Implementation of Leader Election Algorithm Using Sense of Direction

PEIXIN LIU, UIN: 128002261
GUANDA LI, UIN: 626008680
YIRAN HUANG, UIN: 827008769

1 INTRODUCTION

Leader election is a fundamental issue in distributed computing and has been studied in various computation models. A leader can simplify coordination among processors and is helpful in achieving fault tolerance and saving resources. It also provides a way for symmetry-breaking problems.

In the leader election problem, there are n processors in the network, each having a unique identity. Initially, all processors are passive and do not know the identity of any other processor. An arbitrary subset of processors, called the candidates, wake up spontaneously and start the protocol. On the termination of the protocol, exactly one processor announces itself the leader.

In our class, we restrict our attention to the situation in which the topology of the system is an oriented ring. We have learned two basic algorithms for asynchronous rings.

The first one requires $O(n^2)$ messages. In this algorithm, each processor sends a message with its identifier to its left neighbor and then waits for messages from its right neighbor. When the processor receives a message, it checks the identifier of the message. If the identifier is greater than its own identifier, it forwards the message to the left; otherwise, it will not forward the message. If a processor receives a message with its own identifier, it declares itself to be a leader by sending a termination message to its left neighbor and terminating as a leader. A processor that receives a termination message forwards it to the left, terminating as a non-leader.

The second algorithm is based on the first one but is more efficient, which reduces the upper bound of the number of messages from $O(n^2)$ to $O(nlogn)$. In this algorithm, instead of sending messages in one direction, a processor will send $\langle probe \rangle$ messages to both directions. If the processor receiving the probe has an identifier bigger than that of the probe, it will swallow the probe, and if not, it will forward it. A processor that receives its own $\langle probe \rangle$ message sends a termination message around the ring and terminates as a leader.

In our project, we implement the algorithm [1] of electing a leader in an asynchronous complete network, where each pair of processors is connected by a bidirectional link. Previous papers show several algorithms in which a processor captures at least a majority of processors before it declares itself as the leader. In this paper, the authors observe that in the presence of sense of direction, a processor does not have to capture a majority of processors in order to be elected the leader and they use this idea to obtain a protocol which requires $O(n)$ messages and $O(log(n))$ time. This algorithm is message optimal and achieves a significant improvement over existing algorithms for this problem by then. We verify the message complexity and time complexity in our experiment. It is worth mentioning that, we deeply appreciate advises from Professor Jennifer Welch and DistAlgo tips written by Gerald Hu [2], which helps us greatly on our work.

2 ALGORITHM

This algorithm is a work of Gurdip Singh, *Efficient leader election using sense of direction* [1]. It considers the problem of electing a leader in an asynchronous complete network, where each pair of processors is connected by a bidirectional link, with a sense of direction. The network has a sense of direction if there exists a directed Hamiltonian cycle and each edge incident at a processor is labeled with the distance of the processor at the other end of this edge along this Hamiltonian cycle. With the presence of a sense of direction, a processor can declare itself as the leader without capturing a majority of processors. Using this idea, this algorithm achieves time complexity of $O(log(n))$ and message complexity of $O(n)$.

First, we assume that n is a power of 2. Let $k = n/2^{\lceil loglog(n) \rceil}$. The algorithm proceeds in two phases. In the first phase, we reduce the number of candidates to at most k. This is accomplished by logically dividing the ring into k interleaved subrings, each with $log(n)$ processors, and electing at most one leader in each subring. The subring which node i belongs to consists of nodes in the set $i[0], i[k], i[2k], \ldots, i[n-k]$. We use R_i to denote the subring of i. To elect itself the leader of its subring, a processor must capture all processors in its subring. In the second phase of the algorithm, the remaining candidates compete with one another to elect a single leader. All subrings have a representative among the set of nodes which are within distance $n/log(n) - 1$ from i and each node knows the identity of the candidate in its subring. Node i uses these representative nodes to capture other candidates(If i attempts to capture j, j informs i of its candidate). Using this process, the second phase elects a single leader among the leaders of the subrings. Each processor needs to maintain the following variables.

