Visualize Numerical Integration

CSInParallel Project

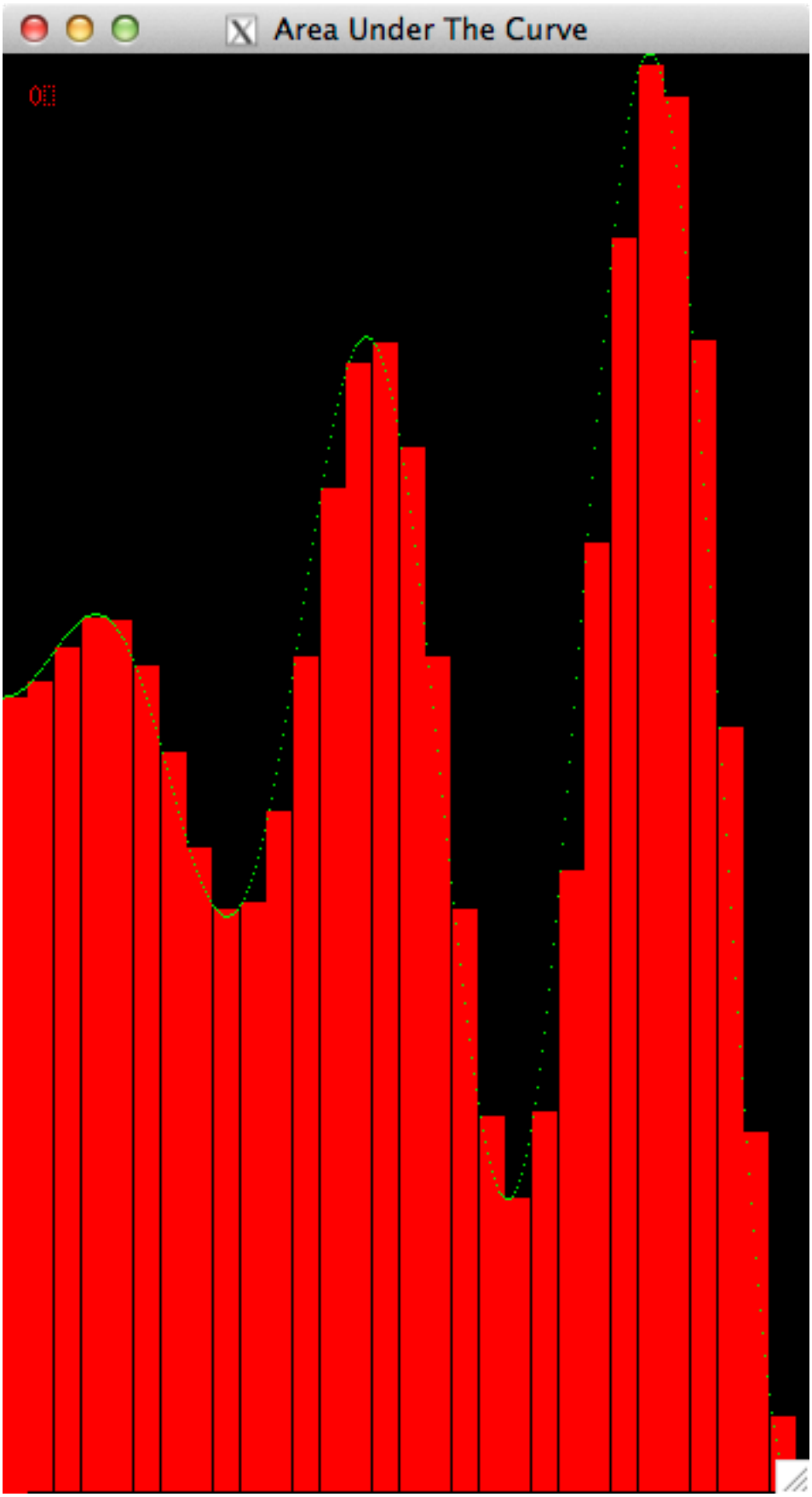
July 21, 2014

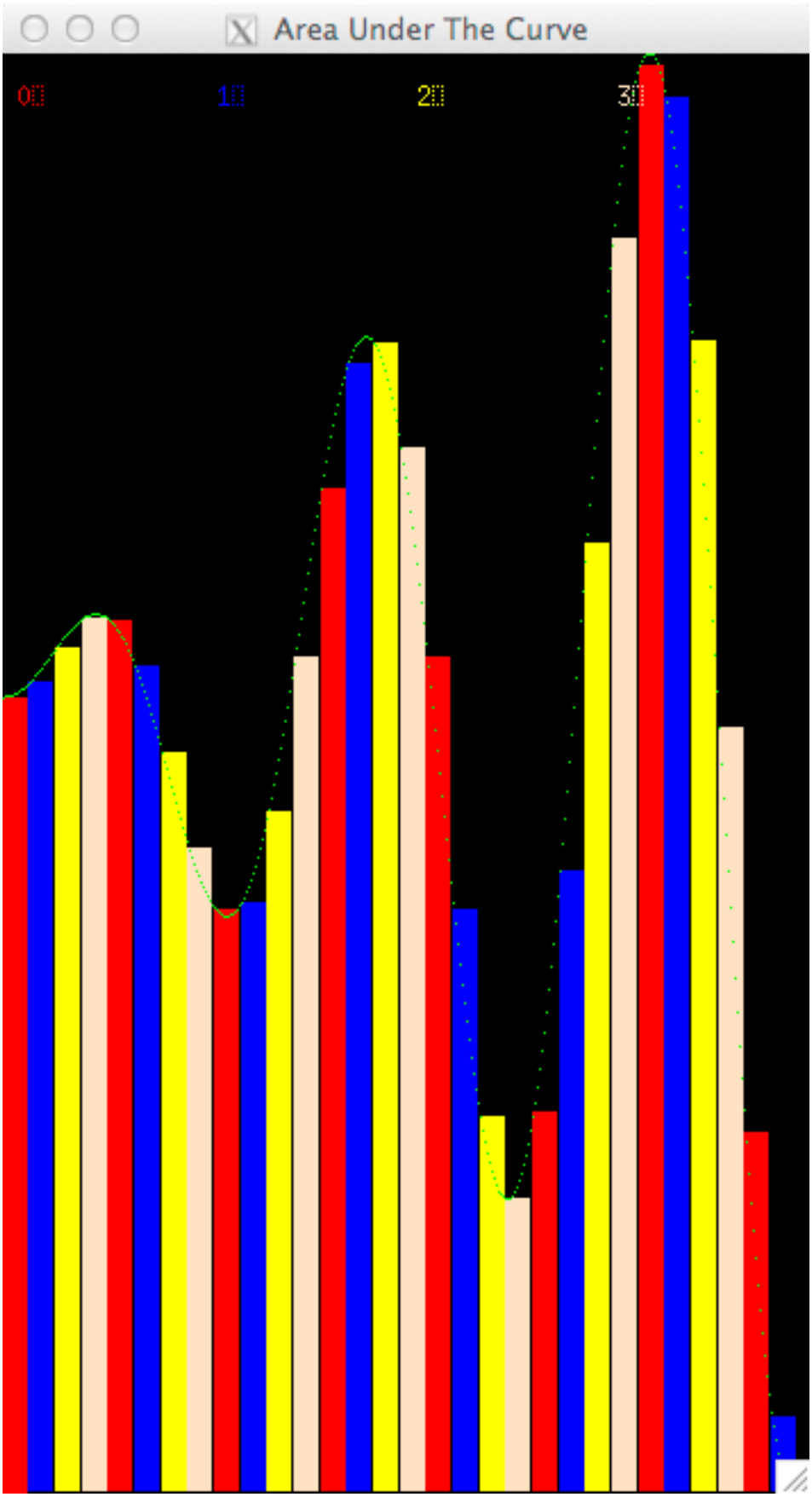
CONTENTS

1 The Numerical Integration Problem **5**
1.1 Programming Numerical Integration 5

2 The Code, Platforms, and Executing a Serial Version **7**
2.1 Parallel Computing Platforms 7
2.2 The Code and Scripts 7
2.3 Notes about building on your installation 8
2.4 Building and Executing the Serial Version 8

3 Parallel Displays of the Area Under the Curve **11**
3.1 OpenMP Versions: Shared Memory multicore with threads 11
3.2 MPI Examples: distributed message passing 14
3.3 Hybrid: MPI plus OpenMP 16


Area Under The Curve


Area Under The Curve
0
1
2
3

Prologue

This is an activity that steps you through some code we have prepared to help you see how several parallel implementations that solve the numerical integration problem might split the work among processing units.

Prerequisites

- Reading the CSinParallel material entitled *Concept: The Data Decomposition Pattern* will provide useful background information regarding the parallel pattern illustrated with these visualizations.
- **You will need to be able to run these programs on a unix machine with X11 installed and with at least one of the following**
  - MPI installed on a single computer or cluster (either MPICH2 or OpenMPI)
  - gcc with OpenMP (most versions of gcc have this enabled already) and a multicore computer
- Some knowledge of building C programs using make may be useful.
- Some understanding of numerical integration: approximating the integral as the area under a curve with left and right boundaries using the 'rectangle rule'.

