Parallel Execution in OpenMP and Java 8 Streams

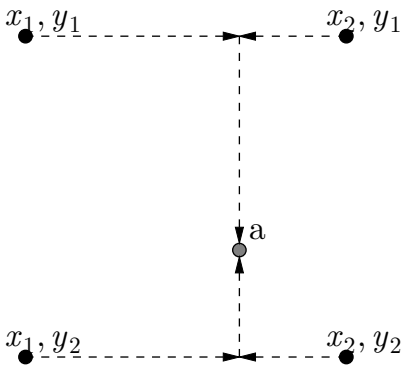
by Natan Deusdedit Vargas

Introduction

Modern computers can utilize multi-threading to improve performance. This is done by partitioning tasks between multiple threads, which can run concurrently. For certain tasks utilizing multiple cores can improve performance. However, since threads may share memory access, they could operate out of order resulting in an strange behavior. 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

OpenMP was unable to locate the loop parameters properly, therefore I rewrote it as a nested loop, which iterates the same as it did before. Since the src array is read-only, and each iteration writes to a different index in the dst image, there are no data dependencies.

```
...
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. `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 for a thread to do work on.

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)` utilizing 4 threads over 16 iterations would assign thread 1, 1 to 2 then 9 to 10; and thread 2, 3 to 4 then 11 to 12.

`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	32	48.86
dynamic	2	48.94
guided	default	49.05
guided	2	49.20
guided	8	49.21
guided	4	49.25
dynamic	4	49.37
static	2	49.74
static	8	49.99
dynamic	default	50.22
dynamic	8	50.27
auto	default	50.48
static	4	50.72
guided	32	50.84
static	default	50.85
static	1	51.15
serial	default	137.46

4 threads

Using Java 8 Streams

By abusing the capability for Java 8 streams to run in parallel, we can easily optimize this algorithm in 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

```
...
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