Area Under a Curve

JASON REGINA
University of Wyoming
May 12, 2016

Summary Use a Riemann sum to approximate the area under a curve

Duration 3 hours

Prerequisites Calculus I or exposure to algebraic summations

1.0 Overview

This module introduces a method to approximate the area under a curve using a Riemann sum. Serial and parallel algorithms addressing shared and distributed memory concepts are discussed, as well as the MapReduce algorithm classification. A method to estimate pi (π) is also developed to demonstrate an example scientific application. Exercises focus on how to measure the performance and scaling of a parallel application in multi-core and many-core environments.

Upon completion of this module, students should be able to: 1) Understand the importance of approximating the area under a curve in modeling scientific problems, 2) Understand and apply parallel concepts, 3) Measure the scalability of a parallel application over multiple or many cores, and 4) Identify and explain the Area Under a Curve algorithm using the Berkeley Dwarfs system of classification.

Much of the material in this module has been adapted from the Area Under a Curve module written by Aaron Weeden[1].

2.0 Approximating the Area Under a Curve

Calculating the area under a curve is an important task in science. The area under a concentration-time curve is used in neuroscience to study changes in endocrine levels in the body over time. In economics, the area under a curve is used in the study of discounting to determine the fee incurred for delaying a payment over time. Applications of the area under a curve are used in many other fields of science, including machine learning, medicine, psychology, chemistry, and environmental science.

2.1 Riemann sums

Several techniques exist which can be used to provide acceptable approximations for the area under a curve. This module considers the Riemann sum method of integration. A Riemann sum is a summation that takes the form:

$$\sum f(x)\Delta x \tag{1}$$

Riemann sums can be used to approximate the area under a curve by dividing the area into multiple component shapes and summing their areas. The specific Riemann method explored in this module involves dividing the domain of integration into several rectangles. The height of an arbitrary rectangle is given by $f(x)$ and the width is Δx. The sum of the areas of the rectangles formed by this method is the approximation of the area under the curve. This module considers the Left Riemann sum, in which $f(x)$ indicates the height of the left-most side of each rectangle. A pictorial example of the Left Riemann sum is shown in Figure 1.

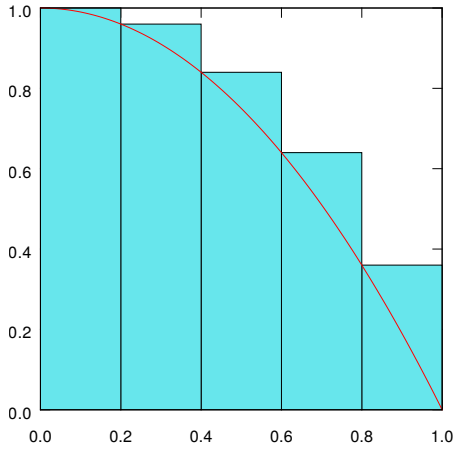

Figure 1: Left Riemann Sum

Quick Review Questions

1. What does “left” refer to in the left Riemann sum?
2. What does “sum” refer to in the left Riemann sum?

2.2 Estimating Pi in Serial

A unit circle has a radius of 1.0 and an area of pi. Given a unit circle centered at the origin and limiting the Riemann sum to Quadrant I, pi can be approximated by:

$$\pi = 4 \sum_{i=0}^{N-1} (1 - x_i^2) \Delta x_i \,, \qquad 0 \leq x < 1 \tag{2}$$

Where i is the index of an arbitrary rectangle, N is the number of rectangles, $(1 - x_i^2)$ is the height of rectangle i, and Δx_i is the width of rectangle i.

For a small number of rectangles, calculations can be performed easily by hand or using a calculator. Beyond just a few rectangles, however, the area needs to be approximated using a computer. Many serial (non-parallel) algorithms for approximating the area under a curve exist with many coded implementations. Such code, running on a single computer, can calculate the areas of millions of rectangles in a Riemann sum in a very small amount of time.

A possible implementation of equation 2 written in pseudocode is shown in Algorithm 1.

