Introduction

This Parallel Computer term project is tackling the problem of path counting in a social, directed, acyclic graph (DAG). More specifically, I am looking to compute a function $f_S: V \to \mathbb{N}$ such that, given a subset of vertices $S \subseteq V$, for all $v \in V$, $f_S(v)$ is the number of paths from any vertex in S to vertex v. This problem has applications in various fields such as path finding and recommendation algorithms. We will choose this S uniformly at random of size $\Theta(|V|)$ from the vertices in the graph.

For $|S| = 1$, there exists a $\Theta(|V| + |E|)$ serial algorithm that computes this function. It does so by, for $S = v_o$, first ordering the vertices in the DAG topologically with respect to v_o to get an ordering $O: V \to N$. For this step, ignore any incoming edges to v_o. This sorting step can be done in $\Theta(|V| + |E|)$ time with a twist on depth-first search (DFS). Note that any vertex not discoverable from v_o will not appear in this ordering.

From there, since every vertex is adjacent only to vertices that have a higher ordering than itself, we can simply iterate over the vertices once and make updates to a memo. This first vertex v_o starts with memo[$O(v_o)$] = memo[0] = 1. The ordering for this iteration is such that the vertex v in the ith iteration has $O(v) = i$. The update step for a vertex v is as follows:

For every edge (v, v'): // where $O(v') > O(v)$

memo[$O(v')$] += memo[$O(v)$];

Once you have done this for all the vertices in the ordering, then the function f_S is $f_S(v) =$ memo[$O(v)$] for v in the ordering O, and $f_S(v) = 0$ otherwise. Proof of correctness is omitted, but the idea is as follows:

> We start with the knowledge that the number of paths from v_o to v_o is 1 (which is why we start with memo[0] = 1). Then, for any neighbor v' of the current vertex v, since there are memo[$O(v)$] paths from v_o to v, we can make a path from v_o to v' by concatenating the edge (v, v') to any path from v_o to v. There are precisely memo[$O(v)$] of these paths.

As you may be able to tell from the implementation of this algorithm, this algorithm for $|S| = 1$ can be generalized to general $S' \subseteq V$ by running the algorithm with $S = v$ for every $v \in S'$, then adding all the functions together. This can be written as $f_S(v) = \sum_{x \in S} f_x(v)$.

In parallel computing, this is what one might call "embarrassingly parallel", as we can compute each $f_{\{x\}}$ independently from $f_{\{y\}}$ for all $y \in S \setminus \{x\}$, and the only processor-to-processor communication required to combine the functions is a reduction sum which can be compute with a simple call to a library function. Still, though, as a reference, I have implemented this "embarrassingly parallel" solution to this problem and will occasionally refer to its runtime results.

Parallelization

In order to properly parallelize a solution to this problem, we need to consider the short-comings of the "embarrassingly parallel" solution. The paper "Guided Walk: A Scalable Recommendation Algorithm for Complex Heterogeneous Social Networks" by R. Levin, et al., provides a Pregel algorithm that computes f_S for arbitrary S:

1. **Vertex Program (vprog):** A vertex receives a combined message of all the messages sent to it in the previous super-step (or an initial message if this is the first super-step). A new state is computed based on the current state and the combined message by applying the provided 'vprog'.
2. **Send Messages: (send):** Outgoing edges [2] of vertices that received messages in the current super-step apply the given 'send' message and can optionally send a message via these edge.
3. **Merge Messages:: (merge):** Each vertex that appeared as a destination in the Send Messages step applies a provided 'merge' method on the messages sent to it to combine them into a single unified message.

Types

ID *description* *notations*	vertex id *(id)*
VD *description* *notations*	vertex content tuple of the form *(new-count, old-count)* *(nc, oc)*
A *description* *notations*	message containing the *change-count* Δ

Methods

vprog	(ID, VD, A) : VD $(id, (nc, oc), \Delta) \rightarrow$ if $(\Delta > 0)$ then $(nc + \Delta, nc)$ else (nc, oc)	update PC based on changes done in the previous super-step
send	$((ID, VD), (ID, VD)) : (ID, A))$ $((sid, (snc, soc)), (did, (\_, \_))) \rightarrow$ if $(snc - soc > 0)$ then $(did, (snc - soc)))$	if vprog changed value of its PC then send change via outgoing edges
merge	$A^K : A$ $\{\Delta_1, \Delta_2, \ldots, \Delta_K\} \rightarrow \sum_{k=1}^{K} \Delta_k$	combined change in PC is sum of PC changes

They also describe that:

"The content of each vertex is a pair containing: (a) the PC updated according to current super-step i and (b) its value before the update took place (i.e., the PC value at superstep $j < i$). The latter is needed to keep track of the change done to the PC in the current super-step by vprog. Initially these

values are set to 1, 0 respectively for vertices in S and 0, 0 for all other vertices - this represents the starting values, i.e., in super-step 0. An initial message of 0 is sent to all vertices in super-step 1."

In a Pregel algorithm, the best way to understand the computer design is to think of each vertex as a single processor. The reference paper can be accessed at the following link: https://dl.acm.org/doi/pdf/10.1145/2959100.2959143

Upon a deeper examination, we can see that this algorithm utilizes the same idea as the algorithm for $|S| = 1$ by starting a counter to be 1 at every vertex in S (and 0 elsewhere), and subsequently sending any update to its count to its neighboring vertices. The difference is that this algorithm does this for $|S| \geq 1$, and in doing so lose the benefit of being able to traverse a topological ordering (because there is no reasonable topological ordering for $|S| > 1$). Still, though, this algorithm is likely to run faster than the "embarrassingly parallel" solution because, for example, if some vertex receives more than one message in a super-step, these messages are merged into a single update, which is in-turn distributed as a single message to each of the neighbors. In the "embarrassingly parallel" solution, such an event would occur independently on separate processors, and so each incoming message would be sent to each neighbor. Most of my parallel algorithms that I will discuss are loosely based on this Pregel algorithm. The difference between this algorithm and my parallel algorithms results from a mixture of experimentation and from logical reasoning regarding possible areas for improvement.

We can also recognize a short-coming of this Pregel algorithm design: since each vertex is on its own processor, we do not have any ability to combine messages from the same vertices. One example wherein we might want to combine the messages of two vertices is if two vertices $v, v' \in V$ have the same set of neighbors N.

Suppose v and v' each, after their respective merge steps, are left with an incoming message of m and m', respectively. In the Pregel formulation, since v and v' are on separate processors, each of these processors need to send their corresponding update to each of their neighbors, resulting in $|N|$ messages from each. However, if v and v' were to be on the same processor, we could combine the messages of v and v' into a single message of $(m + m')$ and send this to each of the $|N|$ neighbors.

To see why this maintains correctness, suppose for an arbitrary $n \in N$ that n receives the messages $\{\Delta_1, \Delta_2, \dots, \Delta_k\}$ from its other incoming edges (besides those from v and v'). In the case where each of m and m' were sent separately, n receives $\{\Delta_1, \Delta_2, \dots, \Delta_k, m, m'\}$ and merges them into a single update with value $m + m' + \sum_{i=1}^{k} \Delta_i$. Whereas when only one message of $(m + m')$ is sent from the processor with v and v', n receives a single message $\{\Delta_1, \Delta_2, \dots, \Delta_k, m + m'\}$ and merges them into a single update value $(m + m') + \sum_{i=1}^{k} \Delta_i$. In both cases, the update to the vertex n is identical.

The tradeoff here is that instead of sending $2|N|$ separate messages, we send $|N|$ messages after computing $m + m'$. As an added benefit to the system, now the processor that owns $n \in N$ does not need to compute $(m + m')$ in the merge step because this value has already been supplied. In summary, this is $|N|$ saved additions (spread evenly across $|N|$ processors) and $|N|$ saved messages at the cost of computing one addition. Therefore, it is desirable to combine the messages m, m' into a single value $m + m'$.

Note that this combination reduction can be performed for an arbitrary set of vertices that have the same set of neighbors and are owned by a single processor. This is the basis for my first attempt at non-trivially parallelizing the problem. To obtain a partition of the vertices onto P processors, I pass the undirected version of the graph to METIS which returns a partition. The METIS algorithm takes several hyperparameters that may be worth investigating to see if the partition can drastically change performance, but I have not done this in this investigation.

Given a partition of the vertices onto P processors, for each processor p, we have a set of vertices that p owns V_p. Partition V_p by the edge set of each vertex. That is to say, every vertex with the same set of neighbors is put into the same piece of the partition. Let this partition be given by $\{V_1, V_2, \dots, V_o\}$ where there are no distinct sets of neighbors among all the vertices owned by p. We will refer to these as edge sets. Each of these V_i have an associated set of receivers E_i by which every message sent by any of $v \in V_i$ is received.

