

Parallel computing

Parallel Ramer-Douglas-Peucker



February 26, 2018

Hogeschool van amsterdam

# Introduction

This document describes my learnings into the process of parallelizing algorithms.

## Ramer-Douglas-Peucker

The Ramer-Douglas-Peucker series reduction algorithm (RDP) reduces the number of points in a sequence. As such it’s used for simplifying geographic data, 3d laser scanning and (htt)

RDP is recursive and has an average time complexity of O(n log n), but the worst case complexity is O(n^2).

A visualization of the algorithm can be found on youtube.

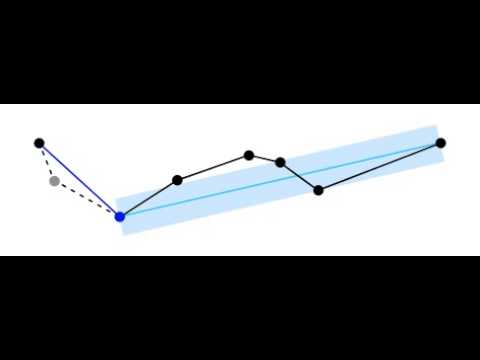
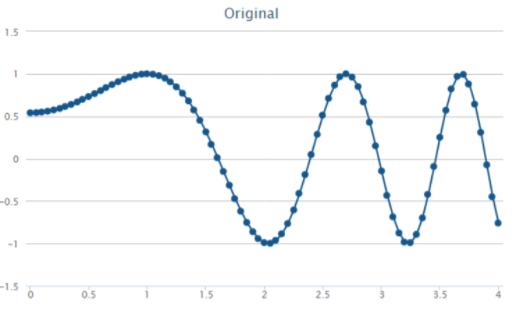
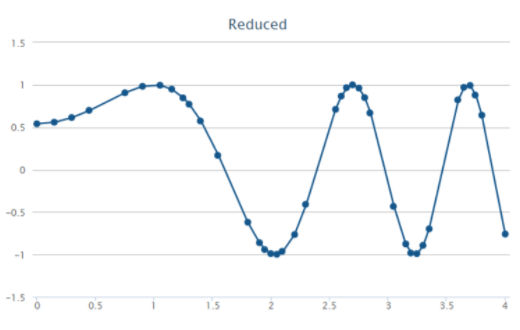
[](https://www.youtube.com/watch?v=TvHmIWM_EUU)

Figure source; Jade Haayen





RDP in pseudocode

* Function Distance(Line, Point), where the index of Point lies between L[p0] and L[Pn]
* threshold value ε representing a normalized minimum distance (irrespective of line length) where ε > 0

Ordered Sequence S (p0, p1 … Pn)

RDP[S] =

Line l = {S0, Sn}

Subsequence = {S1, S2 … Sn-1}

Maxdistance = -1;

MaxIndex = -1;

For each P in Subsequence

If max(distance(P,L)) > ε

Maxdistance = distance(P,L)

MaxIndex = P

If Maxdistance > ε

///break line into two segments, recurse

Return RDP(S0, MaxIndex) + RDP(MaxIndex, Sn)

Else

//return results

Return Lstart, Lend

## Problems and process

## Recursion and decomposition

RDP is a recursive algorithm, which can be parallelized, but hard to distribute. Fork-join is nice, but it’s hard to deconstruct individual parts of the recursive algorithm into discrete elements.

Luckily the recursion is tail-end, which means it can be rewritten into iterative format. In the original, all split lines are immediately recursed into. Instead of immediately going into the split line, you can collect all the split lines on a stack, which can be iterated over.

## Decomposing RDP

RDP basically does four things;

1. Construct a line from the first and last point of sequence
2. Search for the most distant point in sequence from the line
3. Split the line if the most distant point is more than epsilon, or return start- and end-point.
4. If split, repeat the process for the two new sequences

Constructing a line *(1)* from two, two-dimensional points, and calculating the derivative is done in O(1) time complexity . With some guessing, creating a two-dimensional line from two points simply isn’t that taxing. Higher-dimensional lines and derivatives do exist, but lie outside the scope of this document.

