Project 1

Zack Owens

7/15/2023

In [2]:
```
using Base.Threads
Threads.nthreads()
```

```
12
```

1

NOTE need to run sinx.(x,1000) instead of for loops

In [12]:
```
function sinx_seq(x, n = 5; degree = true)
    if degree
        xRad = deg2rad(x % 360)
    else
        xRad = x
    end
    value = xRad
    numer = xRad^3.0
    denom = 6.0
    sign = -1.0
    for i in 1:n
        value += sign*numer/denom
        numer *= xRad^2.0
        denom *= (2.0 * i + 2.0) * (2.0 * i +3.0)
        sign *= -1.0
    end
    return value
end
```

```
sinx_seq (generic function with 2 methods)
```

In [13]:
```
function sinx_seq_simd(x, n = 5; degree = true)
    if degree
        xRad = deg2rad(x % 360)
    else
        xRad = x
    end
    value = xRad
    numer = xRad^3.0
    denom = 6.0
    sign = -1.0
    @simd for i in 1:n
        value += sign*numer/denom
        numer *= xRad^2.0
        denom *= (2.0 * i + 2.0) * (2.0 * i +3.0)
        sign *= -1.0
    end
    return value
end
```

```
sinx_seq_simd (generic function with 2 methods)
```

In [14]:
```
# seq
@time for i in 0:0.000_01:90
    sinx_seq(i,1000)
end
```

```
 30.922111 seconds
```

In [15]:
```
# simd
@time for i in 0:0.000_01:90
    sinx_seq_simd(i,1000)
end
```

```
 30.832594 seconds
```

In [21]:
```
# threaded
@time @threads for i in 0:0.000_01:90
    sinx_seq(i,1000)
end
```

```
  4.587285 seconds (41.64 k allocations: 2.820 MiB, 0.25% gc time, 7.12% compilation
time)
```

In [22]:
```
@time @threads for i in 0:0.000_01:90
    sinx_seq_simd(i,1000)
end
```

```
  4.338472 seconds (22.06 k allocations: 1.536 MiB, 5.36% compilation time)
```

Results

Run Type	Time
Sequential	30.922111 seconds
SIMD	30.832594 seconds
Multi-Theaded	4.587285 seconds
SIMD and Multi-Threaded	4.338472 seconds

Overall, the largest increase is multithreading. This took the runtimes from 30 seconds to less than 5 seconds. The SIMD always increased the performance but not by nearly as much as multi-threading increased the performance. Without using either method of parallel computing performed the worst.

Run Type	Speedup
SIMD	0.0029033236710475933358056088307
Multi-Theaded	6.7408305784358286001414780202233
SIMD and Multi-Threaded	7.1274197459382012837699540298981

Overall, the speedup of SIMD is very minimal on its own being under 1%. The increase of multithreading is 6.7 times faster than sequential computing. This is a major speedup with the use of 12 threads. The SIMD and multi-threaded increase is best speedup of all with over 7 times speedup using both 12 threads and the single instruction multiple data operators in my Ryzen 6 CPU.

2.1

In [1]:
```
using BenchmarkTools;
using Base.Threads;
using Plots;

function setMandelbrotPixel(c, niter=255)
    1 ≤ niter ≤ 255 ? niter : 255
    z = zero(typeof(c))
    z = z*z + c
    for i in 1:niter
        abs2(z)> 4.0 && return (i-1)%UInt8
        z = z*z + c
    end
    return niter%UInt8
end


function MandelbrotSet_threaded(niter=100, width=800, height=600,
    x_start=-2.0, y_start=-1.0, x_fin=1.0, y_fin=1.0)
    pic = Matrix{UInt8}(undef, height, width)

    dx = (x_fin-x_start)/(width-1);
    dy = (y_fin-y_start)/(height-1);

    # Compute pic column by column
    @threads for j in 1:width
        x = x_start+(j-1)*dx
        for i in 1:height
            y = y_fin-(i-1)*dy
            @inbounds pic[i,j] = setMandelbrotPixel(x+y*im, niter)
        end
    end
    return pic
end


function MandelbrotSet(niter=100, width=800, height=600,
    x_start=-2.0, y_start=-1.0, x_fin=1.0, y_fin=1.0)
    pic = Matrix{UInt8}(undef, height, width)

    dx = (x_fin-x_start)/(width-1);
    dy = (y_fin-y_start)/(height-1);

    # Compute pic column by column
    for j in 1:width
        x = x_start+(j-1)*dx
        for i in 1:height
            y = y_fin-(i-1)*dy
            @inbounds pic[i,j] = setMandelbrotPixel(x+y*im, niter)
        end
    end
```

```
    return pic
end
```

```
MandelbrotSet (generic function with 8 methods)
```