Algorithm 1 Area Under a Curve — Serial

Require: N
Ensure: N is an integer greater than 0

```
i ← 0                                  ▷ Initialize i
PI ← 0                                 ▷ Initialize PI
Δx = 1/N                               ▷ Calculate Δx
while i < N do                         ▷ Repeat for all rectangles
    x = i/N                            ▷ Calculate x
    area = (1 − x^2)Δx                 ▷ Calculate rectangle area
    PI ← PI + area                     ▷ Add area to total sum
    i ← i + 1                          ▷ Increment i
end while
PI ← 4 ∗ PI                            ▷ Approximate PI
```

To approximate with even more rectangles, one needs to employ more processing power than is available on a single processor. The concept of parallel processing can be used to leverage the computational power of computing architectures with multiple or many processors working together.

3.0 Introduction to Parallelism

In parallel processing, rather than having a single program execute tasks in a sequence, the program is split among multiple *execution flows* executing tasks in parallel, i.e. at the same time. The term *execution flow* refers to a discrete computational entity that performs processes autonomously. A common synonym is *execution context*; *flow* is chosen here because it evokes the stream of instructions that each entity processes.

Execution flows have more specific names depending on the flavor of parallelism being utilized. In *distributed memory* parallelism, in which execution flows keep their own private memories (separate from the memories of other execution flows), execution flows are known as *processes*. In order for one process to access the memory of another process, the data must be communicated, commonly by a technique known as *message passing*. The standard of message passing considered in this module is defined by the *Message Passing Interface (MPI)*, which defines a set of primitives for packaging up data and sending them between processes.

In another flavor of parallelism known as *shared memory*, in which execution flows share a memory space among them, the execution flows are known as *threads*. Threads are able to read and write to and from memory without having to send messages. The standard for shared memory considered in this module is OpenMP, which uses a series of directives for specifying parallel regions of code to be executed by threads.

A third flavor of parallelism is known as *hybrid*, in which both distributed and shared memory are utilized. In hybrid parallelism, the problem is broken into tasks that each process executes in parallel; the tasks are then broken further into subtasks that each of the threads execute in parallel. After the threads have executed their sub-tasks, the processes use the shared memory to gather the results from the threads, use message passing to gather the results from other processes, and then move on to the next tasks.

Quick Review Questions:

3. What is the name for execution flows that share memory? For those with distributed memory?
4. What is message passing and when is it needed?

3.1 Parallel hardware

In order to use parallelism, the underlying hardware needs to support it. The classic model of the computer, first established by John von Neumann in the 20th century, has a single CPU connected to memory. Such an architecture

does not support parallelism because there is only one CPU to run a stream of instructions. In order for parallelism to occur, there must be multiple processing units running multiple streams of instructions. *Multi-core* technology allows for parallelism by splitting the CPU into multiple compute units called cores. Parallelism can also exist between multiple *compute nodes*, which are computers connected by a network. These computers may themselves have multi-core CPUs, which allows for hybrid parallelism: shared memory between the cores and message passing between the compute nodes.

Quick Review Questions:

5. Why is parallelism impossible on a von Neumann computer?
6. What is a core?

3.2 Motivation for Parallelism

We now know what parallelism is, but why should we use it? The three motivations we will discuss here are speedup, accuracy, and scaling. These are all compelling advantages for using parallelism, but some also exhibit certain limitations that we will also discuss.

Speedup is the idea that a program will run faster if it is parallelized as opposed to executed serially. The advantage of speedup is that it allows a problem to be modeled faster. If multiple execution flows are able to work at the same time, the work will be finished in less time than it would take a single execution flow. Speedup is an enticing advantage. The limitations of speedup will be explained later.

Accuracy is the idea of forming a better solution to a problem. If more processes are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the problem that is being modeled. In order to make a program more accurate, speedup may need to be sacrificed.

Scaling is perhaps the most promising of the three. Scaling says that more parallel processors can be used to model a bigger problem in the same amount of time it would take fewer parallel processors to model a smaller problem. A common analogy to this is that one person in one boat in one hour can catch a lot fewer fish than ten people in ten boats in one hour.

As stated before, there are issues that limit the advantages of parallelism; we will address two in particular. The first, communication overhead, refers to the time that is lost waiting for communications to take place before and after calculations. During this time, valuable data is being communicated, but no progress is being made on executing the algorithm.

