Universiteit
Leiden
The Netherlands

Efficient Parallel Cycle Detection

Cassie Wanjun Xu

Supervisors: Dr. Alfons W. Laarman
Dr. Walter Kosters

Opleiding Informatica

Bachelor Thesis

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

01/06/2020

Abstract

Cycle detection is the problem of identifying repetition in a sequence of iterative function applications. Sequential algorithms are able to solve cycle detection in $O(n)$ time with $O(log(n))$ space. However, there was no existing efficient parallel solution for this problem. In this thesis, we propose an efficient parallel algorithm. The algorithm is based on iterative squaring of a linked list, using $O(n)$ processors within $O(log(n))$ time to identify whether a cycle exists and compute the cycle length and entry points.

Acknowledgements

First, I want to say thank you to my supervisor Alfons Laarman, who taught me a lot during this year, consisting of software engineering, parallel algorithms and academic writing. Besides the direct guidance, I see a passionate and idealistic researcher's figure in Alfons, which also impresses and influences me. It was always enjoyable to learn things from you. Thank you, Alfons.

Romke and Yannik, thanks for the discussion sessions we had. Your ideas inspired me and contributed to the completion of this thesis. Walter Kosters, Jan van Rijn and Fenia Aivaloglou, thank you for constantly communicating and helping Zhe and me for the bachelor project's practical issues.

And I want to express my thankfulness to Leiden University. I feel the lofty elegance of a superior university here. This year which I spent in Leiden, the Netherlands enriches me greatly, both in studies and personal life.

Finally, at the point of my graduation for bachelor study, I want to express my gratitude heartily to Xi'an Jiaotong University, and my fellow students from CS-Shao 61, my roommates from 310, and all the ones from ShaoNianBan. I received an excellent computer science education here and was able to learn and grow together with you amazing people. It is a memorable and valuable journey for me, and I believe it is only the very beginning of my entire adventure.

Contents

1 Introduction

Cycle detection is the problem of identifying repetition in a sequence of iterative function applications. Sequential algorithms for this problem, including Floyd's Tortoise and Hare algorithm[Flo67] and its variations, are able to solve cycle detection in $O(n)$ time with $O(log(n))$ space. However, there was no efficient parallel solution for this problem. The existing common method for the problem is to treat the linked list as a general graph and requires $O(n^\omega)$ [1] processors and $O(log^2(n))$ time. In this thesis, we propose an efficient parallel algorithm for a general version of the cycle detection problem, where we define the function as a partial function. The algorithm is based on iterative squaring of a linked list, using $O(n)$ processors within $O(log(n))$ time on a CREW PRAM model to identify whether a cycle exists and further computes the cycle length and entry points, which improves the efficiency significantly from the former method.

The structure of this thesis is as follows: we will first go through the definitions and existing sequential algorithms for cycle detection. After that, some relevant parallel computing concepts, including the PRAM computational model, NC complexity class, and the current parallel method for the problem, will be introduced. Then we will give our efficient parallel cycle detection algorithm and a proof will be provided to demonstrate the correctness and completeness of our algorithm.

2 Preliminaries

2.1 Cycle Detection

Cycle detection is the problem of identifying a repetition in a sequence of iterative function applications. The original mathematical problem of cycle detection can be defined as follows. For any function f that maps a finite set S to itself, and any initial value x_0 in S, when the sequence of iterated function values

$$x_0,\ x_1 = f(x_0),\ x_2 = f(x_1) = f(f(x_0)) = f^2(x_0),\ \ldots,$$

$$< x_i \mid x_i = f^i(x_i) >$$

contains the same value twice, which means there exists a pair of distinct indices i and j such that $x_i = x_j$. Once this happens, the sequence will continue periodically, repeating the same sequence of values from x_i to x_{j-1}. In this original functional definition of the problem, a cycle is bound to exist due to the pigeon hole principle, since the function maps a finite set to itself. Cycle detection is the problem of finding i and j, given f and x_0.

The cycle detection problem has many applications in computer science: for example, it forms the core of Pollard's rho algorithm [Tes01]; it is used in some methods of finding infinite loops in computer programs [VG87]; Teske [KJ88] described the application in cryptographic algorithms, etc.

The most widely known and used algorithm is Floyd's Tortoise and Hare algorithm. The key idea of Floyd's algorithm, traversing with fast and slow pointers, can also be used in many different problems regarding singly linked lists. Several other algorithms improved the complexity based on Floyd's algorithm, including Brent's algorithm[Bre80] and Gosper's algorithm[BGS72].

[1] The ω is the lower bound for the exponent of matrix multiplication. Further explanation is given in Section 3.1

The input of the algorithm is often given as an array $[n_1, n_2, n_3...]$. Considering this array, we view index as x_i and value as $f(x_i)$. Therefore, we can also view this array as a linked list, in which the index x_i is the node's value, and the value of the array element $f(x_i)$ is the next field of the node. Since there is only one next field for each node, it forms a singly linked list.

The essential idea of Floyd's algorithm[Flo67] is as follows:

1. Traverse the linked list with two pointers, starting from the head.
2. Move one pointer(slow_p) one step at each operation and the other pointer(fast_p) two steps at each operation.
3. The two pointers will meat each other inside the loop.

The method can be understood simply by imagining two people running in a circular track. The two people are running at different speeds. Since it is a circular track, they will eventually meet inside the circle. Furthermore, since we move one of the pointers at the speed of one and the other one at the speed of two, the distance difference of the two pointers increased every time is always one which guarantees that we won't miss one single point inside the cycle.

