Area Under A Curve: Parallelizing the Riemann Sum

Samuel Leeman-Munk

April 11, 2011

Abstract

An explanation of the mathematical basis of Area Under Curve and a study of its parallelization

Contents

1 Background **1**

2 Implementation and Parallelization **2**

3 Laboratory: Observing the scaling of Area-Under-Curve **4**

3.1 Prerequisites 4

3.2 Materials 4

3.3 Procedure 4

3.4 Questions 8

3.5 Assessing Student Understanding 9

1 Background

The Riemann sum, named after German Mathematician Bernhard Riemann, is a method for estimating the area under a curve by splitting it into a finite number of

rectangles. Since its invention Riemann's sum has been improved by using trapezoids instead (Figure 1), which tend to be more accurate and have areas nearly as easy to calculate as rectangles:

$$A = width * \frac{height1 + height2}{2}$$

. The Riemann sum is popular among computer scientists because it presents a simple computational method for estimating definite integrals.

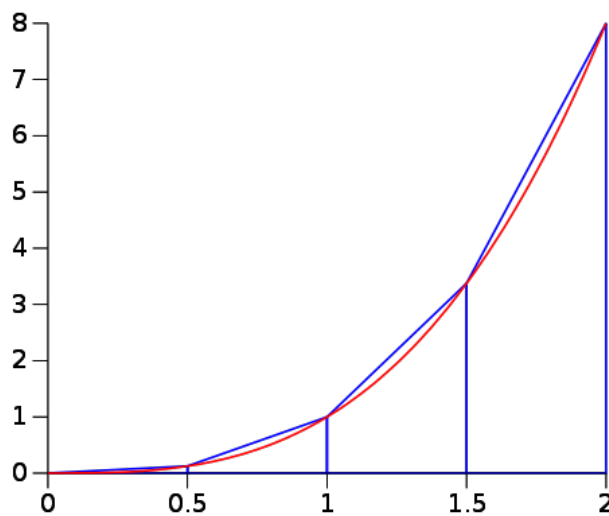

Figure 1: A four-segment Trapezoidal Riemann sum of a function

2 Implementation and Parallelization

The Riemann sum calculator's time to run depends almost exclusively on one factor: the number of segments. Because the calculation of each segment happens exactly once and takes the same amount of time anywhere in the function, the Riemann sum calculator runs at $O(n)$, meaning it scales linearly[1].

The Riemann sum calculator is a popular program among parallel computing educators because of its incredible simplicity. The entire algorithm for a trapezoidal

[1] How could you show this with a graph?

Riemann sum calculator consists of calculating a number of independent areas and summing them together. The main portion of the code is easily separated into an almost limitless number of subtasks that all take the same amount of time and are completely independent of each other. The Riemann sum calculator is an example of a problem often humorously described as "embarrassingly parallel." In other words, you should be embarrassed if you can't parallelize it.

The Riemann sum calculator is an example of the Map-Reduce dwarf [1] because of just this property. The Map-Reduce dwarf describes algorithms that spread a single function across many data and then sum the results together in a single unit[2]. Map-Reduce dwarfs are easy to parallelize because they require almost no communication – just one reduction where each client processor tells the server the value it got. Communication is always just one floating point value per processor no matter how many segments there are, so communication is constant with respect to problem size.

[2]Can you think of any other problems that can be solved via this method?

3 Laboratory:

Observing the scaling of Area-Under-Curve

3.1 Prerequisites

- Prior experience using UNIX command-line interface.
- Knowledge of how to use an available parallel processing environment[3]

3.2 Materials

- Computer (1 per student)
- Parallel Processing Environment[4]
- UPEP Area-Under-Curve Module

3.3 Procedure

1. From *Multidimensional Benchmarking with PetaKit* take the StatKit zip file[5] and unpack it into the directory where all your source code is.
2. In your source code directory, run the following command:

   ```
   make
   ```

   This will build your executables.
3. If you are running on a cluster that requires a scheduler, construct a template for submitting jobs to the scheduler[6]. Making a script template is as simple as filling in all the directives you'll want to be passed to your scheduler, for the values that

[3]In other words, do not attempt this lab unless you or someone with whom you are working can access a parallel machine that he or she can use. This includes but is not limited to clusters and multicore personal computers. Be aware that cluster computers are highly complicated devices, and most of the commands coming up will *not work* if you just copy them directly into your command line. In short, **Read the instructions carefully.**

[4]A cluster is ideal for this purpose, but if one is not available, nearly all modern computers have at least two cores, which can serve as a very small-scale parallelization environment

[5]https://shodor.org/media/content//petascale/materials/UPModules/pkit/StatKit.zip

[6]If you don't know how to submit jobs to your scheduler, speak with someone who does

will change substituting variables specified in the StatKit documentation (perl stat.pl --help). Make sure to include in your script where appropriate $special $prep and $main, in that order and each with its own line. These form the main command that actually runs your program. If you still don't understand, try looking at the existing templates in the script_templates directory to get a better idea of the format, and remember the StatKit documentation.

4. Save your template in the StatKit folder in the script_templates directory. Remember what you named your template. We will refer to it as "myTemplate.sh". We'll be referring to your cluster's scheduler as "myScheduler", also.

5. Now you're ready to begin making performance scaling graphs!

Serial:

Scale the serial code, running over several problem sizes and collecting the times.

