Teaching Parallelism with River Trail

Jaswanth Sreeram  Stephan Herhut  Richard L. Hudson  Tatiana Shpeisman
Intel Labs
jaswanth.sreeram, stephan.a.herhut, rick.hudson, tatiana.shpeisman@intel.com

Abstract
Parallel hardware is today's reality and parallel programming models exist for most mainstream languages. Surprisingly JavaScript, the mother tongue of the web, is still stuck in its sequential past. JavaScript's unique programming model, the web's safety and security requirements, and the expectations of its programmers for determinism have impeded parallelization until now.

Parallel JavaScript (code named River Trail) is a set of JavaScript extensions and an API that provides safe, deterministic parallelism to web applications while fitting within the event-driven, security oriented world of JavaScript applications. River Trail allows programmers to utilize available hardware parallelism on client machines - from SIMD units to multiple cores, using high-level parallel programming patterns.

In this paper we describe River Trail, show how it can be used to provide a gentle introduction to parallelism using our self-contained hands-on tutorial. At the end of the tutorial the student will have mastered the basic concepts of parallel programming and use this knowledge to build a realistic parallel HTML5 web application in River Trail.

1. Introduction
As increasing hardware performance via higher processor frequencies has hit the power wall, hardware manufacturers have switched to adding more power-efficient parallel hardware. Modern processors feature multiple cores, wide vector units, and often an integrated GPU suitable for general purpose computing as well as graphics. To take advantage of this new hardware we need to bring up a new generation of programmers who are fully proficient in parallel programming.

Classical parallel programming models enable software developers to extract maximum performance from the underlying hardware but come with a host of hard to teach concepts such as threads, locks, memory models, data races, deadlock, and so forth. It is not surprising that the majority of software developers are ill equipped to write parallel codes and the majority of software remains sequential.

In this paper we discuss teaching parallel programming in the context of River Trail, a data-parallel extension to JavaScript. We chose JavaScript since it is the programming language at the core of HTML5 web applications. River Trail has been designed to enable parallel programming for web developers - the audience that is mostly unfamiliar with parallel programming and often even lacks formal computer science education. As such, it aims to introduce parallelism gently - concentrating on the fundamental parallel properties while hiding complicated underlying mechanics in the River Trail implementation. River Trail stays within the boundaries of the familiar programming paradigm of JavaScript preserving all the important properties of this programming language. It is as safe, portable, and suitable for rapid prototyping as JavaScript itself. Like JavaScript, River Trail is deterministic and results from River Trail produce the same result, modulo floating point anomalies, regardless of the order realized by the underlying implementation. River Trail has been implemented as an add-on to the Firefox browser and is available as open source software. It is a prototype implementation of Parallel JavaScript which is currently a strawman proposal before the EcmaScript standards committee [4].

At the core of River Trail lies a ParallelArray type with six fundamental methods - map, combine, reduce, scan, filter and scatter. To write a parallel program, the programmers specify the parallel array that contains the data, the method that determines the parallel processing pattern, and the elemental function (kernel) that is applied to the array elements in parallel. The code below shows a simple River Trail program that multiplies each array element in \( a \) by the value of a global variable \( g \).

```javascript
var b = a.map(function(v) {return g*v;});
```

The River Trail programming model is designed to eliminate many factors that complicate teaching parallel programming:

- River Trail allows programmers to easily express a high-level programming pattern without having to spell out how this pattern maps to various types of parallel hardware, such as multiple cores or vector instructions.
- River Trail ensures deterministic race-free execution through read-only shared state. In the above example, the elemental function can read the global variable \( g \). An attempt to modify this variable, however, would result in an exception.
- River Trail has low barrier of entry. It is implemented as part of a browser and requires only tools typical to JavaScript programming such as an external editor.
- River Trail programming is highly interactive. Because the tutorial leverages the dynamic nature of JavaScript, programmers get immediate feedback on the correctness of their program. With the proper choice of examples, the performance effect of parallelism is also easily observable (as for example, a better visual experience due to higher frame rate).