Searching an array (2) has a time-complexity of O(n), and can be executed in parallel. This process is dependent on the line from process (1). This is by far the costliest process in the entire algorithm (aside from recursing).

Splitting sequences and returning results (3) is the last step of the process. Scales O(1) with load.

Recursing into the process until all subsequences are solved (4) scales O(log n) on average, and is actually pretty heavy on system resources. In the worst case (all points within epsilon, last point most distant) this scales O(n)

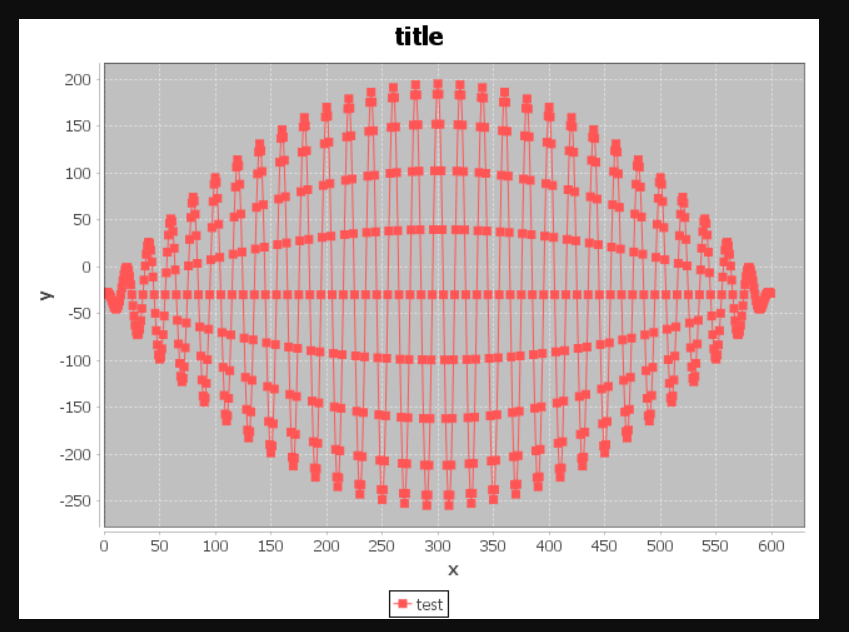
Effectively, steps 2 and 4 incur the costs of the entire algorithm, average O(n log n) and O(n\* n). Luckily, step 2 is embarrassingly parallel, and step 4 can also be executed in parallel after the first *split*.

## Soundness and measurability

To save some time, I grabbed a working RDP implementation from a [GitHub repository](https://github.com/LukaszWiktor/series-reducer). The implementation came complete with unit-tests and extensible interfaces.