Note that these receivers in E_i may be owned by different processors. For this implementation, we need to communicate the subset of receivers to the owning processor of those receivers. This requires that we partition E_i by the processor each receiver is owned by to get a P-way partition $\{E_i^1, E_i^2, \dots, E_i^P\}$ (some of which may be empty if the vertices in V_i have no neighbors on a particular processor). For each of these nonempty sets E_i^q, we communicate the set its associated owner, processor q. With this set, we also include the index from the edge set partition: i. We do this so that, later, when V_i wants to send a message m to its receivers, it need only communicate the message m and edge partition number i for the receiving processor q to know what the message is (m) and which vertices to apply the update to $\left(E_i^q\right)$.

In MPI, a message is also accompanied by a number indicating from which processor the message was received (i.e. the source). Therefore, the receiving processor q is able to differentiate messages from different processors for receiving sets with the same index on the original processor. For example, if processor q received two messages, with form (update, source, index), (m, p, i) and (m', p', i), processor q knows to apply the update m to E_i that it received from processor p, and to apply the update m' to E_i that it received from processor p'.

For the message sending scheme to communicate these edge sets to external processors, I used a standard MPI_Send to communicate the receiving vertices and the index (in the tag argument). For the receiving scheme, one difficulty inherent to this communication is that a processor does not know how many edge sets it will be sent from other processors: it could be as many as $|V| - \left|V_p\right|$ (which happens when, on every other processor, every vertex on that processor has a different set of neighbors), or as few as 0 (if every vertex on the current processor has no edges incoming from a vertex on a different processor, i.e. the subgraph induced by the set of vertices on the current processor is not reachable from any other vertex). Therefore, it is logical to design the edge set receiving scheme such that it can receive an arbitrary number of edge sets. In the following pseudocode, I use Recv(m, s, i, c) to indicate using MPI_Recv to receive a message m of length c with tag i from processor s. I accomplish this receiving scheme as follows:

Use the non-blocking MPI_Iprobe function to check for any incoming messages

// If there is, indeed, an incoming message, MPI_Iprobe returns the source s and tag i of that message

While there is an incoming message:

Use MPI_Get_count to extract the size c of the message // this is the size of the edge set

Recv(m, s, i, c)

//Store the edge set m of size c from processor s where it has an associated edge set index of i

RECEIVERS$[s][i] = m$

Use MPI_Iprobe to check if there are still incoming messages to be received

Preceding this receiving scheme, we must decide how to stall a processor that finished sending its edge sets early and make it wait for the others to finish sending theirs. To this end, I tried a couple different stalling schema. This first scheme that I tried was simply to use MPI_Barrier after sending all the edge sets on every processor in order to guarantee that all edge sets were ready to be received before any processor started checking for incoming messages and receiving them using MPI_Iprobe.

I also tried a non-blocking type of barrier MPI_Ibarrier and an occasional MPI_Test to complete this task. For a given processor, once it was done sending all of its edge sets, it would hit the MPI_Ibarrier and start looking to receive a message (using MPI_Iprobe). If there was a message to receive, it would receive it and store the source, edge set, and edge set index properly for later use (RECEIVERS[source][edges set index] = edge set, just like the above pseudocode). It would continue to receive like this until there were no messages to receive (i.e. MPI_Iprobe returns false), then it would MPI_Test the MPI_Ibarrier again to check if all processors have reached the barrier yet. If not, it repeats this process by calling MPI_Iprobe again. If so, then it goes into one last loop of calling MPI_Iprobe to check for any messages that got sent by the last few processors to reach the MPI_Ibarrier. The potential upside of this receiving scheme is that it permits a processor that has completed all of its MPI_Sends to start receiving any edge sets from other processors, instead of simply waiting for all edge sets on every processor to be sent. The tradeoff here, however, is the additional overhead that comes with non-blocking communication like MPI_Ibarrier and MPI_Test.

After some experimentation, I found that, for this use, the added overhead of the non-blocking communication overcomes the benefit of receiving earlier on the earliest processors. For example, for a graph with $|V| = 500000, K_1 = 6, K_2 = 12$, and $|E| = 37209707$, with P=36 processors, using the receiving scheme with a non-blocking barrier, $14.261 - 11.9979 = 2.2631$ seconds were added to the edge set processing time. As such, my algorithms going forward will adopt the former scheme of receiving communications using a blocking MPI_Barrier.

Now that all edge sets have been communicated and received, the path counting algorithm can begin.

Just like the given Pregel algorithm, we begin by initiating all vertices in S with (1, 0) and others with (0, 0). We can now begin sending updates. As a general overview, the sending scheme stores a running total t_i for each edge set V_i in which we will reduce-sum the updates for each $v \in V_i$. We also have a need to keep track of messages that would be sent from a processor to itself. This occurs when E_i^p is non-empty, i.e. when a vertex has a neighbor that is owned by the same processor. We hope for this to be the usual case as determined by the input partition so as to minimize communication. We will keep

track of these "local messages" with a variable l_i for each edge set. Both t_i and l_i are initialized to 0 for all $i \in \{1, 2, \dots, o\}$.

In the following pseudocode, I use Send$(m, p, i, 1)$ to indicate using MPI_Send to send a message m of length 1 with tag i to processor p. m is length 1 because it is an integer. The sending scheme is as follows:

For each $v \in V_p$:

if v was updated by $m > 0$ in the previous super-step:

get i s.t. $v \in V_i$ // this is v's associated edge set index

$t_i \mathrel{+}= m$

For each $i = 1, 2, \dots, o$ s.t. $t_i > 0$:

For each $q = 1, 2, \dots, P$ s.t. E_i^q is non-empty: // that is to say, edge set V_i has at least one receiver on processor q

If $p == q$: // if this set of receivers is owned by the current processor

$l_i = t_i$

else:

Send$(t_i, q, i, 1)$

This sending scheme reduces all messages for a given edge set V_i into a single sum t_i, then sends t_i to all external processors that have some receiving edge from V_i. In the case that V_i has some received on the local processor, this scheme also stores t_i into l_i for later use in the merge step so that we do not have the processor p Send(t_i, p, i) which is a message to itself. This is a another way in which my methods of parallelization take advantage of having a small number of processors P (as opposed to $P = |V|$ in the Pregel scheme): since we have to put more than one vertex on each processor, we can take advantage by not having to use the MPI communication construct to send messages, but rather just store the would-be messages locally to read from at a later time.

Next, after the send step, we come the merge step. One commonality between my method of parallelization of send step and with the Pregel algorithm's method is that we cannot know precisely how many messages we are going to have to merge in the merge step. One quick and clear way to get an upper bound of such a value is the number of edge sets that we received in the edge set communication preprocessing step (because we will at most receive one message from each incoming edge set). But we are not guaranteed to meet exactly this bound because, for example, an edge set that received no updates in the previous super-step will not send any messages in the subsequent send step (because the update amount in the previous step $m = 0$). Therefore, it is logical to design the merge

step to be able to receive any number of incoming messages. Additionally, in order to properly merge any incoming messages for a vertex $v \in V_p$, we maintain a running total Δ_v for each. Δ_v is initialized to 0 for all $v \in V_p$.

In the following pseudocode, I use Recv$(m, s, i, 1)$ to indicate using MPI_Recv to receive a message m of length 1 with tag i from processor s. m is of length 1 because it is an integer. The merge step goes as follows:

Use the non-blocking MPI_Iprobe function to check for any incoming messages

While there is an incoming message:

Recv$(m, s, i, 1)$

For each receiving vertex $r \in$ RECEIVERS$[s][i]$:

$\Delta_r \mathrel{+}= m$

Use MPI_Iprobe to check if there are still incoming messages to be received

For each $i = 1, 2, \dots, o$ s.t. $l_i > 0$:

For each receiving vertex $r \in$ RECEIVERS$[p][i]$:

$\Delta_r \mathrel{+}= l_i$

Between the send and merge steps, we have the same conundrum as before regarding how to stall a processor that finishes all of its sends earlier than the others. I considered the same stalling schema (blocking barrier vs. non-blocking barrier), and after some experimentation, I found that a blocking barrier for the very first super-step is ideal, whereas a non-blocking barrier is better for the subsequent iterations. For example, for a graph with $|V| = 500000, K_1 = 6, K_2 = 12$, and $|E| = 37209707$, with $P = 36$ processors, utilizing a non-blocking barrier for the later super-steps decreased the computation time for path counting by $53.3402 - 48.7103 = 4.6299$ seconds. From there, utilizing a non-blocking barrier in the very first super-step added $49.2697 - 48.7103 = 0.5594$ seconds to the computation time. Therefore, all future algorithms I discuss will utilize these two results by setting the blocking types for the super-steps accordingly.

After the merge step, we have completed one super-step. As described by the Pregel algorithm, we are to terminate the algorithm if no messages were sent (by any processor) in the most recent super-step. The difficulty lies in that a processor p can only know if it itself sent any messages, or if some other processor sent a message to p. We recognize that the boolean value of whether any processor sent some message is equivalent to $\bigvee_{q=1}^{P}$ (processor q sent some message). We can obtain this value by making use of MPI_Allreduce with the predefined MPI_Operation MPI_LOR. MPI_Allreduce with predefined MPI_Operation MPI_LOR takes in a value b_q from each of the P processors, and returns $\bigvee_{q=1}^{P} b_q$ to all of them.

