Parallel Breadth First Search

Wenjie Huang

Abstract: In this project, we explored the serial BFS search and Parallel BFS. For the PBFS, we use the Bag structure describe in this paper[1] and construct a customer reducer. We also discussed about the race bug and speed up of the parallel algorithm and analyzed the performance according to graph size and number of processors. The program use cilk++ library, running on a share memory system.

1. Introduction

As a well known algorithm, BFS is widely used in graph level searching. It explored nodes in graph level by level and marked the node by depth. Usually, BFS use FIFO queue to keep nodes waiting to be explored.

For the PBFS, there are many data structures can be used to implement the algorithm. In concern of efficiency, we choose the Bag structure instead of linked-list and array. We implement bag as a customer reducer[2], since there may be “stealing” when the cilk scheduler schedule the next strand to be executed.

2. Serial BFS

In this session, we will give a simple overview on serial BFS algorithm.

BFS algorithm[3] explores vertices and edges starting from a specified vertex. Unlike its sibling "Depth First Search",

breadth first search searches nodes one level away from the root node before traversing deeper. BFS and DFS are both create spanning trees with certain properties which are the basis for many graph algorithms.

BFS algorithm has been used in many applications, such as web crawling. Being able to find the "best pages" quickly is important for search engines such as Google, it’s used for searching for pages through the web to yield "high quality" pages.

The pseudo code of BFS is as below:

```
procedure BFS(G,v):
   create a queue Q
   enqueue v onto Q
   mark v
   while Q is not empty:
      t ← Q.dequeue()
      if t is what we are looking for:
          return t
      for all edges e in G.adjacentEdges(t) do
       u ← G.adjacentVertex(t,e)
       if u is not marked:
           mark u
           enqueue u onto Q
return none
```

The time complexity of BFS is: O(V + E), where V is the number of vertices and E is the number of edges.

3. Parallel BFS

In this session, we will give details about cilk++ reducers, read-in graph structure, parallel BFS algorithm and data structure it used.

3.1 Customer Reducer

Cilk++ provides reducers to address the

problem of accessing nonlocal variables in parallel code.

Normally, locks are required in multi-thread programming when threads are trying to access some global variables to ensure the data consistency. But locking will slow down the runtime of program, since other threads need to wait until the lock is released. So we use reducer to solve this problem by creating a local view of global variable for each thread.

Before we explore how reducer works, we need to introduce what is "steal" first.

```
do_somthing();//strand 1
cilk_spawn foo(); //strand 3
foo();          //strand 2
do_somthing();  //strand 4
```

The strand execution flow is illustrated in figure 1.

We call a processor to execute the strand "worker". When there is only a single worker, the strand above is executed in sequential order. In this case, there is no

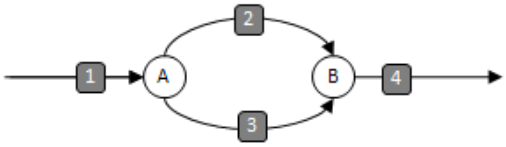

Figure 1

stealing and no data race. But whether the code should be executed in parallel or in sequential is scheduled by cilk++ scheduler. That means, even when there are more than two workers, the code may be executed in sequential order in one worker while other workers are all busy. Figure 2 shows the case that strand is executed in sequential order.

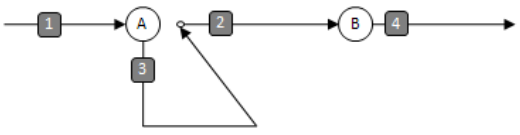

Figure 2

When the number of workers is larger than one, it may happen that strand 2 and strand 3 is executed in parallel. This is aimed at making full use of idle processors. The scheduler will steal strands which have not been executed yet from busy workers to those idle workers. Under this situation, data race would be a serious problem and may affect the data correctness. The figure 3 shows what happened when there is a stealing.

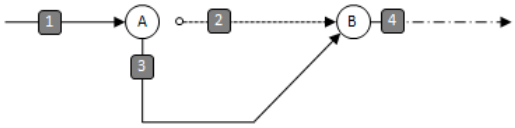

Figure 3

In the figure, after parallel strand 2 and strand 3, the view of variable on point 1 may be generated into two different views. So we need to "reduce" this two different views into one consistent view when arriving point 4. So we need reducers to reduce two views into one view.