Algorithm 1 Floyd's Algorithm

```
t, h ← x, f(x)
while t ≠ h do
    t, h ← f(t), f(f(h))
end while
```

After the two pointers meat each other, we will begin to calculate the length and the starting point of the cycle. To illustrate this, we assume the following notations (see Figure 1):

i: The distance that the slow pointer has passed. Therefore, the distance of the fast pointer has passed is $2i$.
m: The distance from the start node to the cycle entry.
n: The length of the cycle.
k: The length from the cycle entry to the place where the two pointers meet.
p: When meeting, the count of cycles that the slow pointer has passed.
q: When meeting, the count of cycles that the fast pointer has passed.

We can get the following equations:

$$i = m + p * n + k$$

$$2i = m + q * n + k$$

Use the second minus the first one:

$$i = (q - p) * n$$

Therefore, i equals an integral multiple times of the cycle length.

To get the length of the cycle, we leave one pointer at the meeting point and move the other pointer one step each time. When they meet, which means the moving pointer has gone through the cycle again, the number of steps that the pointer has passed is the length of the cycle.

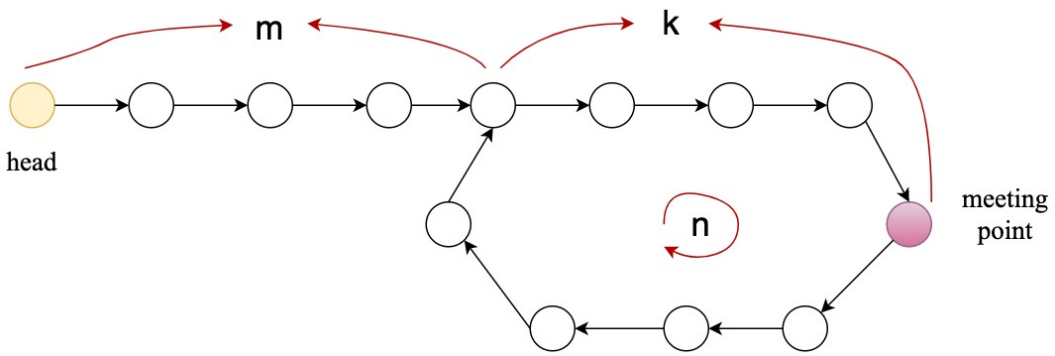

Figure 1: Cycle Detection Illustration

To get the entry point of the cycle, we set one pointer back to the start of the linked list, and the other one remains at the meeting point. Make them move together, one step at each operation. When they move m steps, the length they pass from the beginning is m and $m + i$ separately. And since $i = (q - p) * n$, the distance difference is a multiple of the cycle length, so they will meet at the entry of the cycle. So moving the pointers one from beginning and one from the meeting point, when they meet again, the new meeting point is the start of the cycle.

Brent's cycle detection algorithm is similar to Floyd's algorithm, also using two pointers technique, but improves the time complexity. In Brent's algorithm[Bre80],the fast pointer moves in powers of 2 at each step. After every power, we reset the slow pointer to the previous value of the second pointer. The idea is to find the smallest power of 2^i that is longer than m and n. As proven in Brent's work[Bre80], the time complexity is still $O(n)$, but the algorithm is 24 - 36% faster than Floyd's Algorithm.

In this thesis, we define the function of the cycle detection problem as a partial function: instead of $f : D \to D$, the function is $f : D \to D \cup \{\perp\}$. Not every element has its function value in the domain. The pigeon hole principle no longer holds and a cycle may not exists. The general graphic example is shown in Figure 2. Therefore, in the first place, we aim at solving the decision problem of cycle existence in parallel.

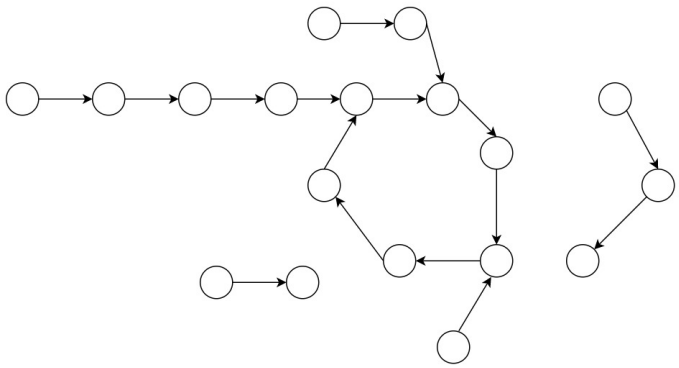

Figure 2: Cycle Detection Example

2.2 Parallelism and Parallel Algorithm

In 1965, Gordon Moore, the CEO of Intel, made the famous observation, Moore's law[M+65],that the number of transistors on a dense integrated chip doubled itself every 18 months. The observation precisely predicted the development of integrated chip for decades. Even though the computing ability of a single CPU has been increased significantly, it has gradually reached its own fundamental physical limit. Meanwhile, people become more interested in distributing computing task onto multiple processors and doing calculation steps simultaneously to achieve higher performance. Today, most PC and laptops are equipped with multiple cores and it's also common to utilize the GPU when a specific task requires large amount of numerical calculations. Distributed computing and cluster computing are adopted when computing as a large scale and among different computers. In this way, parallel computing has gradually become the main paradigm of computer architecture and main stream computing method.

Many researches and experiments are being done to adapt this architecture. Most of the existing algorithms are sequential, which means they are executed by one single operation at a step. In order to fit the change of the computing paradigm mentioned above, algorithms should also be adjusted to perform well on parallel computers. Therefore, it becomes necessary to design algorithms which execute multiple operations on multiprocessors at every step, i.e., parallel algorithms.

