DPP-exam-report

Oleksandr Kryshtalov
Alexandre DA CUNHA–GUIBORAT

January 2024

1 Introduction

Often times, when we have point data, we want to decide whether the point is included in a rectangle. This is a task that could be useful for example in a navigation setting to decide whether something is in a region or not. Or for example for a primitive version of clustering.

A lot of these use cases involve huge datasets or requirements for low latency, which makes it valuable to apply parallel programming as well as optimization techniques, to make the applications better fit its requirements for speed.

When we talk about parallelism and concurrency, often times these things get confused, therefore it is important to clarify this. Concurrency is when our application start and complete tasks in overlapping space of time. This is useful for example in highly IO dependent tasks like an interactive website, which might simultaneously need to download data from a server, process it, as well as handle inputs from the user, while simultaneously showing animated graphical elements. This type of parallelism does not require the usage of multiple threads or similar, it only requires that the starting and stopping can happen in an interleaved manner. On the other hand, we have parallelism, which is the ability to perform a task by splitting it up and, for example, have multiple threads work on the task at once. Since this is a data-parallelism focused course, we will be focusing on the latter.

Furthermore, we have the distinction of task parallelism and data parallelism. Task parallelism is the attribute of having multiple tasks and being able to do them on each of their own threads or similar, this type of parallelism often requires expensive measures such as locks and mutex'. This is because different tasks could depend on each other's state, in other words running the tasks in a different order could yield an incorrect result. An example of this could be using your banking website, and you try to send money, when you send money, it is important that while the transaction is being cleared no one can draw from your account, otherwise you could, for example, send a bunch of transactions at once and only pay for one of them. We also have data parallelism, which is the attribute of having a task which has a dependency graph that is simple to reason about, which means that we can build programs that avoid needing

expensive parallelism constructs by simply reordering the reads and writes. An example of this could be the vector addition, we already know the exact calculations that need to happen and in which order, because of this it becomes quite trivial to compute it in parallel. In general because our code operates in the real world, we often have to contend with task parallelism, but often times the libraries that are used in a task parallel setting are data parallel, in other words even though we cannot achieve data parallelism as a whole we can inject data parallel programs that benefit from all the benefits of data parallelism.

2 Picking the right tool

When tackling a problem with parallelism, we have a lot of options to choose from, all with their of pros and cons. To make a short list, we could write it in a typical language, not designed for parallelism, which comes with issues such as inconsistent performance, for example the compiler might use SIMD, but that is not guaranteed, and we have to keep track of performance because an update could change the performance characteristics. Using parallel constructs a typical language, that is not built for it, can be quite time intensive and complex. Therefore, oftentimes it makes sense to use a dedicated programming model which can achieve the performance that we are seeking, an example of this could be, futhark, ISPC or in some cases using c/c++.

Finally, we picked futhark, mainly because that is the language we were most comfortable with, and it let us compile to different targets, like GPU's.

3 Designing an efficient program by utilizing parallelism

When coming up with ideas, on how to solve it, we started by looking at the brute force solution. We quickly could tell that this algorithm is $O(n * m)$, from the skeleton of this brute force algorithm it was simple to design a parallel algorithm that had $S(1), W(n * M)$, by using two maps. After implementing that algorithm, we would try to design algorithms that would potentially lower this number and make it more efficient. Put differently, the performance of our algorithm depends on how efficient our algorithm is, as in how much total work it has to do asymptotically, and making best use of each clock cycle.

One of the ways of making our code asymptotically more efficient, was exploiting the fact that for each rectangle, in a brute force approach, the algorithm checks way more points than is necessary, and thus we could check fewer points per rectangle. We will be going more in depth with this solution later.

In terms of non-asymptotic efficiency we did not put in any effort into it since that is typically done by the compiler, and we expect it to do a good job.

4 Brute force solution

4.1 Algorithm definition

The implementation of a brute force algorithm is quite straight-forward. We start by defining the function by which we determine whether something is in a rectangle or outside. This function takes a point, and then a rectangle defined as two points, upper-right(ur) and lower-left (ll). This function can be run in constant time, and corresponds to a series of four conditionals.

```
def within(point, rect):
    return  p.x <= rect.ur.x AND
            p.x >= rect.ll.x AND
            p.y <= rect.ll.y AND
            p.y >= rect.ur.y
```

The brute force, algorithm goes through every point, and for every point goes through every rectangle and increments a counter corresponding to each rectangle depending on if it's within the rectangle.

```
def get_counts_simple (points, Rects)
    count = [0]*len(rect)
    for p in Roints
        for (i, r) in enumerate(Rects)
            if within(p, r)
                count[i]++
            else
                nop
    return count
```

4.2 Performance

Since this is a brute force approach, the asymptotics do not look good. There are n iterations of the outer loop, and m iterations of the inner loop, thus it has time complexity of $O(n * m)$. Since this is a sequential algorithm, the parallel analysis is trivial, with a span of $n * m$ and work of $n * m$.

