Parallelization: Area Under a Curve

By Aaron Weeden, Shodor Education Foundation, Inc.

I. Abstract

This module teaches: 1) How to approximate the area under a curve using a Riemann sum, 2) how approximating the area under a curve is used in solutions to scientific problems, 3) how to implement parallel code for Area Under a Curve (including versions that use shared memory via OpenMP, distributed memory via the Message Passing Interface (MPI), and hybrid via a combination of MPI and OpenMP), 4) how to measure the performance and scaling of a parallel application in multicore and manycore environments, and 5) how Area Under a Curve falls into the MapReduce "dwarf" (a class of algorithms that have similar communication and computation patterns).

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) Design a parallel algorithm and implement it using MPI and/or OpenMP, 3) Measure the scalability of a parallel code over multiple or many cores, and 4) Explain the communication and computation patterns of the MapReduce dwarf.

It is assumed that students will have prerequisite experience with C or Fortran 90, Linux/Unix systems, and modular arithmetic.

II. Area Under a Curve: Motivation and Introduction

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 [1]. 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 [2]. Besides these two examples, there are many applications of the area under a curve in many fields of science, including pharmacokinetics and clinical pharmacology [3][4], machine learning [5], medicine [6][7], psychiatry and psychology [8], chemistry [9], environmental science [10], fisheries and aquatic sciences [11], and many others.

The calculus developed by Isaac Newton and Gottfried Leibniz in the 17th century allows for the exact calculation of the area of simple curves through integration, but for many functions integrals do not exist for finding the area under their curves, or these integrals cannot be used to find the area in a reasonable number of steps. To compensate for this, other techniques can be used that provide acceptable approximations for the area under a curve. This module considers the Riemann method of integration, developed by Bernhard Riemann in the 19th century for approximating the area under a curve.

The specific Riemann method explored in this module involves dividing the domain over which we are integrating into segments of equal width that serve as the bases of

rectangles[1]. The heights of the rectangles correspond to the y-value of the function for an x-value found somewhere within the rectangles' widths. 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 each rectangle has the height of the left-most point of its width. A pictorial example of the Left Riemann sum is Figure 1.

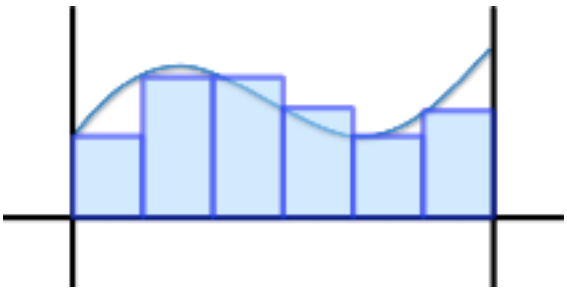


Figure 1: Using 6 rectangles to find a Left Riemann sum

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 the 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.

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.

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?

III. 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.

---

[1] In a Riemann sum, it is not necessary for all rectangles to be of equal width. In fact, it can be advantageous to have rectangles of different sizes to fit the curve better. For the purposes of creating a simple model in this module, we will assume that all rectangles are of equal width.

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.[2] 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.[3]

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?

IV. 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

[2] It should be noted that shared memory is really just a form of fast message passing. Threads must communicate, just as processes must, but threads get to communicate at bus speeds (using the front-side bus that connects the CPU to memory), whereas processes must communicate at network speeds (ethernet, infiniband, etc.), which are much slower.
[3] Threads can also have their own private memories, and OpenMP has directives to define whether variables are public or private.

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"?

V. 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[4]. 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[5] 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

---

[4] Another noteworthy advantage of parallelism, not discussed in much detail here, is that it can be used for applications that require larger amounts of memory than is available on a single system. In such cases, by splitting up the memory among multiple computers, any one computer is able to store less data than it would have to in the serial algorithm.

[5] Note that we refer to "modeling" a problem, not "solving" a problem. This follows the computational science credo that algorithms running on computers are just one tool used to develop *approximate* solutions (models) to a problem. Finding an actual solution may involve the use of many other models and tools.

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 represented as an equation in Figure 2.

$$\text{Speedup} = \frac{1}{1 - P + \frac{P}{N}}$$

where

P = the proportion of the program that can be made parallel

1 – P = the proportion of the program that cannot be made parallel

N = the number of processors

Figure 2: Amdahl's Law

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 represented as an equation in Figure 3.

$$\text{Speedup}(N) = N - (1 - P) * (N - 1)$$

where

N = the number of processors

(1 – P) = the proportion of the program that cannot be made parallel

Figure 3: Gustafson's Law

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. We will further explore the limitations posed by Amdahl's Law and the possibilities provided by Gustafson's Law in Section VIII.

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"?

VI. The parallel algorithm

The parallel algorithm is designed with hybrid parallelism in mind. There will be some number of processes, each of which has some number of threads. 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.

Students can develop the algorithm themselves while reading this module by completing Exercise 1, an attachment to this module.

In order to introduce the parallel algorithm, a good first step is to identify the algorithm's 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). In hybrid parallelism, data structures can be private to each thread, global to the threads in a process but private from other processes, or global to all processes.

A good method for identifying data structures is to draw and label a picture of the problem being modeled by the algorithm. This helps give a clear visual representation of the problem. An example of a labeled picture of the left Riemann sum is in Figure 4.