At a later time, when attempting to speedup the parallel computations, I realized that this MPI_Allreduce operation acts as a blocking barrier for each processor. However, it would be best to use

an optional non-blocking barrier here so that those processors that already know a message was sent can continue on to the next super-step instead of waiting for all processors to continue. As such, I redesign this Boolean OR reduction step as follows:

// b_p holds the Boolean value indicating if processor p sent a message in the most recent super-step

Use MPI_Iallreduce to perform a non-blocking OR reduction, passing parameter b_p, and storing into B

If ($\neg b_p$) :

Use MPI_Wait to wait for the MPI_Iallreduce operation to complete

If ($b_p \vee B$):

Continue to the next super-step

Else:

Terminate the path counting algorithm on all processors

It is clear to see that the termination condition for the algorithm is that b_q = B = false on all processors. This happens if and only if b_q = false for all $q \in \{1, 2, \ldots, P\}$ (meaning that every processor sent no messages), in which case all processors will call MPI_Wait and wait for the reduction operation to complete before proceeding to the final else clause.

This reduction scheme using a non-blocking reduction proved to be an improvement over the reduction scheme that simply used MPI_Allreduce. For example, in a graph with $|V| = 100000, K_1 = 6, K_2 = 12$, and $|E| = 7224554$, with $P = 32$, using the non-blocking reduction scheme improved the path count computation time by $6.41903 - 0.266332 = 6.152698$ seconds which is a tremendous 96% speedup for this test case.

After completing the reduction operation, if we continue into the next super-step, we compute what corresponds to the vprog function in the Pregel algorithm. There is no processor-to-processor communication in this algorithm. It just simply completes the update for each vertex $v \in V_p$ given Δ_v.

Improving on the Parallel Algorithm

After having completed the "Vanilla edge sets" algorithm described above, I thought for a long while about what optimizations I could make. After stepping through the algorithm a few times, I found a possible area for improvement: the above algorithm, in the step where we compute the edge sets, we first partition V_p by their sets of neighbors, then partition each of those sets of neighbors by the processors they are owned by. I realized that it makes more sense to partition by the owning processor first, then partition by sets of receivers. This change makes sense because a vertex v on processor p sends its message once to each processor with a neighbor already, so if there is another vertex on the p with the same set of receivers on a particular processor, we can combine these messages to only send one while maintaining correctness.

Upon further inspection, it is reasonable to expect that this will increase the number of distinct edge sets o because the edge sets in the original partitioning method are all split into at most p partitions, each of which is an edge set in its own right. Hence, we expect that the edge set preprocessing time will increase since there are more edge sets to consider when first matching up all vertices' partitioned-by-P edge sets (there are more sets per vertex to look for, and more sets to look through from previous vertices). It also reasonable to expect the cardinality of each edge set (the number of times the edge set appears in some vertex's partition) to increase since all edge sets are more finely-grained with fewer elements and so are more likely to be the same as another.

The partitioning method has some nice properties such as: if two vertices on the same processor have the same set of neighbors except for one has one additional neighbor, then in the original partitioning method, these vertices would be placed into different edge sets which fails to take advantage of the near-perfect similarity. But the above proposed method would assign these vertices all of the same set numbers except for 1, so almost all of the work we do for one can be applied to both. This is a desirable trait to have as it allows for more granularity in similarity for vertices to share work, whereas the original method is a binary measure of similarity (they have the same neighbors or they do not). I will refer to this variant of the algorithm as the P-ways edge sets algorithm.

The tradeoff here in using this partitioning method is that while we get to combine more messages because edge set cardinality is increased (thereby decreasing communication), whenever a message is received by a vertex, we now have to distribute that message to many edge sets instead of just one edge set like in the previous formulation. In particular, a slowdown could occur in the new version of the sending scheme. In the following pseudocode, EdgeSetIdx$[v]$ is the set of edge set indices associated with vertex v, and EdgeSetIdxToOwnerOfReceivers$[i]$ is the owner of the receiving vertices of the edge set with index i on the current processor p.

For each $v \in V_p$:

if v was updated by $m > 0$ in the previous super-step:

for all $i \in$ EdgeSetIdx$[v]$:

$t_i \mathrel{+}= m$

For each $i = 1, 2, \dots, o$ s.t. $t_i > 0$:

$q =$ EdgeSetIdxToOwnerOfReceivers$[i]$

If $p == q$: // if this set of receivers is owned by the current processor

$l_i = t_i$

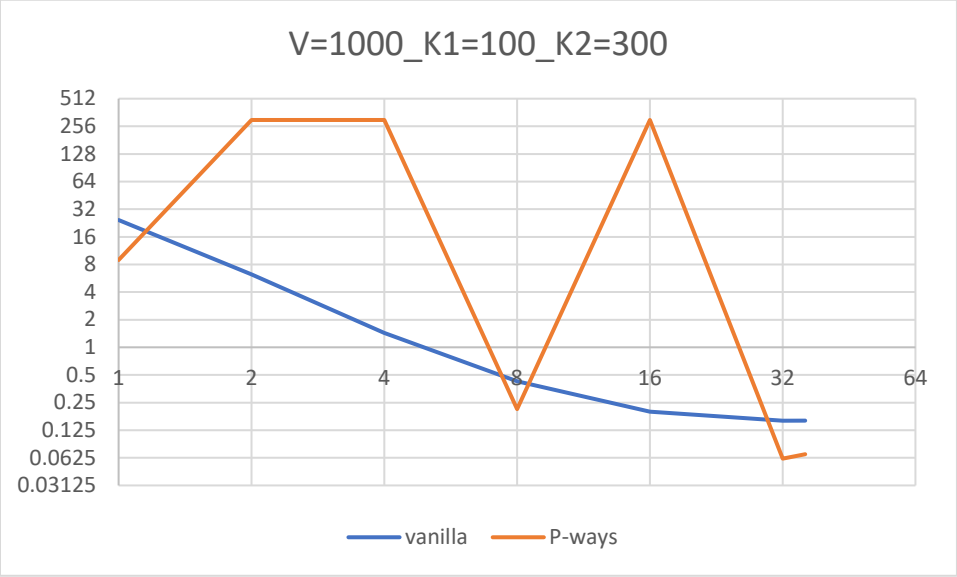
else:

