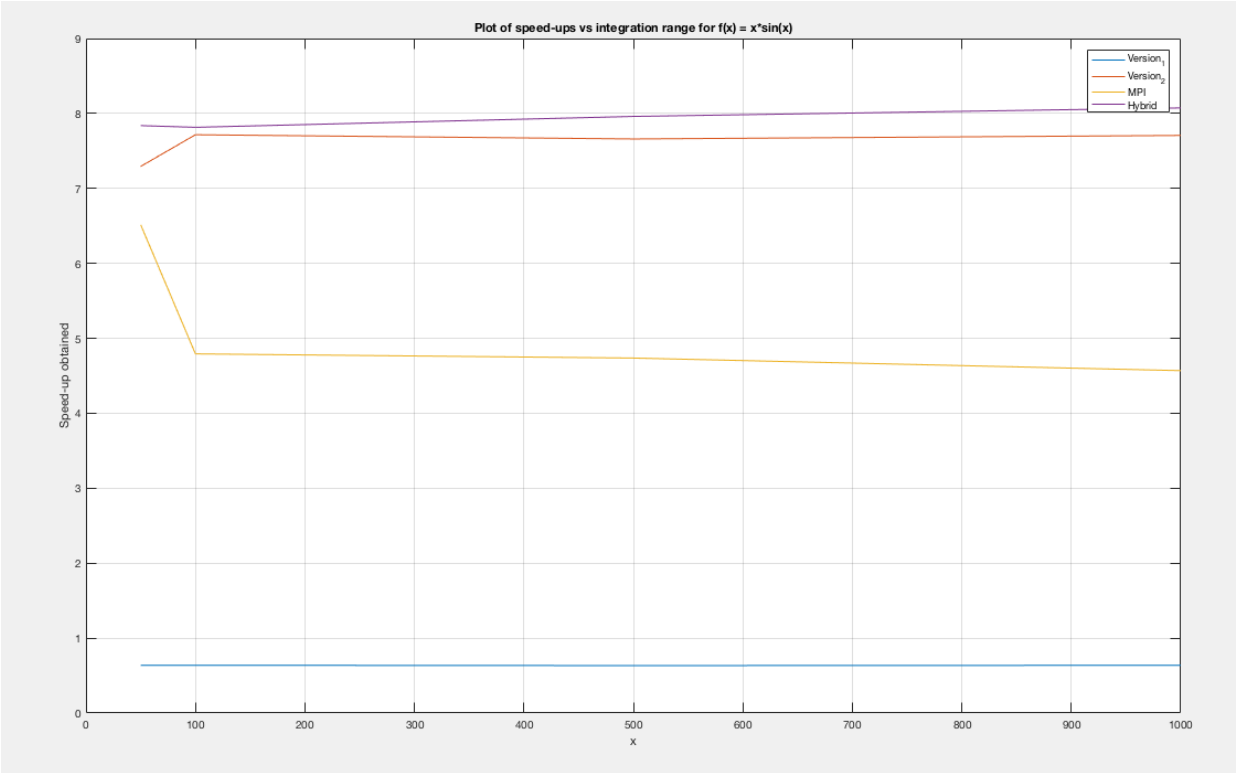

Figure: Speed up obtained w.r.t reference(Simpson's method) with comparable accuracy.

EXPERIMENTAL RESULTS

f(x) = exp(x)*sin(x)