By introducing reducer, we can avoid lock on shared variables, since each thread can keep a local view on variable, and after that the reducer will merge two views into one.

Cilk++ library provides several default reducers. It can reduce integer, string, or list, etc. But what we need is a new reducer since we choose to implement the "Bag" structure. In summary, the reducer should hold per-strand view of bag and merge bags from different strand into one bag if necessary.

3.2 Sparse Matrix

In this project, we will read the RMAT graph. Since the graph may be sparse when the edge factor is very small, we use what's called "compressed sparse adjacency lists". It gives the pointer to the first neighbor of every node. If the graph is dynamic, the list structure may encounter some problems. But in this project, we only consider the static graph. So using the adjacency list is very efficient when loop over all of the neighbors of a given vertex.

3.3 PBFS Algorithm

In session two, we have a short overview of BFS algorithm and give the pseudo code. In serial BFS, it pushes all neighbors of a current

vertex into queue, and explore the pushed neighbors in first-in-first-out order, which is sequential. And all these neighbors are in the same level (depth).

In parallel BFS, we still explore vertex level by level. What can be parallel is the part that accessing neighbors of a current vertex. Also, instead of using queue, we use bag --- an efficient structure to hold the vertices waiting to explore. To simplify, we will give the pseudo code[1] of PBFS to show the basic idea of parallel. Before we do that, explanations of some notation is needed:

d: level to the starting vertex.
IN_BAG: hold the vertices of current level d.
OUT_BAG: hole the vertices of neighbors of current level. That is, the vertices for next level d+1.
BAG-INSERT: insert a vertex into a bag.
BAG-SPLIT: split a bag into two half.

```
REPBFS(G,v0)
   parallel for each vertex v ∈ V(G)−{v0}
      v. dist = ∞
      v0. dist = 0
      d = 0
      IN_BAG = BAG-CREATE()
      OUT_BAG = BAG-CREATE()
      BAG-INSERT(IN-BAG, v0)
   while ¬BAG-IS-EMPTY(IN_BAG)
         PROCESS-LAYER(IN_BAG ,OUT_BAG,d)
         d = d+1
         IN_BAG = OUT_BAG
         Clear OUT_BAG

PROCESS-LAYER(in-bag, out-bag,d)
   if BAG-SIZE(in-bag) < GRAINSIZE
      for each u ∈ in-bag
         parallel for each v ∈ Adj[u]
            if v. dist == ∞
            v. dist = d+1// benign race
            parent[v] = u// benign race
            levelSize[d+1]++     // data race
        BAG-INSERT(out-bag, v)
        return
      new-bag = BAG-SPLIT(in-bag)
      spawnPROCESS-LAYER(new-bag,out-bag,d)
      PROCESS-LAYER(in-bag, out-bag,d)
      Sync
```

From the code above, we can see that it continues on spawn the current level into child thread until the number of nodes in one piece is small than the grandsize. Each time it spawn, a new bag whose number of nodes become half of the original bag is created. It's acting like the divide stage in divide and concur algorithm. And in each child thread, get access to each neighbors of current node is also parallel. At this point, there may be race problem. The line with blue comment has the "benign race", which means the race problem won't actually affect the correctness of this algorithm, because there is no disagreement about what v.dist should be, and if v has multiple parents, it doesn't matter in the end whether parent[v] turns out to be u0 or u1.

The line with red comment will introduce data race which will cause problem. But the problem won't affect the correctness of the algorithm also. The level size is used to calculate how many nodes in one level, that is, for checking the correctness of the algorithm.

The insurance of data race make us hard to evaluate the performance of the algorithm. We call this phenomenon nondeterministic. By carefully introduce lock can eliminate the nondeterministic but will also slow down the run time. The lock can be added as below:

```
set = FALSE
if TRY-LOCK(v)
if v. dist == ∞
    v. dist = d+1
    parent[v] = u
    levelSize[d+1]++
   set = TRUE
RELEASE-LOCK(v)
 if set
   BAG-INSERT(out-bag, v)
```

There is a redundant line:

```
if v. dist == ∞
```

Double check this variable is aim at reduce unnecessary lock so that decrease the time on waiting to lock the variable.

