Numerical Integration using Cavalieri-Simpson methods in Serial and Parallel

MANJANG Amadou, MAHMUD Mukthar Opeyemi, MAHMOOD Irfan, LAZARUS John Success, LAWAL Abubakar Dobi, LARTEY Collins

Parallel Computing Laboratory
DISIM
University of L'Aquila, Italy

June 2023

Contents

Math Problem I

- Integrals are a crucial component of mathematical analysis. Indefinite and definite integrals are the two types of integrals.
- The first is referred to as a primitive function, and it is the inverse of a derivative.
- It is also a broadening of a definite integral. Using the range [a, b], we get a definite integral in this range.
- It is defined as the area between the function graph and the *xy*-axis, with positive values being represented by the sign plus and negative values being represented by the sign minus.
- Integrals are extremely useful not just in mathematics but also in physics computations.
- We can use this interpretation of the integral to calculate values in physics, with the result.

Math Problem II

- In this project, integration will be performed with the Cavalieri-Simpson's Numerical Method.
- This will be implemented both in serial and parallel and a comparison will be made.
- The function of the integrand to be considered is:

$$f(x) = 6x^4 - 7x^3 + 4x$$

Numerical Method I

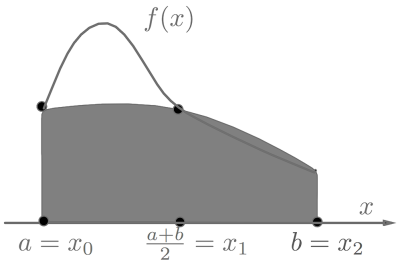

Figure 1: Cavalieri-Simpson formula

- Let f be a real integrable function over the interval $[a, b]$.

Numerical Method II

- Computing explicitly the definite integral $I(f) = \int_a^b f(x)dx$ may be difficult or even impossible.
- One can obtain the integral by replacing f with an approximation f_n, depending on then integer $n \geq 0$, then computing $I(f_n)$ instead of $I(f)$.
- Letting

$$I_n(f) = \int_a^b f_n(x)dx, \quad n \geq 0$$

- Using the interpolating Lagrange polynomial of f over a set of $n+1$ distinct nodes $\{x_i\}$, with $i = 0, \ldots, n$.
- It is noticed that

$$I_n(f) = \sum_{i=0}^{n} \alpha_i f(x_i)$$

Numerical Method III

- The Cavalieri-Simpson formula can be obtained by replacing f over $[a, b]$ with its interpolating polynomial of degree 2 at the nodes $x_0 = a, x_1 = (a + b)/2$ and $x_2 = b$ (see Figure 1. The weights are given by $\alpha_0 = \alpha_2 = (b - a)/6$ and $\alpha_1 = 4(b - a)/6$, and the resulting formula reads

$$I_2(f) = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

- The error is given by

$$E_2(f) = -\frac{h^5}{90}f^{(4)}(\xi), \quad h = \frac{b-a}{2}$$

provided that $f \in C^4([a, b])$, and where ξ lies within (a, b).

Serial Algorithm for Cavalieri-Simpson Method I

1. Define the function f(x) that you want to integrate over a given interval [a, b].
2. Choose the number of subintervals, N (must be an even number), to divide the interval [a, b] into.
3. Calculate the width of each subinterval, h, using the formula h = (b - a) / N.
4. Initialize the variables sum_{odd} and sum_{even} to 0.
5. Loop through the subintervals from i = 1 to N-1:
   - Calculate the midpoint of the current subinterval, $x_i = a + i * h$.
   - If i is odd, add $f(x_i)$ to sum_{odd}.
   - If i is even, add $f(x_i)$ to sum_{even}.
6. Calculate the integral estimate using the Cavalieri-Simpson formula: $integral = (h/3) * (f(a) + 4sum_{odd} + 2sum_{even} + f(b))$.
7. Output the calculated integral as the result.

Parallel algorithm and MPI code for Cavalieri-Simpson methods I

To parallelize the Cavalieri-Simpson method, you can divide the intervals among multiple processes, with each process calculating its local integral. The local integrals are then combined to obtain the global integral. Here's a parallel algorithm for the Cavalieri-Simpson method:

1. Initialize MPI and get the rank and size of the communicator.
2. Determine the number of intervals, n, and the integration limits, a and b.
3. Calculate the local number of intervals, local_n, for each process based on n and size.
4. Calculate the local start and end points, local_start and local_end, for each process based on rank and local_n.
5. Calculate the step size, h, using (b - a) / n.
6. Initialize the local integral, local_integral, to 0.0 for each process.

Parallel algorithm and MPI code for Cavalieri-Simpson methods II

7. Perform the local integration:
   - Each process iterates from local_start to local_end.
   - For each interval, calculate the function values at the endpoints and the midpoint of the interval.
   - Accumulate the local integral by summing the function values using the Simpson's rule weights (1, 4, 1).
8. Reduce the local integrals to obtain the global integral on process 0 using MPI_Reduce.
9. Print the result on process 0.
10. Finalize MPI.

Results I

- The number of subintervals used was 1000000.
- Execution time for the serial was 0.033970 sec.
- The approximate integral for the parallel method was a value averaging 2704.77271.
- The approximate integral for the serial method was 2704.05981.
- The exact integral was 2704.8.