The communication overhead of a program can quickly overwhelm the total time spent modeling the problem, sometimes even to the point of making the program less efficient than its serial counterpart. Communication overhead can thus mitigate the advantages of parallelism.

A second issue is described in an observation put forth by Gene Amdahl and is commonly referred to as *Amdahl's Law*. Amdahl's Law says that the speedup of a parallel program will be limited by its serial regions, or the parts of the algorithm that cannot be executed in parallel. Amdahl's Law posits that as the number of processors devoted to the problem increases, the advantages of parallelism diminish as the serial regions become the only parts of the code that take significant time to execute. Amdahl's Law is given by:

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)} \tag{3}$$

Where $S(n)$ is the theoretical speedup of the program, B is the portion of the program that is strictly serial, and n is the number of processors.

Amdahl's Law provides a strong and fundamental argument against utilizing parallel processing to achieve speedup. However, it does not provide a strong argument against using it to achieve accuracy or scaling. The latter of these is particularly promising, as it allows for bigger classes of problems to be modeled as more processors become available to the program. The advantages of parallelism for scaling are summarized by John Gustafson in Gustafson's Law, which says that bigger problems can be modeled in the same amount of time as smaller problems if the processor count is increased. Gustafson's Law is given by:

$$S(n) = n - B(n - 1) \tag{4}$$

Where $S(n)$ is the theoretical speedup of the program, B is the portion of the program that is strictly serial, and n is the number of processors.

Amdahl's Law reveals the limitations of what is known as *strong scaling*, in which the number of processes remains constant as the problem size increases. Gustafson's Law reveals the promise of *weak scaling*, in which the number of processes varies as the problem size increases.

Quick Review Questions:

7. What is Amdahl's Law? What is Gustafson's Law?
8. What is the difference between strong scaling and weak scaling?

3.3 Estimating Pi in Parallel — MPI

The Message Passing Interface (MPI) allows communication among independent processes in an execution flow that uses distributed memory. MPI consists of various *data structures*. Data structures consist of constants (structures whose values do not change throughout the course of the algorithm) and variables (structures whose values do change). Access to data structures may be limited to an individual process (private) or accessible to all processes (global).

In the context of Area Under a Curve, the entire algorithm will be executed once for each process. This will require the implementation of certain data structures to manage the division of work among different processes. The primary data structure used to pass information between processes will simply be called MPI. MPI will be used to inform each process of the total number of processes (np), as well as the process' own ID ($rank$)[1]. Finally, MPI is used to collect all the work from the individual processes. MPI and np are public data structures and remain constant for all processes. $rank$ is a private data structure and will be different for each process.

With reference to Equation 2, N and Δx_i are public, while i and x_i are private. The MPI algorithm operates on the same principles as Algorithm 1, however each process sums over a subset of the range $0 \leq x < 1$.

The MPI algorithm is shown in Algorithm 2.

3.4 Estimating Pi in Parallel — OpenMP

Open Multi-Processing (OpenMP) supports shared memory multiprocessing. OpenMP allows multiple execution threads to access the same block of memory without message passing. Similar to MPI, OpenMP has various data structures that may be shared (global) or private to each thread. We will refer to the abstract OpenMP data structure as **OMP**. To use OpenMP for Area Under a Curve, we need only provide the desired number of threads (t) and indicate which part of the algorithm to execute in parallel. In this case, we will execute the while loop of the Area Under a Curve algorithm (Algorithm 1) in parallel.

Additionally, we must also specify which data structures are shared (N,Δx) and which data structures are private (i)[2]. OpenMP will then automatically assign a subset of the range of rectangles to each thread. The OpenMP algorithm is shown in Algorithm 3.

[1] MPI assigns process ranks starting at zero. For example, three processes will be ranked 0, 1, and 2.

[2] Each thread will have its own range of i values. Unlike MPI, OpenMP will automatically determine $first$ and $last$ for each thread.