- $state_i$: Initially, $state_i$ is passive. When i wakes up spontaneously to initiate the algorithm, it changes $state_i$ to the candidate. If node i has been captured then $state_i = captured$. If i has been elected then $state_i$ is $elected$.

- $level_i$: Initially, $level_i$ is 0. At any other time, $level_i$ denotes the number of processors i has captured in the first phase.

- $step_i$: Initially, $step_i$ is 1. The election in the second phase proceeds in a sequence of steps. For a candidate, $step_i$ denotes the current step number of i in the second phase.

- $owner_i$: For a candidate processor i, $owner_i = 0$. For a captured processor, $i[owner_i]$ denotes the edge leading to the leader of R_i.

- $phase_i$: When i wakes up (either spontaneously or on receiving a message), it sets $phase_i$ to 1. $phase_i$ is 2 if i has entered the second phase as a candidate or has received a message of the second phase.

The following is the list of messages used in the algorithm.

- $CAPTURE(phase_i, level_i/step_i, i)$: This message is sent by i to capture another node. If $phase_i$ is 1 then the second argument is $level_i$, else it is $step_i$.

- $INFORM(dist)$: In the second phase, a node uses this message to inform the sender of a CAPTURE message regarding the distance to its owner.

- *ACCEPT*($phase_i, level_i$): This message is sent in response to a *CAPTURE* message indicates that i was captured as a result of the *CAPTURE* message. The second argument is omitting if $phase_i$ is 2.

- *OWNER*(id): This message is sent by the leader to all members of its subring at the end of the first phase to announce the fact that it has been elected in its subring.

- *ACK*: This message is sent to acknowledge the receipt of the *OWNER* message.

- *ELECTED*: This message is sent at the end of the second phase by a node to announce the fact that it has been elected as the leader in the network.

Algorithm 1 Leader Election Algorithm Using Sense of Direction:

Algorithm for a candidate node j:

1: $state := candidate$; $level := 0$; $phase := 1$; $owner := 0$
2: **while** $state = candidate$ and $phase = 1$ **do**
3: **if** $level \leq n/k$ **then**
4: send $OWNER(j)$ to all nodes in R_j; $Response := 0$
5: **else**
6: send $CAPTURE(1, level, j)$ to $j[(level + 1) * k]$
7: **end if**
8: $Received := false$
9: **while** $state = candidate$ and $phase = 1$ and $\neg Received$ **do**
10: receive M on edge $j[e]$
11: Case M of
12: ***CAPTURE***$(1, l, i)$:
13: if $(level, j) < (l, i)$ then
14: send $ACCEPT(level)$ over $j[e]$; $state := captured$
15: ***ACCEPT***$(1, l)$:
16: $level := level + l + 1$; $Received = true$
17: ***ACK***(j):
18: $Response := Response + 1$
19: if $Response = n/k - 1$ then $phase := 2$
20: ***CAPTURE***$(2, step, i)$:
21: $state := captured$; $phase := 2$
22: send $ACCEPT(2, i)$ over$j[e]$
23: ***ELECTED***(i):
24: $state := captured$
25: **end while**
26: **end while**
27: **if** $state = candidate$ and $phase = 2$ **then**
28: $step = 1$; $elected := false$
29: **while** $state = candidate$ and $\neg elected$ **do**
30: send $CAPTURE(2, step, j)$ to $j[k/2^{step}], \ldots, j[(2^{step-1} - 1)k/2^{step}]$
31: $Received := false$; $Response := 0$;
32: **while** $state = candidate$ and $\neg elected$ and $\neg Received$ **do**
33: receive M over $j[e]$

```
34:         Case M over j[e]
35:           CAPTURE(1, l, i):
36:             skip
37:           CAPTURE(2, l, i):
38:             if(l, i) > (step, j) then
39:               send ACCEPT(2) over e; state := captured
40:           INFORM(x):
41:             send CAPTURE(2, step, j) to j[e + x]
42:           ACCEPT(2):
43:             Response := Response + 1
44:             if Response = 2^(step−1) then
45:               step := step + 1
46:             if step = log(k) + 1 then elected := true else Received := true
47:       end while
48:     end while
49: end if
```