In the StatKit folder that you downloaded earlier in this module (see materials), enter the "stats" folder and run the following command[7]:

```
perl stat.pl --cl '../serial_simple_integration 0 1 $problem_size' \
-t <myTemplate.sh> --problem_size 100,1000,10000,50000,100000 \
--repetitions 5 --proxy-output --tag mySerialArea \
--scheduler <myScheduler>
```

Remember to replace values in angle brackets (<>) with values appropriate to your cluster!

problem_size is the most significant variable in a program. The problem size is, in general, the variable that has the greatest effect on runtime. For instance, in area-under-curve the problem size is the number of segments. In StatKit, "repetitions" decides how many times you'll run at each problem size. Proxy output simulates the output that StatKit expects so the user doesn't have to program it into his

[7] The problem sizes given here are based on the runtimes of a single system. If your runs seem to be over too quickly and are registering as zero time units long, try multiplying each problem size by powers of ten until the fastest run is at least a hundredth of a second.

or her program in order to benchmark it[8].

While you wait for these runs to complete, consider what sort of scaling you will expect. Scaling is how the runtime of a problem changes as one increases some part of it. In this case, you're scaling over problem size.

Once all the runs are finished, return to your main directory and run:

```
 perl PlotKit.pl \
--independent problem_size  --dependent walltime \
--datafile stats/output.txt -o mygraph.ps mySerialArea
```

StatKit outputs its results into a file called output.txt. Output.txt remembers the variable names so you don't have to remember which column they're in. Take a look at output.txt to see how it's layed out. PlotKit.pl collects the relevant data, averages repeats and builds a graph. Independent and dependent should be self-explanatory, they decide which variables populate the x and y axes respectively. The last option must always be the tag(s) of the runs you want to see.

In whatever way you can on your machine, look at mygraph.ps. [9] If you can view graphical output directly on your machine, feel free to omit the '-o mygraph.ps" option to get a graph sent directly to your screen.

Shared Memory Parallelism:

Now that we have some parallelization, we'll scale AUC over the number of threads.

Be aware that if you are not running through a scheduler you will need to define OMP_NUM_THREADS to externally restrict the number of threads a shared memory parallel program uses. The "–special" argument will let you do this. In Bourne Again SHell (BASH), the value you'll want to give it is as follows:

```
 --special 'OMP\_NUM\_THREADS=$threads'
```

[8]More detailed information can be obtained by building StatKit output into the code. See the module associated with StatKit for more information

[9]If your computer can't handle .ps, try ps2pdf, or for Windows GhostScript http://pages.cs.wisc.edu/ ghost

In the StatKit folder, run the following command:

```
 perl stat.pl --cl '../omp_simple_integration 0 1 $problem_size' \
-t <myTemplate.sh> --problem_size 100,1000,10000,50000,100000 --threads 1-4\
--scheduler <myScheduler> --repetitions 5 --proxy-output --tag myOmpArea
```

```
 perl PlotKit.pl \
--independent threads  --dependent walltime --split problem_size \
-o mygraph.ps --datafile stats/output.txt myOmpArea
```

Notice the new "split" option, which allows us to "split" a graph into a different graph for each value of a variable. Observe your graph. How does the problem scale over the number of threads? How does the scaling change with respect to problem size? What does this tell you about parallel processing?

Distributed Memory Parallelism:

Now we'll compare distributed and shared scaling over number of processors and threads respectively.

Distributed Memory implementation is more challenging because it depends more heavily on cluster implementation. The following command is just an example, and you may need to work with your cluster's system administrator to figure out the correct command-line template.

```
 perl stat.pl \
--cl 'mpirun --bynode -np $processes --hostfile <myHostFile> \
 ../mpi_simple_integration 0 1 $problem_size' \
-t <myTemplate.sh> --problem_size 100000 --processes 1-8\
--scheduler <myScheduler>\
--repetitions 5 --proxy-output --tag myMPIArea
```

For comparison, generate some more shared memory data.

```
 perl stat.pl --cl '../area-omp 0 1 $problem_size' \
```

```
-t myTemplate.sh --problem_size 100000 --threads 1-4 --scheduler <myScheduler>\
--repetitions 5 --proxy-output --tag myOmpArea2
```

```
 perl PlotKit.pl \
--independent threads  --dependent walltime --split tag \
-o mygraph.ps --datafile stats/output.txt myOmpArea2,myMPIArea
```

To graph more than one tag, combine them with commas. In this example we then split over tag to compare the two runs. Notice that although MPI uses processes, PlotKit.pl still refers to the output as threads. For purposes of parsing convenience, StatKit always outputs the number of units working on a problem as threads.

From this graph, what do you notice about distributed memory parallelism as compared to shared memory parallelism? Why do you think this is? What advantages do we get for distributed memory parallelism?

3.4 Questions

1. In Shared memory, what problem size (number of segments) does Area Under Curve (AUC) need for a distinct shape to appear when scaling over the number of threads?
2. How does AUC scale over threads? Is it how you expected it to scale?
3. What's the name of the curve that AUC's scaling over threads or processes takes?
   (a) What shape would you expect if you varied the problem size proportionally to the number of processes or threads?
   (b) Try it, run OMP or MPI stats again with –pfunc linear and –tag 'weak scaling' Don't forget to use PlotKit.
   (c) What shape did you get?
4. Given your scaling information, which version of AUC is best in which situation? For instance, Which runs most quickly when only one core is available? What

about when several cores are available, but each core is on a different compute node? Explore, and explain your findings.

3.5 Assessing Student Understanding

A student that has or can quickly generate performance graphs for each parallel version of AUC and can answer all of the questions above has gained full understanding of this lab.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] S. Leeman-Munk, A. Weeden, A. F. Gibbon, B. Johnson-Stalhut, M. Edlefsen, G. Schuerger, D. Joiner, and C. Peck. SIGCSE Milwaukee, 2010.