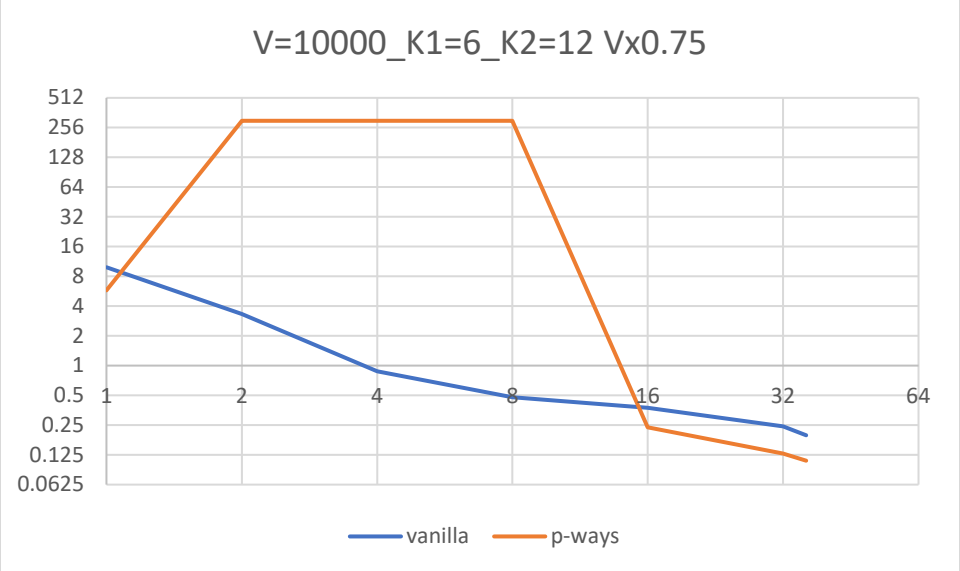
Send$(t_i, q, i, 1)$

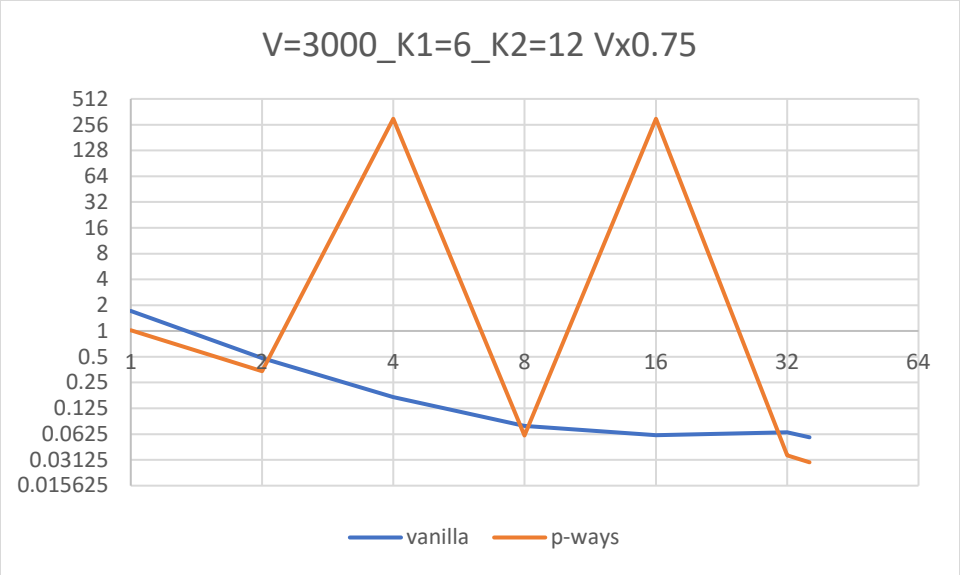
In the first for loop, each vertex's update m has to be added to t_i for all $i \in$ EdgeSetIdx$[v]$ instead of just a single t_i like before. Therefore, while would typically expect runtime to improve by using this partitioning method, it is feasible that the runtime can become worse. This would happen in cases

where the number of sets per vertex (which is $\leq p$) is very large compared to the number of edge set intersections (low cardinality), resulting in minimal benefit to combining messages while suffering in having to update multiple edge sets. For very large P, we do not expect to see this phenomenon because the edge sets are very finely grained and so we expect many intersections (high cardinality). And for very small P, we also do not expect to observe this because the number of edge sets per vertex is still small (because it is no more than p). However, for mid-range P, it is feasible to observe this phenomenon.

Here are a few graphs that very clearly demonstrate this phenomenon in practice:







It is worth noting that all values at 300 seconds resulted from a job timeout. As can be seen, this partitioning method is usually an improvement, but the runtimes can be unreliable. It can happen that some update on a processor takes an exceptionally long time, causing the whole program to go over time. I will refer to this as getting "stuck", but know that the algorithm is not actually stuck, but rather forced to finish a long-winded processor.

As I worked even more to try to shave off time from the parallel algorithm, I realized that a likely holdup on runtime improvements is the communication volume. In particular, I realized that the number of single-data communications is very large, and each communication brings with it additional overhead. Hence, I redesigned the message-passing scheme so that, in a particular super-step, all messages from processor p to processor q are passed with a single communication by communicating 2 times the number of messages I would have sent previously. To do so, in the send function, whenever I would have sent a message $(m, q, i, 1)$ in the original scheme, I instead append both m and i to an array. After all message-tag pairs have been appended, I then send the entire array as one message (array, q, 0, numMessages $\times$ 2). This redesigned messaging scheme also forces a redesign of the receiving scheme in the merge function to allow for arbitrarily long messages of even length. After some experimentation, my hypothesis proved correct when I found that for a graph with $|V| = 500000, K_1 = 6, K_2 = 12$, and $|E| = 37209707$, with $P = 36$ processors, the runtime decreased by $68.8411 - 53.3402 = 15.5009$ seconds which is a 23% decrease.

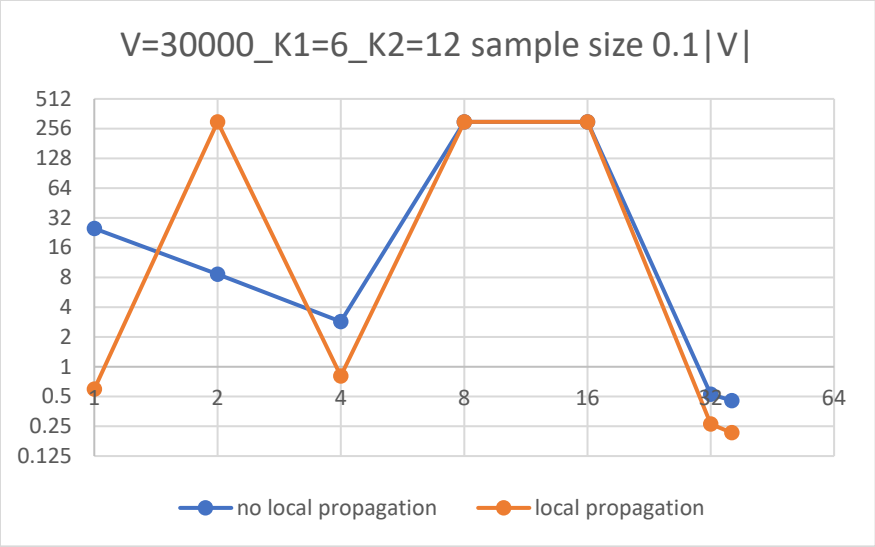
At this point, I was running low on ideas for optimizations, so I started analyzing the original Pregel algorithm to see if there were any shortcuts I could take. It was at this point that I realized that the barrier between the send operation and the merge operation is not required! Even with the barrier gone, it is clear to see that correctness is maintained in that:

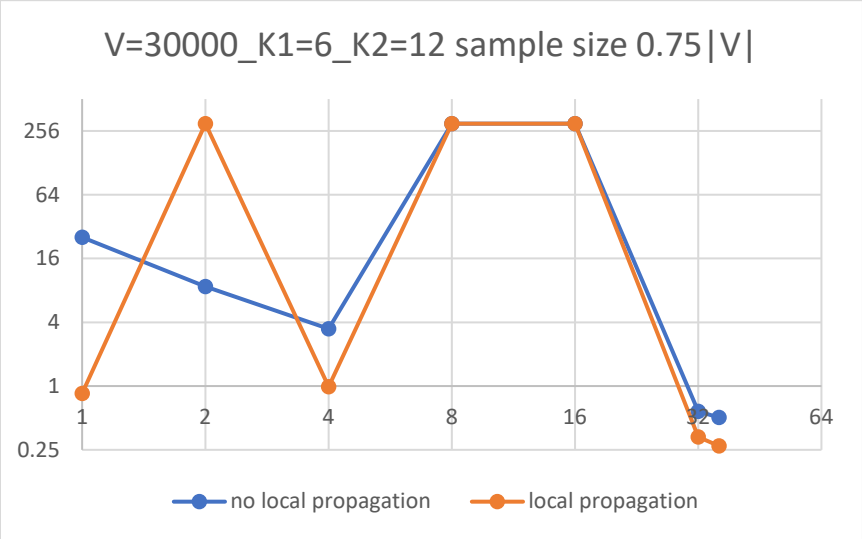
If a processor p does not receive a message sent to it by q because p finished its sends and receives before q sent the message, then p will fail to take into account that messages updates and distribute

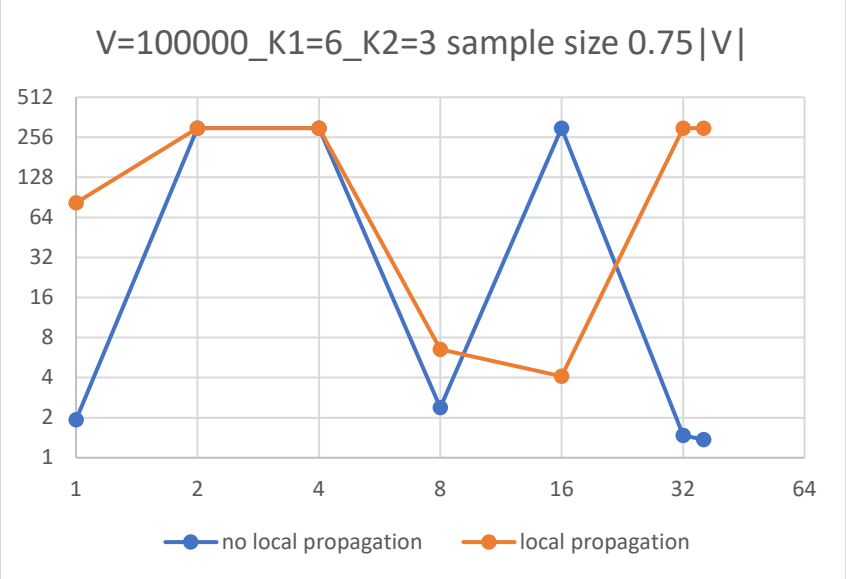
them accordingly. That is, until the next super-step (the next super-step is guaranteed because q sent a message) when p probes for incoming messages again. Even though p received this update in the wrong super-step, this is okay because when p receives it later, the same updates will be made and the updated will be distributed in the same manner, perhaps merged with message from different super-steps, but this is okay as addition is associative and, eventually, the message will be distributed everywhere it needs to go. Hence, I made an effort to improve the runtime by removing the barrier between the send and merge operations. The results, however, were surprising:

As can be seen in the graphs below, removing the barrier did not make much of a change for these graphs. Usually, the time increased after removing the barrier, but there are a few instances where removing the barrier did end up decreasing the runtime. As for whether or not this change should be adopted, it most certainly should not because there are no perceivable gains in the given examples, and for a graph with $|V| = 500000, K_1 = 6, K_2 = 12$, and $|E| = 37209707$, the algorithm was not able to terminate in the 5-minute time limit with the barrier removed, but was able to terminate in about a minute with the barrier present. Therefore, we have seen that this change offers no perceivable runtime benefit at the risk of a very large increase in runtime, so I do not adopt this change into my algorithms. After stepping through the algorithm in debug mode, I've realized that the barrier not only serves as a guarantee that all messages get received in the same super-step, but also serves to gain a larger benefit from the merge function by forcing that all available messages are merged each time. This corresponds to more messages included in each update, corresponding to fewer updates being sent, corresponding to less communication.