For these reasons River Trail is well suited for teaching entry level parallel programming.

The remainder of this paper consists of excerpts from the tutorial and is organized as follows:

- The first module of the tutorial is presented in Section 2. It introduces the River Trail API along with short code examples illustrating the syntax and semantics of the API components.
The second module is presented in Section 3. In this module, the participants will start with a skeleton of a video effects application that includes sequential versions of various filters such as brightening, desaturation, sepia toning, color adjustment, edge detection, sharpening, stereoscopic 3D rendering, and color-based face detection. Participants will use the constructs introduced in the first module to implement parallel versions of these filters and observe the performance impact.

2. Module I: The River Trail API

In this module, participants learn the River Trail API by writing small examples using the browser-based River Trail environment [3]. This environment allows participants to experiment with and quickly prototype River Trail programs without needing to install River Trail. However, the programs they write will be executed sequentially. Participants who install River Trail as a browser plugin will be able to run programs utilizing the full range of parallelism available in their hardware from SIMD units to multiple cores.

The central component of River Trail is the ParallelArray type. ParallelArray objects are essentially ordered collections of scalar values. They have a few important properties introduced below:

2.1 Multi-Dimensional and Uniform

ParallelArrays can represent multi-dimensional collections of scalars. All ParallelArrays have a shape that succinctly describes its dimensionality and size. The shape of a ParallelArray is represented as an array of numbers whose $i^{th}$ element corresponds to the number of elements in the $i^{th}$ dimension. A 4x5 matrix of numbers would be represented as a ParallelArray whose shape is [4, 5]. Similarly, a 2D image in which each pixel has RGBA values can be represented as a ParallelArray object with the shape [height, width, 4].

Multi-dimensional ParallelArrays are required to be uniform (also called rectangular). That is, the length of all inner arrays in a particular dimension must be the same. For example, 

```
< <0, 1>, <2, 3>, <4, 5>>
```

would be disallowed since it is not uniform.

2.2 Immutable

Once created ParallelArrays are immutable. Invoking ParallelArray methods typically produces new freshly minted ParallelArrays.

2.3 Constructors

ParallelArrays can be created in a variety of different ways as the code below illustrates. The first few examples are self-explanatory and provide the expected functionality.

In addition to the first four obvious mechanisms line 20 shows how to create ParallelArray objects using comprehension. Comprehension requires the length of the ParallelArray (3) and an elemental function. As the name suggests this elemental function is invoked for each index i, i.e., $i = 0$, $i = 1$ and $i = 2$. Each function invocation returns a 2-element array consisting of $i$ and $i + 1$. After the elemental function has been invoked for each $i$, River Trail aggregates the results into the freshly minted ParallelArray $< < 0, 1 >, < 1, 2 >, < 2, 3 >>$. Note that the order in which the elemental function is called is irrelevant. We will learn more about writing these functions in the next section.

The comprehension constructor can also create multi-dimensional array as shown in line 26. We simply need to supply a shape vector instead of a length and an elemental function that takes a vector index as an argument. In this case, the shape we specify is [3, 2]

and the elemental function will be invoked with a 2-element vector argument $iv$.

Finally, line 35 shows how to create a new ParallelArray object directly from an HTML5 canvas object. The canvas object is used for drawing 2D shapes, pictures and video on a webpage. We will see how this is useful later when we build a video web app.

```
1 // Create an empty ParallelArray
2 var pa0 = new ParallelArray(); // pa0 = []
3 // Create a ParallelArray with a nested JS array.
4 // Note the inner arrays are also ParallelArrays
5 var pa1 = new ParallelArray([[0, 1], [2, 3], [4, 5]]); // pa1 = < <0, 1>, <2, 3>, <4, 5> >
6 // Create a ParallelArray from another ParallelArray
7 var pa2 = new ParallelArray(pa1);
8 // pa2 = < <0, 1>, <2, 3>, <4, 5> >
9 // Create a ParallelArray from several other ParallelArrays
10 var pa3 = new ParallelArray(<0, 1>, <2, 3>);
11 // pa3 = < <0, 1>, <2, 3> >
12 // Create a one-dimensional ParallelArray of length 3 using the comprehension constructor
13 var pa4 = new ParallelArray(3, function(i){return [i, i+1];});
14 // pa4 = < [0, 1], [1, 2], [2, 3] >
15 // Create a two-dimensional ParallelArray with shape [3, 2] using comprehension
16 var pa5 = new ParallelArray([3, 2], function(iv){return iv[0] * iv[1];});
17 // pa5 = < [0, 0], [0, 1], [0, 2] >
18 // Create a ParallelArray from canvas. This creates a PA with shape [w, h, 4], with the width and height of the canvas and the RGBA values for each pixel.
19 var pa6 = new ParallelArray(canvas);
20 // pa6 = CanvasPixelArray
```