To illustrate the basic idea of parallel algorithms, we will use computing the sum of an integer array as an example here. The sequential method is going through the array and keeping a running sum of the integers which have been seen so far. It takes $O(n)$ time. In the parallel algorithm, we use multiple processors at the same time and they do the computation steps simultaneously. The parallel solution is as follows: In the first step, $\lceil n/2 \rceil$ processors will compute the sum of every two adjacent numbers and store the $\lceil n/2 \rceil$ results in different memory cells. In the second step, $\lceil n/4 \rceil$ processors will compute the sum of every two adjacent results from the first step, etc. Finally, one processor will compute the total sum of the whole array at the $\lceil log(n) \rceil$-th step. It is a natural question to ask how will multiple processors behave when they simultaneously make a request to read or write to the same memory resource and we will discuss these parallel models and concurrent conflicts in the section 2.3.

The associative property and the independence of sub problems of the above example make it highly parallelizable. And the idea of this algorithm forms the foundation of many parallel algorithms. However, several computational problems are believed as "inherently sequential" [Gre92], which means the operations rely on previous operation, such as depth first search, linked list algorithms. Also, parallel algorithms usually require more resources than the sequential solutions. Considering the above example, sequentially, n steps of addition can finish the summation. But in parallel, we use n processors in $log(n)$ time, which makes the total work into $nlog(n)$ amount.

When dealing with the "inherently sequential" problems, usually much more resources are required, to overcome the dependence of different operations by doing redundant operations in every processor. One example is the transitive closure which we will discuss in Section 3.2. There are linear-time sequential algorithms for graph reachability. But the parallel algorithms for the problem, although fast, require a great many processors and thus require much more computation overall than the sequential algorithms.

When parallelizing these computational problems, a dedicated parallel algorithms' design need to be considered in order to reduce the requirement of resources and improve efficiency as much as possible.

2.3 The Parallel Random Access Machine Model

When designing and analyzing sequential algorithms, an abstract memory model called random access memory (RAM) is generally used. In this model, the single processor is connected the memory and each CPU operation, including arithmetic operations, memory operations and logical operations, will take one step of time. The designer is able to only focus on the logic of the algorithm instead of considering too much about the detail of implementation.

In the field of parallel algorithms, the Parallel Random Access Machine (PRAM) is the most commonly used abstract model, which is an analogy and generalization of the RAM model in sequential algorithms. In PRAM model, all the multiple processors are connected to a shared memory space and are able to access any memory cell in unit time. The basic idea of the model is shown in Figure 3.

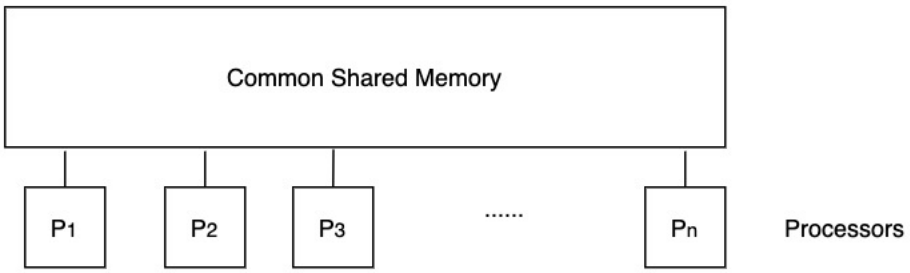

Figure 3: Parallel Random Access Model

People have been debating about the PRAM model because the PRAM model is based on an ideal assumption: the time cost for an arbitrary number of memory accesses is the same for every processor in one access [Wyl79]. The PRAM model neglects some practical issues, such as synchronization and communication method and cost. But the same criticism can be leveled to RAM, which is widely used as the idealized computational model [Ski98]. Likewise, PRAM algorithms capture the "parallelizability" of problems well, mimicking the low-level circuit model, making the model suitable for our purposes here.

When doing multiple operations simultaneously, read/write conflicts occur. Inside the execution of the algorithm, multiple processors may request to read or write into the same cell at the same particular unit time. The strategy to read/write conflicts splits the model into four categories:

1. Exclusive Read Exclusive Write (EREW) — Only one processor can read or write into one memory cell at one time.

2. Concurrent Read Exclusive Write (CREW)— Multiple processors can read a memory cell at the same time, but only one processor can write into the memory cell at one time.

3. Exclusive Read Concurrent Write (ERCW)— This model is less practical. It is not useful to have a model that can only read exclusively but write concurrently.

4. Concurrent Read Concurrent write (CRCW)— Multiple processors can read and write in simultaneously. A CRCW PRAM is also known as the concurrent random-access machine.[Imm89]

When concurrent writes are allowed, a rule is used to determine which value will be written into the memory cell. This includes arbitrary select, priority select and specific rules, such as SUM or AND.

In this thesis, considering the characteristic of our algorithm, we will propose the algorithm on the CREW PRAM model. Further explanation about the concurrent problem will be discussed in Section 4.2 after the algorithm.

2.4 Parallel Complexity

Time and space complexity are used to describe the time and resources which are required to run an algorithm. Instead of computing the precise amount of resources required by an algorithm, which differs on every different problem and different machine, the asymptotic method is commonly used when describing complexity, and it is usually expressed using the big O notation. For example, the time complexity $O(n)$ indicates an algorithm runs in worst-case time linear in the input size n. For instance, solving connectivity in general graphs can be done in linear time, since the description of the graph requires $|V| + |E|$ space and each vertex and edges is considered only once in a standard sequential search algorithm.