Nomenclature

A **Processing Unit** is an element of software that can execute instructions on hardware. On a multicore computer, this would be a *thread* running on a core of the multicore chip. On a cluster of computers, this would be a *process* running on one of the computers. On a co-processor such as a Graphics Processing Unit (GPU), this would be a *thread* running on one of its many cores.

A program that uses only one processing unit is referred to as a *serial* or *sequential* solution. You are likely most familiar with these types of programs.

Code You will Need

You should download `VisArea.tgz` to obtain the code that you will run for the examples shown in the following sections of this reading.

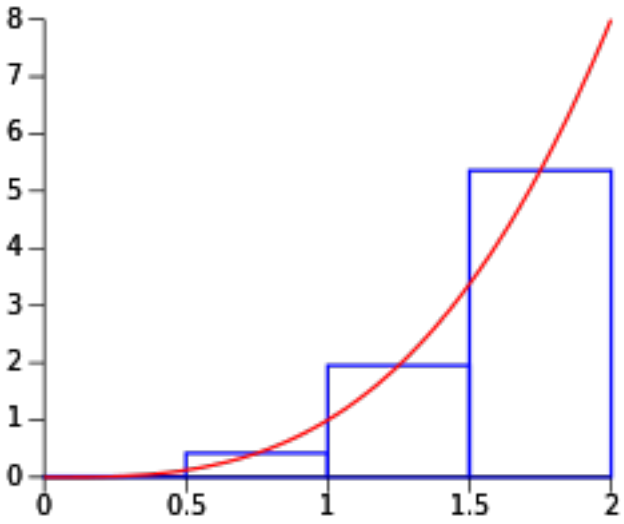
CHAPTER

ONE

THE NUMERICAL INTEGRATION PROBLEM

A famous mathematical problem that is easy to compute is to approximate the numerical value of a definite integral of a function from a to b as the area under the curve of that function using the rectangle method.

Given a function, we can approximate its definite integral over the interval (a, b) by adding the values of the areas of contiguous small rectangles whose heights are the value of the function along the interval. Zoomed in on a small portion of a curve for a function, these rectangles look like this:



This image is courtesy of Qref on Wikimedia commons.

1.1 Programming Numerical Integration

Programs written for solving this problem on a given function are fairly straightforward and involve determining the number of rectangles desired over a given interval and computing the value of the function for each rectangle, adding it to an overall sum for the value of the integral.

For a sequential implementation of such a program, the run time is on the order of the number of rectangles. To obtain an accurate result, a large number of rectangles should be used.

The sample code we provide in `area.tgz` contains a program that will enable you to visualize several ways that we can make numerical integration run faster by splitting the work of computing the area of each of the rectangles across

multiple processing units. In these examples, we illustrate the types of *data decomposition patterns* that can be used. In this problem, the data to be computed is the area of each rectangle and we can think of those rectangles as lining up linearly along the x axis as shown above. In the following sections we will walk you through the execution of this code for several different data decompositions of this linear collection of rectangles, using a simple function.

The code you will use is not necessarily meant for you to examine, but instead to run with different hardware and software combinations and see how the decomposition took place. We built the code to execute in real time, saving the results of which processing unit computed which rectangle. We then graphically display the results by replaying the computations, drawing the rectangles with a time delay so you can visualize what happened.

Continue to the next chapter to begin your journey!

CHAPTER

TWO

THE CODE, PLATFORMS, AND EXECUTING A SERIAL VERSION

2.1 Parallel Computing Platforms

The code we provide will compile and run with the following combinations of platforms. Any computer you use will need linux or unix with X11 (this is how we coded the visualizations). We have examples that use OpenMP alone, MPI alone, and a combination of MPI and OpenMP (sometimes called hybrid solutions).

The OpenMP examples will run on:

- A single multicore machine with an OpenMP-compatible C compiler

The MPI examples will run on:

- A single multicore computer *or* a cluster with MPICH installed and MPE installed (MPE is a visualization library for MPI based on X11 that comes with MPICH).
- A single multicore computer *or* a cluster with OpenMPI installed and MPE installed (with OpenMPI, you need to install MPE separately).

The MPI + OpenMP examples will require a single computer or a cluster with MPI, MPE, and an OpenMP compiler all installed.