2.4 ParallelArray Methods

ParallelArray objects created with the above constructors come with several methods to manipulate them. These methods typically produce a new ParallelArray object. One notable exception is the reduce method which produces a scalar value.

2.4.1 Map

The first method discussed is map, probably the most prominent and well-known data-parallel skeleton. The map method expects a function as its first argument that given a single value returns a new value. In the following, we will call such a function an elemental function as it is used to produce the elements for the resulting ParallelArray. The map method computes a new ParallelArray out of an existing ParallelArray by applying the elemental function to each element in the source array and storing the result in the corresponding position in the resulting array. Let us look at a simple example: increment

```
1 var source = new ParallelArray([1, 2, 3, 4, 5]);
2 var plusOne = source.map(function inc(v) {
3     return v+1;
4 });
```

First, we define a new ParallelArray object source that contains the numbers 1 to 5. We then call the map method of our source array with the function inc that adds 1 to its argument. Thus, plusOne
contains the values 2 to 6. The map method is shape preserving so `plusOne` has the same shape as the original array source.

As you may have noticed, the map method does not provide an index to the elemental function it calls. We refer to this as index free computations. Not using an index has the advantage that no indexing errors can be made. However, this added simplicity comes at the cost of expressiveness. With map, one cannot inspect neighboring values, as commonly required for convolution style codes.

### 2.4.2 Combine

The combine method addresses this issue. Similar to map, combine can compute a new ParallelArray by inspecting an existing ParallelArrays element. Unlike map, the elemental function of combine is provided with access to the current index in the source array, along with a reference to the source array itself (via the implicit variable `this`).

As an example, consider the following code that reverses the order of elements in a ParallelArray:

```
var source = new ParallelArray([1,2,3,4,5]);
var reverse = source.combine(function rev(iv) {
  return this.get(this.length-iv[0]-1); });
```

The elemental function `rev` exploits the access to the index to compute the reversed index in the source array. Note that computations are driven by the index position in the result, not the index position in the source. We therefore use the expression `this.length-iv[0]-1` to compute the source index of the reversed element for position `i` in the result array. This code makes use of the length property of the ParallelArray object that, similar to JavaScript’s Array object, gives the number of elements in the array.

It is important to note here that the index `i` is not a scalar value but actually a vector of indices. In the above example, we therefore have to use `i[0]` in the computation of the source index. So far, all our examples computed on vectors and the use of an index vector in combine was of no help. However, ParallelArray objects in River Trail can have multiple dimensions. The map method always operates on the outermost dimension only, i.e., on the dimension that corresponds to the first element of the shape vector. With combine, the programmer can choose how deep to traverse. For this, an optional first argument to the combine method is used. As an example, let us generalize the above reverse operation into a reverse operation on matrices:

```
var source = new ParallelArray([[1,2,3,4,5],[6,7,8,9,10]]);
var reverse = source.combine(2, function rev(iv) {
  return this.get(this.length-iv[0]-1, iv[1]-iv[1]-1); });
```

We use a depth of 2 and, consequently, the index vector `iv` passed to the elemental function contains two indices, corresponding to the two outermost dimensions of the source array. We also use the `getShape` method, which is the multi-dimensional counterpart to length: It returns a vector that gives the length for each dimension of a ParallelArray. `this.getShape()[0]-iv[0]-1` computes the index at the reversed position within the source array along the first dimension. Note here that `get` also accepts an index vector as argument.