Algorithm for a captured node *j*:

```
50: done := false
51: while ¬done do
52:   receive M on edge j[e]
53:   if M is the first message received then
54:     state := captured; owner := 0; phase := 1
55:   end if
56:   Case M of
57:     CAPTURE (1, l, i):
58:       if phase = 1 then send ACCEPT(1, 0) over j[e];
59:     CAPTURE(2, l, i):
60:       if phase = 1 or owner = 0 then
61:         send ACCEPT(2) over j[e]; phase := 2
62:       else send INFORM(owner) over j[e];
63:     OWNER(i):
64:       if phase = 1 then
65:         owner := Cycle.index(i); phase := 2; send ACK(i) over j[e]
66:     ACCEPT(1, l, i):
67:       skip
68:     ACCEPT(2):
69:       skip
70:     ACK(j):
71:       skip
72:     ELECTED(i):
73:       done = true
74:     INFORM(x):
75:       skip
76: end while
```

In the pseudocode above, it is assumed that n is a power of 2. If n is not a power of 2, then

the protocol has to be modified. Let m be the smallest number greater than n such that m is a power of 2. Let $k = m/2^{\lceil loglog(n) \rceil}$.

Another modification is that in the first phase, i attempts to capture nodes in R_i as described before. After capturing these nodes, it sends a *CAPTURE* message simultaneously to all nodes j such that $i \in R_j$. After capturing this set of nodes, i can enter the second phase. In the second phase, i sends message to all nodes which are at distances that are multiples of $k/2^l$ in the lth step. In addition, it also sends messages to all nodes j such that i is at a distance which is a multiple of $k/2^l$ from node j. This modification ensures that at most one node will announce itself the leader. The number of messages and the time required is only increased by a constant factor.

3 IMPLEMENTATION

We use DistAlgo to implement the algorithm. Although the algorithm runs on a complete graph, a Hamiltonian cycle is used to implement the sense of direction. The id of a neighbouring processor is not necessary, but a processor must know how long is the distance from itself to another processor on Hamiltonian cycle. So in our implementation, the Hamiltonian cycle is achieved by defining a list of processors *ps* in *main*() function. The list is passed to every processor when initializing, such that every processor knows the distance from itself to another processor acrossing Hamiltonian cycle. Also the algorithm requires every processor to distinguish every channel, this is easily done by $from$ parameter in $receive$() function and *to* parameter in *send*() function.

In the algorithm, the response to receiving a message is categorized into three parts by the local states of receiver. So in the implementation, we define the action of receiving in $receive$() function and chose computation steps by checking $state$ and *phase* variable in the function. The algorithm always checks the state of the processor after receiving a message. To do this, it iterates through while loops when waiting for messages, and receives at most one message every loop. So in our implementation, a boolean variable $receiving\_flag$ helps us to achieve this. At the beginning of every loop, $receiving\_flag$ is $False$. When a message is processed, $receiving\_flag$ is set to be $True$. In *run*() function, the processor waits for $receiving\_flag$ to be true, then takes steps and go to the next phase. This ensures processors always checking their states after receiving a message.
Not all processors wake up spontaneously, some wake up upon receiving a message. These processors will never become candidates. To achieve this, we use a variable $wake\_rate$ and a random number $awaken$ set when processors are initialized. In practice, $wake\_rate$ is set to be 0.8. When $awaken$ is smaller than $wake\_rate$, the processor wakes up as a candidate; when $awaken$ is larger than $wake\_rate$, the processor wakes up but immediately becomes captured and skips candidate phase, taking no step until receiving a message.

However during the implementation, due to the limitation of the pseudocode provided in the paper, we encountered several problems and solve them as follows.

- The pseudocode has some errors in it. In original pseudocode in the paper, the indent of line 45 was one indent less, and indent of $56 - 75$ was two indents more. Without statements like "$end\ if$" and "*end while*", this error results in some confusion about under which condition should the equations be. We test under different cases and find out the correct intent.