3.4 Bag Data Structure

There are many ways to implement the bag. That is, we can choose array, list or tree or whatever structure that can hold the nodes and implement split, insert and union operation. In this session, we will start with array and list. Review the pseudo code in session 3.3, we know that the bag should have the flowing function: insert, split and union.

By analyze the time complexity of operation on array and lists, we give the reason why we choose pennant structure.

Array

Insert: The insert operation of array is O(1). Just append the element to the end of array.

Split: The split operation of array is O(1), since we know the index and we can locate the middle point in constant time.

Union: The union operation of array is O(n). Because union two array need to scan the entire array.

List

Insert: The insert operation of list is O(1). Just append the element to the end of list.

Split: The split operation of list is O(n). Because for each operation it has to scan the entire list to find the middle point.

Union: The union operation of list is O(1).

Pennant

We will introduce the pennant structure here. And give the details about why the union and slit operation of pennant is more fast than array and list. Bag is a collection of pennant.

A pennant is a tree of 2^k nodes, where k is a nonnegative integer. Each node x in the tree contains two pointers x.left and x.right to its children. The root of the tree has only left child, which is a complete binary tree of the remaining elements.

Union: The union operation of pennant can complete in O(1) by the flowing function:

```
PENNANT-UNION(x, y)
    y. right = x. left
    x. left = y
    return x
```

Figure 4 illustrate the process of union.

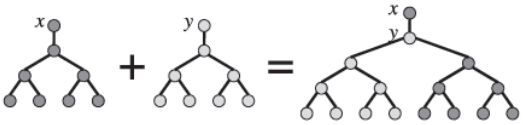

Figure 4

Split: The split operation of pennant is to split the tree into two subtree that each of the tree contains the same number of nodes. The flowing function of split only has O(1) complexity.

```
PENNANT-SPLIT(x)
    y = x. left
    x. left = y. right
    y. right = NULL
    return y
```

Bag

As stated above, bag is a collection of pennants. Figure 5 illustrate the structure of bag:

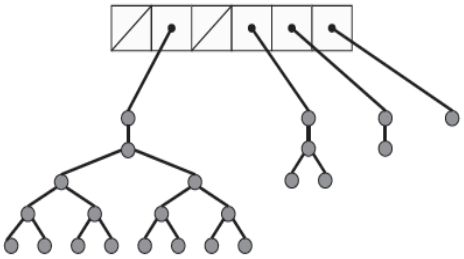

Figure 5

In the figure 5, we can implement the bag as an array, and each element contains a pointer to the pennant. And the number of nodes of each pennant is the power of 2. And the power is increase as the index of array. The flowing insert function can achieve this:

```
BAG-INSERT(S, x)
    k = 0
    while S[k] _= NULL
      x = PENNANT-UNION(S[k],x)
    S[k++] = NULL
    S[k] = x
```

The complexity of insertion of bag is O(r), where r is the length of bag.

When doing the union operation of bag, it is actually union the pennant at the corresponding position one by one. Since we have to make sure that after union, the number of nodes in each pennant is still increase by the index of array, we will use the flowing method:

```
BAG-UNION(S1,S2)
    y = NULL //The "carry" bit.
    for k = 0 to r
      (S1[k], y) = FA(S1[k],S2[k], y)
```

The function FA is a function that union the pennant with "carry bit". The idea is very similar to the addition with carry bit. If the position is "full", than move the content to next position by set a carry bit. The table of rules of carry bit is illustrated as below:

x	y	z	s	c
0	0	0	NULL	NULL
1	0	0	x	NULL
0	1	0	y	NULL
0	0	1	z	NULL
1	1	0	NULL	PENNANT-UNION(x,y)
1	0	1	NULL	PENNANT-UNION(x,z)
0	1	1	NULL	PENNANT-UNION(y,z)
1	1	1	x	PENNANT-UNION(y,z)

Figure 6

The complexity of union of bag is O(r), where r is the length of bag.

The split of bag is essentially split the pennant of each position into half and make a new bag. The function of bag-spilt is showed as below:

```
BAG-SPLIT(S1)
    S2 = BAG-CREATE()
    y = S1[0]
    S1[0] = NULL
   for k = 1 to r
     if S1[k] != NULL
       S2[k−1] = PENNANT-SPLIT(S1[k])
    S1[k−1] = S1[k]
    S1[k] = NULL
    if y != NULL
      BAG-INSERT(S1, y)
return S2
```

The complexity of split operation is also O(r), where r is the length of bag.