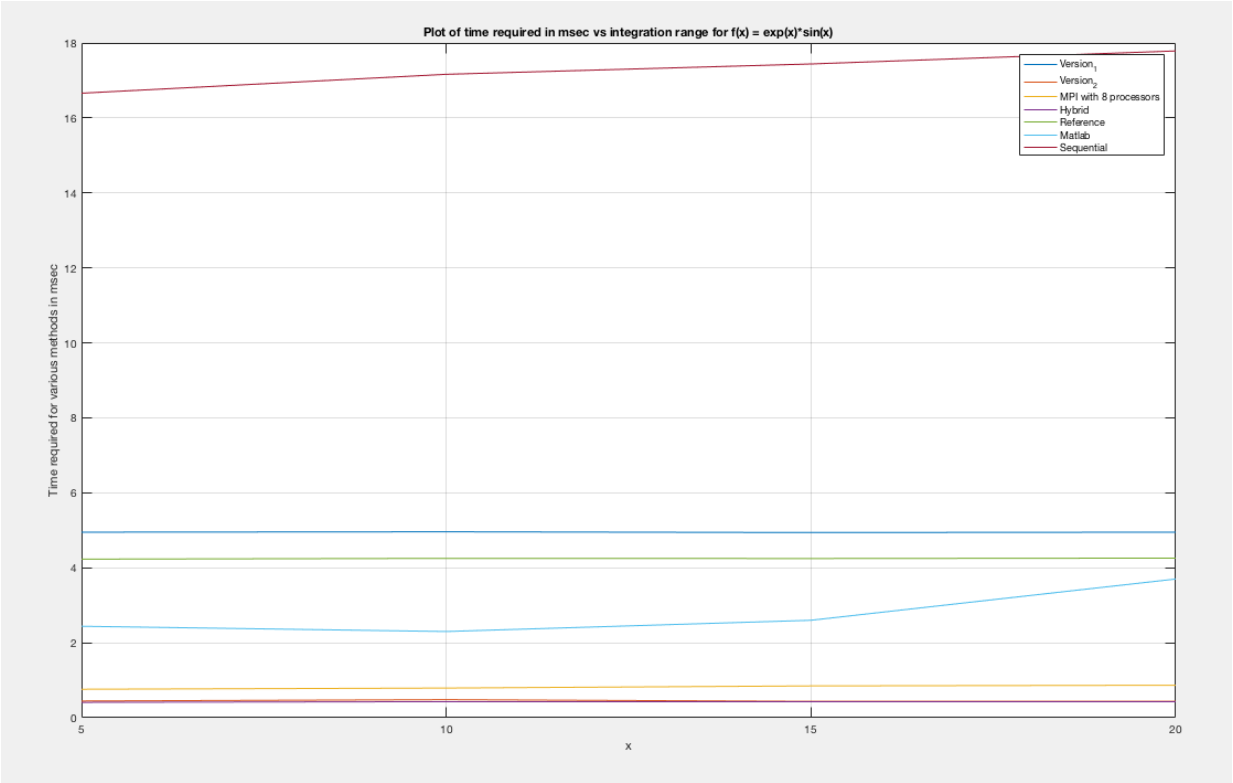

Figure: Time required for various methods

EXPERIMENTAL RESULTS

f(x) = exp(x)*sin(x)

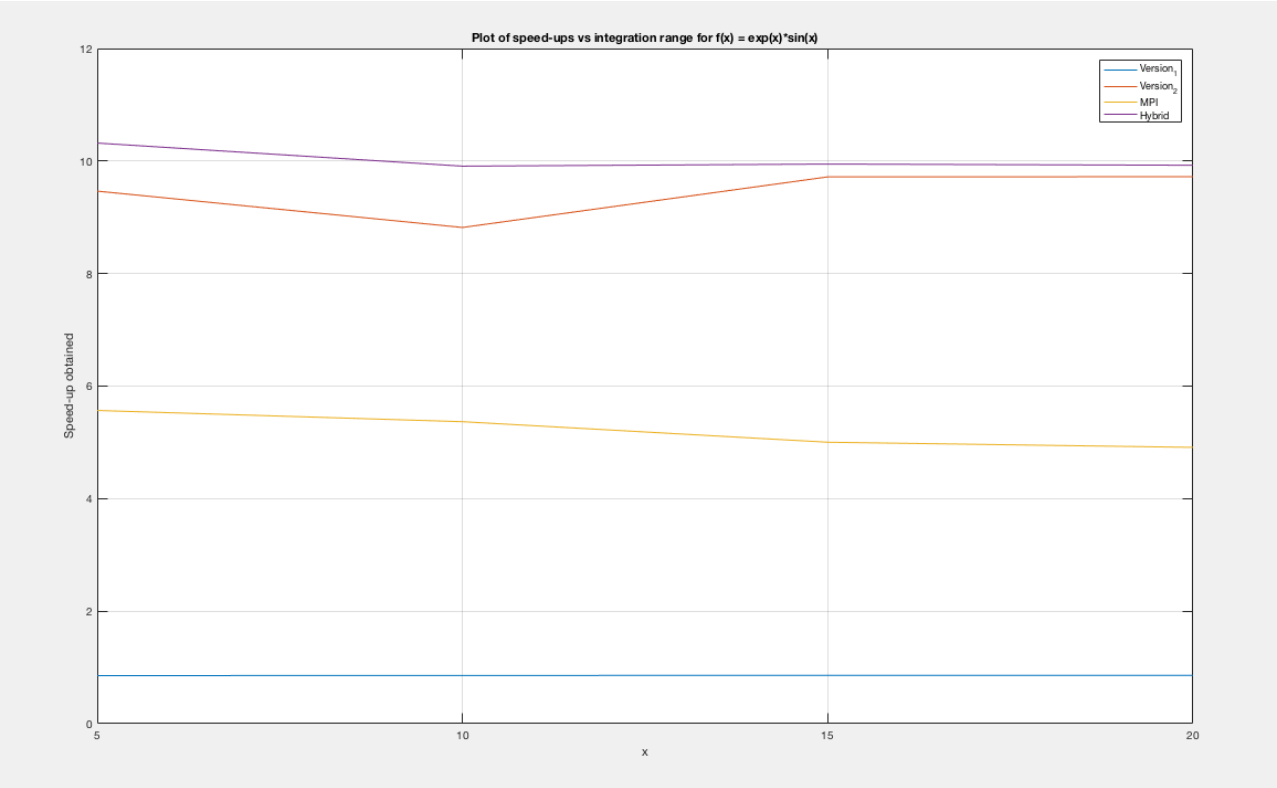

Figure: Speed up obtained w.r.t reference(Simpson's method) with comparable accuracy.

Conclusion

- Keeping the order of accuracy same for all the methods we are getting a considerable speed-up with our second algorithm.
- Even for MPI and the hybrid implementation we are getting a good speed-up. The maximum speed-up is obtained from the hybrid(CPU+GPU) implementation.
- On finite precision machines, Romberg integration is more prone to numerical errors, due to the divisions (by $4^j - 1$) involved at every steps of the algorithm.

QUESTIONS?

THANK YOU