Because of the analogy with the later-described parallel complexity classes, we reproduce some classical results of complexity theory here. An interested reader should look up precise definitions in [Joh90]. First we introduce some concepts fundamentally related to complexity classes:

- Decision problem: Problems that can be posed as a yes-no question of the input values.
- Verification: Given a set of existing potential answers, verification is recomputing the accepting path by a non-deterministic turning machine, and giving the result whether a solution is correct or not.
- Reduction: A reduction is a process of transforming a problem A into another problem B. A polynomial-time reduction is a reduction that can be done within polynomial time. This further implies that if we can solve problem B in polynomial time, we can then solve problem A also in polynomial time (by doing the reduction process and solving B).

The fundamental complexity classes for sequential algorithms are as follows:

- P: The set of problems that can be solved by a deterministic Turing machine in polynomial time. In order words, we can "quickly" find the answer to a P problem.
- NP: The set of problems that can be solved by a non-deterministic Turing machine in polynomial time. In order words, we can "quickly" verify an answer of an NP problem whether is correct or not.
- NP-hard: If every problem G in NP can be reduced into another problem H in polynomial time, the set of problem H is called NP-hard problems. In other words, if we can find a polynomial algorithm for H, we also get a polynomial solution for G. Finding out a polynomial solution for NP-hard problems would give polynomial solutions for all NP problems.
- NP-complete: The set of problems that are both NP and NP-hard.

It remains an important open question whether P equals NP, which indicates that does quickly verifying a problem's answer equal to finding an answer.

In parallel computing, the complexity theory is analogous and relevant to the above theory of NP-completeness. The class "NC" (for Nick's Class) [AB09] is the set of problems that can be solved by using a polynomial number of processors within poly-logarithmic time. As P problems could be considered as being tractable on a sequential computer, NC problems are considered as tractable on a parallel computer. NC is a subset of P since parallel operations can also be stimulated on a sequential computer. Instead of doing the computations of n processors in $log(n)$ time, we can do the computations on a single processor in $nlog(n)$ time, which makes the problem in P class. Like the definition of polynomial-time reduction, NC-reduction is the reduction that can be operated in poly-logarithmic time on a parallel computer with a polynomial number of processors.

It is not known whether NC = P. Just as it is widely doubted by computer scientists that P does not equal NP, it is also widely doubted that NC does not equal P. Similarly to the use of NP-complete problems to analyze the P = NP question, the P-complete problems, the problems which are in P and every problem in P can be reduced to them by NC-reduction or logarithmic space reduction, viewed as the "probably not parallelizable", serves a similar function in analyzing the whether NC equals P question.

NC class can be divided by the depth $O(log^i(n))$ of the uniform boolean circuits with a polynomial number of gates of at most two inputs. This forms the NC-hierarchy:

$$\mathsf{NC}^1 \subseteq \mathsf{NC}^2 \subseteq \cdots \subseteq \mathsf{NC}^i \subseteq \cdots \mathsf{NC}$$

Besides the above-discussed complexity classes defined by time, complexity classes can also be defined by required writing memories. L is the set of decision problems that can be solved by a deterministic Turing machine using a logarithmic amount of writable memory space. L is a subset of NC class, because those problems can only have $2^{O(log(n))} = n^{O(1)}$, a polynomial number of different configurations, which means: we can use n processors to execute the different n possibilities in $O(1)$ time. Furthermore, we can get the relation of $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NC}^2$ [Sip97]. The inclusion relationship of different complexity classes is shown in Figure 4.

To sum up, we list the following concepts and definitions:

- NC: The set of decision problems can be solved by a polynomial number of processors within poly-logarithmic time.
- NC reduction: A reduction which can be operated by a polynomial number of processors within poly-logarithmic time.
- L: The set of decision problems that can be solved by a deterministic Turing machine using a logarithmic amount of writable memory space.
- Log-space reduction: A reduction which can be computed by a deterministic Turing machine using logarithmic space.
- P-complete: The set of problems which is in P and every problem in P can be reduced to it by log-space reduction or NC reduction.

Back to the cycle detection problem. Analyzing the Floyd's algorithm, only two pointers are used. The pointer is used to indicate the index of the array element. Given the total size is n and

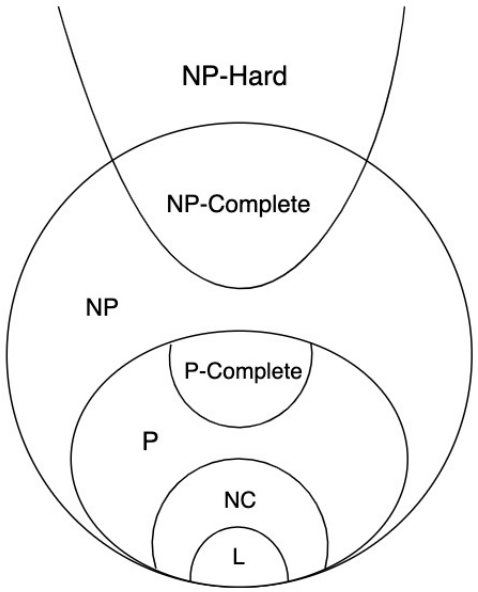

Figure 4: Complexity Class

the index will be stored in binary format in a pointer, logarithmic space $log(n)$ is sufficient to hold the constant number of pointers. Therefore, the problem of cycle detection lies in the L complexity class, and L $\subseteq$ NC [Bor77], which makes it parallelizable, which gives us the confidence to find a better solution than the existing method for general graphs in Section 3.2.