Algorithm 2 Area Under a Curve — MPI

Require: N, np
Ensure: N and np are integers greater than 0
  $MPI(np)$ ▷ Initialize MPI processes
  $MPI(rank)$ ▷ Inform this process of its $rank$
  $first = rank * (N/np)$ ▷ Determine $first$ rectangle for this process
  $last = first + (N/np))$ ▷ Determine $last$ rectangle for this process
  if $rank = np - 1$ **then** ▷ If this is the last process
    $last = N$ ▷ Finish the remaining rectangles
  end if
  $i \leftarrow first$ ▷ Initialize i for this process
  $sum \leftarrow 0$ ▷ Initialize sum for this process
  $\Delta x = 1/N$ ▷ Calculate Δx
  while $i < last$ **do** ▷ Repeat for this subset of rectangles
    $x = i/N$ ▷ Calculate x
    $area = (1 - x^2)\Delta x$ ▷ Calculate rectangle $area$
    $sum \leftarrow sum + area$ ▷ Add $area$ to sum
    $i \leftarrow i + 1$ ▷ Increment i
  end while
  $sum \leftarrow 4 * sum$ ▷ Approximate partial sum for this process
  $PI \leftarrow MPI(sum)$ ▷ Collect all partial $sums$

Algorithm 3 Area Under a Curve — OpenMP

Require: N, t
Ensure: N and t are integers greater than 0
  $i \leftarrow 0$ ▷ Initialize i
  $PI \leftarrow 0$ ▷ Initialize PI
  $\Delta x = 1/N$ ▷ Calculate Δx
  OMP (Loop) ▷ Run the next loop in parallel
    shared $N, \Delta x, PI$ ▷ All threads have access
    private i ▷ Each thread has its own
    while $i < N$ **do** ▷ Repeat for all rectangles
      $x = i/N$ ▷ Calculate x
      $area = (1 - x^2)\Delta x$ ▷ Calculate rectangle $area$
      $PI \leftarrow PI + area$ ▷ Add $area$ to total sum
      $i \leftarrow i + 1$ ▷ Increment i
    end while
  EndOMP
  $PI \leftarrow 4 * PI$ ▷ Approximate PI

3.5 Estimating Pi in Parallel — Hybrid

The hybrid algorithm combines MPI and OpenMP. There will be some number of processes, each of which has some number of threads. In this case, the parameter t indicates the number of OpenMP threads to execute per MPI process. As we'll see, the shared memory version of the algorithm can be refined out of the hybrid version by assuming only one process with multiple threads. The distributed memory version can be refined out of the hybrid version by assuming multiple processes, each with only one thread. The serial version can also be refined out of the hybrid version by assuming only one total process with one total thread.

The hybrid algorithm is shown in Algorithm 4.

Algorithm 4 Area Under a Curve — Hybrid

Require: N, np, t
Ensure: N, np, and t are integers greater than 0
  $MPI(np)$ ▷ Initialize MPI processes
  $MPI(rank)$ ▷ Inform this process of its $rank$
  $first = rank * (N/np)$ ▷ Determine $first$ rectangle for this process
  $last = first + (N/np))$ ▷ Determine $last$ rectangle for this process
  if $rank = np - 1$ **then** ▷ If this is the last process
    $last = N$ ▷ Finish the remaining rectangles
  end if
  $i \leftarrow first$ ▷ Initialize i for this process
  $sum \leftarrow 0$ ▷ Initialize sum for this process
  $\Delta x = 1/N$ ▷ Calculate Δx
  OMP (Loop) ▷ Split the next loop across threads for this process
    shared $N, \Delta x, sum$ ▷ All this process's threads have access
    private i ▷ Each thread has its own
    while $i < last$ **do** ▷ Repeat for this subset of rectangles
      $x = i/N$ ▷ Calculate x
      $area = (1 - x^2)\Delta x$ ▷ Calculate rectangle $area$
      $sum \leftarrow sum + area$ ▷ Add $area$ to sum
      $i \leftarrow i + 1$ ▷ Increment i
    end while
  EndOMP
  $sum \leftarrow 4 * sum$ ▷ Approximate partial sum for this process
  $PI \leftarrow MPI(sum)$ ▷ Collect all partial $sums$

3.6 Generalization of the algorithm using the Berkeley Dwarfs

The Berkeley *dwarfs* are equivalence classes of important applications of scientific computing. Applications are grouped into dwarfs based on their computation and communication patterns. When an application is run in parallel, a certain percentage of time is spent performing calculations, while another percentage is spent communicating the results of those calculations. The dwarf captures the general trends of these percentages.