- Another error in the pseudocode is that in line 40, $owner = i$ should be $owner = Cycle.index(i)$. That is to say, instead of informing sender id of the owner, the processor should inform the distance from itself to its owner across the Hamiltonian cycle. Only in this way can sender find the channel to owner with a sense of direction.

- Some details about the algorithm are not clear enough in the paper. For example, a processor sends *CAPTURE* and *OWNER* to nodes including itself. But the algorithm only gives the steps of processors other than the sender itself, resulting in some kind of infinite loop or ties, especially when $n = 2$. We detected and fixed this by adding equations and functions during the implementation.

- The algorithm also does not specify steps and state changes when a leader is elected. We complemented this part in our code.

- The algorithm only works when the number of processors n is the a power of 2, and the paper only gives a sketch of modifications to the algorithm when n is not a power of 2. The modifications are not clear enough in some details. We detected bugs brought by this and fixed it during implementation. For example, in phase 1, processors i should simultaneously send *CAPTURE* to the nodes j such that $i \in ring_j$. We implemented this by putting all such js in a list $ring_{i,belong}$, and add $ring_{i,belong}$ to the tail of $ring_i$. Meanwhile, when i receives *ACCEPT* messages with $l > 0$ from processor k, it skips some nodes in the ring to send *CAPTURE* because they are already captured by k. However, when receiving messages from node k in original $ring_i$, i should not skip sending messages to node j in $ring_{i,belong}$, since k does not capture j. But during our experiment, we discovered that since $ring_i$ is extended with $ring_{i,belong}$, some nodes in $ring_{i,belong}$ is incorrectly skipped. We added an if statement to separate nodes in the original $ring_i$ and $ring_{i,belong}$ when receiving such a message, thus prevent this situation from happening.

- When n is not a power of 2, after $ring_i$ is extended in phase 1, in phase 2 processors should ignore those nodes when sending *CAPTURE* messages. However, it is possible with certain *step*, after ignoring those processors, the number of receivers becomes 0. At this time, we should go directly into the next step. Moreover, if it is the last step, the processor should then elect itself as the leader.

- We modified all judgment conditions of *response* and *step* from "==" to ">=", to avoid infinite loop especially when $n = 2$.

- When running the code using a process for every node, the largest number of nodes it can run with is less than 100, which is not enough for our experiment. After using multiple threads to run nodes, the upper bound raises to being larger than 5000.

4 EXPERIMENTATION

In our experiment, we verified the message complexity to be $O(n)$ and time complexity to be $O(log(n))$.

The experiment is done on a PC with the environment of: DistAlgo 1.1.0b13, Python 3.6.8, Anaconda3 2019.03, 8 core Intel i7-7700HQ @2.8GHz CPU, 8 GB RAM, x64 platform.

To perform the experiment, we ran the code with n processors, such that n increases from 2 to 100, each time by 1. For every n, we repeated the code for 10 times to measure running time and number of messages of each repeat, and take the average of 10 repeats as the result.

Since DistAlgo neither allows us to access attributes or functions of nodes, nor supports public variable. To measure time and messages, we printed out running time and number of messages for every processor on screen and analyzed them.

Figure 1 shows the message-processor relation.

