Parallel Programming with Prim's Algorithm

Erik Lopez
Math/CS Major, Reed College
Portland, Oregon
lopeze@reed.edu

1. ABSTRACT

Parallel Programming can be a very useful way to work through big data sets and get results much quicker than had you used a serial implementation of an algorithm. Not only can it be more efficient but it can also push the architecture of your system to the maximum. I will explore what multithreading and multiprocessing python modules can do for us when using them on an embarrassingly parallel problem and then scale up to Prims, a minimum spanning tree graph algorithm.

2. INTRO

Many modern computers with an Intel i5 or i7 processor or AMD equivalents have at least a quad-core processor which means that we can utilize 4 to 8 different processes with many threads, or if the resources are readily available, switch to using a graphics card unit which can utilize thousands of cores which are made to do many small calculations.

2.1 Embarrassingly Parallel Problems

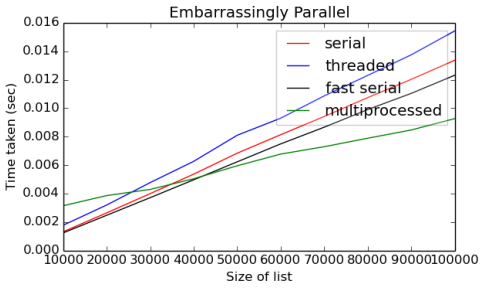
An embarrassingly parallel problem is one in which the work that needs to be completed does not depend on any other data, which we call data independent. A simple example of a data independent problem is if we wanted to add five to each element in a list because we do not need any information about any other index to add five to any one of the index elements. To conceptualize a parallel solution to this problem, we can use a divide and conquer approach where we split the array up by however many threads or processes our architecture allows for, and have each thread work on a subsection of data. Afterwards, each thread finishes and pieces together their portion of problem to give a complete array. This is the basis for how parallel computing works.

2.2 Multithreading and Multiprocessing

Threads are the smallest set of instructions an operating system or kernel can launch. Threads are usually launched by processes, and if launched in the same process they can use shared memory, or if launched by different processes they will have their own memory. Multithreading is the act of letting multiple threads access a problem or data set to complete their set of instructions. This could mean that an array that needs to have its elements doubled could be solved by giving each thread a portion of the array and have each thread individually work on their portion. Multiprocessing is very similar concept except it has threads completing the task and all processes have their own memory, unlike threads that are normally launched by a process and have shared memory.

2.3 Small benchmarks

To test how python's multiprocessing and multithreading modules perform I decided to create an array and add one to each element in the array if it was even as my embarrassingly parallel problem. To test how much faster these versions can run I created two more versions as baselines, one serial and one serial with the container overhead that is in the multithreaded and multiprocessing versions. The serial with overhead version creates containers as if it is going to have an array of threads or processes, then splits up the array into sections to work on, but does all the actual work serially.



The graph depicts the serial with overhead version as serial and serial with no overhead version as fast serial. We can see that fast serial is quicker than serial as expected, but notice that threaded is slower than all of the versions. This came as a surprise given that I thought that we would achieve a faster run time, but did see that multiprocessing did outperform the serial and fast serial versions once the size of the array started to pass a size of 40,000 elements. After further research into the multithreading module and python, I found that the reason why multithreading ran so slowly was that because python has a global interpreter lock(GIL) that locks any computations to be one thread run at any given moment.

Bio331 Fall 2016, Reed College, Portland, OR

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

3. PRIMS ALGORITHM

3.1 Minimum Spanning Tree

A minimum spanning tree(MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices of the original graph with the minimum possible edge weight total. There can be multiple minimum spanning trees for a graph if the totals of each of them are equal to each other.

3.2 What is Prims?

Prim's algorithm is a greedy graph algorithm that finds the minimum spanning tree for a connected, weighted undirected graph. This algorithm is very similar to Kruskal's because it finds a MST, but has a slightly different run time.

4. METHOD

I will show pseudo code for two serial Prim's algorithm, talk about its runtime, then show how the serial implementation can be multithreaded. Afterwards, we will take into consideration how to create accurate tests for recording actual run times of the implementations.

4.1 Prims

For Prims algorithm, it receives as input a graph(V,E), its weights, and an adjacency list.
It starts with an empty forest which is a subset of the input graph, and a nonforest set which will start as the full graph. We place one node into the forest to begin. We then iteratively add a node to the forest if it has an edge with the lowest possible weight for connecting a node in our nonforest to the node in the forest. This is a greedy algorithm because it focuses on making the "best" decision at any given moment to pick an edge to add to the minimal spanning tree.