In [2]:

```
@btime mandel = MandelbrotSet();
```

```
  35.526 ms (2 allocations: 468.86 KiB)
```

In [9]:

```
@btime mandel_threaded = MandelbrotSet_threaded();
```

```
  8.380 ms (72 allocations: 477.27 KiB)
```

The threaded version of the Mandelbrot Set was 4.1365 times faster. The sequential version took 34.664 milliseconds while the threaded version took 8.380 milliseconds to complete. This test was run on a AMD Ryzen 7 3700X using only 12 threads.

2.2

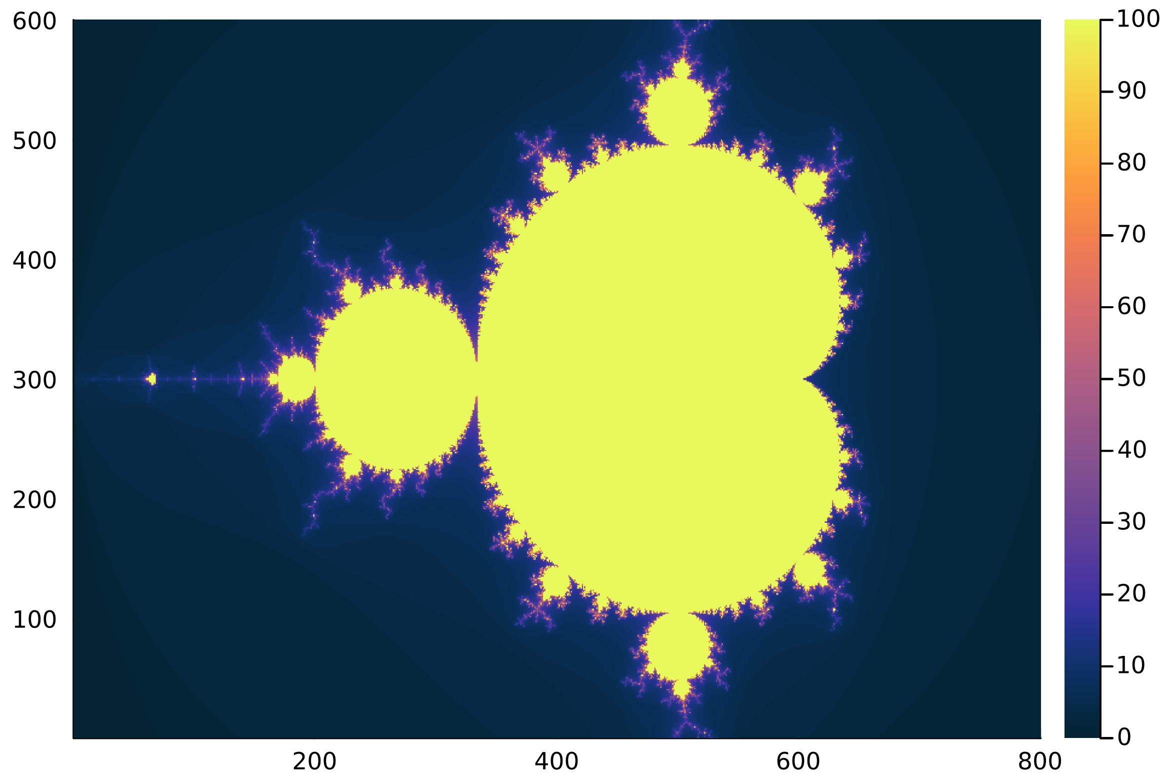
In [25]:

```
function MandelbrotSet_threaded_modified(array,niter=100, width=800, height=600,
    x_start=-2.0, y_start=-1.0, x_fin=1.0, y_fin=1.0)
    pic = Matrix{UInt8}(undef, height, width)

    dx = (x_fin-x_start)/(width-1);
    dy = (y_fin-y_start)/(height-1);

    # Compute pic column by column
    @threads for j in 1:width
        x = x_start+(j-1)*dx
        for i in 1:height
            y = y_fin-(i-1)*dy
            @inbounds pic[i,j] = setMandelbrotPixel_t(x+y*im, array, niter)
        end
    end
    return pic
end

function setMandelbrotPixel_t(c, array, niter=255)
    1 ≤ niter ≤ 255 ? niter : 255
    z = zero(typeof(c))
    z = z*z + c
    for i in 1:niter
        array[threadid()] += 1
        abs2(z)> 4.0 && return (i-1)%UInt8
        z = z*z + c
    end
    return niter%UInt8
end
```

