School of Parallel Programming & Parallel Architectures for HPC (ICTP)

Instructor: Ekpe Okorafor

Numerical Integration of a Set of Discrete Data

This problem uses a master-worker model where the master process divides up the data and sends it to the workers, who perform local computations on the data and communicate results back to the master. There is no data dependency between workers (they don't need to communicate with each other). This is an example of what is called an "embarrassingly parallel" problem.

Consider the numerical integration of a large set of discrete data values, which could represent points sampled from a function.

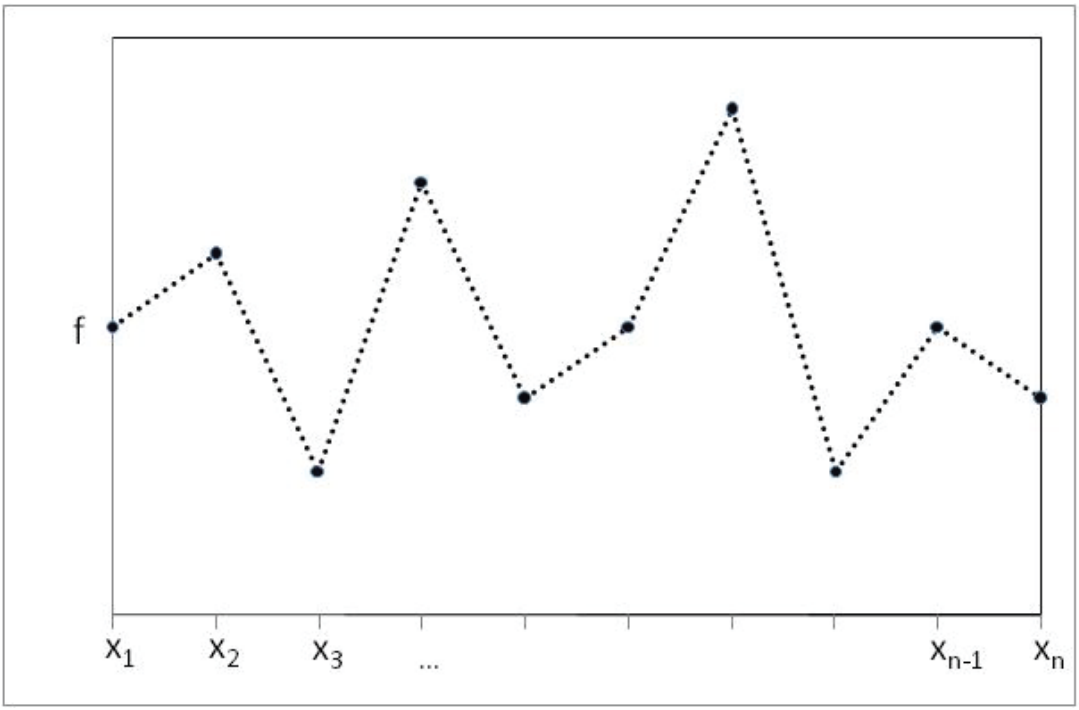

Figure: Discrete data values, f(xi), where i = 1,2,3,…,n.

To approximate the integral, we can fit straight lines between each pair of points and then compute the sum of the areas under each line segment. This is the trapezoid formula:

$$Integration = \sum_{i=1}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2} (x_{i+1} - x_i)$$

The locations x_i may not be evenly spaced. An example FORTRAN 90 code is given below.

```
program integrate
! A program to numerically integrate discrete data from the file "discrete.dat"
implicit none
integer, parameter::n=960000! Number of points in data file
integer::i
real(kind=8)::integral
real(kind=8), dimension(n)::x,f
! Open data file and read in data
open(unit=21,file="discrete.dat",status="old")
do i=1,n
read(21,*)x(i),f(i)
end do
! Now compute global integral
integral=0.0_8
do i=1,n-1
integral=integral+(x(i+1)-x(i))*(f(i)+f(i+1))/2.0_8 ! trapezoidal formula
end do
! Ouput result
write(*,*)"The integral of the data set is: ",integral
stop
end program integrate
```

Note: Generate yours or use an existing data set for "discrete.dat" file. Feel free to experiment with number of points in data file

Now parallelize this program using the master-worker model. The master process (choose process 0, which is always present) reads in data from the file, divides it up evenly and distributes it to the workers (all other processes). The workers compute the integral of their portions of the data and return the results to the master. The master sums the results to find the global integral and outputs the result. If you keep things simple, you should be able to write the parallel program in less than 60 lines of code.

Some tips and hints:

- If the data array is very large, the master process may not have enough local memory to store the entire array in memory. In this case it would be better to read in only part of

the data set at a time and send it to a worker(s), before reading in more data (over-write previous values) and send to other workers, etc.

- In order to make this algorithm efficient, we need to minimize the idle time of the workers (and the master) and balance the computational work as evenly as possible. If the number of processes is small, and the data set is large, we may want the master process to help compute part of the integral while it is waiting for the workers to finish. Also, if the amount of data communicated to each worker is large (lots of communication overhead – bandwidth related) other workers will be idling while they wait for their data. Would it be more efficient to send smaller parcels of data to each worker so that they all get to work quickly, and then repeatedly send more data when they finish until all the work is done? But if the number of messages gets too large, then we will have increased latency-related overhead.
- You can try using non-blocking communication calls on the master process so that it can do other tasks while waiting for results from workers.
- You can also try using the scatter and reduce collective communication routines to implement the parallel program.

Investigate Parallel Performance

Measure the parallel performance of your code and examine how the efficiency varies with process count and problem size.

- Implement timing routines in your parallel code as well as in a sequential version, and write run time to standard output. When submitting timed parallel jobs to the queue, you want to make sure that resources are used exclusively for your job (i.e. other applications are not running at the same time on the same CPU). Also, the run time of your code may be affected by the mapping of processes to cores/sockets/nodes on the machine so experiment with this. It might be a good idea to launch the code several times and average the run time results.
- Measure the parallel efficiency and speedup of your code on different numbers of processes. You may also want to repeat the measurements on larger/smaller domains to examine the effects of problem size. The single process run time T_1 can be used to calculate speedup, or a tougher measure is to use the sequential code run time T_s. Plot a curve of speedup versus number of processes used. Also plot efficiency versus number of processes.
- How well does your code scale? How does the problem size affect the efficiency? Are there ways that the parallel performance of your code can be improved? You may want to consider operation count in critical loops, memory usage, compiler optimization, communication overhead, etc. as ways to improve the speed of your code.