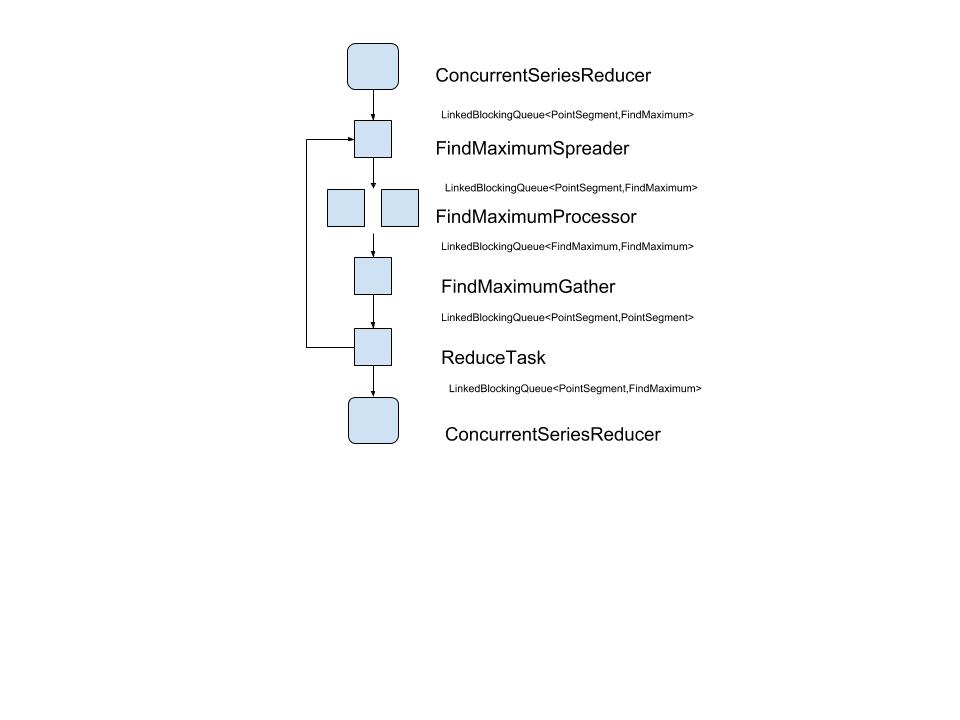
Even when “*cheating”*, the algorithm’s results were not obvious. To make sense of the inner workings of the algorithm, I needed something more than just a console. I wanted to visualize the results, and for that end I used jFreeChart to chart the complete sequence. jFreeChart was ultimately limiting, and generating visualizations (proper scale) ended up costing me a lot of time.

Measurements in speedups also required some tools, a basic stopwatch and a little bit of *Julia* to generate some results. It required some serious wet-work to find a proper *Epsilon* that didn’t retain all points OR removed them all. For that end, I constructed a test-set with gradual deltas, shown below.



# Threaded producer-consumer

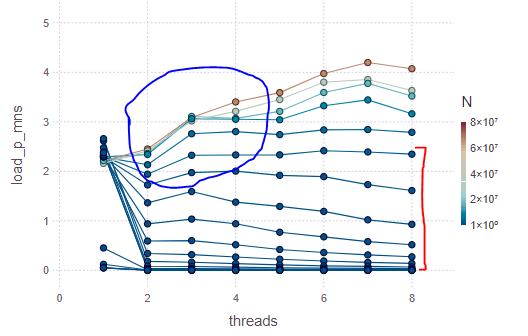
My threaded version of RDP splits up the 4 previously describes operations into 4 discrete processes, of which searching for the furthest point from line is done in parallel. The code is available



## Measurements

Measuring improvements was done with 2 variables; the size of the sequence (N) and the number of *FindMaximumProcessor* threads. Tests were done on a 4-core i7 with hyper-threading, hence the 8 *virtual* cores.

Any threading overhead (pools) was included in the test-results.



The above graph shows the spawned number of threads (X-axis) used vs ((time in ns) / N ) (Y-axis), and N as a colored line, ranging from 128 points to ten million points.

### Observations

At the red mark, the relative speedup actually goes flat, proving Gustafson’ law; with more resources execution, we might handle bigger workloads in the same time, but not necessarily reduce the time taken.

At the blue mark, we can see a definitive “S” curve in the grey to light-blue workloads. As resources improve, speed improves until the non-parallel portion of the algorithm presents a bottleneck. After the bottleneck, improved resources are subject to diminishing return. This effectively proves Amdahl’s law.

Also noticeable is the tradeoff between single-threaded and multi-threaded execution. Below N’s of ~20 million, overhead introduced by threading outweighs actual improvement.

### Magic numbers

In the FindMaximumSpread task, the batchsize has an optimum of around 12500 points.

