CS214, Winter 2022, **Project #2 Final-Report**

Name: Prince Choudhary **Student ID: 862254827**

1 Parallel BFS

Two Grace Days - I would take two grade days for this project assignment. The reason is my laptop had screen flickering problem which got more intense during the last week and I had to replace it. The new laptop needed extra setup time.

Git-link - https://github.com/ucrparlay-class/project2-non-cilk-princerokn

1.1 Explanation

We are provided with data in CSR format.
The bfs.cpp has already implemented codes to parse it and form offset[] and Edge[] arrays. The sequential algorithm bfs is already implemented that will be used for verifying the correctness of our code. We only have to implement the parallel bfs function.

1.1.1 Logic for BFS

1) We need to implement logic for switching between sparse and dense modes depending on the size of current frontier. We also need to keep calculating the distance of the nodes and update the dist[] array. Our exit condition would be when frontier has size 0.
2) First, we will make all the entries in dist[] -1 and all the nodes not visited.
3) Since we have a source we will put this in currentFrontier[] and make its distance 0 and mark it as visited.
4) We will define several variable here to be used in bfs function or in dense or sparse functions. This is to avoid redundant reallocation of memory resource.
5) We will start a while loop which would exit only when the current frontier size is 0, or there is no node left to visit in that connected graph.
6) Our first thing would be to decide which mode we should run, For this we would declare two bool variable. One to decide which one to go for and another to keep track of what was our previous mode.
7) Since logic of my sparse and dense mode make use of different structure of current frontiers, We need to know when we are switching as we need to change the structure.
8) If we are repeating same mode, then there won't be any need to change the structures of current frontiers as they would be the same.
9) For dense mode we are using current frontier as a bool array of size n, this

would eliminate our need to pack the elements each time, along with this it would be simpler to access the elements.
10) For sparse mode we are using packed structure with size equal to number of elements in current frontier as our current frontier.
11) When we are switching from sparse to dense mode, we will have a our current dense frontier of n size. To get this we will check all the elements in current frontier, and set the corresponding bool values to true and all other values would be false
12) When we are switching from dense to sparse mode, we will find the prefix sum of our current dense frontier and pack them to get the elements which had bool values 1 and pack them.
13) Now depending on mode we had decided, we will call the dense or sparse mode function.
14) Both the functions would return array of next Frontier.
15) We will keep track of size of next frontier using reference variable.
16) We will update the distances of nodes at each iteration and the current distance from source would keep increasing with the iterations

1.1.2 Logic for Sparse Mode

1) In the sparse mode we will use our logic to flatten, pack and scan, we will combine the flatten and pack functions so as to avoid passing the same set of arguments to pack function and get the return and then re-route to the bfs function.
2) We will implement our scan function we will be using coarsening in this function, where if size is small then we would just run it sequentially.
3) For flattening we will find the all the neighbors of our current frontier, so as to do this in parallel we will first find the locations where these neighbors should go, we will make use of scan algorithm to find that.
4) Now that, we have the locations where each node can go, we will place all the neighbors bit which are supposed to be in a flattened array as many of them would have already been visited, we will use parallel for and compare and swap to achieve this efficiently.
5) Now its time to pack them, we will run scan to get the locations of each node which can be included in the next frontier.
6) We will place the elements in the packed array using parallel for.
7) This packed array is our next frontier and we will return this.

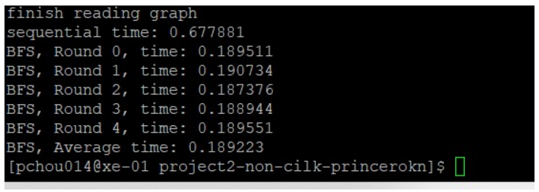
1.1.3 Logic for Dense Mode

1) In the dense mode we will have current dense frontier array of length n and all the nodes which are in this with bool value as 1 would be considered present and the one which are having value 0 are considered absent.
2) We will traverse through each element in visited[] array, if any of the element is not visited before. we will check if its neighbor is in the current dense frontier array or not.

3) If its there that means this node can go to our next frontier, we will add it to the array of next frontier and break.
4)When we have checked all the items, we will have our next frontier ready and we will return it.