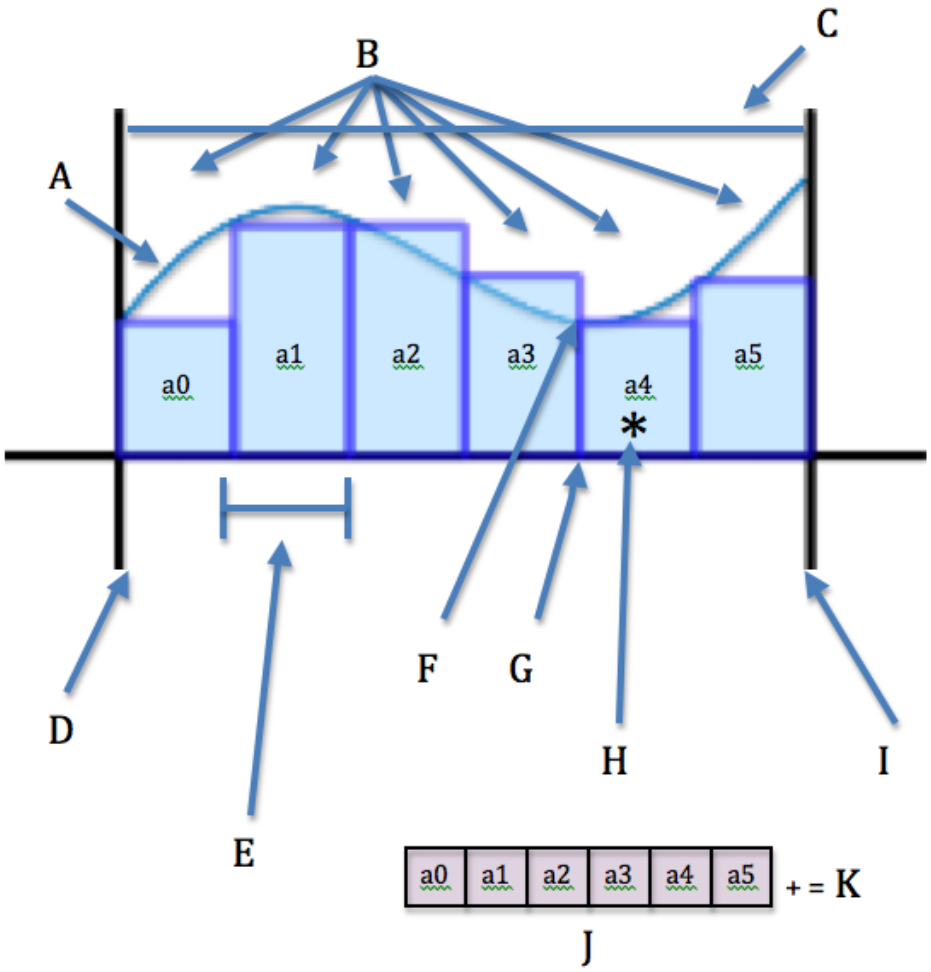


Figure 4: A picture of the left Riemann sum algorithm

After identifying the data structures, a good next step is to describe what they are and give them names. This helps to represent the data structures in writing, and it will be useful when we translate the algorithm in pseudocode later. An example of naming data structures is in Table 1.

Data structures		
Label in picture	**Written Representation**	**Name**
A	The function under whose curve we are approximating the area	FUNC( )
B	The total number of rectangles	NUMBER_OF_ RECTANGLES
C	The overall width of the domain of the function	WIDTH
D	The left x-boundary	X_LEFT

E	The width of a rectangle	RECTANGLE_ WIDTH
F	The height of the current rectangle	current_rectangle_ height
G	The x-value of the left side of the current rectangle	current_rectangle_left
H	The identifier of the current rectangle	current_rectangle_id
I	The right x-boundary	X_RIGHT
J	The array of the areas of the rectangles	areas[ ]
K	The total sum of the areas of all the rectangles	the_total_sum

Table 1: Written representation of data structures

There are also data structures that control the parallelism. Any parallel algorithm is likely to have data structures like these. These are listed in Table 2.

Data structures that control parallelism	
Written Representation	**Name**
The rank[6] of a process	RANK
The number of processes	NUMBER_OF_PROCESSES
The thread number[7] of a thread	THREAD_NUM
The number of threads	NUMBER_OF_THREADS

Table 2: Written representation of data structures

After we have a written representation of the data structures, we next indicate their scope, which tells whether they are public or private to the threads and processes that use them. One way to indicate scope is by changing the name of the data structures. If the data structure is public to all processes, its name does not change. If the data structure is public to all threads but private to each process, the word "our" is prepended to the front.[8] If the data structure is private to each thread, the word "my" is prepended to the front. An

[6] "Rank" is the terminology used by MPI to indicate the ID number of a process.
[7] "Thread number" is the terminology used by OpenMP to indicate the ID number of a thread.
[8] We take a thread-centric view here, because threads are the lowest form of execution flow in the hybrid parallelism we are using. From a thread's perspective, a public data structure is something that is shared among other threads like it. A thread would thus say that it is "our" data structure.

example of name changes is summarized in Table 3. Note that some data structures have been added because they are private copies of other, public data structures. These are indicated with asterisks (*) for their old names.