2.2 The Code and Scripts

You will need to set up the code. First get and uncompress the `VisArea.tgz` file like so (these are linux commands):

```
tar -zxvf VisArea.tgz
ls -R -1
```

You should get some output like this:

```
.:
area
Makefile

./area:
area.c
area.c-mpi
area.h
colors.h
makeExecutable
```

```
Makefile
MPEarea.h
MPEcolors.h
Readme
run_mpi_chunksOfOne
run_mpi_chunksOfOne_openmp_equalChunks
run_mpi_equalChunks
run_mpi-openmp_chunksOfOneDynamic
run_mpi-openmp_chunksOfOneStatic
run_mpi-openmp_equalChunks
run_openmp_chunksOfOneDynamic
run_openmp_chunksOfOneStatic
run_openmp_equalChunks
run_serial
structs.h
Xarea.h
```

Note that there is a directory called **area** with its own Makefile and a global Makefile. You will work inside the directory called **area**. From there, we have set up shell scripts to compile and run various types of data decomposition with different hardware and software compinations. Each of these scripts begins with the prefix *run_*.

2.3 Notes about building on your installation

You might need to change the path to your X libraries. The place to do that is to edit the LDFLAGS in the the global Makefile (the one outside of the source code folder called *area*). You can try the first script shown below to test whether this will be a problem for you.

2.4 Building and Executing the Serial Version

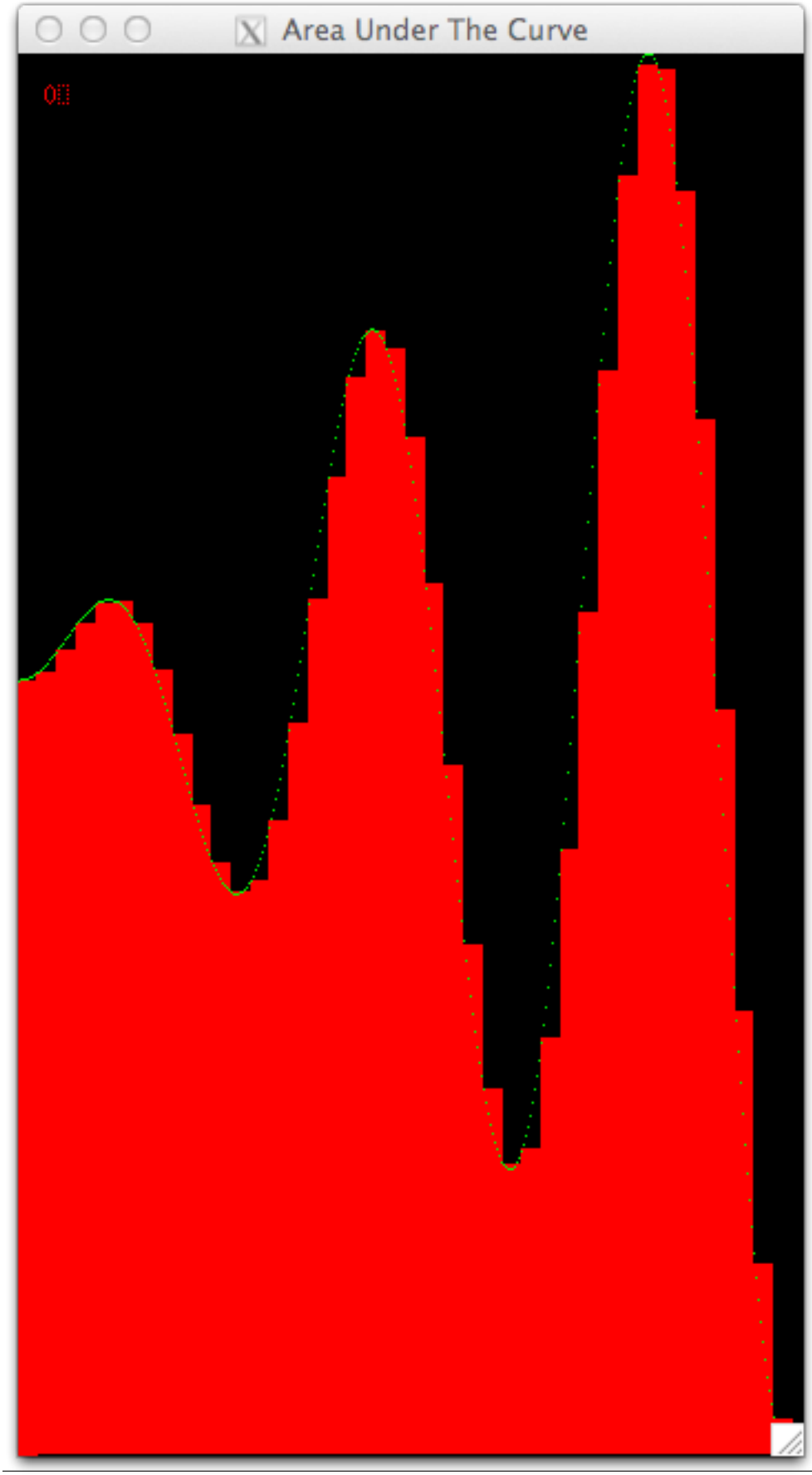
You will need to be logged directly into the computer with the code, MPI, OpenMP, and X11 installed in order to see the X11 visualization that the code produces. You can also have an X11 client installed on a remote machine in which you use ‘‘ssh -X ‘‘ to remotely log in to the server with the parallel software and this code installed.