### 2.4.3 Reduce

So far we have concentrated on parallel patterns that produce a new array out of an existing array. The reduce method implements a further important parallel pattern: reduction operations. As the name suggest, a reduction operation reduces the elements from an array to a single result. A good example to start with is computing the sum of all elements of an array:

```
var source = new ParallelArray([1,2,3,4,5]);
var sum = source.reduce(function plus(a,b) {
  return a+b; });
```

As the example shows, reduce expects an elemental function as the first argument that, given two values as parameters, produces a new value as its result. In our example, we use plus which adds two values. A reduction over plus then defines the sum operation.

Note that the reduction may be computed in any order so to ensure deterministic results River Trail requires that the elemental function be commutative and associative. Since the River Trail runtime does not check this property results might differ between calls.

### 2.4.4 Scan

The reduce method reduces an array into a single value. For some use cases, it can be interesting to preserve the intermediate results. One commonly used example is the prefix sum operation that, given a vector of numbers, computes another vector of numbers such that each position contains the sum of all elements of the source vector up to that position. To implement this parallel pattern, River Trail’s ParallelArray features the scan method:

```
var source = new ParallelArray([1,2,3,4,5]);
var psum = source.scan(function plus(a,b) {
  return a+b; });
```

The same rules of parallel execution that apply to reduce also apply to scan. Although less obvious, scan can be computed in parallel by reordering the reduction steps. Therefore, as with reduce we only guarantee a deterministic result if the elemental function is commutative and associative.

### 2.4.5 Scatter

So far we have seen map and combine used to produce new arrays out of existing arrays. However, both methods are driven by result indices, i.e., they define for each index position in the result how it is to be computed. Sometimes, this mapping is difficult to specify or costly to compute. Instead, it is preferable to specify for a certain source index where it should be stored in the result array. This pattern is supported by the scatter method:

```
var source = new ParallelArray([1,2,3,4,5]);
var reorder = source.scatter([4,0,3,1,2]);
```

We again first compute our source array source. In a second step, we apply the scatter method with a single argument: the scatter vector `[4,0,3,1,2]`. Thereby, we specify that the first element of source is to become the fifth element of the result (indexing starts with 0), the second value in source becomes the first in the result and so on. Overall, the above example produces the array `<2,4,5,3,1>`. The actual tutorial would also present how conflicts are resolved and how other corner cases, such as missing indices, are dealt with.

### 2.4.6 Filter

The filter method takes an elemental function that is given an index argument and returns a truth value that indicates whether the element at that index in the source array should be included in the output. In the following example, we use this method to remove all odd numbers from a ParallelArray:

```
var source = new ParallelArray([1,2,3,4,5]);
var even = source.filter(function even(iv) {
  return (this.get(iv) % 2) == 0; });
```

As before, we first produce a source array containing the values one to five. Next, we apply the filter method using even as elemental argument.
function, which returns true for all even elements. Thus, we remove all odd elements from the source array, producing even containing < 2, 4 >.

3. Parallelizing video filters

The participants are instructed to download and install River Trail according to the instructions [2] provided. The River Trail distribution contains a skeleton for a video processing application that participants use in the rest of this tutorial. Video processing was chosen for several reasons - it is compute-intensive, it is a realistic HTML5 workload and highlights the performance benefit of parallelism with clear “visual impact”.

The first step is to load the index.html file for this skeleton in the browser. They will see the default screen for the application skeleton as in Figure 6.

The large box in the center is a Canvas that is used for rendering the video output. The video input is either a HTML5 video stream embedded in a video tag or live video captured by a webcam. On the right of the screen one sees the various filters that can applied to the input video stream - sepia toning, lightening, desaturation etc. To switch to webcam video, click the “Webcam” toggle in the top-left corner. The sequential JavaScript versions of the filters on the right are already implemented and the participants are given time to explore what each of these effects do. They will then begin implementing the “parallel” versions of these filters using River Trail.