5 Parallel algorithm in futhark

5.1 Implementation

The parallel implementation in futhark is almost exactly the same as the brute force implementation, but we use maps instead of loops:

```
-- counts the number of points in the rect
def points_in_rect points  rect =
    reduce (+) 0 (map (\p -> if within p rect then 1 else 0) points)

def rangeQuery2d points rects =
    map (points_in_rect points) rects
```

5.2 Performance

As mentioned earlier this version of the algorithm is parallel, this means that our work and span analysis is a bit more interesting. We are using two nested maps, and in the innermost map we perform a constant time operation. This means that the span of the algorithm is $S(c)$. Furthermore, since we are not duplicating any of the calculations, the work remains the same $w(n*m)$. All of this means, is that for a big enough dataset, as the constant performance penalties from using parallelism diminishes, we can keep scaling up the number of used threads and get better and better performance. Furthermore, the amount of total work remains the same, which means it is work efficient.

6 Grid algorithm

6.1 Idea

In order to speed up the computation, we can subdivide the plan like a grid. Calling each rectangle in the grid a cell, we can then maintain for each cell a list of points within that specific cell. This part consist of the preprocessing.

Then we can iterate through all rectangle and depending on the crossed cells, we consider only a subset of all the points and brute force on this subarray.

6.2 Preprocessing

Note : To stick to the standard of programming, the upper left corner of the plan is chosen to be the origin, with x going left to right, and y going top to bottom.

To create the grid, we decided to subdivide the plan into equal rectangles. As the plan is a rectangle of size 1 by 1. We can compute, for a given depth (d) and index (i), the position of the upper right (ur) and bottom left corner (ll) of each cell.
$ll = \frac{1}{2^d}(i\%(2^d), \frac{i}{2^d} + 1)$, and $ur = \frac{1}{2^d}(i\%(2^d) + 1, \frac{i}{2^d})$.

This way we can create each cells. To maintain the list of all points in the cell, we can brute force over all points.

6.3 Sub-Array

Once the preprocessing done, for each rectangle, we can check which cells are crossed by the given rectangle. Constructing this list, we can then gather all points within those cells, and brute force on those points.

6.4 Implementation

```
def rangeQuery2d_grid [m] [n] (depth : i64) (rectangles : [m]Rectangle)
                              (points : [n]Point) : [m]i64 =

    let d = depth
    let cells = preprocess_create_grid_depth d points


    let solution =  map(\r ->
                        let subarray_pts : []Point =
                            get_subarray_point r cells points

                        in nb_points_in_rectangle r subarray_pts
                    ) rectangles
    in solution
```

6.5 Optimisation

This implementation had a major flaw actually. We used to have two filter call. Those filter were use in the get_subarray_point function. Those two filter were slow.

Thankfully, we got rid of them both using scans, maps and scatter in order to speed up that part. Those changement can be observe on the _bis.fut version of the grid implementation.

This proved to have a huge impact on performances.

6.6 Performance

This code is almost fully parallel. But sadly, for big data set, the memory usage is too high and we can't run it on our computers.

This can be fixed by changing the flag array in each cell by an array of unknown size (at compilation time) containing the indices of the points in the points array. This way we can save on memory.

But even and very small data set (5k), our implementation is barely faster than the pbbs one on a data set of (1M).

```
benchmark_all.fut:bench_simple_parallel (no tuning file):
2DinCube (5k):        322µs (95% CI: [     310.9,      338.8])

benchmark_all.fut:bench_point_grid (no tuning file):
2DinCube (5k):     241725µs (95% CI: [  239974.0,   242729.7])

benchmark_all.fut:bench_point_grid_bis (no tuning file):
2DinCube (5k):      35760µs (95% CI: [   35124.1,    36173.6])
```

```
PBBS times (s)
2DinCube (1M):     '0.583', '0.584', '0.58', geomean = 0.583
2Dkuzmin (1M):     '0.582', '0.584', '0.578', geomean = 0.581
2DinCube (10M):     '9.737', '9.813', '9.845', geomean = 9.798
2Dkuzmin (10M):     '10.15', '10.673', '10.578', geomean = 10.465
```

To reproduce those result, go to tools and then run : $ make name_size.in. Go to src/benchmark_all.fut and change the entry list as well as the data input as you need them.
Finally just run : $make bench_all_backend , to benchmark using a specific backend.

6.7 Improvement

This implementation was inspired by the quadtree method. In fact, the quadtree method might be even faster than this one and could be an implementation to explore.

In order to lower the memory usage, we can use irregular array in cells to store points. But this will imply overhead.

7 Running the code

In /tools we have a Makefile to generate test data, as well as to run the benchmark. To run the benchmarks we have Makefile rules bench_all_c bench_all_multicore bench_all_opencl are all there to benchmark the code with various backends.