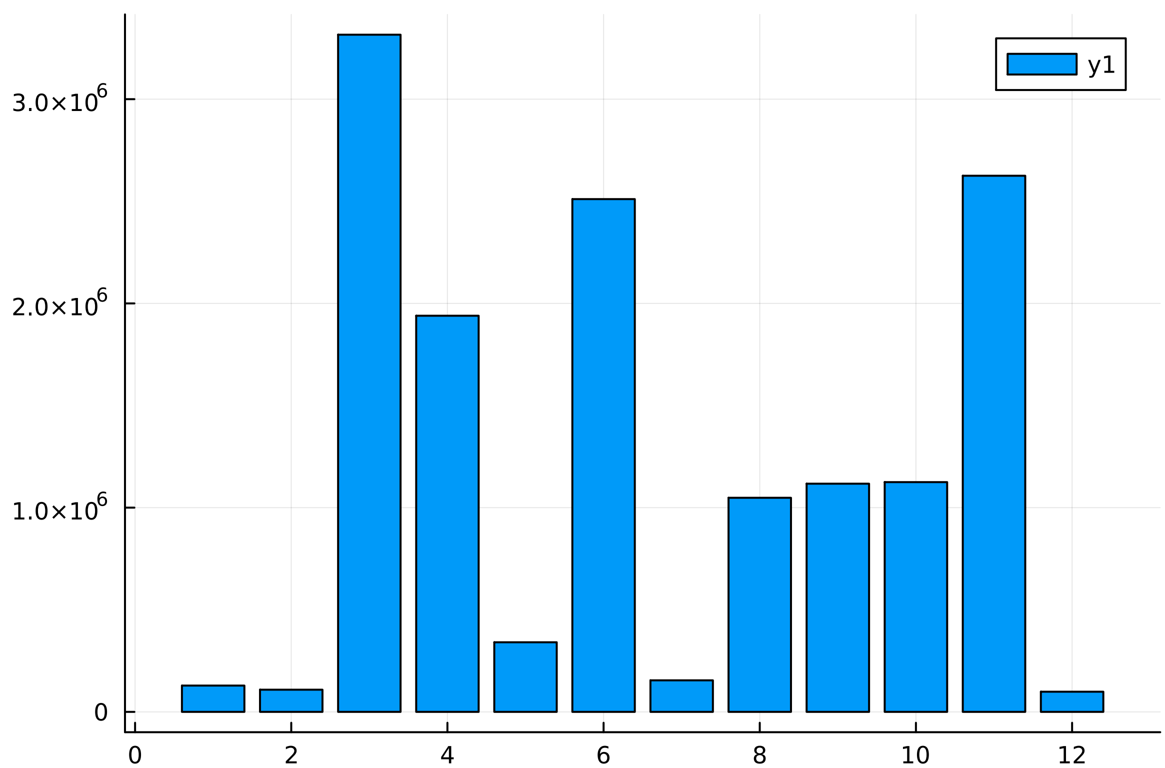
```
setMandelbrotPixel_t (generic function with 2 methods)
```