4.1.1 Primbad

This is my first implementation of Prim's that I created that focuses on adding one node and edge to our forest by iterating through all of the edges of the original graph.

```
while nforest: # while the forest is not complete
  edged = []
  minn = 1000
  for edge in edges:
    if weights[edge[0],edge[1]] < minn:
    # check edges with nodes in forest and nforest
      if edge[0] in forest and edge[1] in nforest:
        minn = weights[edge[0],edge[1]] # smallest edge
        nforout = edge[1] # node to take out
        edged = [edge[0],edge[1]] # save edge position
      if edge[1] in forest and edge[0] in nforest:
        minn = weights[edge[0],edge[1]]
        nforout = edge[0]
        edged = [edge[1], edge[0]]
```

Primsbad goes through the while loop V times $O(V)$, and then iterates through all edges $O(E)$ checking to see if edge[0] is in forest and edge[1] is in nonforest or vice versa where both forest and nonforest are lists so it takes $O(4V)$ to check if edge[0] and edge[1] are in either forest or nonforest. This results in a total run time of $O(V * E * 4V) = O(EV^2)$, which will be the worst run time of Prims I created while still following Prims methodology.

4.1.2 Primbasic

Primsbasic is the default method of doing Prims by using an adjacency list to change the node key values that share an edge to a node in the forest. By having nodes contain key values, it allows us to only search our list of node keys to find the lowest weight edge, and gives a much faster run time of $O(V^2)$.

```
while nonforest: # while the forest is not complete
 minn = 1001

 # search for smallest edge to add to forest
 for anode in nonforest:
   if nonforest[anode][key] < minn:
     minn = nonforest[anode][key]
     minode = anode

 # put our new node and its edge into forest
 forest[minode][parent] = nonforest[minode][parent]
 forest[minode][key] = nonforest[minode][key]
 del nonforest[minode]

 # change the weights of  nodes in nonforest if they have a
 # low weight edge with node in forest and node in nonforest
 for node in adjlist[minode]:
   if (minode, node) in weights and node in nonforest:
     if weights[(minode, node)] < nonforest[node][key]:
       nonforest[node][key] = weights[(minode, node)]
       nonforest[node][parent] = minode

 if (node, minode) in weights and node in nonforest:
   if weights[(node, minode)] < nonforest[node][key]:
     nonforest[node][key] = weights[(node, minode)]
     nonforest[node][parent] = minode

return forest
```

We iterate finding a node to add to our forest by finding the minimum key, then change all the keys of nodes who share an edge to the new node in the forest. This allows us to just look at our key list to find the lowest weight edge to add to our forest. In the section where we have to find the minn and minode, the minimum value edge and node to add to nonforest, we can see that checking each node is data independent, so we can use a multithreaded version that splits up the work of finding their minimum key and node in each section they work on.

4.1.3 Primthreaded

Primsthreaded works exactly like Primsbasic but we split up the work of finding the minimum key and node to add to our forest. We first assign each thread a portion of the work, start each thread to go work on its given portion, then call .join() to wait for all threads to finish their work before continuing. By doing this we make sure that no thread's work gets left out and potentially choose an edge that was not the lowest weight edge. Now that all the work of each thread is there, we serially go through and choose the minimum of the min values and nodes that each thread did.

```
left = 0
right = division
```

```
for i in range(threadnum):
  if i == (lenque - division):
    threadslist[i] = Thread(target = choosesmallest,
                     args = (nodelist[left:],
                     nonforest, xs, i,))
  else:
    threadslist[i] = Thread(target = choosesmallest,
                     args = (nodelist[left:right],
                     nonforest,xs, i,))
  threadslist[i].start()
  left = right
  right += division

  # Wait for all threads to finish
  for i in range(threadnum):
    threadslist[i].join()
```

4.2 Density

For the density of the generated graphs, I decided that density would be measured as a function of V, where V^n would give us the number of edges for the graph. The value of n can be no greater than 2 and no less than 1 because anything more than 2 would mean there would be repeated edges with adding more edges than there are in a complete graph, and a value of n = 1 can give us a connected graph(at best) because there are just enough edges to connect the vertices. A value of n = 1.25 was what I considered to be a sparse graph and n = 1.75 is what I considered a dense graph and were the n values used in my tests.

4.3 Statistics

We need to be able to record how long our algorithm will take so we can benchmark the different implementations of Prim's. To time the implementations we will use timeit, a python module, that will be able to take the execution time of any code it is given. By using timeit.timeit() we will be able to take the execution time of just Prims algorithm itself, not including any of the overhead that comes with creating the containers(which includes vertices, edges, etc). Below is the pseudo-code used to benchmark the different versions of Prims.