Data Structures	
Old Name	**New Name**
FUNC( )	FUNC( )
NUMBER_OF_ RECTANGLES	NUMBER_OF_ RECTANGLES
*	OUR_NUMBER_OF_ RECTANGLES
WIDTH	WIDTH
*	OUR_WIDTH
X_LEFT	X_LEFT
*	OUR_X_LEFT
RECTANGLE_ WIDTH	RECTANGLE_ WIDTH
current_rectangle_ height	my_current_rectangle_ height
current_rectangle_left	my_current_rectangle_left
current_rectangle_id	my_current_rectangle_id
X_RIGHT	X_RIGHT
*	OUR_X_RIGHT
areas[ ]	our_areas[ ]
the_total_sum	the_total_sum
*	our_total_sum
RANK	OUR_RANK
NUMBER_OF_PROCESSES	NUMBER_OF_PROCESSES
THREAD_NUM	MY_THREAD_NUM
NUMBER_OF_THREADS	OUR_NUMBER_OF_THREADS

Table 3: Name changes for indicating scope of data structures

After identifying the scope of the data structures, the next step is to identify how and when they interact. We start with how they interact and look to represent these interactions both in writing and through equations. This helps us get one step closer to representing the algorithm in pseudocode. An example of this is in Table 4.

Interactions of Data Structures	
Written Representation	**Equation Representation**
The width of the domain of the function is the difference of the right and left x-	WIDTH = X_RIGHT – X_LEFT

boundaries.	
The width of a rectangle is overall width of the domain of the function divided by the overall number of rectangles.	RECTANGLE_WIDTH = WIDTH / NUMBER_OF_RECTANGLES
The number of rectangles for which a process is responsible is determined by dividing the number of rectangles by the number of processes.[9]	OUR_NUMBER_OF_ RECTANGLES = NUMBER_OF_RECTANGLES/ NUMBER_OF_PROCESSES
If there are leftover rectangles, the last process is responsible for them.	if OUR_RANK == NUMBER_OF_PROCESSES – 1, then OUR_NUMBER_OF_ RECTANGLES = OUR_NUMBER_OF_ RECTANGLES + NUMBER_OF_ RECTANGLES mod NUMBER_OF_PROCESSES
The left x-boundary of a process's rectangles is obtained by multiplying the rank of the process by the number of rectangles for which each process is responsible, and adding it to the overall left x-boundary.	OUR_X_LEFT = (OUR_RANK * (NUMBER_OF_RECTANGLES / NUMBER_OF_PROCESSES)) * RECTANGLE_WIDTH + X_LEFT
The x-value of the current rectangle's left side is obtained by calculating how many rectangle widths it is away from that rectangle's thread's process's left x-boundary.	my_current_rectangle_left = OUR_X_LEFT + my_current_rectangle_id * RECTANGLE_WIDTH
The height of a rectangle is obtained by using the x-value of the left side of that rectangle to evaluate the function under whose curve we are approximating the area.	my_current_rectangle_ height = FUNC(my_current_ rectangle_left)
The area of a rectangle is obtained by multiplying the width of the rectangle by the height of the rectangle.	our_areas[my_current_ rectangle_id] = RECTANGLE_WIDTH * my_current_rectangle_ height
The total sum of the areas of the rectangles for which a process is responsible is obtained by adding the areas of that	our_total_sum = our_areas[0] + our_areas[1] + … + our_areas[OUR_NUMBER_

[9] It should be noted that if the number of rectangles is less than the number of processes, the last process will receive all of the rectangles, which is not an efficient method. For simplification purposes, we will assume in this module that the number of rectangles is greater than or equal to the number of processes. If the number of rectangles is greater than the number of processes, the decimal of the result is removed.

process's rectangles.	OF_RECTANGLES – 1]
The total sum is the sum of all the areas of the rectangles of all the processes.	For each process, the_total_sum = the_total_sum + our_total_sum

Table 4: Interactions of data structures

Now that we know how the data structures interact with each other, we next identify when they should interact with each other. "When" refers here to both "during which step of the algorithm" and "under which conditions." To indicate when the interactions should take place, we use each row in a table as a step of the algorithm. In the second column of the table, we indicate the conditions under which the step takes place. We also indicate how many times the step should take place, and which threads and processes should execute the step. An example of this is Table 5.

Steps of the algorithm (order of interactions)					
Step	**Interaction**	**Condition**	**Number of times**	**Which process?**	**Which thread?**
0	WIDTH = X_RIGHT – X_LEFT	always	1	all	0
0	OUR_NUMBER_OF_ RECTANGLES = NUMBER_OF_ RECTANGLES/ NUMBER_OF_ PROCESSES	always	1	all	1 mod OUR_NUMBER_OF_THREADS
0	OUR_X_LEFT = (OUR_RANK * (NUMBER_OF_ RECTANGLES/ NUMBER_OF_ PROCESSES)) * RECTANGLE_WIDTH + X_LEFT	always	1	all	2 mod OUR_NUMBER_OF_THREADS
1	RECTANGLE_WIDTH = WIDTH / NUMBER_OF_ RECTANGLES	always	1	all	0
1	OUR_NUMBER_OF_ RECTANGLES = OUR_NUMBER_OF_ RECTANGLES + NUMBER_OF_ RECTANGLES mod NUMBER_OF_	always	1	NUMBER_OF_PROCESSES - 1	1 mod OUR_NUMBER_OF_THREADS