In [26]:

```
array1 = Int128[0,0,0,0,0,0,0,0,0,0,0,0]
mandel1 = MandelbrotSet_threaded_modified(array1)
heatmap(1:size(mandel1,2),1:size(mandel1,1), mandel1, color = :thermal)
```



In [28]:
```
using Plots
bar(array1)
```



In [29]:
```
@btime MandelbrotSet_threaded_modified(array1)
```

  8.839 ms (75 allocations: 477.36 KiB)

```
600×800 Matrix{UInt8}:
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
    ⋮                                ⋮  ⋱              ⋮
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
```

The bar chart above shows the number of iterations that ran on each thread. As seen above the spread is not very even as some pixels require more iterations than others. This creates a large number of iterations on certain threads. In this case, thread 3 had the largest amount by over a million iterations. This discrepancy causes the program to be slowed as this thread takes longer than other threads to complete task.

2.3

In [21]:

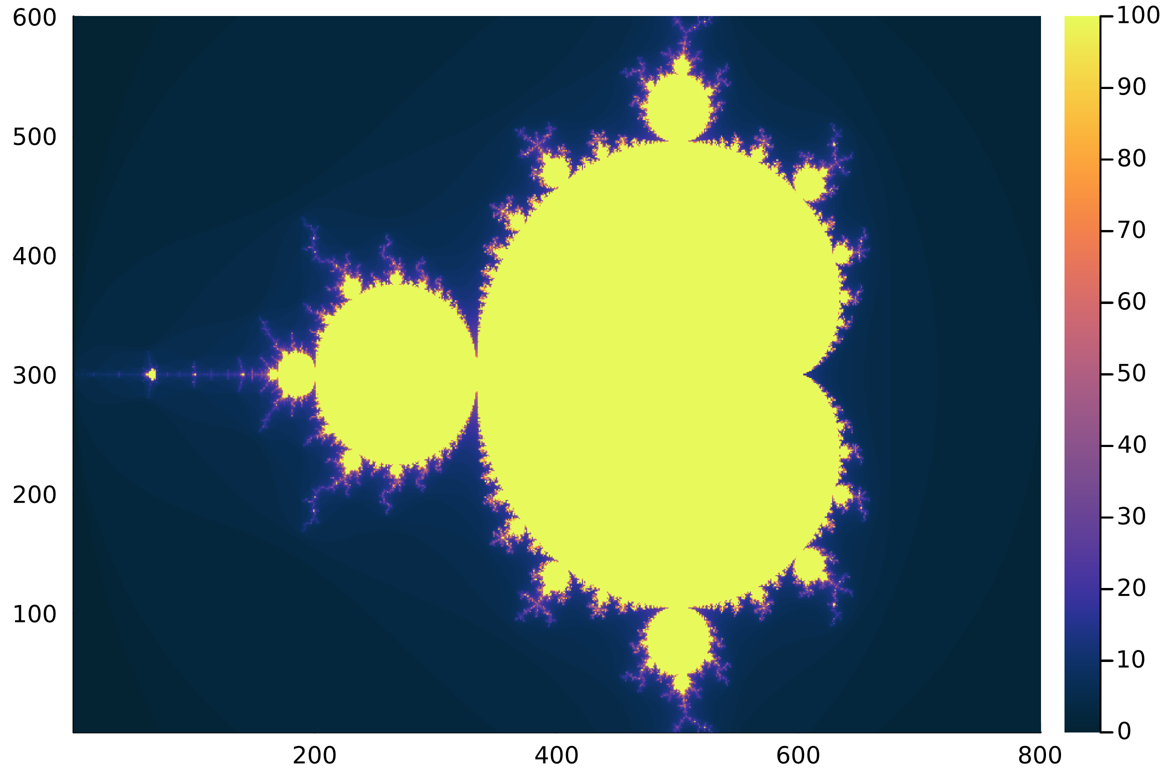
```julia
function MandelbrotSet_threaded_modified_fixed(array,niter=100, width=800, height=600,
    x_start=-2.0, y_start=-1.0, x_fin=1.0, y_fin=1.0)
    pic = Matrix{UInt8}(undef, height, width)

    dx = (x_fin-x_start)/(width-1);
    dy = (y_fin-y_start)/(height-1);

    # Compute pic column by column
     for j in 1:width

        x = x_start+(j-1)*dx
        for i in 1:height
            @spawn begin
                y = y_fin-(i-1)*dy
                @inbounds pic[i,j] = setMandelbrotPixel_t(x+y*im, array, niter)
            end
        end
    end
    return pic
end
```

```
MandelbrotSet_threaded_modified_fixed (generic function with 8 methods)
```