The last change to my parallel algorithm that I made to improve runtime was to take further advantage of having more than one vertex on a processor. I modified the P-way edge set partition algorithm so that when we are keeping track of the local messages to be sent to local vertices, instead of compiling them all into some buffer and merging them with the rest of the external messages in the merge step, we can instead propagate the local message to the local vertices, causing us to update their respective edge set update values (some of which are local), and we can recurse! We continue to recurse until all messages are outgoing and there are no local messages (this is guaranteed to happen because the graph is a DAG). Only at this point do we actually send the outgoing updates to their respective processors. One potential downside of this change is that if some processor has lots of local updates to process and another has very few, then the latter may find itself waiting at the subsequent barrier for the first one to finish which could result in an overall slowdown. I will refer to this update algorithm as P-ways edge sets with local propagation.


V=30000_K1=6_K2=12 sample size 0.1|V|
512
256
128
64
32
16
8
4
2
1
0.5
0.25
0.125
1
2
4
8
16
32
64
no local propagation
local propagation


V=30000_K1=6_K2=12 sample size 0.75|V|
256
64
16
4
1
0.25
1
2
4
8
16
32
64
no local propagation
local propagation


V=100000_K1=6_K2=3 sample size 0.75|V|
512
256
128
64
32
16
8
4
2
1
1
2
4
8
16
32
64
no local propagation
local propagation

This recursion step on handling all local messages will cause many more would-be outgoing messages into a single outgoing message, thereby reducing total communication.

As I was running some timing runs for this algorithm and comparing it against my fastest serial algorithm, I quickly realized that it was, in fact, not *the* fastest serial algorithm when I saw that this parallel algorithm was able to outperform the serial by a good deal. Upon further inspection, the local message propagation method outlined above suffices as a serial algorithm on its own. This serial algorithm is very fast, as you can see in the next section.

Scalability

The algorithm has 2 inputs: the social DAG and a subset of vertices S. After discussing with the professor way to generate social graphs that fit this problem, we settled on the following:

The social DAG has 3 parameters: the number of vertices $|V|$, and 2 associativity constants K_1 and K_2. Start with a complete DAG with $K_1 + 1$ vertices.

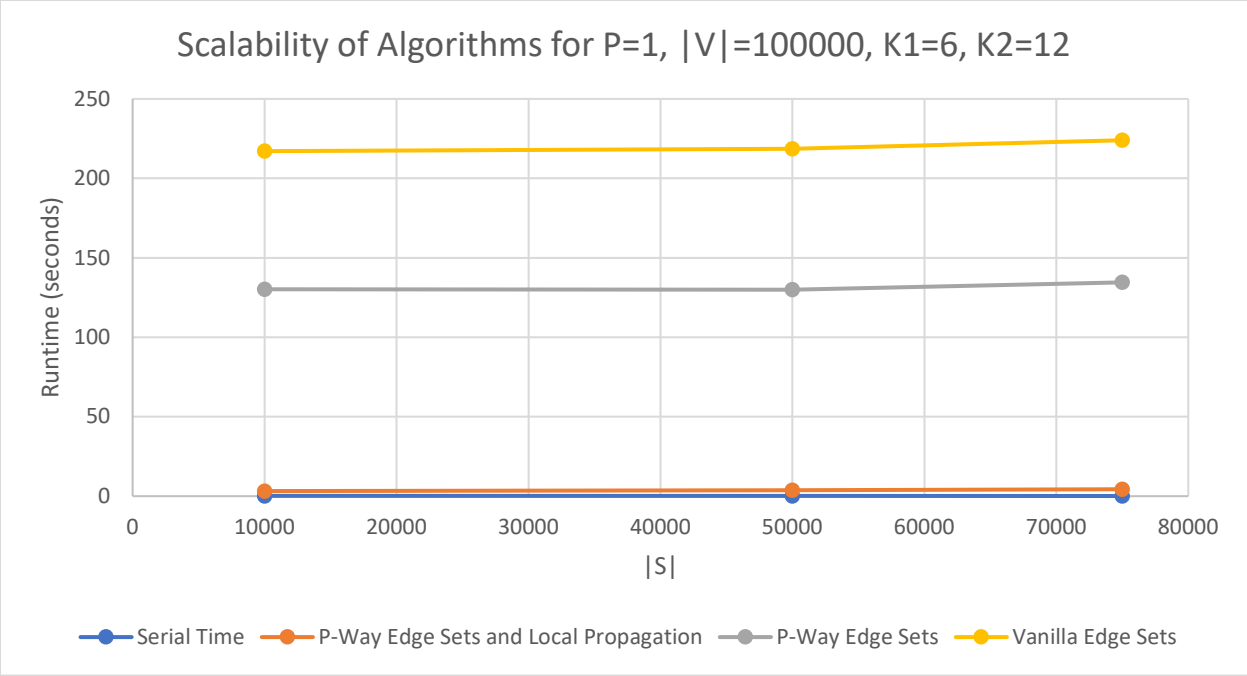
For every vertex v you add from $K_1 + 2$ to $|V|$, connect v to a random sample of K_1 of the vertices already present.

For each of those K_1 vertices v just connected to, connect v to a random sample of K_2 of their neighbors.

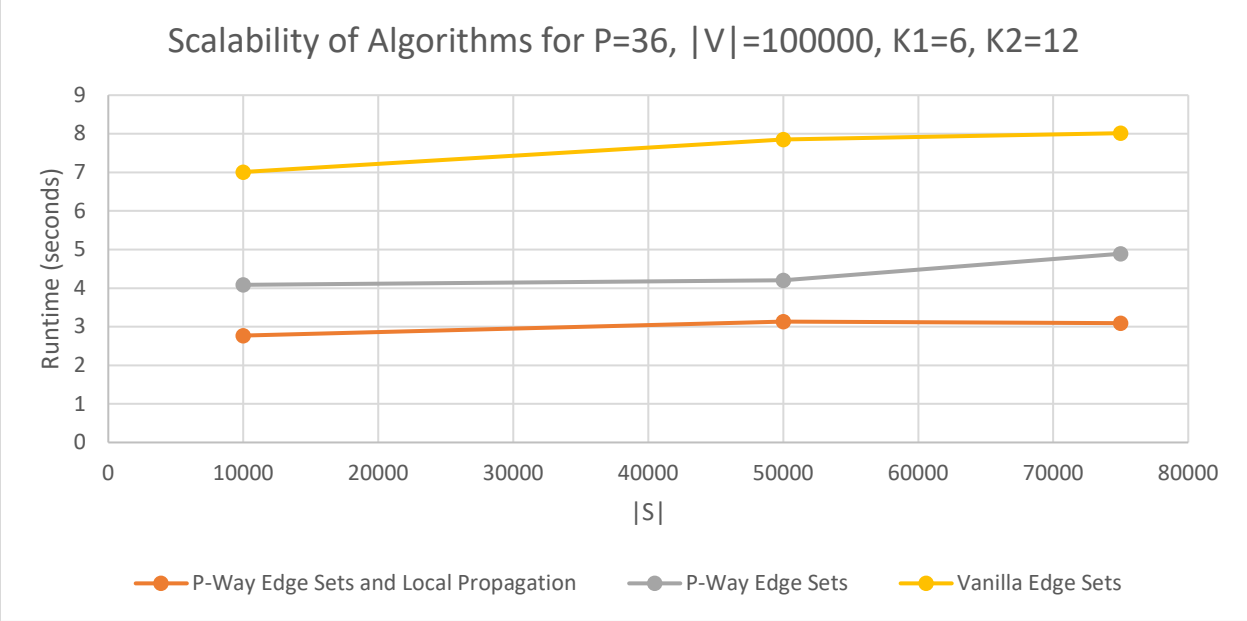
In this way, we generate a social DAG where each vertex has no more than $K_1 + K_2 \times K_1 = K_1(K_2 + 1)$ neighbors.

Hence, the graph has no more than $|V|K_1(K_2 + 1)$ edges.

First, I will reason about the scalability of the proposed algorithms w.r.t. the size of S.