	PROCESSES				
2	my_current_rectangle_ left = OUR_X_LEFT + my_current_rectangle_ id * RECTANGLE_WIDTH	always	Once for each rectangle	all	all
3	my_current_rectangle_ height = FUNC(my_current_ rectangle_left)	always	Once for each rectangle	all	all
4	our_areas[my_current_ rectangle_id] = RECTANGLE_WIDTH * my_current_rectangle_ height	always	Once for each rectangle	all	all
5	our_total_sum = our_total_sum + our_areas[my_current_ rectangle_id]	always	Once for each rectangle	all	0
6	the_total_sum = the_total_sum + our_total_sum	always	Once for each process	0	0

Table 5: Steps of the parallel algorithm

The table of steps gives us an outline for the algorithm that we can now translate into pseudocode. Interactions that need to happen multiple times are represented by loops in this pseudocode. An example of such pseudocode is shown in Figure 5.

```
All processes do the following:
0-1) Calculate the overall width of the domain of the
     function and the width of a rectangle.
     • If MY_THREAD_NUM == 0,
     • WIDTH = X_RIGHT – X_LEFT
     • RECTANGLE_WIDTH = WIDTH / NUMBER_OF_RECTANGLES
0-1)  Calculate the number of rectangles for which the
     process is responsible.
     • If MY_THREAD_NUM == (1 mod OUR_NUMBER_OF_THREADS),
     • OUR_NUMBER_OF_RECTANGLES = NUMBER_OF_RECTANGLES/
       NUMBER_OF_PROCESSES
     • If OUR_RANK == NUMBER_OF_PROCESSES – 1,
     • OUR_NUMBER_OF_RECTANGLES = OUR_NUMBER_OF_RECTANGLES
       + NUMBER_OF_RECTANGLES mod NUMBER_OF_PROCESSES
0)  Calculate the left x-boundary of the process.
     • If MY_THREAD_NUM == (2 mod OUR_NUMBER_OF_THREADS),
```

```
    • OUR_X_LEFT = (OUR_RANK * (NUMBER_OF_RECTANGLES/
      NUMBER_OF_PROCESSES)) * RECTANGLE_WIDTH + X_LEFT
2-4)  For each rectangle, parallelized by thread,
    2)  Calculate the x-value of the left side of the
        rectangle:
      • my_current_rectangle_left = OUR_X_LEFT +
        my_current_rectangle_id * RECTANGLE_WIDTH
    3)  Calculate the height of the rectangle:
      • my_current_rectangle_height =
        FUNC(my_current_rectangle_left)
    4)  Calculate the area of the rectangle:
      • our_areas[my_current_rectangle_id] =
        RECTANGLE_WIDTH * my_current_rectangle_height
5)  Calculate the total sum for the process.
  • If MY_THREAD_NUM == 0,
  • for each rectangle,
  • our_total_sum = our_total_sum +
    our_areas[my_current_rectangle_id]
6)  Calculate the overall total sum.
  • Each process sends our_total_sum to Rank 0
  • Rank 0 adds sums to the_total_sum
```

Figure 5: Pseudocode for the algorithm

Note that steps 2-4 have been combined into one loop. Note also that step 6 contains the only communication of the algorithm; each process sends data to Rank 0.

We have now completed the process of defining the algorithm and developing pseudocode for it.

VII. Code implementation

Now that the pseudocode has been developed, the code can be implemented. It should first be noted that MPI processes each execute the entire code, but OpenMP threads only execute the sections of code for which they are "spawned", or created. This means that sections of code will only be executed with shared memory parallelism if they are contained within parallel OpenMP regions, but all code will be executed with distributed memory parallelism (except for sections of the code contained in conditionals that are only executed by a process of a certain rank).

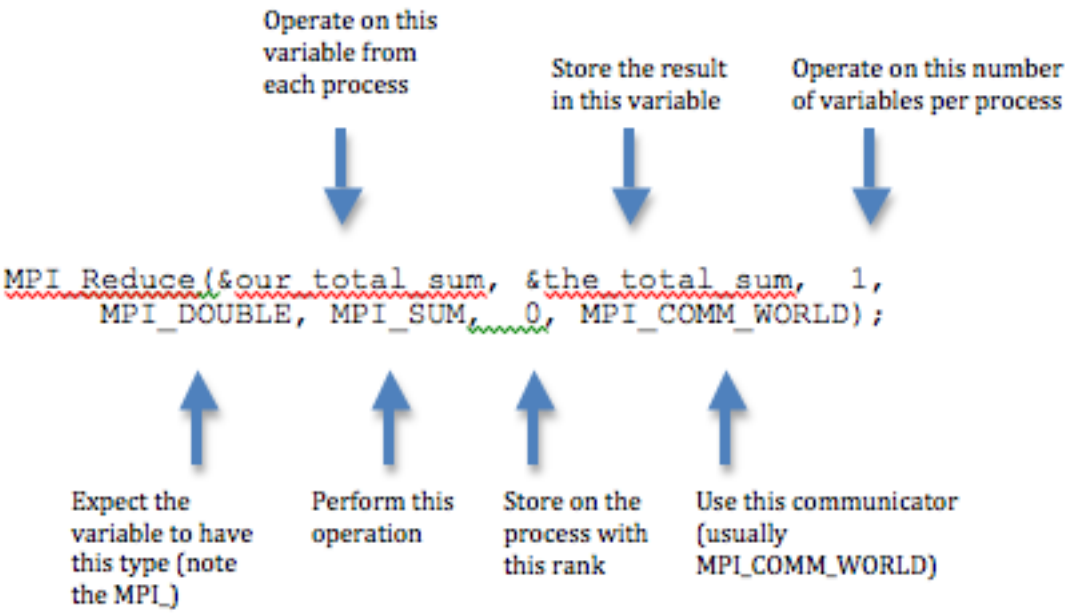
Before delving into this section, it is likely to be helpful to first run through Exercise 2 (an attachment to this module).