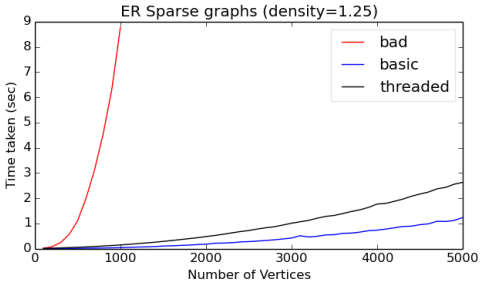
Algorithm 1 Capturing Prim version runtimes

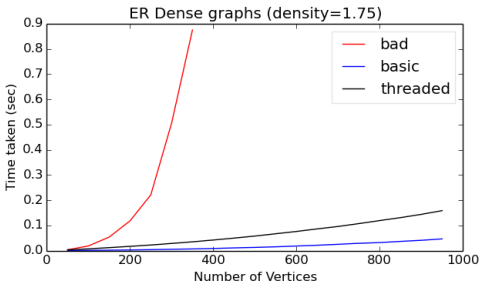
```
t = 0
for x in range(sizeofgraph) do
  V, E, weights, adjlist = makegraph(x, density)

  t = t + timeit(stmt='primsbad(V,E,weihts,adjlist',setup
  = 'V,E,weights,adjlist',iters)
  primsbadfile.write(t)

  t = t + timeit(stmt='primsbasic(V,E,weihts,adjlist',setup
  = 'V,E,weights,adjlist',iters)
  primsbasicfile.write(t)

  t = t + timeit(stmt='primsthreaded(V,E,weihts,adjlist',setup
  = 'V,E,weights,adjlist',iters)
  primsthreadedfile.write(t)
end for
t = t / (totalgraphs * iters)
return t
```

We first generate our graph using the erdos renyi model which fills our containers; V is nodes, E is edges where $E = V^n$ for a given density n, weights is a dictionary of edges->edge weights, and adjlist is the adjacency list. For each generated graph we then run the timeit function to run prims a total of "iters" times on that graph. We will run through this loop depending on the sizes of vertices used for the graphs and the increment counter which was different for each density level. For all of the tests, I decided to use an iters = 100 to make sure that the tests would be finished quickly enough, due to limitations on computing time available. The reason why we didn't choose to test different graphs is that even though generated graphs have different edges and weights, different generated graphs should not run at a different speed. One thing to note is that the number of vertices and edges are set and the weights are uniformly chosen from 1 to 100.

5. RESULTS

Because we have Primsbad with a runtime $O(EV^2)$, Primsbasic with $O(V^2)$, and Primsthreaded with $O((V^2)/threads)$, we expect that Primsbad runs slowest, basic runs next slowest, and primsthreaded runs fastest.





As we can see, Primsbad runtime grows so quickly to where it just explodes in run time and Primsthreaded actually runs slower than Primsbasic. The reason why Primsbad has such a high run time is because E is $V^{1.25}$ for sparse graphs and $V^{1.75}$ for dense graphs so our runtime could also be written in only terms of V and is $O(V^{3.25})$ or $O(V^{3.75})$ depending on the generated graphs we are looking at. In the generated data I cut off calculations for Primsbad after 1000 vertices for sparse graphs and 350 for dense graphs because it became too long to time and would not show any legible results for Primsbasic and Primsthreaded with how stretched the graph would be to plot primsbad's run time. Primsthreaded again runs slower because of GIL as we tested in the simple embarrassingly parallel problem. While I was able to include the multiprocessing version in the embarrassingly parallel problem, I was not able to scale this up to the Prim's algorithm due to the fact that the multiprocessing module "pickles" each argument to the process, which means

that a copy of the arguments are handed to the process not allowing us to modify a list in python as we normally expect it to. This means that when a list is passed to a process and the process tries to change that list, it only changes a copy of that list which makes the python list property of mutability not apply here. There was not enough time to look into shared memory between processes but that would be very interesting to look into.

5.1 Human Interactome

I ran all three implementations of Prim's on a human interactome from "Pathways on Demand: Automated Reconstruction of Human Signaling Networks" that was a protein interaction network with edge weights showing the likelihood of any two nodes having an interaction.[1] What the minimal spanning tree result from Prim's algorithm means is that it will find the set of interactions that are least likely to be interactions in the network because of the minimum weight edge total. Primsbasic ran at 10.3 seconds and Primsthreaded ran at 21.6 seconds, and Primsbad finished after 26 hours.

6. CONCLUSIONS

Parallel programming while very tricky to do correctly can be a very useful and be a satisfying endeavor. The python modules and python itself were not ideal because low level languages such as C have much less overhead and can perform much faster. Before attempting to try any sort of parallel programming, it very useful to take into account the performance gains that can be availed and how to set up a solution otherwise you might end up finding unexpected results such as python's GIL locking us to one thread or that Python "pickles" arguments to processes that don't allow for shared work. As useful as parallel programming is, we could have implemented a binary heap to find the smallest node edge to add to our forest and would reduce our runtime to $O(|E|log|V|)$, which is smaller than parallel programming would have provided us. This project was meant to explore parallel programming even if the run time of doing so is not ideal.

7. REFERENCES

[1] Anna Ritz, Christopher L Poirel, Allison N Tegge, Nicholas Sharp, Kelsey Simmons, Allison Powell, Shiv D Kale, and TM Murali. Pathways on demand: automated reconstruction of human signaling networks. *npj Systems Biology and Applications*, 2:16002, 2016.