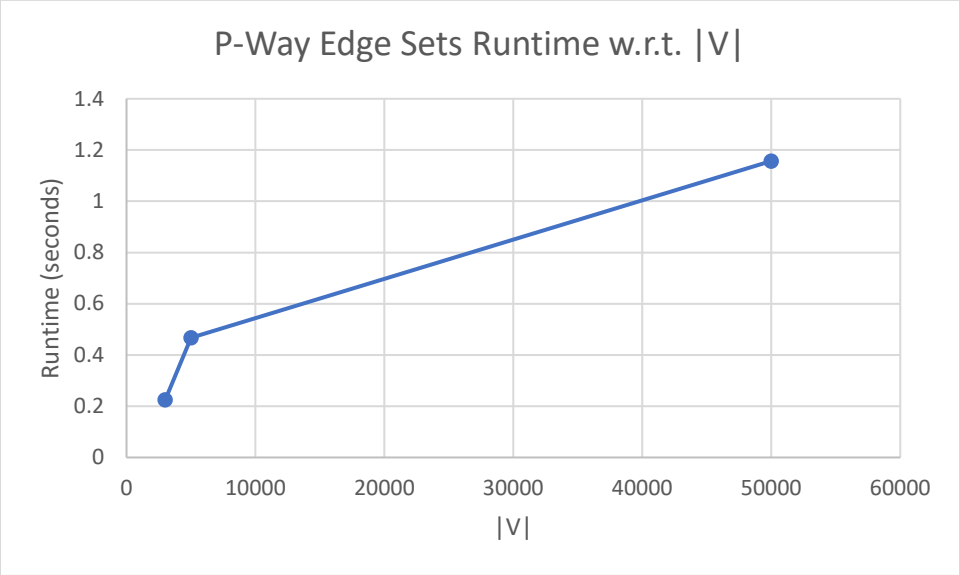


As seen in the graph above, it is clear that, for $P = 1$, the runtimes of these solutions are mostly independent of $|S|$. This makes sense because the number of times each algorithm loops is determined by the length of the longest path starting in S (this is because the 1 starting on that node at the start of the longest path has to traverse the entire path before the algorithm can terminate). Adding more vertices to S just increases the likelihood that that vertex is the start of an even longer path, but this is not likely to increase the length of the longest path from S by much after adding just a few vertices. There does seem to be a slight upward trend which can be explained by one of the added vertices make the longest path from S slightly longer, but I would still consider this trend to be mostly constant. This trend is maintained for larger P as well.

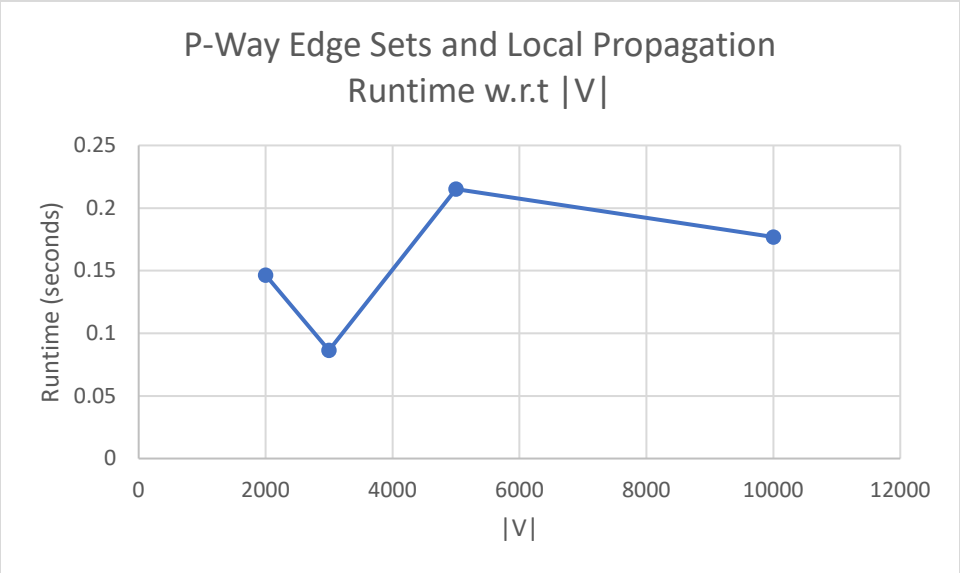


Henceforth, my analyses will not be concerned with the size of S because it appears to be irrelevant to the runtime.

Now, to reason about the scaling w.r.t. input parameter $|V|$, I compare the runtimes for graphs that were generated have similar values for $|E|$, regardless of the inputs K_1 and K_2, since $|V| + |E|$ is what determines the actual size of the graph input.

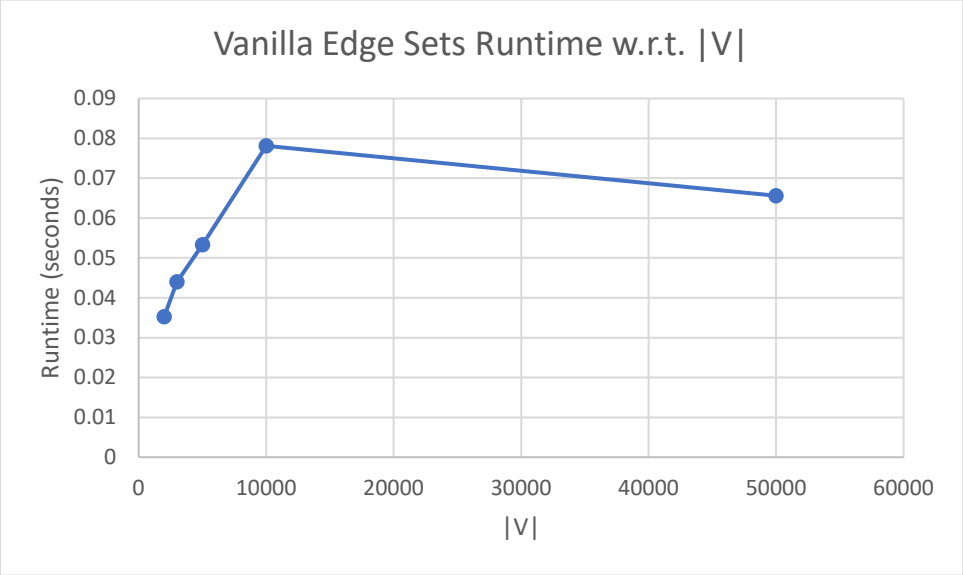


Note that this graph was obtained using $P = 4$. The P-way edge sets algorithm's runtime appears to grow approximately logarithmically with $|V|$. One likely explanation of decreasing slope as $|V|$ increase is that for larger $|V|$, each added vertex is unlikely to add much more work by introducing new edge sets, and so the rate of increase of the runtime will slow as $|V|$ increases.



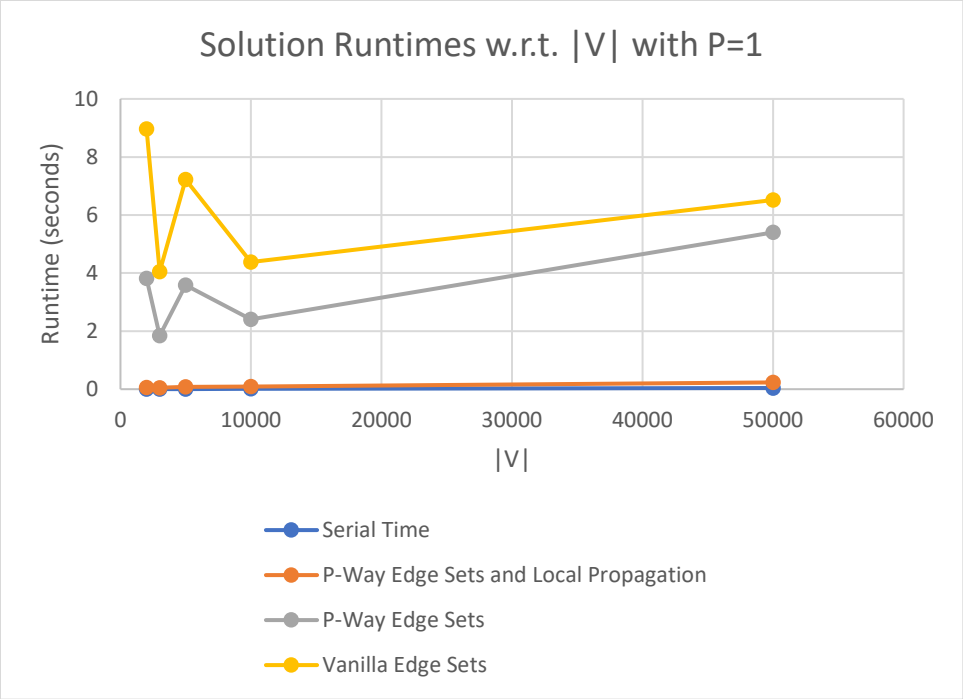
Note that this graph was obtained using $P = 4$. While P-way edge sets algorithm's runtime appears to follow an upward trend as $|V|$ increases, I am unable to propose any possible mathematical relationship

due to the high variance. This high variance is likely due to the occasional processor getting "stuck" in the local propagation phase.