Let's start by trying an example that runs serially, or sequentially, without any parallelization. The script for this is called `run_serial`. You can compile the code and run it like this.

```
cd area
./run_serial
```

We will run all the rest of the example scripts from the **area** directory.

An X window displaying how the code executes the computation of the area of each rectangle should appear. When it completes, it should look like this:


Area Under The Curve
0

The default curve we use is shown (see below for some more options). Each rectangle is drawn with a single color because one processing unit computed each rectangle, one after another. In other examples, colors will be used to show which processing unit computed which rectangle.

To close the window, you can type the 'q' key if the focus is still in the window. Otherwise, place focus back in the terminal window where you ran the script and type control-c.

We chose some default settings, which you can change to see what happens. For all scripts, you can use the `-?` option as follows to see what your options are:

```
./run_serial -?
```

You should get some output like this:

```
./run_serial: illegal option -- ?
usage: ./run_serial options

This script runs the visualization of computing the area under the curve serially.

OPTIONS:
-t  Throttle time in microseconds
-n  Number of rectangles
-f  Function kind (1, 2 or 3, default if not specified)
-r  Righthand boundary of the curve
```

The throttle time will change how quickly the rectangles will be displayed. These visualizations are designed to be 'played' so that you can see what assignments were made. You can change this time to help you see what is happening.

You can change the number of rectangles under the curve. Making this smaller or larger might help you envision what is happening in each of the situations we describe in the next few sections. For eaxmple, if you try this:

```
./run_serial -n 12
```

You can see how this small amount of rectangles does not approximate this curve very well.

However, if you use too many rectangles, it is very hard to see each one. Really high numbers of rectangles will mean that it takes less than one pixel to draw them; this is a situation you want to avoid. About 60 or so is likely the most you should opt for with `-n`.

The `-f` option will let you pick other curves to display.

- Option *1* is the straight-line function $f(x) = x$.
- Option *2* is the positive quarter circle, $f(x) = +\sqrt{r^2 - x^2}, 0 \leq x \leq r$
- Option *3* is half the period of the sine function, $f(x) = \sin(x), 0 \leq x \leq \pi$
- Option *4*, the default in these example scripts, is the curve $f(x) = x\sin(0.05x)$

The `-r` option can only be changed with the `-f` option *1* or *4*. With `-f 1`, values between 400 and 1000 make the most sense to use for `-r` (any larger and the window gets too large; any smaller and the window is too small and hard to see.) With `-f 4`, the default shown in these scripts is 350 for the right-hand value of x in the function. We recommend going now higher than 600 or so, unless you have a really large monitor.

Now that you know the options, let's you on and try parallel versions of this code.

CHAPTER

THREE

PARALLEL DISPLAYS OF THE AREA UNDER THE CURVE

Here we will run the code for various types of data decomposition patterns using OpenMP, MPI, or a combination of MPI and OpenMP.

3.1 OpenMP Versions: Shared Memory multicore with threads

In code that uses OpenMP, the processing unit executing the computation of the area of each rectangle is a *thread* executing on a core of a shred-memory multicore computer. Let's try a few examples of how OpenMP can map threads to rectangles to be computed. We consider these rectangles to be computed from left to right, forming a linear set of data values to be computed.

In these visualizations, we run the code in real time as it would get run in openMP, keeping information about which thread computed which rectangle. The program then displays back which thread computed each rectangle by playing back the recorded information and coloring the rectangles with different color for each thread.

3.1.1 Equal chunks of rectangles

Try running this script like this on the command line, in the **area** directory:

```
./run_openmp_equalChunks
```

You should see that four threads each computed consecutive equal-sized chunks of rectangles (we sometimes also call this division of work the 'blocks' decomposition). This is a well-used pattern of how to split the work, or decompose the problem, among the threads. This same pattern is also used frequently in distributed, message passing situations.