3 Related Work

In this chapter, we discuss the existing method to solve the cycle detection in parallel. We will introduce about matrix multiplication and parallel transitive closure to calculate the connectivity of a graph. We will use this to compare it with our method in Chapter 4.

3.1 Matrix Multiplication

Matrix operations are required in many numerical and other algorithms.

Given two matrices, if A is an $m \times n$ matrix $\begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{m1} & a_{m2} & ... & a_{mn} \end{bmatrix}$, and B is an $n \times p$ matrix

$\begin{bmatrix} b_{11} & b_{12} & ... & b_{1p} \\ b_{21} & b_{22} & ... & b_{2p} \\ ... & ... & ... & ... \\ b_{n1} & b_{n2} & ... & b_{np} \end{bmatrix}$, then $C = AB$ is defined to be an $m*p$ matrix $\begin{bmatrix} c_{11} & c_{12} & ... & c_{1p} \\ c_{21} & c_{22} & ... & c_{2p} \\ ... & ... & ... & ... \\ c_{m1} & c_{m2} & ... & c_{mp} \end{bmatrix}$, where

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + ... + a_{in}b_{nj} = \sum_{k=1}^{n} a_{in}b_{nj}$$

We can see that sequential matrix multiplication requires three loops therefore $O(n^3)$ time complexity when done naively. Due to the two separate loops and the array sum (as mentioned in section 2.2), matrix multiplication is highly parallelizable. If we assume $m = n = p$, we set n processors to each of the n^2 numbers in C, where each set of n processors will compute the value in $O(logn)$ time. Therefore, this algorithm requires $O(n^3)$ processors and $O(log(n))$ time and therefore makes the algorithm in the NC complexity class. Sequentially, it is known that matrix multiplication can be improved. It was first proposed by Strassen [Str69] in 1969 to decrease the complexity to $O(n^{2.807})$. The lower bound ω for the exponent has been improved gradually and so far $\omega = 2.37286$, proposed by François Le Gall in 2014 [LG14]. By using the same principle in the multiplication procedures, the total processors of parallel matrix multiplication required is $O(n^\omega)$.

Algorithm 2 Parallel Matrix Multiplication

```
1: (m, n) := dimensions(A)
2: (n, p) := dimensions(B)
3: parfor i ← 1, m do
4:     parfor j ← 1, p do
5:         C_ij := sum(A_ik * B_kj : k ∈ [1, n])
6:     end parfor
7: end parfor
```

A boolean matrix is a matrix where every entry is 0 or 1. Boolean matrix multiplication is performed by using AND for * operation and OR for + operation. It is used when we only care about the logical calculation of two matrices instead of numerical results. The time complexity of boolean matrix multiplication is the same as normal matrix multiplication.

3.2 Parallel Transitive Closure

A directed graph is composed of vertices and edges with directions. A directed graph of n vertices can be stored and represented by an $n \times n$ adjacency matrix, where the value of element on (i, j) shows the existence of the path from i to j. If matrix A is the adjacency matrix for graph G, then $A_{i,j} = 1$ when there is a direct edge connected from vertex i to vertex j in graph G. Otherwise, $A_{i,j} = 0$ when there is no edge.

Floyd-Warshall algorithm [FM71] gives the method to obtain the reachability by doing calculations on the adjacency matrix. The reachability matrix is called the transitive closure of the graph and we refer to this technique as "iterative squaring". We explain the method as follows: Consider the element on position (i, j) of $A \cdot A$. It will only be 1 when there is a k where (i, k) and (k, j) both equal to 1 (note that during the calculation we are using boolean addition.) The result

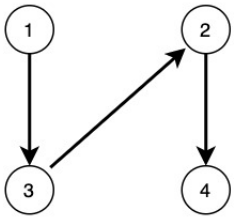

Figure 5: A^1

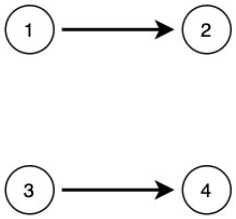

Figure 6: A^2

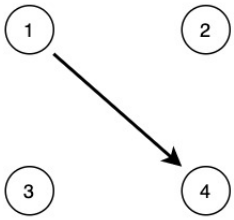

Figure 7: A^3

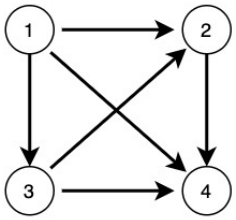

Figure 8: A^+

indicates the existence of a path of length 2 from i to j by passing k. Therefore, A^2 shows all the node pairs connected by paths of length 2. Continue this process inductively, we will get all the node pairs reachable by paths of length 3 by calculating the result of A^3, till the paths of length n by calculating A^n. Adding all the results from A^1 to A^n, which are the connections by paths from length 1 to n, will give us the result of the reachability of the whole graph, i.e. the transitive closure.

The process is shown in Figure 5, 6, 7, 8. Further formal proof and explanation can be found in [FM71].

Considering our cycle detection problem, a singly linked list can be seen as a general graph. Therefore, it can be solved by computing the transitive closure. If there exists 1 in the diagonal of the reachability matrix, which means there is a path from the node to itself, a cycle is found. And the diagonal entries with value 1 are the nodes lying inside the cycle. Based on the discussion in Section 3.1, this method solves the problem with $O(n^\omega)$ processors in $O(log^2 n)$ time.

But it is noticeable that this is not an efficient method. Considering the linked list as a graph, the out-degree for every vertex is always one and the adjacency matrix would be a very sparse one, which takes much unnecessary space and also leads to much unnecessary operations. To improve this, we have to use the property of linked lists instead of loosely treating it as general graphs. The iterative squaring of a linked list and how to detect a cycle will be discussed in the next chapter.