4. Experiment

The experiment is based on running on Triton. We will test for different size of graph and processors. And calculate the speed up and efficiency according to the problem size and number of processors. Consider of efficiency, we remove the lock stated in session 3.3 and take the calculation of levesize out of the parallel part. All experiment use grandsize 128.

For the small graph, it's hard to get a linear speed up. The effort to spawn thread and manage the reducer eliminates the benefit from parallelism. The table below has showed the result from small graph:

Case 1:
Vertices: 8192 Edges: 131072
PBFS reached 6 levels and 5692 vertices

Processors	Time /s
1	4.445791e-03
2	3.308058e-03
4	3.937960e-03
8	3.904104e-03
16	4.547119e-03
32	7.192850e-03

The speed up and efficiency would be:

Processor	Speedup	Efficiency
1	1	1
2	1.34	0.67
4	1.12	0.28
8	1.13	0.14
16	0.98	0.06
32	0.62	0.02

From the table above we can see that PBFS is highly inefficient when the graph size is small. And the serial BFS is faster than PBFS in this case. Figure 7 shows the speed up and efficiency according to the number of processors of case 1.

For a larger graph, we can get approximately linear speed up. But how linear it is doesn't only depend on the size of graph. It also depends on the structure of graph. If the graph is sparse and highly unconnected, or the starting vertex only has a small sub graph, it will not give you a good linear speed up. Because actually it didn't has a lot of work as the total number of vertices to do. It only explores a small portion of the big graph. We have several test cases that cover the conditions: highly connected large graph, highly unconnected large graph.

Case 2:
Vertices: 2^{20} Edges: 2^{20}

Processors	Time /s
1	1.304522e-01
2	8.925009e-02
4	4.934216e-02
8	2.851987e-02
16	2.309513e-02

The speed up and efficiency would be:

Processor	Speedup	Efficiency
1	1	1
2	1.56	0.73
4	2.74	0.66
8	4.67	0.57
16	5.65	0.35

In this case, PBFS reached 9 levels and 173330 vertices. Though it's only 17% of the graph size but it's a considerable amount of work. And the speed up and efficiency is better than the small graph in the first case.Figure 8 shows the speed up and efficiency according to the number of processors of case 2.

Case 3:
Vertices: 2^{20} Edges: 2^{23}

Processors	Time /s
1	3.707099e-02
2	2.036500e-02
4	1.192904e-02
8	9.295940e-03
16	9.577036e-03

The speed up and efficiency would be:

Processor	Speedup	Efficiency
1	1	1
2	1.82	0.91
4	3.11	0.77
8	3.99	0.49
16	3.87	0.24

In this case, the edge size is larger than case 2 but the starting vertex can only reached 3 levels and 2018 vertices. Even when the total size of problem increases, the work in fact decreases. Overall the speed up and efficiency is not as good as case 2.Figure 9 shows the speed up and efficiency according to the number of processors of case 3.

Case 4:
Vertices: 2^{25} Edges: 2^{25}

Processors	Time /s
1	1.155691e+00
2	6.664321e-01
4	4.065211e-01
8	2.197721e-01
16	2.150211e-01

The speed up and efficiency would be:

Processor	Speedup	Efficiency
1	1	1
2	1.73	0.86
4	2.84	0.71
8	5.26	0.65
16	5.37	0.34

In these cases, the number of vertices and edges are both larger than case 2. The PBFS reached 14 levels and 1433066 vertices, which is 10 times larger than case 2. But overall the speed up doesn't have a big improvement. Figure 10 shows the speed up and efficiency according to the number of processors of case 4.

We know that we can't get a better speed up by increasing the problem size when the problem size is large enough. Change the grandsize can get a better result, but since we can't find the optimal formula to get the grandsize based on

number of vertices and edges, since the graph is randomly generated and the actual work is related to the structure of graph, we hard coded it as 128.