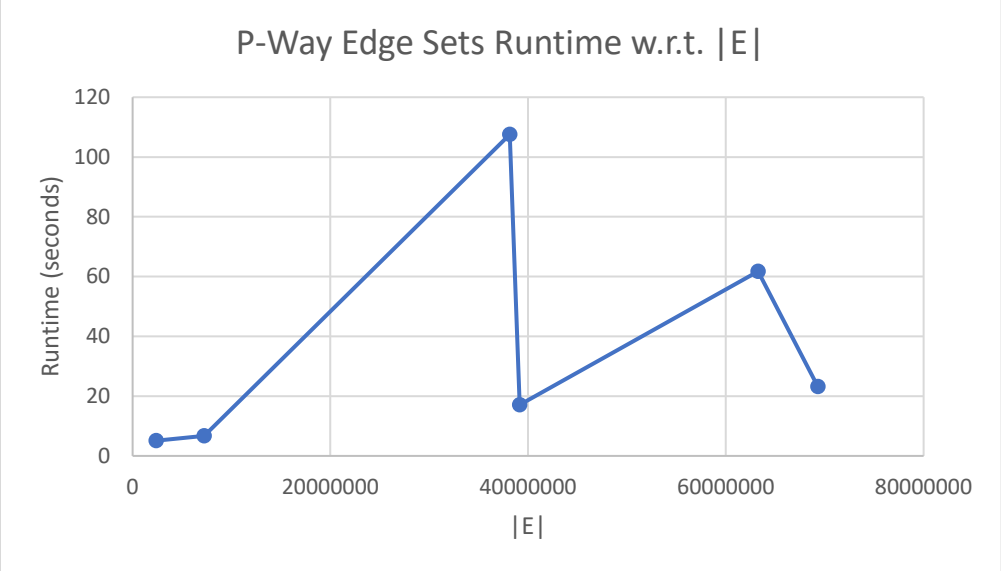
Note that this graph was obtained using $P = 36$. Vanilla edge sets algorithm's runtime also appears to grow linearly with $|V|$ with the exception of the outlier at $|V| = 50000$. It makes sense for the runtime to increase linearly with $|V|$ because each add vertex is likely to add a new edge set to the scenario, adding a constant amount of new work to the problem. The outlier at $|V| = 50000$ is likely a lucky occurrence wherein the partition and sample for this particular arrangement of edges was particularly good.

We can also examine how the runtimes are affected as a function of $|V|$ in the case when $P = 1$:

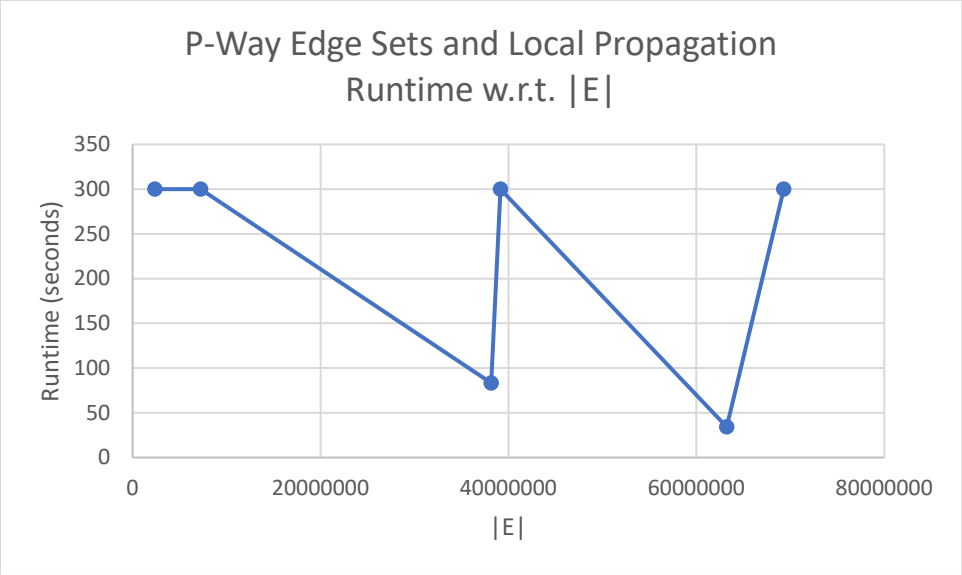


We can see that all of the runtimes grow approximately linearly with $|V|$ (which makes sense because everything has to be processed serially since $P = 1$), but the P-way edge sets and Vanilla edge sets algorithms exhibit a high variance for low $|V|$. In fact, they all exhibit the same variances (i.e. both quickly decrease then quickly increase, then quickly decrease again) which leads us to the conclude that this variance is likely a result of the hardnesses of the associated problems that were solved to obtain these data points (i.e. the problem with |V|=3000 seems to have been particularly easy).

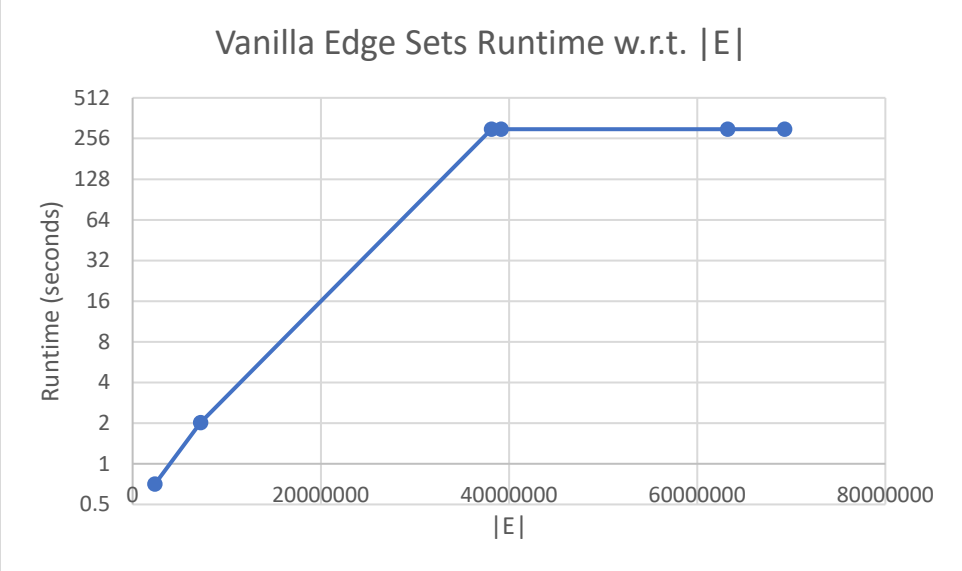
After examining the scalability w.r.t. $|V|$, we now move to examine the same w.r.t. $|E|$ (using $P = 16$):



Here, the runtime P-way edge sets runtime exhibits a strange behavior with no apparent trend. Perhaps this is indicative of the associated hardnesses of the problem that corresponds to each data point.

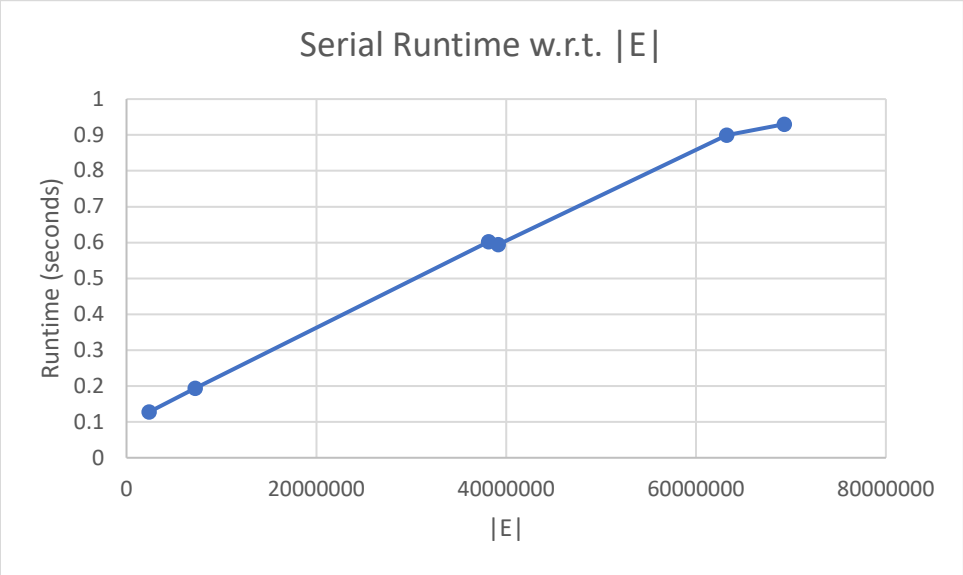


As can be seen above, the P-way edge sets and local propagation algorithm exhibits the exact opposite behavior of the P-way edge sets (without local propagation) algorithm. This is very revealing about a relationship between the two approaches: for any graph for which the local propagation approach will do well, a strictly P-way edge sets approach is likely to do poorly, and vice versa! What this means is that these two approaches are complementary in some sense, each taking advantage of opposite properties about a given graph so that when one does well, the other may suffer. Exactly what these opposite properties could be is an excellent topic for future work.