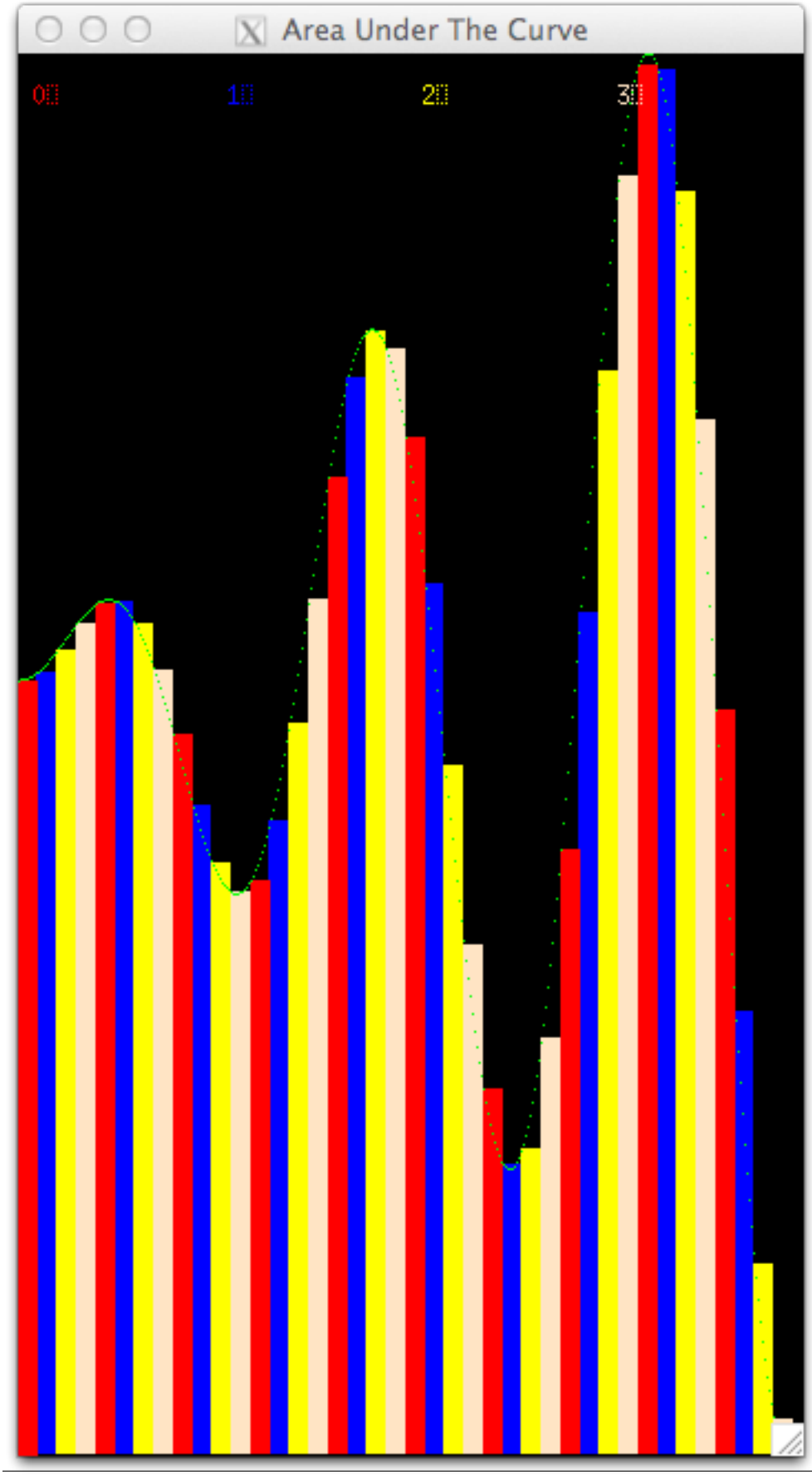
Note: Though the playback in these visualizations may appear to be serial, the threads did run concurrently. We unfortunately have no way of displaying rectangles on the screen in parallel!

3.1.2 Chunks of one, statically assigned

Rather than equal chunks where the size of a chunk that a thread works on is n/p, where p is the number of threads and n is the number of rectangles, another possible distribution of work is to have each thread compute one rectangle of the first p rectangles, then go on a compute one of the next group of p rectangles, and so on. If the thread to rectangle assignment is done statically ahead of time, it would look like the results of running this script:

```
./run_openmp_chunksOfOneStatic
```

It should look like this when it completes:


Area Under The Curve
0
1
2
3

Note the legend at the top of the window tells you the color for each thread, which are numbered from 0 through 3 for this case. Study this to be certain you understand the assignment of threads to rectangles.