Results II

Table 1: Table of results

No. of proc.	Time (s)	Speed up	Efficiency
1	0.033528	1.0132	1.0132
2	0.019088	1.7796	0.8898
3	0.012809	2.6520	0.8840
4	0.009463	3.5897	0.8974
5	0.007665	4.4313	0.8863
6	0.006283	5.4061	0.9010
7	0.008811	3.8550	0.5507
8	0.004327	7.8502	0.9813
9	0.007213	4.7091	0.5232
10	0.003679	9.2326	0.9233

Speed up I

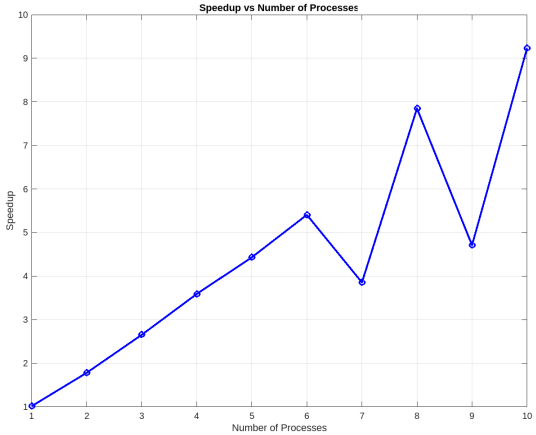

Figure 2: Speed up vs Number of Processes

Speed up II

As seen in figure 2 above, the speed-up of parallel program for the Cavalieri-simpson numerical method for different number of MPI Proceses. it could be seen that as the number of process increases, the speed up of the parallel program increase. However, it could also be seen that for processes corresponding to odd processes, the speed drops drastically, which may be due to the cost in communication between the processors involved. The speed up is given by:

$$S_p = \frac{T_s}{T_p}$$

where T_s is the execution time for serial and T_p is the execution time for parallel.

Efficiency I

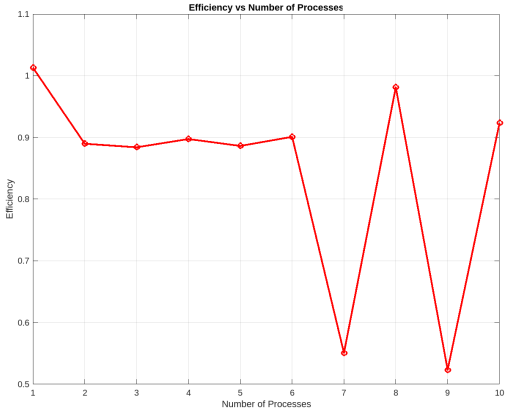

Figure 3: Efficiency vs Number of Processes

Efficiency II

In the figure 3, it can be seen that using one processor for the parallel process, the efficiency is almost equal to 1 and there is a stable value for processes 2 through 6. However, there is a sharp decrease to about 0.55 and 0.51 for processes numbers of 7 and 9. This can be explained in the same way as the reasoning behind the speed up values calculated. The efficiency is given by:

$$Eff = \frac{S_p}{n_p}$$

where S_p is the speedup and n_p is the number of processors.

Parallel code used for the simulations I

```
! Initialize MPI
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
! Number of intervals
n = 1000000
! Integration limits
a = 1.0
b = 5.0
! Calculate local number of intervals for each process
local_n = n / size
! Calculate local start and end points for each process
local_start = rank * local_n
```

Parallel code used for the simulations II

```
local_end = (rank + 1) * local_n - 1
! Calculate step size
h = (b - a) / real(n)
! Initialize local integral value
local_integral = 0.0
! Start the timer for parallel version
start_time = MPI_Wtime()
! Perform local integration
do i = local_start, local_end
x = a + i * h
! Calculate the function value at x
local_integral = local_integral + f(x) + 4.0 * f(x + h/2.0) + f(x + h)
end do
```

Parallel code used for the simulations III

```
! Multiply by h/6 to complete the local integration
local_integral = h/6.0 * local_integral
! Reduce local integrals to obtain the global integral
call MPI_Reduce(local_integral, parallel_integral, 1, MPI_REAL, MPI_SUM, 0,
MPI_COMM_WORLD, ierr)
! End the timer for parallel version
end_time = MPI_Wtime()
! Calculate the parallel execution time
parallel_time = end_time - start_time
! Print parallel result and execution time from rank 0
if (rank == 0) then
write(*,*) "Parallel Approximate integral: ", parallel_integral
write(*,*) "Parallel Execution Time: ", parallel_time
```

Parallel code used for the simulations IV

end if

Conclusion I

- The approximate integrals of the serial and parallel methods were very close to the exact integral which was 2704.8.
- The project demonstrated that finite difference algorithms can be effectively parallelized. While the solution for the single-processor case can be transferred to the multi-processor case, it may require significant modifications to ensure proper parallel execution and efficient utilization of resources.
- While a common purpose of using parallel algorithms is to reduce the computation time needed to calculate a solution, it is important to note that in some cases, using multiple processors may not always lead to a reduction in computation time. Several factors can contribute to this observation, such as overhead, load imbalance, and communication costs.

Conclusion II

- The effect of using multiple processors in parallel is much more significant when increasing the number of points to be evaluated, primarily due to the larger computational effort each processor has to carry in every iteration. As the problem size grows, the parallelization becomes more beneficial, as it allows distributing the workload across multiple processors and taking advantage of their combined computational power.

References

Quarteroni, A, Sacco, R and Saleri, F. (2006): Numerical Mathematics. Spring-Verlag New York, Inc.