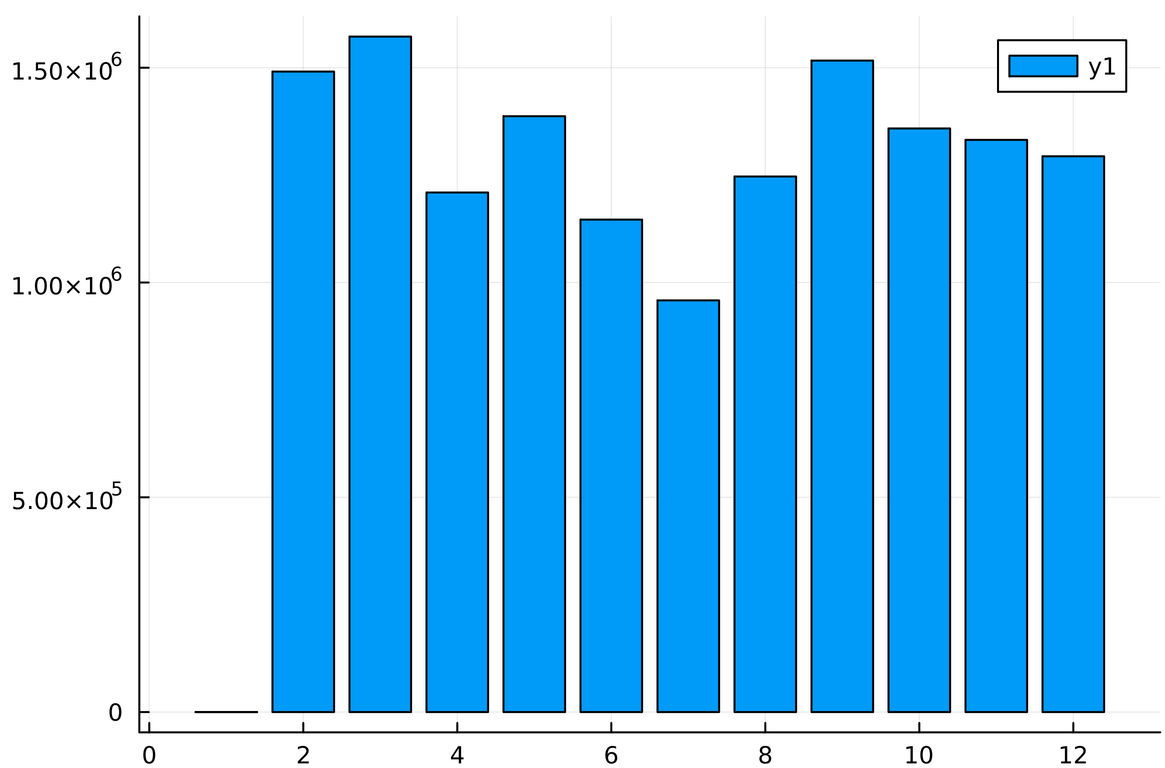
In [22]:

```julia
using ColorSchemes
array2 = Int128[0,0,0,0,0,0,0,0,0,0,0,0]
```

```julia
mandel2 = MandelbrotSet_threaded_modified_fixed(array2)
heatmap(1:size(mandel2,2),1:size(mandel2,1), mandel2, color = :thermal)
```