3.1 Manipulating pixels on Canvas

The participants open up main.js in a code editor. This file implements all the functionality in this web application except the filters themselves. When the HTML page is opened in a browser, the doLoad() function is called after the body of the page has been loaded. This function sets up the drawing contexts, initializes the input video stream - sepia toning, lightening, desaturation etc. To switch to webcam video, click the “Webcam” toggle in the top-left corner. The sequential JavaScript versions of the filters on the right are already implemented and the participants are given time to explore what each of these effects do. They will then begin implementing the “parallel” versions of these filters using River Trail.

After this video frame is drawn to canvas, we need to capture the pixels so that we can apply the various filters on them. This is done by calling getImageData() on the context containing the image to be captured.

Now we have an ImageData object called frame. The data attribute of this object contains the pixel information and the “width” and “height” attributes contain the dimensions of the image we have captured. The data attribute contains RGBA values for each pixel in a row-major format. That is, for a frame with h rows of pixels and w columns, it contains a 1-dimensional array of length w * h + 4 as shown in Figure 1.

For example to get the color values of a pixel in the 100th row and 50th column in the image, we would do:

Remember from the previous snippet that the frame.data buffer contains color values as a linear sequence of rgba values. The for loop in line 823 iterates over this buffer and for each pixel it reads the red, green and blue values (which are in pix[i], pix[i+1] and pix[i+2] respectively). It computes a weighted average of these colors to produce the new red, green, blue values for that pixel. It then writes them back into the “data” buffer. When the loop is finished, we have replaced the RGB values for all the pixels with their sepia-toned values and we can now write the image back into the output context ctx with the putImageData() method. The result should look like the image in Figure 8.

3.3 Can we make this parallel?

Looking closely at the sepia_sequential function above, we notice that each pixel can be processed independently of all other pixels as its new RGB values depend only on its current RGB values. And each iteration of the for loop does not produce or consume side-effects. This makes it easy to parallelize this operation with River Trail. Recall that the ParallelArray type has a constructor that takes a canvas object as an argument and returns a freshly minted ParallelArray object containing the pixel data.

This creates a 3-dimensional ParallelArray pa with shape [h, w, 4] as illustrated in Figure 2.
0, 0, 0

h

w

Figure 2: The structure of a ParallelArray object instantiated from a Canvas object

So for canvas pixel at coordinates \((x, y)\), \(pa.get(x, y, 0)\) will produce the red value, \(pa.get(x, y, 1)\) the green value and \(pa.get(x, y, 2)\) the blue value. The input ParallelArray that is given to the filter(s) (line 253, main.js):

```javascript
if (execution_mode === "parallel") {
    stage_output = new ParallelArray(input_canvas);
    w = input_canvas.width; h = input_canvas.height;
}
```

stage_input and stage_output are ParallelArrays that contain the input and output pixel data for each filtering “stage”. Now lets look at the code that creates the filters to be applied (line 271, main.js):

```javascript
if (execution_mode === "parallel") {
    switch(filterName) {
        case "sepia":
            stage_output = new ParallelArray([h, w],
                kernelName, stage_input);
            break;
        ...

    }
```

This code block is wrapped in a for loop that iterates over all the available filters. To implement a particular filter, we add code to produce a new ParallelArray object containing the transformed pixel data and assign it to stage_output. So for example, for the sepia filter, we would write:

```javascript
var pa = new ParallelArray(shape_vector,
    elemental_function, arg1, arg2...);
```

where elemental_function is a JavaScript function that produces the value of an element at a particular index in pa. Recall that the input to our filter stage_input is a \([h, w, 4]\) shaped ParallelArray. You can think of it as a two-dimensional ParallelArray with shape \([h, w]\) in which each element (which corresponds to a single pixel) is itself a ParallelArray of shape \([4]\). The output ParallelArray we will produce will have this same shape - we will produce a new ParallelArray of shape \([h, w]\) in which each element has a shape of \([4]\), thereby making the ParallelArray have a final shape of \([h, w, 4]\). Modify line 275 to this:

```javascript
    case "sepia":
        stage_output = new ParallelArray([h, w],
            kernelName, stage_input);
        break;
```