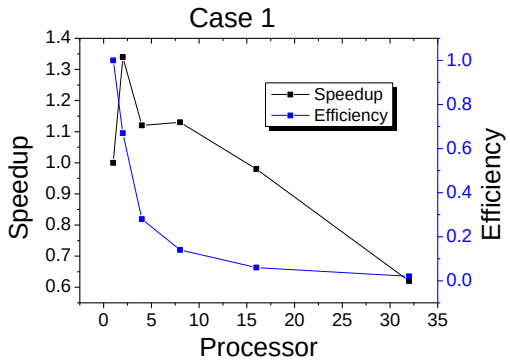

Figure 7

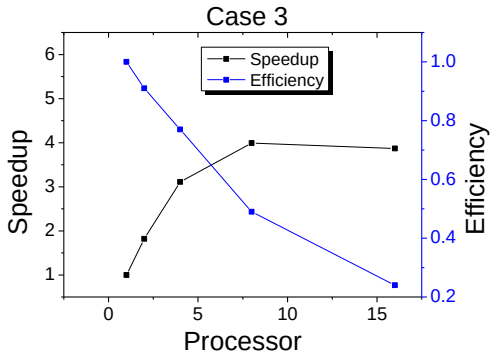

Figure 9

We also compare the average instructions per strand of these 4 cases:

Case	Instructions/strand
1	685
2	905
3	676
4	1417

If the instructions/strand is small, we say that the overhead of spawn is high.

The graph size of case 3 is much larger than case 1 and case 2, but it has a larger overhead when spawn the child. It proves that not only the size of graph but also the structure of the graph have impact on the speed up.

Every time there is a "work stealing", the runtime system creates a local view for that strand. We call that frame. When strands met the sync point, their local views need to be reduced into one view. So the runtime system will dynamically create and destroy frames. When the create and destroy operation happen too frequently, it will slow down the runtime. Cilkview tool provides a statistic of frame count.

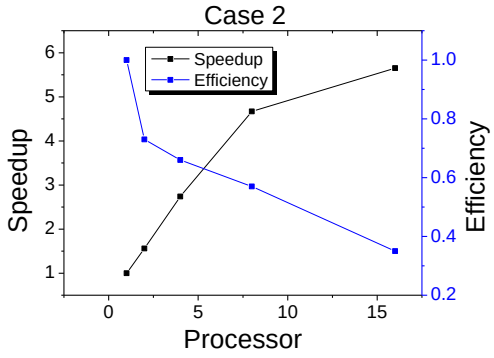

Figure 8

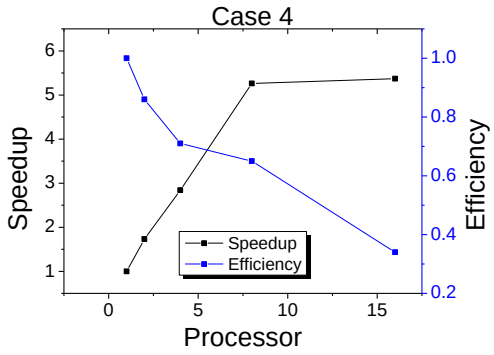

Figure 10

Case	Frame count
1	264883
2	1819757
3	16436275
4	9311738

We can see that the case with highest overhead also has the highest frame count.

5. Conclusion

In this project, we explore both details of PBFS algorithm and technique in Cilk++.

The bag structure use a smart way to eliminate the effort of insertion , union and split. And it allocates the same amount of memory each time it is constructed, the runtime system will optimize this process by remembering the size.

By analyzing the speed up according to problem size, we realize that the graph problem is different from other problem with a well defined structure. In practice, the strategy to solve a

problem may be more complexity according to how sparse or dense of a graph is, since the performance may vary based on the structure of graph.

The statistic data from cilkview provides hint and idea when optimize the program. If the overhead of parallelism is too high, it may be the reason that the program parallels the problem into chunks which are too small. Modify the grandsize may result a better parallelism.

6. References

[1] Charles E. Leiserson, Tao B. Schardl. A Work-Efficient Parallel Breadth-First Search Algorithm. In SPAA 10' Proceedings of the 22nd ACM symposium on Parellelism in algorithms and architectures, page 303-314.

[2] Intel Cilk++ Programming Guide

[3] Wiki: Breadth-first search http://en.wikipedia.org/wiki/Breadth-first_search