OpenMP can decide for itself which threads will work on which computational elements. This is know as dynamic scheduling. Try this script for that and see what you observe. This could be different on different machines and could change each time you run it.

```
./run_openmp_chunksOfOneDynamic
```

3.2 MPI Examples: distributed message passing

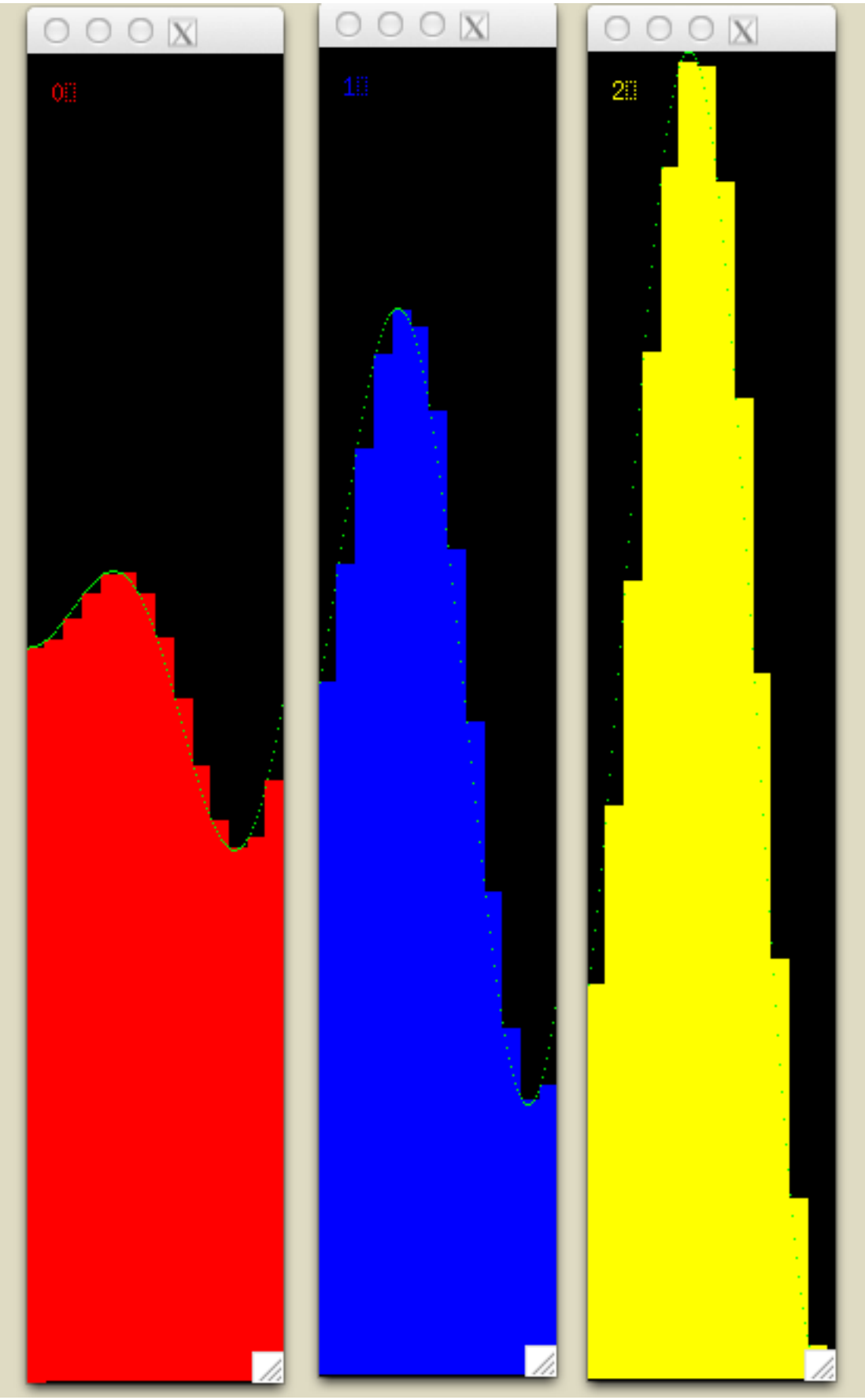
In MPI, the processing units that run in parallel are *processes*, which are either running on the same machine or more likely across a cluster of machines, where each machine starts its own process.