For serial programmers, the most confusing sections of the code are likely to be the MPI functions and OpenMP regions. The MPI functions that are used in the code are listed

and given explanations in Table 6. MPI_Reduce() is a bit more complicated (it takes a value from each of the processes, performs a calculation using all those values, and stores the result in a variable owned by Rank 0), so it is explored in more detail in Figure 6. The relevant OpenMP regions and functions are listed and given explanations in Table 7.

MPI Function	**Explanation**
MPI_Init()	This initializes the MPI environment. It must be called before any other MPI functions.
MPI_Comm_rank()	This assigns the rank of the process in the specified communicator to the specified variable.
MPI_Comm_size()	This assigns the number of processes in the specified communicator to the specified variable.
MPI_Reduce()	This takes a value from each process and performs an operation on all the values. The result is stored in a variable owned by one of the processes.
MPI_Finalize()	This cleans up the MPI environment. No other MPI functions may be called after this one.

Table 6: MPI Functions used by the code




```
CALL MPI_REDUCE(our_total_sum, the_total_sum, 1, &
     MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD,
     ierror)
```

(Fortran only) Store any error codes in this variable

Figure 6: Details about the arguments to MPI_Reduce()

OpenMP pragma (C)	**OpenMP region (Fortran 90)**	**Explanation**

#pragma omp parallel
{
…
}
	!$ OMP parallel
	…
	!$ OMP end parallel
		This spawns a pre-defined number of threads and parallelizes the section (indicated here by "…").

omp_get_thread_num()	OMP_GET_THREAD_ NUM()	This returns the number of threads that have been spawned in the

		parallel section in which it is called.

#pragma omp parallel for
for(…)
{
…
}
	!$ OMP parallel do
	DO …
	…
	END DO
	!$OMP end parallel do
		This spawns a pre-defined number of threads and parallelizes the for loop.

#pragma omp parallel private (…)
{
…
}
	!$ OMP parallel private (…)
	!$ OMP end parallel
		This spawns a pre-defined number of threads and parallelizes the section. Each thread has a private copy of each variable specified.

Table 7: OpenMP regions and functions used by the code

The parallel code should be implemented step-by-step and in small pieces. To start with, we look back at the constants, variables, and data structures, and decide how to declare them in the language of our choice. If the language is strongly typed, like C or Fortran90, the data type of the constant, variable, or data structure will need to be determined. Also, if we decide to allocate the array dynamically as we do here, we make sure to declare it as such. To guard against possible problems in the code later, we initialize all of the constants and variables to default values. Our C code declares the constants, variables, and data structures as in Figure 7, while our Fortran 90 code declares them as in Figure 8.

```
/* Declare the constants, variables, and data
 *  structures. */
int OUR_RANK = 0, OUR_NUMBER_OF_THREADS = 1,
   OUR_NUMBER_OF_RECTANGLES = 0, NUMBER_OF_RECTANGLES = 10,
   NUMBER_OF_PROCESSES = 1, MY_THREAD_NUM = 0,
   my_current_rectangle_id = 0;
double OUR_X_LEFT = 0.0, X_LEFT = 0.0, X_RIGHT = 10.0,
   WIDTH = 0.0, RECTANGLE_WIDTH = 0.0, our_total_sum = 0.0,
   the_total_sum = 0.0, my_current_rectangle_left = 0.0,
   my_current_rectangle_height = 0.0;
double * our_areas;
```

Figure 7: Declaration of constants, variables, and data structures in C

```
! Declare the constants, variables, and data
!  structures.
INTEGER :: OUR_RANK = 0, &
   OUR_NUMBER_OF_THREADS = 1, &
   OUR_NUMBER_OF_RECTANGLES = 0, NUMBER_OF_RECTANGLES &
   = 10 NUMBER_OF_PROCESSES = 1, &
   MY_THREAD_NUM = 0, &
   my_current_rectangle_id = 0
DOUBLE PRECISION :: OUR_X_LEFT = 0.0, X_LEFT = 0.0, &
   X_RIGHT = 10.0, WIDTH = 0.0, RECTANGLE_WIDTH = &
   0.0, our_total_sum = 0.0, the_total_sum = 0.0, &
   my_current_rectangle_left = 0.0, &
   my_current_rectangle_height = 0.0
DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: & our_areas
```

Figure 8: Declaration of constants, variables, and data structures in Fortran 90

MPI requires that we initialize the environment before any MPI code is executed. We then use MPI functions to determine the ranks of processes and number of processes. These steps are shown in Figure 9 for C and in Figure 10 for Fortran 90.

```
/* Initialize the MPI environment */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &OUR_RANK);
MPI_Comm_size(MPI_COMM_WORLD, &NUMBER_OF_PROCESSES);
```

Figure 9: MPI Initialization in C

```
! Initialize the MPI environment
CALL MPI_INIT(ierror)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, OUR_RANK, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, &
               NUMBER_OF_PROCESSES, ierror)
```