## Problems

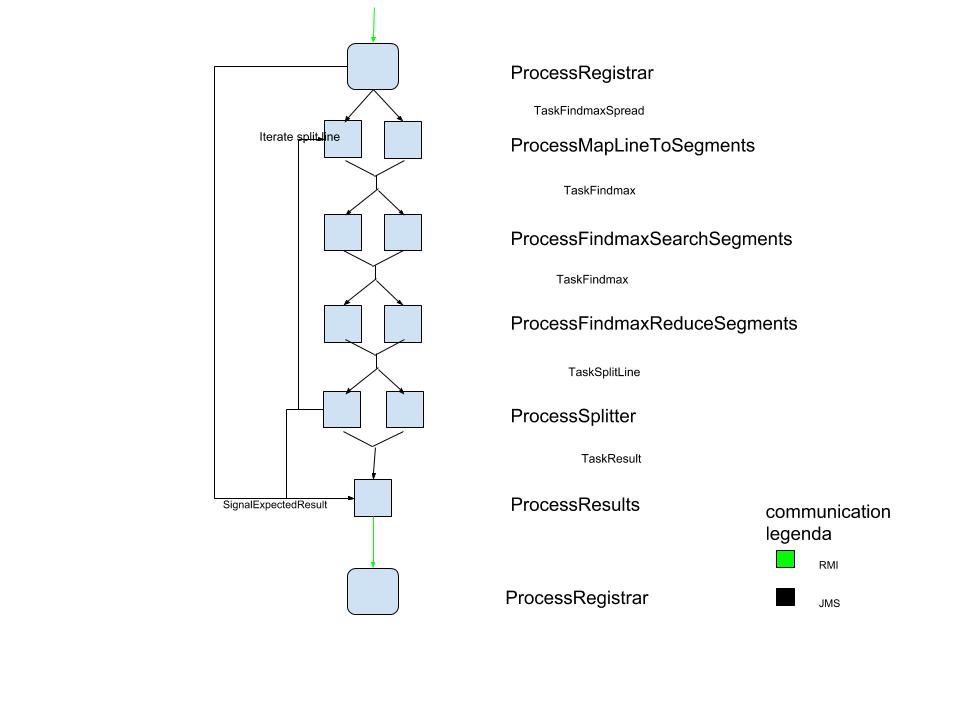
My threaded version has no definitive *end-condition,* andrelies solely on the statusflags (START,WAITING) of existing workers. This is bad, mostly because the entire thing can be spun down much, much, earlier.

Only the most expensive part (search) of RDP is improved upon. After some more research (see *Distributed producer-consumer* chapter), more communication can be introduced to parallelize other parts of the algorithm.

## Conclusion

# Distributed producer-consumer

The distributed RDP is architecturally similar to the threaded version, but runs on several separate processes. The



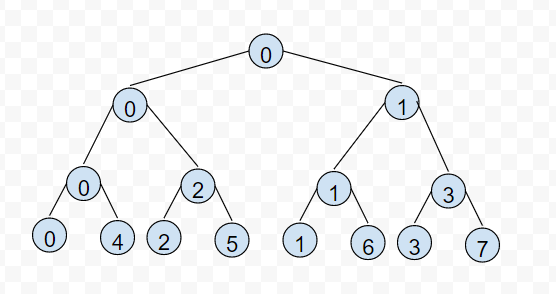
## Improvements over threaded version

Instead of communicating through memory, this version communicates through messaging, allowing it to scale beyond a single machine.

### Tree decomposition

In the threaded version, task completion was determined by the status of individual processes. This was improved by actually looking at the way we’re splitting lines, and thus updating our expectations about the results.

Splitting lines effectively composes a binary tree, in which each node must return a result. By comparing the expected results with the received results of each node (line), we know when the process has finished.



### More parallel

Virtually all the processes can be run in parallel, eliminating bottlenecks. The processes that require *state,* communicate this state via the ProcessRegistrar.

#### ProcessMapLineToSegments

This process stencils lines into segments for parallel searching. To later reattach all searched stencils into a single line (*ProcessReduceSegments*), this process attaches an integer ID (*spreadID*) to all segments. Keeping unique identifiers in the process consequently introduces state into the process, which prevents parallel execution.