3.2.1 Equal chunks of rectangles

In the case of a cluster of machines, there is one data decomposition pattern that makes the most sense, and that is to partition the work into equal chunks. Try this script to visualize that case:

```
./run_mpi_equalChunks
```

You should see multiple windows displayed, one for each process, with rectangles assigned and computed in the order that they appear inside the function. It should look like this:


0
1
2

3.2.2 Chunks of one, statically assigned

Another possible assignment, especially for a single machine running MPI processes, is to have the processes compute one rectangle each, similar to the single chunk case for OpenMP. Run this and you will see it looks the same as the mapping for OpenMP:

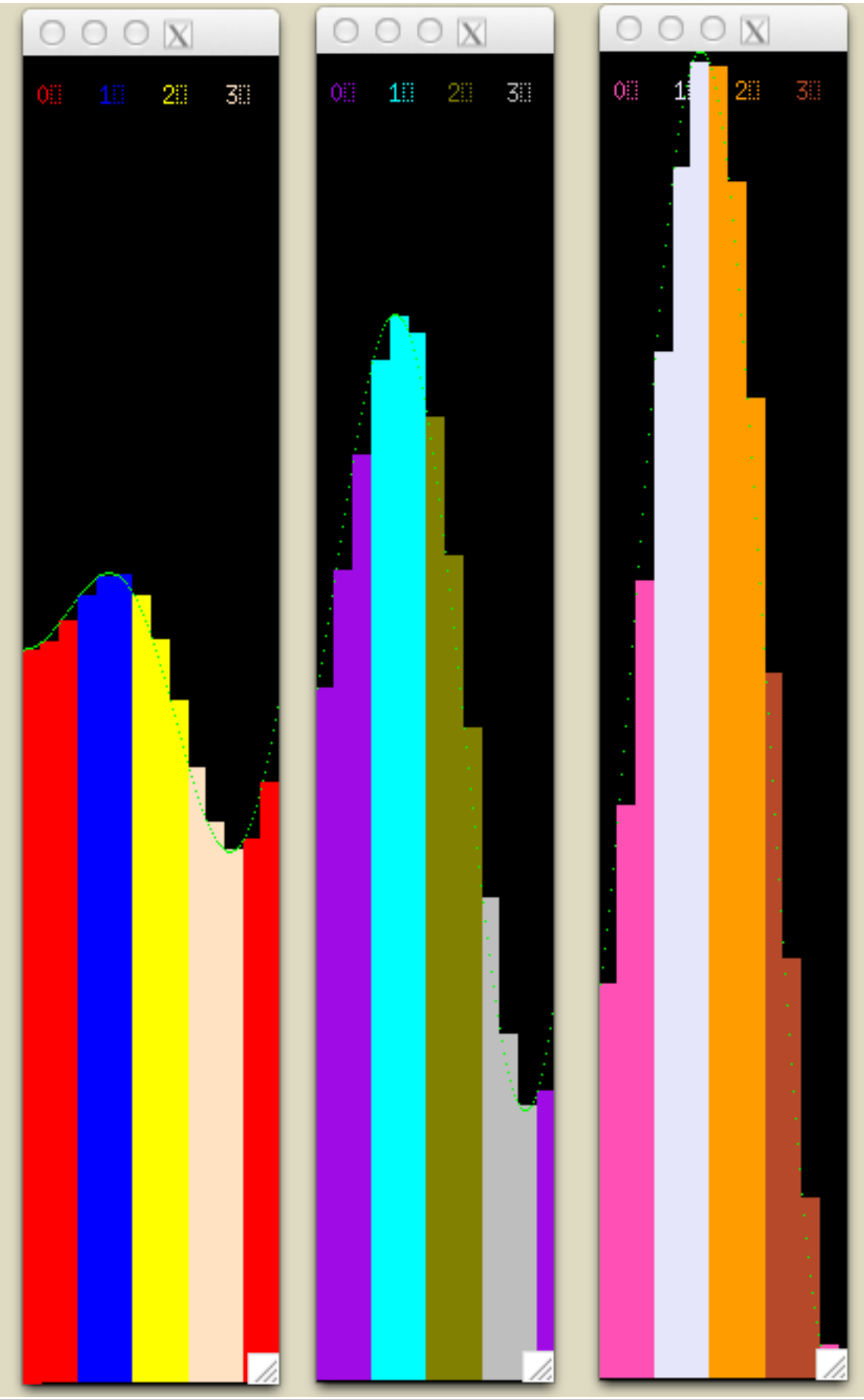
```
./run_mpi_chunksOfOne
```

3.3 Hybrid: MPI plus OpenMP

When using MPI across a cluster of machines, if those machines have multicore processors, then each of them can execute code using OpenMP. The following example script runs this case, where each MPI process gets an equal chunk of the original rectangles, and those chunks are further divided by OpenMP threads.

```
./run_mpi-openmp_equalChunks
```

The result for this decomposition looks like this:


0
1
2
3
0
1
2
3
0
1
2
3

There are other possible hybrid combinations- please experiment with some of the other scripts beginning with `run_` that we haven't mentioned here yet to see if you can determine these more complicated patterns.