Figure 10: MPI Initialization in Fortran 90

Since we have chosen to allocate the array of areas dynamically, our next step is to allocate memory for it. The allocation in our C and Fortran 90 codes are shown in Figures 11 and 12, respectively.

```
/* Allocate the array. */
our_areas = (double*) malloc(NUMBER_OF_RECTANGLES *
   sizeof(double));
```

Figure 11: Allocation of array in C

```
! Allocate the array.
allocate(our_areas(NUMBER_OF_RECTANGLES))
```

Figure 12: Allocation of array in Fortran 90

Now that all of our constants, variables, and data structures have been declared and/or allocated, we are ready to implement the algorithm, starting by translating steps 0-1 of the pseudocode. The C code for steps 0-1 is in Figure 13, and the Fortran 90 code is in Figure 14. Note the OpenMP parallel region.

```
#pragma omp parallel
{
     MY_THREAD_NUM = omp_get_thread_num();

     if(MY_THREAD_NUM == 0)
     {
          /* Calculate the overall width of the
           *  domain of the function and the width of a
           *  rectangle.
           */
          WIDTH = X_RIGHT - X_LEFT;
          RECTANGLE_WIDTH = WIDTH /
               NUMBER_OF_RECTANGLES;
     }
     if(MY_THREAD_NUM == 1 %
               OUR_NUMBER_OF_THREADS)
     {
          /* Calculate the number of rectangles for
           *  which the process is responsible. */
          OUR_NUMBER_OF_RECTANGLES =
               NUMBER_OF_RECTANGLES /
               NUMBER_OF_PROCESSES;
          if(OUR_RANK == NUMBER_OF_PROCESSES - 1)
          {
             OUR_NUMBER_OF_RECTANGLES +=
             NUMBER_OF_RECTANGLES %
             NUMBER_OF_PROCESSES;
          }
     }
     if(MY_THREAD_NUM == 2 %
               OUR_NUMBER_OF_THREADS)
     {
```

```
          /* Calculate the left x-boundary of the
           *  process. */
          OUR_X_LEFT = (OUR_RANK *
                        (NUMBER_OF_RECTANGLES /
                        NUMBER_OF_PROCESSES)) +
                          X_LEFT;
     }
} /* pragma omp parallel */
```

Figure 13: Steps 0-1 of the algorithm in C

```
!$OMP parallel
     MY_THREAD_NUM = OMP_GET_THREAD_NUM()

     IF(MY_THREAD_NUM == 0) THEN
          ! Calculate the overall width of the
          !  domain of the function and the width of a
          !  rectangle.
          WIDTH = X_RIGHT - X_LEFT
          RECTANGLE_WIDTH = WIDTH / &
               NUMBER_OF_RECTANGLES
     END IF
     IF(MY_THREAD_NUM == MODULO(1, &
               OUR_NUMBER_OF_THREADS) THEN
          ! Calculate the number of rectangles for
          !  which the process is responsible.
          OUR_NUMBER_OF_RECTANGLES = &
               NUMBER_OF_RECTANGLES / &
               NUMBER_OF_PROCESSES
          IF(OUR_RANK == NUMBER_OF_PROCESSES - 1) THEN
            OUR_NUMBER_OF_RECTANGLES = &
            OUR_NUMBER_OF_RECTANGLES +
            MODULO(NUMBER_OF_RECTANGLES,
            NUMBER_OF_PROCESSES)
          END IF
     END IF
     IF(MY_THREAD_NUM == MODULO(2, &
               OUR_NUMBER_OF_THREADS)) THEN
         ! Calculate the left x-boundary of the
         !  process.
         OUR_X_LEFT = (OUR_RANK * &
                        (NUMBER_OF_RECTANGLES / &
                        NUMBER_OF_PROCESSES)) + &
                          X_LEFT
     END IF
```

```
!$ OMP end parallel
```

Figure 14: Steps 0-1 of the algorithm in Fortran 90

Because of their brevity and the fact that they exist in the same loop, we combine steps 2-4. Note that the loop is parallelized among all the threads using the "parallel for" OpenMP region. The C code for these steps is in Figure 15, and the Fortran 90 code is in Figure 16.

```
#pragma omp parallel for private(my_current_rectangle_
id, my_current_rectangle_left, my_current_rectangle_he
ight)
for(my_current_rectangle_id = 0;
     my_current_rectangle_id <
     OUR_NUMBER_OF_RECTANGLES;
     my_current_rectangle_id++)
{
     /* Calculate the x-value of the left side of the
      *  rectangle */
     my_current_rectangle_left = OUR_X_LEFT +
          my_current_rectangle_id * RECTANGLE_WIDTH;

     /* Calculate the height of the rectangle */
     my_current_rectangle_height =
          FUNC(my_current_rectangle_left);

     /* Calculate the area of the rectangle */
     our_areas[my_current_rectangle_id] =
          RECTANGLE_WIDTH *
               my_current_rectangle_height;
}
```