In [23]:
```julia
using Plots
bar(array2)
```



In [24]:
```julia
@btime mandel2 = MandelbrotSet_threaded_modified_fixed(array2)
```

```
  308.250 ms (2731130 allocations: 310.86 MiB)
600×800 Matrix{UInt8}:
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
    ⋮                             ⋮  ⋱           ⋮
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00  …  0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
 0x00  0x00  0x00  0x00  0x00  0x00     0x01  0x01  0x01  0x01  0x01  0x01
```

This version of the Mandelbrot Set function was slower than the original multi-threaded version using the @threads macro. This version used the @spawn macro and took 34.874 times longer to compute. The efficiency of the threading was much better than the @threads macro. This is since all of the threads for this version ran within 500,000 iterations of one other. This is more effective when it comes to CPU core usage but worse performance.

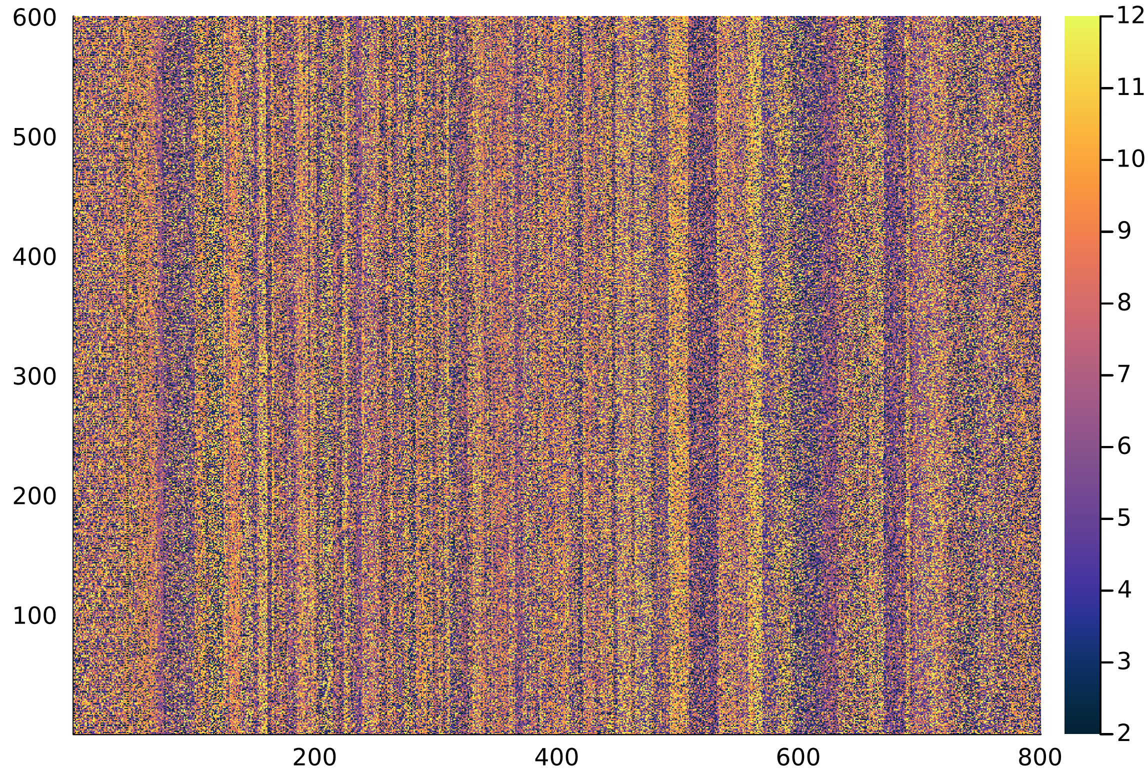
2.4

In [37]:

```julia
function MandelbrotSet_threaded_modified_fixed_threadMap(array,niter=100, width=800, h
    x_start=-2.0, y_start=-1.0, x_fin=1.0, y_fin=1.0)
    pic = Matrix{UInt8}(undef, height, width)

    dx = (x_fin-x_start)/(width-1);
    dy = (y_fin-y_start)/(height-1);

    # Compute pic column by column
     for j in 1:width

        x = x_start+(j-1)*dx
        for i in 1:height
            @spawn begin
                y = y_fin-(i-1)*dy
                setMandelbrotPixel_t(x+y*im, array, niter)
                @inbounds pic[i,j] = threadid()
            end
        end
    end
    return pic
end
```

```
MandelbrotSet_threaded_modified_fixed_threadMap (generic function with 8 methods)
```

In [38]:
```
using ColorSchemes
array3 = Int128[0,0,0,0,0,0,0,0,0,0,0,0]
mandel3 = MandelbrotSet_threaded_modified_fixed_threadMap(array2)
heatmap(1:size(mandel3,2),1:size(mandel3,1), mandel3, color = :thermal)
```



By looking at the heat map, we can see that the threads are more evenly spread the millions of runs needed for each pixel.

3

In [5]:
```
using Random;
using BenchmarkTools;
using Base.Threads;
```

In [7]:
```
array3 = Int128[0,0,0,0,0,0,0,0,0,0,0,0]
random_numbers = rand(Int32,10_000_000)
```

```
10000000-element Vector{Int32}:
  1018666248
 -1675220526
  1657841983
  1697461629
 -1247728381