Notice the logarithmic Runtime scale in the above graphic. The right-most data points on this line resulted from algorithms that timed out due to the 5-minute time limit. Using the 3 left-most data points, we can see that the logarithm of the runtime tends to grow linearly with $|E|$ (i.e. runtime grows

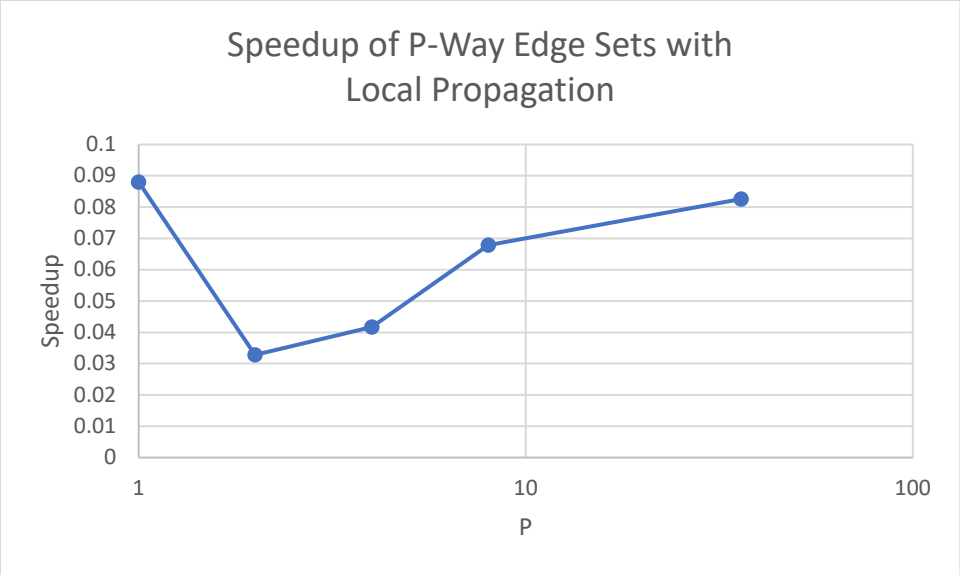
exponentially in $|E|$). This makes sense because every added edge is likely to make a new edge set which can add $O(P)$ work (because this new edge set has to communicate with up to $P-1$ other processors), and each receiving vertex from this edge set has to process an additional update in each iteration. This additional update in turn is likely to instigate an update for their neighbors, then to their neighbors, and so on a which can take $O(|E|)$ time propagate to all eventual receivers. This exponential growth in runtime causes the program to time out for relatively small $|E|$.



As expected, the serial algorithms' runtime grown very linearly with $|E|$ which makes sense because each new edge instigates only one additional message to be sent from the out-side of the new edge, after which the algorithm can process as normal.

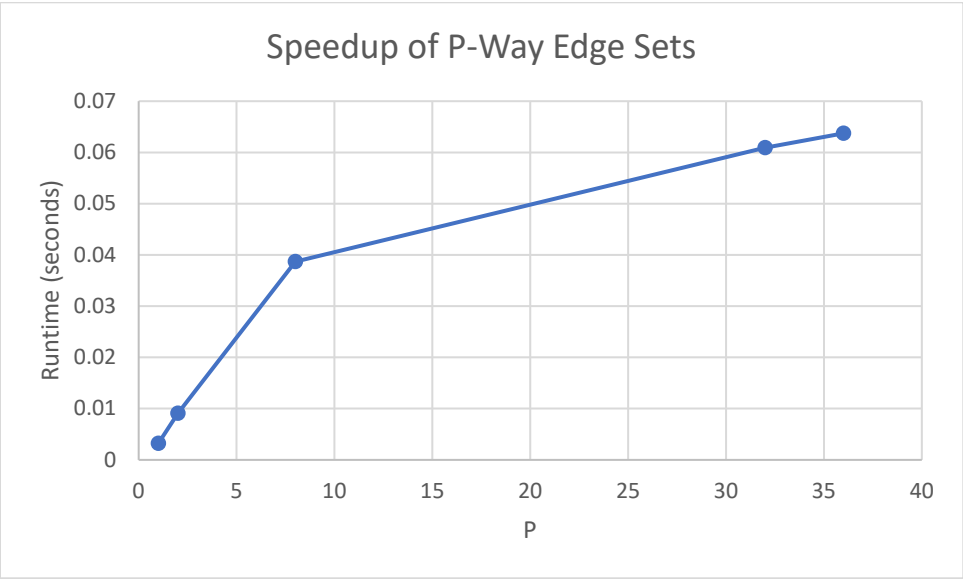
Finally, I example the trend in runtime w.r.t. the number of processors P. These data points were all collected using a graph with parameters $|V| = 3000, K_1 = 6, K_2 = 12$, resulting in $|E| = 178296$. This relatively small graph was selected so as to make termination very likely for every algorithm, and because, after examining the data, the corresponding trends were very apparent. I tried to collect more data points using other, bigger graphs, but most of the jobs timed out, and so no trend could be inferred.

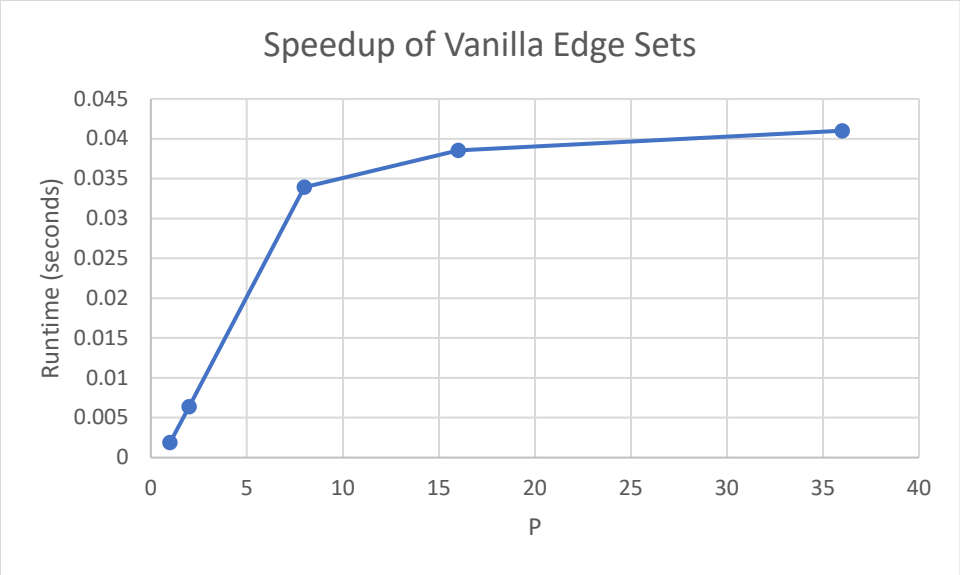
The fastest serial algorithm I could make for this problem solved this instance in 0.00327454 seconds. I display the resulting graphs of Speedup for analysis:



Notice the logarithmic P scale. Here, $P = 1$ appears to be an outlier. Notably, $P = 1$ essentially corresponds to the fast serial algorithm which explains why this datapoint appears to be irregularly fast. As for $P > 1$, the speedup appears to be linear with the logarithm of P (that is to say, the speedup grows logarithmically with P). This makes sense because each additional processor relieves the vertex-load of the others by some factor (assuming this processor takes on its fair share of vertices) by relieving them of the duty of updating those taken vertices, which is a disproportionally larger amount of work.

We see similar trends for each of the other algorithms as well:





Conclusion

To conclude, after much effort and experimentation, while we were unable to make any parallel algorithm that scales nicely, nor achieve any speedup of greater than 1 (for $P \leq 36$, as enforced by the hardware), we are left with good insight into how some of these graph algorithms interact and respond to varying social graph inputs.

On a happier note, we are left with a very fast serial algorithm for solving this problem that beats out (by a long shot) any parallel algorithm I could come up with. I have been unable to find evidence of this algorithm anywhere online, so perhaps this fast serial algorithm is my main contribution to the problem of computing a path-count function from a set of source vertices in a DAG.

Future work on this topic might include investigating the role of structures in the graph in determining what can make a parallel algorithm get "stuck" as we have seen, and also what graph properties determine which parallel approach would work best. It also might prove fruitful to run these algorithms on $P > 36$ processors to see when and where the speedup levels off (and if it can ever achieve a speedup of greater than 1).