Figure 15: Steps 2-4 of the algorithm in C

```
!$ OMP parallel do private (my_current_rectangle_id,
my_current_rectangle_left, my_current_rectangle_heigh
t)
DO my_current_rectangle_id = 0, &
     OUR_NUMBER_OF_RECTANGLES - 1, 1
     ! Calculate the x-value of the left side of the
     !  rectangle
     my_current_rectangle_left = OUR_X_LEFT + &
          my_current_rectangle_id * RECTANGLE_WIDTH

     ! Calculate the height of the rectangle
```

```
     my_current_rectangle_height = &
          FUNC(my_current_rectangle_left)

     ! Calculate the area of the rectangle
     our_areas(my_current_rectangle_id + 1) = &
          RECTANGLE_WIDTH * &
               my_current_rectangle_height
END DO
!$OMP end parallel do
```

Figure 16: Steps 2-4 of the algorithm in Fortran 90

The penultimate step of the algorithm is step 5. Figure 17 is the C code for step 5, and Figure 18 is the Fortran 90 code.

```
/* Calculate the total sum for the process */
for(my_current_rectangle_id = 0;
     my_current_rectangle_id <
     OUR_NUMBER_OF_RECTANGLES;
     my_current_rectangle_id++)
{
     our_total_sum +=
          our_areas[my_current_rectangle_id];
}
```

Figure 17: Step 5 of the algorithm in C

```
! Calculate the total sum for the process
DO my_current_rectangle_id = 1, &
     OUR_NUMBER_OF_RECTANGLES, 1
     our_total_sum = our_total_sum + &
          our_areas(my_current_rectangle_id)
END DO
```

Figure 18: Step 5 of the algorithm in Fortran 90

Finally, step 6 of the algorithm calculates the total sum of all the areas of all the rectangles. To accomplish this, all processes send their sums to Rank 0, and it calculates the total sum. Figure 19 shows how the C version of MPI_Reduce() is used to find the total sum, and Figure 20 shows how the Fortran 90 version is used.

```
/* Calculate the overall total sum */
```

```
MPI_Reduce(&our_total_sum, &the_total_sum, 1,
          MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Figure 19: Step 6 of the algorithm in C

```
! Calculate the overall total sum
CALL MPI_REDUCE(our_total_sum, the_total_sum, 1, &
     MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD,
     ierror)
```

Figure 20: Step 6 of the algorithm in Fortran 90

We have now finished the algorithm as described by the pseudocode. In order to obtain the result of the total sum, our code prints the final value. Also, because we have chosen to allocate the array of areas dynamically, we should deallocate it before the end of the program to avoid a memory leak. We also need to finalize the MPI environment. These final steps are shown in Figure 21 for C and in Figure 22 for Fortran 90.

```
/* Print the total sum */
if(OUR_RANK == 0)
     printf("%f\n", the_total_sum);

/* Deallocate the array */
free(our_areas);

/* Finalize the MPI environment */
MPI_Finalize();
```

Figure 21: Printing the result, deallocating the array, and finalizing the MPI environment in C

```
! Print the total sum
IF(OUR_RANK == 0) THEN
     PRINT *, the_total_sum
END IF

! Deallocate the array
DEALLOCATE(our_areas)