 -1184119826
  1141042292
 -1725925291
  1878417021
   982630976
           ⋮
  1460876765
  1861170015
  1242172441
  2025718649
 -1740062139
 -2027662547
 -1523049364
  -132150021
 -1873862810
```

In [8]:
```
function sum_seq(x)
    s = 0
    for i in x
        s +=i
    end
    return s
end

@btime sum_seq(random_numbers)
```

```
  1.441 ms (1 allocation: 16 bytes)
-2669141116162
```

In [9]:
```
@btime Base.sum(random_numbers)
```

```
  1.560 ms (1 allocation: 16 bytes)
-2669141116162
```

In [9]:
```
Threads.nthreads()
```

```
12
```

In [13]:
```
function partialSums(x)
    if Threads.nthreads() == 1
        print("only one thread using 1 thread")
        return sum(x)
    end
    t=Threads.nthreads()
    split = round(Int64,length(x) / t)
    partial_sums = [0 for x in 1:t]
    #println(split,partial_sums)
    @threads for i in 1:t
        if(i==t)
            partial_sums[i] = sum(x[(i-1)*split+1:length(x)])
            #println((i-1)*split,'x',length(x))
            continue
        end
        if(i==1)
            partial_sums[i] = sum(x[1:split*i])
```

```
            #println(0,'i',split*i)
            continue
        end
        partial_sums[i] = sum(x[(i-1)*split+1:split*i])
        #println((i-1)*split,' ',split*i)
    end
    #println(partial_sums)
    return sum(partial_sums)
end

@btime partialSums(random_numbers)
```

```
  4.042 ms (99 allocations: 38.16 MiB)
-2669141116162
```

In [14]:

```
using Folds
@btime Folds.sum(random_numbers)
```

```
  1.311 ms (557 allocations: 21.80 KiB)
-2669141116162
```

Usage	Timing
Sequential	1.441 ms
Partial Sum	4.042 ms
Base.sum	1.560 ms
Folds.sum	1.311 ms

Overall, the values for each of the values besides Partial Sum were very similar. Partial sum took the longest at over 4 millisecond which was over 2 times slower than all other methods. This experiment is a good example of throwing more threads at a problem will not always create a better solution. The Base.sum method was the seconds slowest taking 1.560 milliseconds which is similar to the sequential runtime of 1.441 milliseconds. The fastest was the Folds.sum which is multithreaded using all 12 threads and took 1.311 milliseconds. The partial sums solution is much slower due to having to splitting of the array and creating a threads to complete the split sums.