Now all we have to do above is produce a new ParallelArray object on the right-hand-side of the statement above. We can produce this new ParallelArray in one of two ways - by using the powerful ParallelArray constructor or by using the combine method. Let us look at the constructor approach first. Recall the the comprehension constructor has the following form:

```javascript
var pa = new ParallelArray(shape_vector,
    elemental_function, arg1, arg2...);
```

The first argument \([h, w]\) specifies the shape of the new ParallelArray we want to create. kernelName is a Function object pointing to the sepia elemental function (that we will talk about in a moment) and stage_input is an argument to this elemental function. This line of code creates a new ParallelArray object of shape \([h, w]\) in which each element is produced by executing the function kernelName. This new ParallelArray is then assigned to stage_output. Finally, we have to create the elemental function that produces the color values for each pixel. As described in Section II, we can think of it as a function that when supplied indices, produces the ParallelArray elements at those indices. Create a function called sepia_parallel in filters.js as follows:

```javascript
function sepia_parallel (indices, frame) {
    // elemental function for sepia
    var i = indices[0];
    var j = indices[1];
    var old_r = frame[i][j][0];
    var old_g = frame[i][j][1];
    var old_b = frame[i][j][2];
    var a = frame[i][j][3];
    var r = old_r*0.393 + old_g*0.769 +
        old_b*0.189;
    var g = old_r*0.349 + old_g*0.686 +
        old_b*0.168;
    var b = old_r*0.272 + old_g*0.534 +
        old_b*0.131;
    return [r, g, b, a];
}
```

In lines 1-7 we grab the indices and read the RGBA values from the input ParallelArray frame. As in the sequential version we mix these colors in lines 9-11 and return a 4-element array consisting of the new color values for the pixel at position \(i, j\). We save this file, switch to the browser, select the “River Trail” toggle on the left of the video panel and play the video. We will see the sepia toning effect as shown in Figure 8.
The River Trail compiler takes the elemental function above and parallelizes its implementation over the iteration space making use of available parallelism both at the SIMD instruction level as well as at the multi-core level. Note that we did not have to create or manage threads, write any non-JavaScript code or deal with race conditions and deadlocks.

### 3.4 Edge Detection and Sharpening

Let’s move on to something a little more complicated - edge detection and sharpening. Edge detection is a common tool used in digital image processing and computer vision that seeks to highlight points in the image where the image brightness changes sharply. To see it in action, we select the edge detection effect and click play.

**Figure 3: A high level look at 2D convolution**

![Figure 3: A high level look at 2D convolution](image)

**Figure 4: Convolution for a single pixel**

There are many diverse approaches to edge detection but we are interested in the most popular 2D discrete convolution based approach. At a high level, discrete convolution on a single pixel in an image involves taking this pixel (shown in dark blue above) and computing the weighted sum of its neighbors that lie within some specific window to produce the output pixel (shown in dark red above). The weights and the window are described by the convolution kernel. This process is repeated for all the pixels to produce the final output of the convolution.

Consider a 5x5 matrix convolved with a 3x3 kernel as shown in Figure 4. For simplicity, we are only interested in the input element highlighted in blue. The weighted sum for this element is: 

\[
(1*2)+(1*3)+(2*0)+(2*0)+(2*1)+(1*3)+(1*3)+(5*0)+(0*3) = 13. 
\]

The value of this element in the output matrix is therefore 13. You can copy this over to the sequential version. And we also need a 4 element array `neighbor_sum` to hold the weighted sum.

The first two lines of the body are the same as the beginning of the parallel implementation. \((m,n)\) is now the position of a pixel in the input ParallelArray frame. The variable `ekernel` is the 5x5 kernel we will be using for convolution (you can copy this over from the sequential version). And we also need a 4 element array `neighbor_sum` to hold the weighted sum.