! Finalize the MPI environment
CALL MPI_FINALIZE(ierror)
```

Figure 22: Printing the result, deallocating the array, and finalizing the MPI environment in Fortran 90

This completes the parallel code for approximating the area under the curve. The code is available in its entirety as attachments to this module. A Makefile is also attached to facilitate easy compilation of the code. Note that the code uses #ifdefs to surround sections of MPI and OpenMP code. This allows all four versions of the program (serial, MPI, OpenMP and Hybrid) to be built from the same source code.

Students should now be able to run the code using various input parameters. The attached code allows for command line arguments to the function, which specify the number of rectangles, the left x-boundary, and the right x-boundary. See the block comment at the beginning of the code for documentation on using these command-line options.

VIII. 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. However, 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.

Exercise 3 (an attachment to this module) describes how to scale the code on a cluster. The results of this exercise are likely to produce something like Table 8 and Table 9, which were produced by following Exercise 3's instructions.

# of nodes used	# of cores per node used	Total # of cores	Total # of Rectangles	Serial	OpenMP	MPI	Hybrid
1	1	1	100,000,000	3.006	0.745	2.402	0.888
1	2	2	100,000,000	3.001	0.745	2.276	2.175
1	4	4	100,000,000	3.008	0.803	1.719	2.892
2	1	2	100,000,000			1.401	0.585
2	2	4	100,000,000			1.828	1.029
2	4	8	100,000,000			1.527	2.799
4	1	4	100,000,000			0.827	0.426

4	2	8	100,000,000			1.541	1.976
4	4	16	100,000,000			1.434	2.907
8	1	8	100,000,000			0.577	0.371
8	2	16	100,000,000			1.444	2.016
8	4	32	100,000,000			1.669	2.960

Table 8 – Strong Scaling

# of nodes used	# of cores per node used	Total # of cores	Total # of rectangles	Serial	OpenMP	MPI	Hybrid
1	1	1	100,000,000	3.010	0.745	2.399	0.865
1	2	2	200,000,000	6.023	1.441	3.418	2.641
1	4	4	400,000,000	12.014	2.823	3.439	4.084
2	1	2	200,000,000			2.636	0.961
2	2	4	400,000,000			2.654	2.893
2	4	8	800,000,000			3.702	4.440
4	1	4	400,000,000			2.674	1.020
4	2	8	800,000,000			3.702	3.079
4	4	16	1,600,000,000			3.734	4.500
8	1	8	800,000,000			3.723	1.235
8	2	16	1,600,000,000			3.740	3.073
8	4	32	3,200,000,000			1.366	2.892

Table 9 – Weak Scaling, 100,000,000 Rectangles per Core

Now that we have an idea of how the algorithm scales, we can extrapolate what we have learned to a generalization of a larger classification of problems.

IX. Generalization of the algorithm using the Berkeley Dwarfs

The Berkeley "dwarfs"[10] [12] 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.

[10] The Berkeley group originally started with seven equivalence classes; the name "dwarf" was chosen as an allusion to the seven dwarves in the *Snow White* fairy tale.

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 application, 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, just as our area under the curve code does a final sum using MPI_Reduce of the areas that were calculated by each process. Applications in the MapReduce dwarf are "embarrassingly parallel,"[11] 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.

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 as we did in this module, the results would likely be similar. A meaningful extension to this module would be to consider the scaling of other applications in the MapReduce dwarf, comparing the results to those found in this module.

Quick Review Questions:

9. What is a Berkeley "dwarf"?

X. Post-assessment Rubric

This rubric is to gauge students' knowledge and experience after using the materials and completing the exercises presented in this module. Students can be asked to rate their knowledge and experience on the following scale and in the following subject areas:

Scale

1 – no knowledge, no experience
2 – very little knowledge, very little experience
3 – some knowledge, some experience
4 – a good amount of knowledge, a good amount of experience
5 – high level of knowledge, high level of experience

Subject areas

Approximating the area under a curve
Serial (non-parallel) Algorithm Design
Parallel Algorithm Design
Parallel Hardware
MPI programming
OpenMP programming
Using a cluster

[11] The term "embarrassingly parallel" refers to the fact that it is easy to parallelize the problem (it would be embarrassing not to). The origin of the term is unknown.

In addition, students are asked the following questions:

1. What did you find to be the most useful aspect of this module?
2. What did you find to be the least useful aspect?
3. What did you learn about that you would be excited to learn more about?
4. What did you find particularly unuseful or irrelevant?

XI. 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) Research two other Berkeley "dwarfs" and describe their communication and computation patterns.

3) Write `volume.c`, which computes the volume under the graph of a function z = f(x,y), over some rectangular region in the x-y plane. If the function is z = x * y, does your program return an area of about 0.25 in the region from (x,y) = (0.0, 0.0) to (1.0, 1.0)?

XII. References

[1] Pruessner, Jens C., Kirschbaum, Clemens, Meinlschmid, Gunther, and Hellhammer, Dirk H. (October 2003). Two formulas for computation of the area under the curve represent measures of total hormone concentration versus time-dependent change. Psychoneuroendocrinology, 28(7), 916-931. doi:10.1016/S0306-4530(02)00108-7.

[2] Myerson, J., Green, L., and Warusawitharana, M. (September 2001). Area under the curve as a measure of discounting. Journal of the Experimental Analysis of Behavior, 76(2), 235–243. doi:10.1901/jeab.2001.76-235.

[3] Dix, S. P., and others. (1996). Association of busulfan area under the curve with veno-occlusive disease following BMT. *Bone marrow transplantation, 17(2), 225-230*. Retrieved from http://cat.inist.fr/?aModele=afficheN&cpsidt=2975645

[4] Shaw, L. M., and others. (June 2000). Mycophenolic acid area under the curve values in African American and Caucasian renal transplant patients are comparable. *The Journal of Clinical Pharmacology, 40(6), 624-633*. Retrieved from http://jcp.sagepub.com/content/40/6/624.short

[5] Bradley, Andrew P. (July 1997). The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recognition, 30(7), 1145-1159. doi:10.1016/S0031-3203(96)00142-2.

[6] Le Floch, Jean-Pierre, Escuyer, Philippe, Baudin, Eric, Baudon, Dominique, and Perlemuter, Léon. (February 1990). Blood Glucose Area Under the Curve: Methodological Aspects. Diabetes Care, 13(2), 172-175. doi:10.2337/diacare.13.2.172

[7] Orenstein, D. M. and Kaplan R. M. (October 1991). Measuring the quality of well-being in cystic fibrosis and lung transplantation: the importance of the area under the curve. CHEST, 100(4), 1016-1018. doi:10.1378/chest.100.4.1016.

[8] Goldberg, D. P. and others. (1997). The validity of two versions of the GHQ in the WHO study of mental illness in general health care. *Psychological Medicine, 27, 191-197.* Retrieved from http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=25271

[9] Gutowska, Anna. and others. (May 2005). Nanoscaffold Mediates Hydrogen Release and the Reactivity of Ammonia Borane. Angewandte Chemie International Edition, 44(23), 3578–3582. doi:10.1002/anie.200462602.

[10] Carnaval, Ana Carolina. (February 2009). Stability Predicts Genetic Diversity in the Brazilian Atlantic Forest Hotspot. Science, 323(5915), 785-789. doi:10.1126/science.1166955.

[11] English, K. K., Bocking, R. C., Irvine, J. R. (1992). A Robust Procedure for Estimating Salmon Escapement based on the Area-Under-the-Curve Method. Canadian Journal of Fisheries and Aquatic Sciences, 49(10), 1982-1989. doi:10.1139/f92-220.

[12] Asanovic, Krste and others. (2006). The landscape of parallel computing research: A view from berkeley. University of California at Berkeley. Technical Report No. UCB/EECS-2006-183.