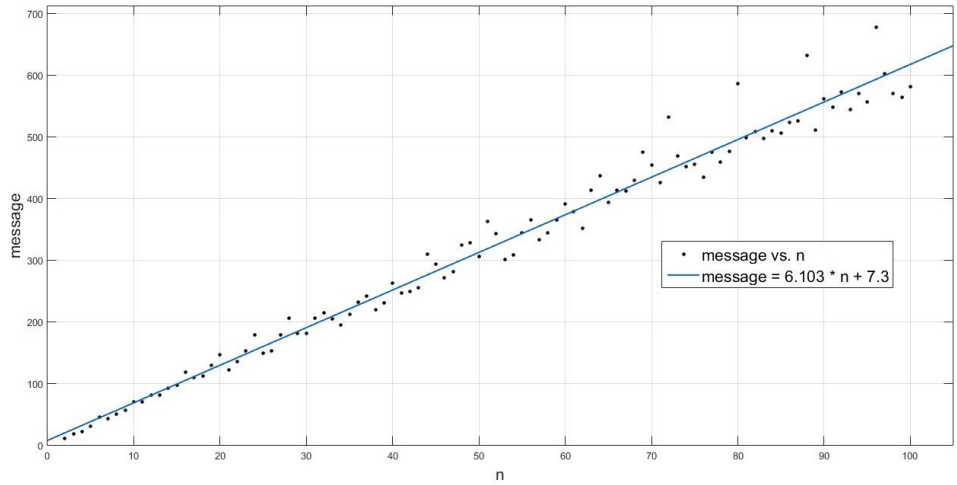

Figure 1. message-processor relation

As we can see in Figure 1, The relationship between the number of messages and the number of processors is $message = 0.613 \times n + 7.3$. R-square value for the fitting is 0.9802, which means the linear function is an excellent fit for message-processor relation. So the experiment verifies $O(n)$ message complexity declared in the paper.

However, there are some errors between experiment results and theoretical analysis, due to the following reasons: The relative order of processor id and the order of processors taking steps in the execution are different in every execution. It brings random errors to the experiment results. The random number in the implementation also affects a lot to the steps taken in execution and brings some uncertainty to the number of messages.

When measure time, we set all message delay to be 0.1 seconds. Figure 2 shows the time-processor relation.

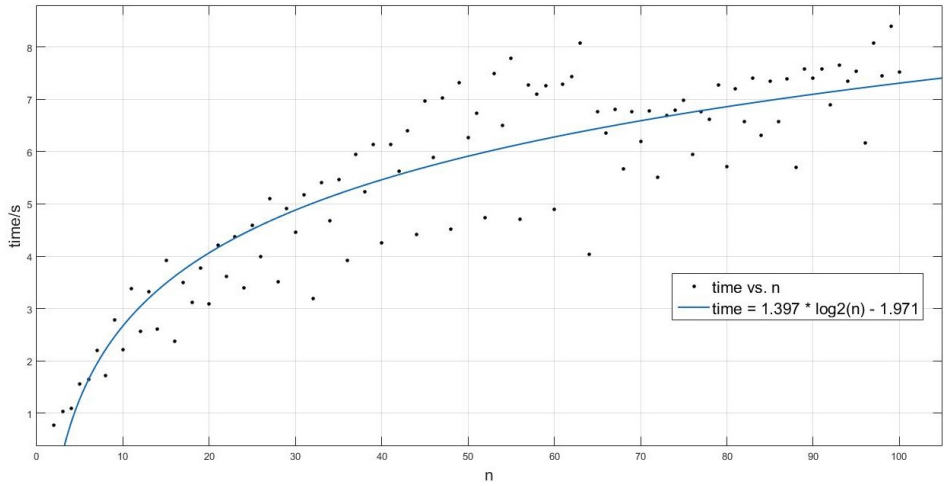

Figure 2. time-processor relation

As we can see in Figure 2, the relation between running time and number of processors is $time = 1.397 \times log_2(n) - 1.971$. The R-square value for the fitting is 0.8157, which means $log(n)$ relation is a good fit for message-processor relation. So the experiment verifies $O(log(n))$ time complexity in the paper.

Furthermore, we added n term into the fitting function, and fits the time-processor relation as shown in Figure 3.