At this point we have an input frame \((\text{frame})\) and a specific pixel \((m,n)\) which we will call the "input pixel". Now we need to define a “window” of neighboring pixels such that this window is centered at this input pixel. We can define such a window by using a nested loop as follows:

```javascript
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1, 1,2,2,2,1, [1,2,-32,2,1, [1,2,2,2,1, [1,1,1,1,1]];
    // kernel_width will be '2' for this kernel
    var kernel_width = (ekernel.length-1)/2;
    var neighbor_sum = [0, 0, 0, 255];
    for(var i = -1*kernel_width; i <= kernel_width; i++) {
        for(var j = -1*kernel_width; j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
            neighbor_sum[2] += frame[x][y][2] * weight;
            neighbor_sum[1] += frame[x][y][1] * weight;
            neighbor_sum[0] += frame[x][y][0] * weight;
        }
    }
    var x = m+i; var y = n+j;
    }
}
```

Now we have an iteration space \((x,y)\) that goes from \([m-2, n-2]\) to \([m+2, n+2]\) which are precisely the set of neighboring pixels we want to add up. That is, \(\text{frame}[x][y]\) is a pixel within the neighboring window. So let us add them up with the weights from `ekernel`:

\[
\begin{align*}
\text{weight} &= ekernel[i+\text{kernel_width}][j+\text{kernel_width}]; \\
\text{neighbor_sum}[0] &= \text{frame}[x][y][0] * \text{weight}; \\
\text{neighbor_sum}[1] &= \text{frame}[x][y][1] * \text{weight}; \\
\text{neighbor_sum}[2] &= \text{frame}[x][y][2] * \text{weight};
\end{align*}
\]

There is a detail we have ignored so far. What do we do with pixels on the borders of the image for which the neighbor window goes out of the image? There are several approaches to handle this situation - we could pad the original ParallelArray on all 4 sides so that the neighbor window is guaranteed to never go out of bounds. Another approach is to wrap around the image. For simplicity, we will simply clamp the neighbor window to the borders of the image.

Now we have an elemental function that takes in a single pixel in the video frame whose position is described by the "index" vector and produces a new value for that pixel. We create the following elemental function in `filters.js`.

```javascript
function edge_detect_parallel(index, frame, w, h) {
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1, 1,2,2,2,1, [1,2,-32,2,1, [1,2,2,2,1, [1,1,1,1,1]];
    // kernel_width will be '2' for this kernel
    var kernel_width = (ekernel.length-1)/2;
    var neighbor_sum = [0, 0, 0, 255];
    for(var i = -1*kernel_width; i <= kernel_width; i++) {
        for(var j = -1*kernel_width; j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
            neighbor_sum[2] += frame[x][y][2] * weight;
            neighbor_sum[1] += frame[x][y][1] * weight;
            neighbor_sum[0] += frame[x][y][0] * weight;
        }
    }
    var x = m+i; var y = n+j;
    }
}
```

After the loops are done, we have our weighted sum for each color in `neighbor_sum` which we return. The complete function should look like the following:
function edge_detect_parallel(index, frame, w, h)
{
    var m = index[0];
    var n = index[1];
    var ekernel = [[1,1,1,1,1], [1,2,2,2,1],
    [1,2,-32,2,1], [1,2,2,2,1], [1,1,1,1,1]];
    var kernel_width = (ekernel.length-1)/2;
    var neighbor_sum = [0, 0, 0, 255];
    var weight;

    for(var i = -1*kernel_width;
        i <= kernel_width; i++) {
        for(var j = -1*kernel_width;
            j <= kernel_width; j++) {
            var x = m+i; var y = n+j;
            if(x < 0) x = 0; if(x > h-1) x = h-1;
            if(y < 0) y = 0; if(y > w-1) y = w-1;
            weight = ekernel[i+kernel_width][j + kernel_width];
            neighbor_sum[0] += frame[x][y][0] * weight;
            neighbor_sum[1] += frame[x][y][1] * weight;
            neighbor_sum[2] += frame[x][y][2] * weight;
        }
    }

    return neighbor_sum;
}

Now we have to use this elemental function in our constructor. So just like we did for sepia we will invoke the constructor in main.js as follows:

switch(filterName) {
    case "edge_detect":
        stage_output = new ParallelArray([h, w],
            kernelName, stage_input, w, h);
        break;
}