4 Parallel Cycle Detection Algorithm

First, in Section 4.1, we show how iterative squaring can be done for linked lists using only n processors and log(n) time. Then in Section 4.2, we apply a similar technique to find cycles in parallel, by passing along the successor vertices with the minimum index. This results in a log(n) time cycle finding algorithm that uses only n processors.

4.1 Traversing Linked List

James C. Wyllie first introduced a technique called "doubling" to count the number of elements in a linked list[Wyl79], which gives a way to do the iterative squaring of a linked list. Given a linked list L, linked to next node via next field, with the end of the list showing by a null next field, compute $|L|$. The algorithm is given in Algorithm 3.

Algorithm 3 Count the Length of Linked List

1: **parfor** $i \leftarrow 1, n$ **do** → assign a processor to each element of L, assign the processor named head to the list head
2: $far(i) := next(i)$
3: $span(i) := 1$
4: **while** $far(i) \neq null$ **do**
5: $far(i) := far(far(i))$
6: $span(i) := span(i) + span(far(i))$
7: **end while**
8: **end parfor**
9: **return** $span(head)$

We assigned each node of the linked list a processor. In each iteration of the while loop, we update $far(i)$ pointing to the element which its current far pointer's $far(i)$ points to, and $span(i)$ to keep the distance of i to $far(i)$, therefore keeps the loop invariant $far(i) = next^{span(i)}i$. Since we assume that there is no cycle in a linked list in this context of counting length, $far(head)$ while eventually become null, which means that the head node has reached the end of the linked list, then $span(head)$ will be the length of the linked list.

The method provides a useful idea on how to do iterative squaring on a linked list: by assigning a processor on each node, and doubling the pointing distance in each iteration, we are able to traverse the linked list in $log(n)$ time.

4.2 Parallel Cycle Detection Algorithm

The parallel cycle detection algorithm is based on the doubling technique to go through a linked list in parallel in the previous section. The novel idea is to keep passing along the id of the minimum successor, i.e., the node it can reach with the lowest vertex id. A cycle will be found once we detect the smallest index is equal to the index of the node itself. This will happen on the node with the smallest index inside the cycle, if and only if the cycle exists. Because otherwise the reachable index could only go down to the sink instead of reaching itself. The algorithm is described in Algorithm 4.

The method of the algorithm can be explained as follows:

1. During initialization, set *far* and *min* to the index of its successor node, if it has successor node. *Far* and *min* record the current furthest and smallest id it could reach.

2. During each iteration inside the while loop, $far(i)$ keeps jumping two times further to the furthest reachable node. $Min(i)$ keeps the smallest index it could reach, which means the smallest index between i and $far(i)$. Therefore, during the execution of Line 9 of Algorithm

Algorithm 4 Parallel Cycle Detection

```
1: parfor i ← 1, n do          → assign a processor for each node of the linked list
2:     if next(i) ≠ ⊥ then
3:         far(i) := next(i)
4:         min(i) := next(i)
5:     else
6:         far(i) := ⊥
7:     end if
8:     while far(i) ≠ ⊥ do
9:         min(i) := min(min(i), min(far(i)))
10:        if i == min(i) then
11:            return True
12:        end if
13:        far(i) := far(far(i))
14:    end while
15: end parfor
16: return False
```

4, we update the $min(i)$ to the smaller value of the smallest index between i to old $far(i)$ and old $far(i)$ to new $far(i)$. Therefore after each iteration $min(i)$ will be the smallest index between i to its current $far(i)$.

3. The algorithm terminates when a $i == min(i)$ happens (a cycle found) or all $far(i) == \perp$ (Every node reaches the end of the linked list and no cycle has been found).

In Figure 9 10 11 12, we show the updating process of *min* in the red number.

The algorithm is given on the CREW PRAM model based on the following properties: 1) Concurrent read is required when two nodes have the same $far(i)$ node during the iteration. 2) The properties $(far(i), min(i))$ are stored locally in each node's processor and no new data is written into the original linked list, therefore exclusive write fulfills our needs.