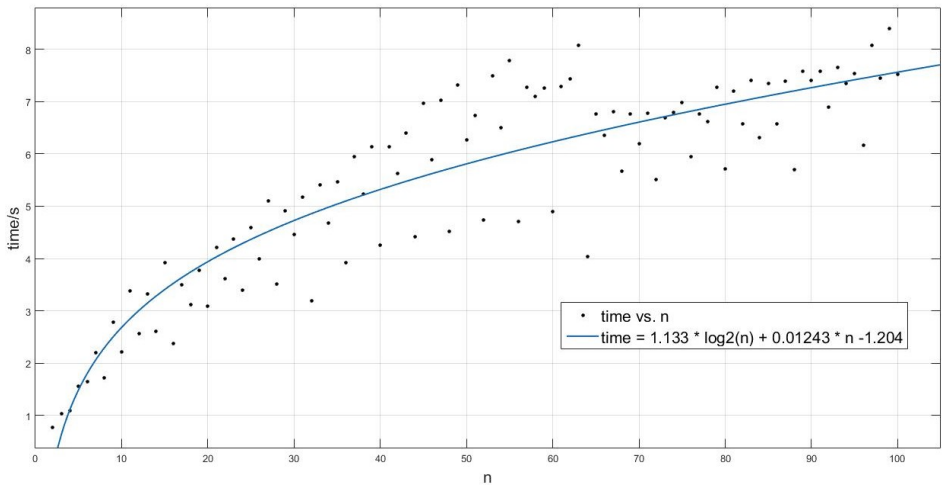

Figure 3. time-processor relation

As we can see in Figure 3, the relation between running time and number of processors is $time = 1.133 \times log_2(n) + 0.01243 \times n - 1.204$. The R-square value for the fitting is 0.8214, which means this function is a better description of message-processor relation than the previous one. The reason for this is that in the paper, computational time is ignored. But in practice, the computational time cannot be fully ignored. And since our message delay is set to be only 0.1 seconds,

the linear growing computational time will have some effects on the time we measured, thus adds a term of n in the fitting function. The coefficient of n term is very small compared to that of $log_2(n)$ term, meaning that n term does not dominate time increment theoretically, especially if some modifications are made in the code to reduce computational time and increasing message delay.

However there are also some errors between experiment results and theoretical analysis, due to the following reasons: Many random factors, including the instability of computer performance, background programs, will cause random errors in measuring time. Especially when the number of processors is small, random errors may become greater. The relative order of processor id, the order of processors taking steps and random variable in implementation are different in every execution. It brings uncertainty to experiment results.

5 ASSESSMENT

DistAlgo is a simple and powerful language that enables us to write and run distributed algorithms easily and quickly. It can express distributed algorithms at a high-level, combining advantages of pseudocode, formal specification languages, and programming languages. It enables us to simulate multiple processors on a single machine and send messages between each other.

Unfortunately, we met some problems with DistAlgo when performing certain tasks. For example, DistAlgo does not allow us to access attributes or functions of nodes, because nodes are not the kind of class we used in Python. It does not compile global variables, either. We met this problem when testing time and message complexity. In order to solve the problem, we wrote a counter to count messages and running times through standard outputs. It will be more efficient if DistAlgo provides APIs for testing time and message complexity. Also, DistAlgo offers low-quality randomness, so that the results often remain the same in different executions when testing time and message complexity.

We are not sure about the realization of asynchronous message passing in DistAlgo. It will be better if DistAlgo provides an API for us to choose from asynchronous and synchronous message passing.

6 FUTURE WORK

In our code, there are a lot of if statements for processor to get which status it is in, which makes the code ugly. We hope to optimize our code in the future to make it more efficient.

When testing the time and message complexity, we only test in networks which contain at most 100 processors, each repeating 10 times, due to the limitation of our machine. If we have better machines, we can test a bigger network with more processors, repeat more times and add more message delay to get a more precise result of the running time and number of messages.

When testing time complexity, the message delay is set to 0.1s in our code. However, the message delay may be different between different processors in reality. In the future, we can set random numbers to simulate the transmission delay in reality to achieve more precise results.

7 APPENDIX

We have three team members. All of us involve in finding appropriate topics and papers for the project, as well as writing report. Guanda Li is responsible for analyzing the algorithm and build up

the structure of the code. Yiran Huang is responsible for processing screen output data, make slides and combine the report. Peixin Liu is responsible for writing code and performing experiments.

REFERENCES

[1] G. Singh, Efficient leader election using sense of direction. Distributed Computing. 10 (1997) 159 – 165.
[2] G. Hu. 2019. DistAlgo–Tips. Retrieved April, 2019 from https://github.com/Piezoelectric/DistAlgo-Tips