Now we save our changes, switch to the browser and refresh the page. We can see this function in action by selecting the “River Trail” execution option and then selecting “edge detection” in the right panel. We will see the edges in the video highlighted as shown in Figure 9. The “Sharpen” filter is similar to the edge detection filter except that it uses a different convolution kernel (ekernel).

4. Related Work

Although there is no universal consensus, many educators [6–8] have argued that introducing parallel programming concepts from the top-down is desirable. In other words, this approach entails starting with high-level parallel programming patterns first, with compelling examples and then drilling into low-level mechanisms for implementing these patterns so that they are correct and performant. In [8] the author describes his experiences in teaching parallel programming to undergraduates using both high-level and low-level languages. He argues that teaching high-level parallelism is essential if parallel programming is to be widely adopted. In [9] the author identifies the following four factors that typically make parallel programming (and teaching parallel programming) hard: the lack of isolation between fine-grained concurrent executions, the lack of compositional reasoning, the lack of determinism, and finally the lack of safety. Speciﬁcally, he argues for teaching abstract models of concurrency and deterministic parallel languages.

The NESL language [6] was designed to facilitate teaching parallel algorithms and parallel programming. The author notes that there is a wide gap between low level languages that require the programmer to orchestrate details irrelevant to the algorithm and languages that are so high-level that the performance implications of the various constructs is unclear.

Several authors [5, 10, 11] have noted the usefulness of programming projects that have a clear “visual impact” in sustaining interest among beginner programmers. In [11] the authors advocate presenting parallel programming exercises whose results have broad appeal - [12, 13] mention image processing and encryption as examples of domains that are well-suited.

5. Conclusion

In this paper we present excerpts from a short tutorial for gently introducing programmers to parallelism with River Trail. We introduce them to ParallelArray, a single new data type with 6 methods representing common high-level parallel programming patterns. The tutorial, as well as ParallelJavaScript, does not require understanding the low-level details of synchronization and scheduling allowing the student to focusing on the underlying algorithm’s parallelism and on expressing that parallelism using high level parallel constructs. The tutorial’s interactive web based design provides immediate feedback not only on correctness but also on the performance benefits of their implementations. River Trail and this tutorial are open-source and can be found at [2] and [1] respectively.

References

ACM 39, 3 (March 1996), 85-97
Ebbers , Christopher T. Mitchell, Julian H. Dale “Teaching Parallel
Computing with Higher-Level Languages and Compelling Examples”,
in Proceedings of the 43rd ACM Technical Symposium on Computer
Science Education (SIGCSE), 2012.
[8] Yi Pan, “Teaching Parallel Programming Using Both High-Level and
Low-Level Languages”, in Proceedings of the International Conference
on Computational Science 2002.
[10] Donald Johnson David Kotz and Fillia Makedon, “Teaching Parallel
Computing to Freshmen”, in the Proceedings of the Conference on Par-
Parallelism, OOPSLA 2009.
Students for a Multicore World”, in Proceedings of the 13th Annual Con-
ference on Innovation and Technology in Computer Science Education
(ITiCSE ‘08), Madrid, Spain, June/July 2008.
Gardner, Michael Haupt, and Peter Hinsbeck, “Strategies for preparing
computer science students for the multicore world”, in the Proceedings of the 2010 ITiCSE working group reports (ITiCSE-WGR ‘10)
Figure 5: The River Trail online REPL environment running in a browser. It can be accessed online at the URL in [3]

Figure 6: Skeleton of the web video app used in this tutorial

Figure 7: Original Image

Figure 8: After Sepia Toning

Figure 9: After Edge Detection