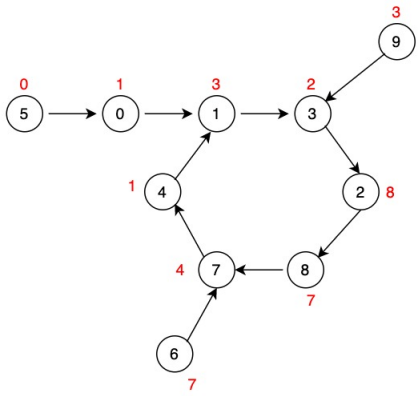

Figure 9: Initialization

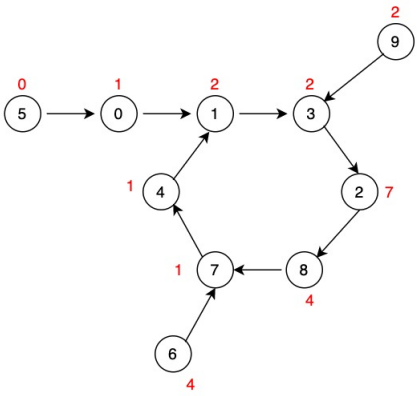

Figure 10: After first iteration

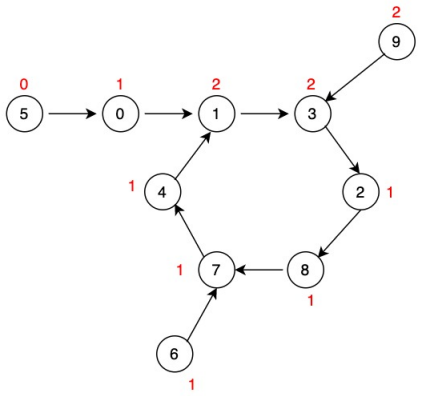

Figure 11: After second iteration

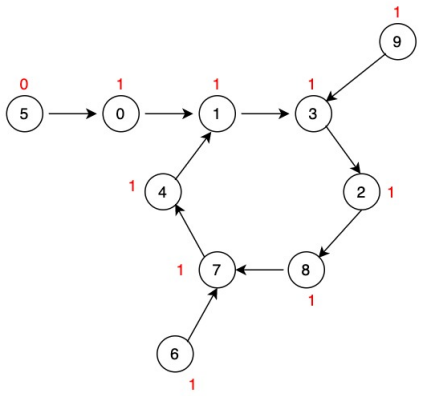

Figure 12: After third iteration(finds a loop)

Below, we show soundness, completeness and termination of the algorithm 4 in Theorems 1 2 3. To prove these theorems, we first propose the Lemma 1, which states the loop invariant inside the algorithm.

Within the lemma and theorems, we use the same notations *next*, *far*, *min* as in the algorithm, which separately store the next linked field, the furthest successor's id and the smallest successor's id, n for the total amount of nodes, and $|D|$ for the length of cycle when there is a cycle. Furthermore, we define the $next^k(i)$ notation:

$$next^1(i) = next(i), next^2(i) = next(next(i)), next^3(i) = next(next^2(i)), \ldots$$
$$next^{k+m}(i) = \perp, \ if \ next(next^k(i)) = \perp, m \in \mathbb{N}^+$$

Lemma 1 (Loop invariant)**.** *At the start of the k-th iteration of the while loop in Algorithm 4, for all i in [1..n]:*

$$far(i) = next^{2^k}(i) \tag{1}$$

$$min(i) = min\{next^y(i) \mid y \ in[1...2^k]\} \tag{2}$$

Proof. Initialization: At the beginning of the first while loop, k equals to 0. From Line 3, $far(i) := next(i) = next^{2^0}(i)$, therefore we get Statement 1. From Line 4, $min(i) := next(i) = next^1(i) = min\{next^y(i) \mid y \ in[1...2^0]\}$, therefore we get Statement 2.
Maintenance: Assume that the loop invariant holds at the start of iteration k. To prove the maintenance of Statement 1: from the loop invariant of iteration k, we have $far(i) = next^{2^k}(i)$. Suppose $far(i)$ points to node j, then we also have $far(j) := next^{2^k}(j)$.
Then:

$$\begin{aligned} far(i) :&= far(far(i)) \\ &= far(j) \\ &= next^{2^k}(j) \\ &= next^{2^k}(far(i)) \\ &= next^{2^k}(next^{2^k}(i)) \\ &= next^{2^{k+1}}(i) \end{aligned} \tag{3}$$

Thus the loop invariant Statement 1 holds again at the beginning of the next (k+1)-th while loop. To prove the maintenance of Statement 2: from the loop invariant of iteration k, we have $min(i) = min\{next^y(i) \mid y\ in[1...2^k]\}$. Since $far(i) = next^{2^k}(i)$, we also have

$$\begin{aligned} min(far(i)) &= min\{next^y(far(i)) \mid y\ in\ [1...2^k]\} \\ &= min\{next^y(next^{2^k}(i)) \mid y\ in\ [1...2^k]\} \\ &= min\{next^{y+2^k}(i) \mid y\ in\ [1...2^k]\} \\ &= min\{next^y(i) \mid y\ in\ [2^k+1...2^{k+1}]\} \end{aligned} \tag{4}$$

$$\begin{aligned} min(i) :&= min\{min(i), min(far(i))\} \\ &= min\{min\{next^{y_1}(i) \mid y_1\ in[1...2^k]\}, min\{next^{y_2}(i) \mid y_2\ in[2^k+1...2^{k+1}]\}\}, \\ &= min\{next^y(i) \mid y\ in[1...2^k, 2^k+1...2^{k+1}]\} \\ &= min\{next^y(i) \mid y\ in[1...2^{k+1}]\} \end{aligned} \tag{5}$$

Thus in this case the loop invariant Statement 2 holds again at the beginning of the next (k+1)-th while loop. □

Theorem 1 (Soundness)**.** *Algorithm 4 is sound. If Algorithm 4 returns true, in other words if there exists $i == min(i)$ among all the nodes, then there is a cycle in the linked list.*

Proof. From Lemma 1, we have $min(i) = min\{next^y(i) \mid y\ in[1...2^k]\}$ at the beginning of the while loop at Line 8. As calculating in Equation 4 and Equation 5 inside the proof of Lemma 1, after Line 9, we have $min(i) = min\{next^y(i) \mid y\ in[1...2^{k+1}]\}$. If the algorithm returns true, which means $i == min(i)$, then we have $i = min(i) = min\{next^y(i) \mid y\ in[1...2^{k+1}]\}$. This suggests that i is among $\{next^y(i) \mid y\ in[1...2^{k+1}]\}$, i.e., i has itself inside its own successor nodes. Therefore, there must be a cycle in the linked list. □

Theorem 2 (Completeness)**.** *Algorithm 4 is complete. If Algorithm 4 returns false, there is no cycle in the linked list.*

Proof. The algorithm returns false after each parallel process finishes. It means the while loop terminates for each node, i.e. each $far(i) == \perp$. This implies all the nodes have reached the sink of the linked list, through the downwards path instead of iterating in a cycle. Therefore, there is no cycle in the linked list. □

Theorem 3 (Termination)**.** *Algorithm 4 terminates.*

Proof. Each iteration increases the reaching distance $far(i)$ twice as before, as implied by Lemma 1. Therefore, the algorithm terminates in $log(|D|)$ steps if a cycle is found or in $log(|L|)$ steps when all the nodes reach the sink. □

Above we have proved the soundness, completeness and termination of the algorithm. Below we will analyze the complexity of the algorithm, where makes it more efficient than the existing one.