The application of approximating the area under a curve falls into the MapReduce dwarf. Applications in this class are characterized by a single function that is applied in parallel to unique sets of data. In the case of area under a curve, the function is simply the calculation of the area of a rectangle. The sets of data are the rectangles. In MapReduce applications, after the calculations have been mapped to the execution flows, they are communicated to a single execution flow (usually Rank 0 or Thread 0), where they are reduced. In the case of area under a curve, this "reduction" takes the form of a summation. Applications in the MapReduce dwarf are *embarrassingly parallel* because the calculations can be performed in their entirety without need for communication in between. Communication happens once, at the end, to perform the reduction.

Once we have an idea of how the area under a curve algorithm scales, we can extrapolate what we learn to a generalization of a larger classification of problems. Because the applications in the MapReduce dwarf have roughly the same computation and communication patterns, it is likely that the results of our scaling exercise could be generalized to all applications in the dwarf. In other words, if another MapReduce application were to be scaled in the same way, the results would likely be similar.

Quick Review Question:

9. What is a Berkeley dwarf?

4.0 Experimenting with Area Under a Curve

If this is your first time using OnRamp, refer to the appropriate module to learn how to get started using OnRamp.

4.1 Scaling the Algorithm

Now that we have developed a parallel algorithm, a natural next question is, "does the algorithm scale?" Because of the limitations revealed by Amdahl's Law, we can be assured that the algorithm will not scale far if we merely

increase the number of cores devoted to the problem (strong scaling); the code will initially run faster but will see diminished returns as communication overhead overwhelms the total amount of time spent running the code. As the number of processes or threads increases with a fixed number of rectangles, each process or thread will have less and less work to do.

Gustafson's Law promises that if we increase the problem size as we increase the core count (weak scaling), we will be able to model a bigger problem in the same amount of time. The goal of this section is to see if strong scaling of our parallel code fails as predicted by Amdahl's Law and to see if weak scaling of our parallel code succeeds as predicted by Gustafson's Law.

4.1.1 Strong Scaling

For strong scaling, the problem size stays fixed but the number of processing elements are increased. This is used as justification for programs that take a long time to run (something that is *cpu-bound*). The goal in this case is to find a "sweet spot" that allows the computation to complete in a reasonable amount of time, yet does not waste too many cycles due to parallel overhead. In strong scaling, a program is considered to scale linearly if the speedup (in terms of work units completed per unit time) is equal to the number of processing elements used. In general, it is harder to achieve good strong-scaling at larger process counts since the communication overhead for many/most algorithms increases in proportion to the number of processes used[2].

For this exercise you may use a table similar to Table 1. On the OnRamp interface, submit a job using 1 process, 1 thread, and 100,000,000 rectangles, and record the time (t_1) for reference. Then continue to submit jobs and record the times, altering the number of processes and/or the number threads per process, each time, but keeping the number of rectangles the same.

If the amount of time to complete a work unit with 1 processing element is t_1, and the amount of time to complete the same unit of work with N processing elements is t_N, the strong scaling efficiency, E_s (as a percentage of linear), is given as[2]:

$$E_s = \frac{t_1}{N * t_N} * 100\% \tag{5}$$

Table 1: Strong Scaling with 100,000,000 Rectangles

Processes	Threads per Process	Rectangles	Time
1	1	100,000,000	
1	2	100,000,000	
1	4	100,000,000	
1	8	100,000,000	
2	1	100,000,000	
2	2	100,000,000	
2	4	100,000,000	
2	8	100,000,000	
4	1	100,000,000	
4	2	100,000,000	
4	4	100,000,000	
4	8	100,000,000	

Questions

1. Did you identify a "sweet spot"? Where did it occur?
2. Did the program scale linearly?
3. In general, how did the number of processes affect the run times? Number of threads?
4. Would you expect performance to change on another cluster? Why or why not?

