Parallel Execution in OpenMP and Java 8 Streams

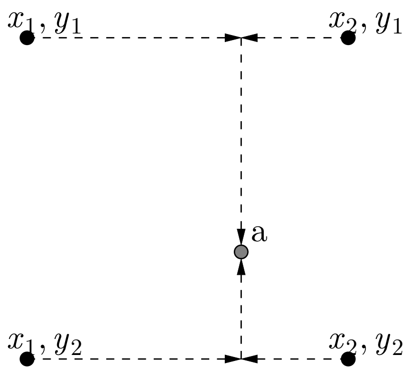
by Natan Deusdedit Vargas

Introduction

To improve performance modern computers utilize multi-threading. This is done by partitioning a process into multiple tasks. Each task is scheduled to run on a concurrent thread. Threads may share memory or instantiate their own instance of a variable. Sharing memory access can be problematic because threads may operate out of order. I will discuss my technique for using OpenMP and Java 8 streams to parallelize Gaussian elimination and bilinear interpolation and how I avoid collisions between threads.

Bilinear Interpolation

Bilinear interpolation is an extension of linear interpolation, which is used to approximate intermediate values of linear functions. Bilinear interpolation is used to interpolate values between two axes.



interpolating point a

A common application of Bilinear interpolation is scaling images. This method of scaling interpolates the color value of the new intermediate pixels when the image is resized.

Image scaled 130% using bilinear interpolation

Using OpenMP

We must begin by identifying possible critical sections, a section of code where memory sharing could cause issues. For this case, there are no critical sections, because we are reading from a `const` pixel array and each thread writes a pixel to a different index. We do not have to do any calculations on the newly generated pixels. The only thing we need to do to make this code parallel is add a OpenMP directive. Since, OpenMP was unable to locate the loop parameters properly, I rewrote it as a nested loop, which has the same range as it did before. Below is the comparison of the concurrent loop and the OpenMP loop.

```
...
for(x=0, y=0; y < newHeight; x++) {
    if(x > newWidth) {
        x = 0; y++;
    }
    ...
```

Original Loop

```
...
#pragma omp parallel for private(x, y)
for(x=0; x < newWidth; x++) {
    for(y=0; y < newHeight; y++) {
    ...
```

Parallelized Loop

It is worth considering using different scheduling to further increase performance. The `schedule` argument is used to specify the technique OpenMP will use to partition and assign chunks to threads. `schedule` takes two parameters. The first is the scheduling type, they are: `runtime`, `auto`, `static`, `dynamic`, and `guided`. The second parameter is the chunk-size. Chunks are a continuous range of iterations that can be tasked to a thread.

The default, `static`, assigns chunks amongst the available threads equally at startup. You can optionally define the chunk size, and they will be assigned to each thread in a round-robin pattern. i.e. `schedule(static, 2)` whill create chunks with two continuous indices: 0-1, 2-3, etc. An equal number of chunks will then be assigned to each thread.

`dynamic` will create a queue of chunks at runtime, and whenever a thread becomes available, it is assigned a chunk from the front of the queue. `dynamic` ensures that none of the threads go idle if their workload is less than the others. However, the queue creates additional overhead that may cause a loss of performance. The chunk size defaults to 1, but you can choose a larger chunk size if you benefit from looping over continuous iterations.

The last scheduling technique `guided` will shrink the chunk size at runtime so that the initial chunks are much larger than the last. The optional chunk size parameter defaults to 1, and it defines the minimum size of a chunk.

Below are the runtimes resulting from different scheduling techniques and various chunk sizes:

Benchmark Results		
schedule	chunk-size	runtime (seconds)
dynamic	8	49.71
guided	8	50.03
dynamic	default	50.04
auto	default	50.07
static	default	50.21
dynamic	2	50.23
guided	2	50.31
static	1	50.36
static	8	50.39
dynamic	32	50.45
guided	32	50.59
guided	default	50.61
guided	4	50.67
static	4	50.87
dynamic	4	50.97
static	2	51.02
serial	default	166.17

AMD FX(tm)-6300 Six-Core Processor

From my own testing, all of the scheduling options I tried had about the same runtime. I would recommend the default static for this algorithm, because guided and dynamic are best suited for jobs where the runtime of each iteration can be different. That is not the case for this algorithm. The extra overhead from dynamic and guided could be avoided, although for this algorithm, it is not impactful.

Using Java 8 Streams

By abusing the capability for Java 8 streams to run in parallel, we can easily optimize this algorithm in Java. Data sharing is done using lambda variable captures[1] .

```java
...
for(int x = 0; x < newWidth; x++) {
   for(int y = 0; y < newHeight; y++) {
       int rgb = blerpPixel(src, newWidth, newHeight, x, y);
       dst.setRGB(x, y, rgb);
   }
}
...
```

Original Java Bilinear Interpolation

```java
...
IntStream.range(0, newWidth).parallel().forEach(
   x -> IntStream.range(0, newHeight).parallel().forEach(
       y -> {
           int rgb = blerpPixel(src, newWidth, newHeight, x, y);
           dst.setRGB(x, y, rgb);
       }
   )
);
...
```

Parallelized using Java 8 Streams

Reading List