Theorem 4 (Complexity)**.** *Algorithm 4 uses n processors and finishes cycle detection in $O(log|L|)$ time, where n is the amount of nodes and $|L|$ is the longest path inside the linked list.*

Proof. At the beginning of the algorithm, we assign each node a processor, therefore the resource requirement is n processors. From Lemma 1 Statement 1, $far(i) = next^{2^k}(i)$, the algorithm will terminate when $far(i) == \perp$, i.e, when $2^k > |L|$. Therefore the time complexity is $O(log|L|)$. □

Moreover, if there is a cycle, the algorithm stops immediately when it finds a circle (which means the algorithm is "on-the-fly"), the run time in this situation is $O(log(|D|))$, where $|D|$ is the cycle length.

4.3 Calculating the Cycle Length

When calculating the length of the cycle, we combine the techniques of counting the length from Algorithm 3 and the *min* index idea from Algorithm 4. We will use the *span* property to store the steps from the current node to its *far* node, and *distance* to store the distance from current node to its smallest successor node. In this way, when the algorithm returns true, $distance(i)$ would be the length of the cycle, since it indicates how far it is from the smallest index. The algorithm is given in Algorithm 5. After knowing the length D, by traversing starting from the node i, which has $i == min(i)$, and we will reach all the nodes inside the cycle.

Algorithm 5 Calculating Cycle Length D

```
1: parfor i ← 1, n do          → assign a processor for each node of the linked list
2:     if next(i) ≠ ⊥ then
3:         far(i) := next(i)
4:         min(i) := next(i)
5:         span(i) := 1
6:         distance(i) := 1
7:     else
8:         far(i) = ⊥
9:         span(i) := 0
10:        distance(i) := 0
11:    end if
12:    while far(i) ≠ ⊥ do
13:        if min(far(i)) < min(i) then
14:            distance(i) := span(i) + distance(far(i))
15:            min(i) := min(far(i))
16:        end if
17:        if i == min(i) then
18:            return distance(i)
19:        end if
20:        span(i) := span(i) + span(far(i))
21:        far(i) := far(far(i))
22:    end while
23: end parfor
24: return 0
```

The method to calculate cycle length can be explained as follows:

1. During initialization, set *far* and *min* to the index of its successor node, if it has successor node. *Far* and *min* record the current furthest and smallest id it could reach. Set $span(i)$ and $distance(i)$ to 1, which is the current distance from i to $far(i)$ and $min(i)$.

2. During each iteration inside the while loop, if $min(far(i)) < min(i)$, which means the new $min(i)$ will be a node between $far(i)$ and $far(far(i))$, we update $min(i)$ and $distance(i)$. Otherwise we remain $min(i)$ and $distance(i)$ the same. Then $far(i)$ keeps jumping to the furthest reachable node and $span(i)$ updates accordingly.

3. When we detect $i == min(i)$, the current $distance(i)$ is the distance from i to $min(i)$, which is the distance from i to itself, i.e., the length of the cycle. Then the algorithm returns $distance(i)$.

5 Conclusions

Based on the proof and analysis, we introduce an efficient parallel cycle detection algorithm, which requires $O(n)$ processors and $O(log(n))$ time. We introduce our algorithm on a CREW PRAM model, which makes it easily adapted on different parallel computers. The efficiency and simplicity of our algorithm make it significantly improved from the existing methods. The algorithm also provides a useful insight which can be further referenced when dealing with linked list in parallel and the transitive closure bottleneck problem[KK93].

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[BGS72] Michael Beeler, R William Gosper, and Richard Schroeppel. Hakmem. 1972.

[Bor77] Allan Borodin. On relating time and space to size and depth. *SIAM journal on computing*, 6(4):733–744, 1977.

[Bre80] Richard P Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

[Flo67] Robert W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14, 1967.

[FM71] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 129–131. IEEE, 1971.

[Gre92] Raymond Greenlaw. A model classifying algorithms as inherently sequential with applications to graph searching. *Information and Computation*, 97(2):133–149, 1992.

[Imm89] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18(3):625–638, 1989.

[Joh90] David S Johnson. A catalog of complexity classes. In *Algorithms and complexity*, pages 67–161. Elsevier, 1990.

[KJ88] BS Kaliski Jr. R. l. rivest, and a. t. sherman. is the data encryption standard a group. *Journal of Cryptology*, 1:336, 1988.

[KK93] Ming-Yang Kao and Philip N Klein. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. *Journal of Computer and System Sciences*, 47(3):459–500, 1993.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303, 2014.

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[Sip97] M Sipser. Introduction to the theory of computation, pws pub. *Co., Boston*, 1997.

[Ski98] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[Tes01] Edlyn Teske. On random walks for pollard's rho method. *Mathematics of computation*, 70(234):809–825, 2001.

[VG87] Allen Van Gelder. Efficient loop detection in prolog using the tortoise-and-hare technique. *The Journal of Logic Programming*, 4(1):23–31, 1987.

[Wyl79] James C. Wyllie. The complexity of parallel computations. *Technical Report TR-79-387, Department of Computer Science, Cornell University*, 1979.