4.1.2 Weak Scaling

For weak scaling, the problem size (workload) assigned to each processing element stays constant and additional elements are used to solve a larger total problem (one that wouldn't fit in RAM on a single node, for example). Therefore, this type of measurement is justification for programs that take a lot of memory or other system resources (something that is *memory-bound*).

In the case of weak scaling, linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors[2].

For this exercise you may use a table similar to Table 2. On the OnRamp interface, submit a job using 1 process, 1 thread, and 100,000,000 rectangles, and record the time (t_1) for reference. Then continue to submit jobs and record the times, altering the number of processes and/or the number threads per process, each time. Each time you change the number of processes or threads, increase the number of rectangles commensurately.

If the amount of time to complete a work unit with 1 processing element is t_1, and the amount of time to complete N of the same work units with N processing elements is t_N, the weak scaling efficiency, E_w (as a percentage of linear), is given as[2]:

$$E_w = \frac{t_1}{t_N} * 100\% \tag{6}$$

Table 2: Weak Scaling

Processes	**Threads per Process**	**Rectangles**	**Time**
1	1	100,000,000	
1	2	200,000,000	
1	4	400,000,000	
1	8	800,000,000	
2	1	200,000,000	
2	2	400,000,000	
2	4	800,000,000	
2	8	1,600,000,000	
4	1	400,000,000	
4	2	800,000,000	
4	4	1,600,000,000	
4	8	3,200,000,000	

Questions

1. How did the run time vary with the workload?
2. Did the program scale linearly?
3. In general, how did the number of processes affect the run times? Number of threads?
4. Would you expect performance to change on another cluster? Why or why not?

5.0 Diving Deeper

Student Project Ideas:

1. Describe another MapReduce application–one in which a set of instructions is mapped to data and then reduced–and design a parallel algorithm for it.
2. Consider the scaling of other applications in the MapReduce dwarf, comparing the results to those found in this module.
3. Research two other Berkeley dwarfs and describe their communication and computation patterns.
4. Write an algorithm which computes the volume under the graph of a function z = f(x,y), over some rectangular region in the x-y plane.

Other Modules:

1. Refer to the Monte Carlo module for a different algorithm that could be used to estimate pi.

References

1. Aaron Weeden, "Petascale: Area Under a Curve." http://www.shodor.org/petascale/materials/UPModules/AreaUnderCurve/. Accessed 17 September 2015.
2. SHARCNET Documentation, "Measuring parallel scaling performance." https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance. Accessed 15 October 2015.

Area Under a Curve

Scaling Study Worksheet

Name **Instructor**

Date **Course**

Type of Scaling Study

Parallel Computing Environment (PCE) Information

Hostname	
Available Nodes	
Cores per Node	

Processes	Threads per Process	Rectangles	Time	Efficiency

Instructions

1. Log on to the OnRamp web-server
2. Select a workspace
3. Select a PCE
4. Select the AUC module
5. The right-hand column summarizes important concepts from the module documentation. At this point, decide whether to conduct a *strong scaling* or a *weak scaling* study.
6. Input a 'job_name.' This will be used by the PCE to track your job status.
7. Input parameters into the relevant fields that are appropriate to the PCE and type of scaling studying being conducted. The AUC program accepts 5 parameters:
   (a) **onramp np:** Number of MPI processes
   (b) **onramp nodes:** Number of compute nodes
   (c) **AUC threads:** Number of OpenMP threads per process
   (d) **AUC rectangles:** Number of rectangles for the Riemann Sum
   (e) **AUC mode:** Character indicating version of AUC to run
      i. 's' = serial
      ii. 'o' = OpenMP only
      iii. 'm' = MPI only
      iv. 'h' = hybrid OpenMP and MPI
8. Click the 'Launch Job' button
9. Find your 'Run Name' among the list of 'My Jobs.' Click 'View Details.'
10. Record the details of the job output including the input parameters and the time.
11. Repeat the process for different parameters, depending on the scaling study being conducted.

Questions

1. Did you identify a "sweet spot"? Where did it occur?

2. How did the run time vary with the workload?

3. Did the program scale linearly?

4. In general, how did the number of processes affect the run times? Number of threads?

5. Would you expect performance to change on another cluster? Why or why not?