By moving that state to the ProcessRegistrar, we can execute this process in parallel again. By batching unused spreadIDs to individual *ProcessMapLineToSegments* processors, we can massively reduce the overhead and expand parallel execution.

#### ProcessSplitter

Similar to *ProcessMapLineToSegments,* the ProcessSplitter creates new lines and thus creates state. This problem was similarly solved by moving the creation of IDs to the ProcessRegistrar.

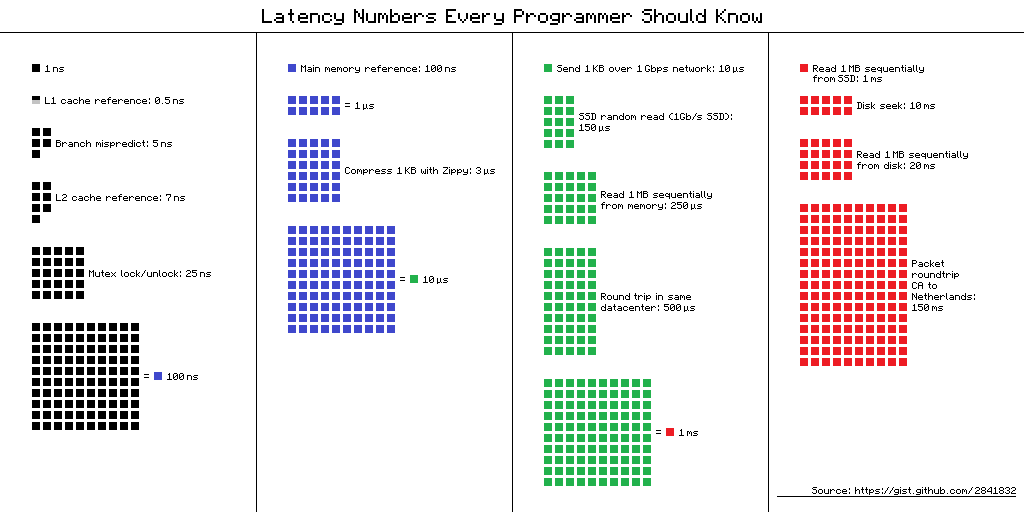
#### ProcessFindmaxReduceSegments

This process reattaches searched segments into a single line again. To do this, it needs to store individual segments until all segments are available. For sole sake of complete parallel execution, all parts are stored at the ProcessRegistrar.

This violates the “don’t move data, move computation” paradigm, but the overhead could be justified with a sufficiently large (millions) search-segment size. We’re already working distributed, the offset is already large. However; in any *real* setup, this temporary storage would require a dedicated machine.

## Measurements

No measurements were done on the distributed RDP variant. Inferring from previous measurements, I’m very sure that the overhead would kill any improvement over the single-threaded version, and I only have 6 cores in my own household; i.e. not enough to improve upon the multithreaded application.



Spitballing some interpolation to justify;

* thread creation on windows, 50 ms
* round-trip in same datacenter: 500 ms
* de/serialization : ??

If multi-threading only becomes attractive at an N of 20 million, then a safe factor of around ~20 times (500+??/(50)) would justify distributing the entire process without wasting money.

## Possible improvements

As earlier described *ProcessFindmaxReduceSegments* requires *state* to collect subsegments and reconstitute lines. I’ve solved this through moving segment storage to another service, but this might also be done though some sort of deterministic message-balancing (i.e. implementing a custom ObjectBrowser).

This should be considered as a proof-of-concept, and not an actual implementation. In order to bring this to any sort of realistic environment;

* Signal new calculations and cleanup through topics
* Introduce orchestration of services
* Introduce proper logging aggregation
* Use multi-processing nodes
* Several processing steps (with O(1) complexity) can be packed into one
* Binary serialization

# Learnings

If you need parallel execution, go big or go home.