1.2 Input and Results

I created several test cases to test my code for various scenarios and corner cases, I wrote a script that would generate the CSR graph values similar to the input provided to us. I have pushed few of the tests in the Github repository as well. Besides custom test cases, I tested my code for the 3 data files provided to us, with taking different points as source.
Here is the screenshot of my test from "soc-LiveJournal_sym,adj" with source as 0.



```
finish reading graph
sequential time: 0.677881
BFS, Round 0, time: 0.189511
BFS, Round 1, time: 0.190734
BFS, Round 2, time: 0.187376
BFS, Round 3, time: 0.188944
BFS, Round 4, time: 0.189551
BFS, Average time: 0.189223
[pchou014@xe-01 project2-non-cilk-princerokn]$
```

Figure 1: Output

1.3 code

Below is my code in the file bfs.h

```
//Prince's Prefix sum
int* scan(int arr[], int start, int end) {
    int n = end - start + 1;
    int* arr1 = new int[n/2];
    int* res = new int[n];
    res[0] = arr[0];
    if(n > 1) {
        if (n <= 100) { //coarsening in scan
            for(int i = 0; i < n/2; i++){
                arr1[i] = arr[2 * i] + arr[2 * i + 1];
            }
            int *arr2 = scan(arr1, start, start + (end - start) / 2);
            for(int i = 1; i < n; i++){
                if (i % 2 == 1)
                    res[i] = arr2[i / 2];
                else
                    res[i] = arr2[(i - 1) / 2] + arr[i];
            }
            parallel_for(1, n, [&](int i) {
                if (i % 2 == 1)
                    res[i] = arr2[i / 2];
                else
                    res[i] = arr2[(i - 1) / 2] + arr[i];
            });
            delete[] arr2;
        }
        else{
            parallel_for(0, n / 2, [&](int i) { arr1[i] = arr[2 * i] +
                 arr[2 * i + 1]; });
            int *arr2 = scan(arr1, start, start + (end - start) / 2);
            parallel_for(1, n, [&](int i) {
                if (i % 2 == 1)
                    res[i] = arr2[i / 2];
                else
                    res[i] = arr2[(i - 1) / 2] + arr[i];
            });
            delete[] arr2;
        }
    }
    delete[] arr1;
    return res;
}

//Prince's Packing parallel
//Prince' Flatten parallel
//Combined packing and flattening
int* flattenandPack(int* curFrontier, int curFrontSize, int* offset,
     int* E, bool* visited, int& nextFrontierSize){
    int* sizeOfArrays = new int[curFrontSize]; //This array will store
        the size of each neighbor of nodes in frontier
    int* offsetForArrays = new int[curFrontSize]; //This array will
        offset values for arrays in frontier.
    //This will determine size of neighbor of each node in frontier and
        store.
    parallel_for (0, curFrontSize, [&] (int i) {
                    sizeOfArrays[i] = offset[curFrontier[i]+1] -
                        offset[curFrontier[i]];
                }
    );
    offsetForArrays = scan(sizeOfArrays, 0, curFrontSize); //This will
        set the offset values in the array

    int sizeofflattenedArray = offsetForArrays[curFrontSize-1];
    int* flattenedArray = new int[sizeofflattenedArray]; //array with
        original values
    int* flattenedArrFlag = new int[sizeofflattenedArray]; //flag array
        with 0 or 1
    parallel_for (0, sizeofflattenedArray, [&] (int i)
        {flattenedArrFlag[i] = 0;}); // make all flags false initially.
    //Flatten the array and store it
    parallel_for (0, curFrontSize, [&] (int i) {
                    int off = (i>0)?offsetForArrays[i-1]:0;
                    int node = curFrontier[i];
                    int location = offset[node];
                    parallel_for(0,sizeOfArrays[i], [&] (int j) {
                        int y = E[j+location];
                        flattenedArray[off+j] = y;
                        if(!visited[y] &&
                            __sync_bool_compare_and_swap(&visited[y],
                            false, true))
                            flattenedArrFlag[off+j] = 1;
                    });
                }
    );
    // flattenedArray contains elements after being flattened
    // flattenedArrFlag contains elements after CAS
    delete[] sizeOfArrays;
    delete[] offsetForArrays;
    //Flattening starts
    int* prefixSumArr = new int[sizeofflattenedArray];
    prefixSumArr = scan(flattenedArrFlag,0,sizeofflattenedArray);
    nextFrontierSize = prefixSumArr[sizeofflattenedArray-1];
    int* packedArray = new int[nextFrontierSize];
    //This will pack the flattened array.
    parallel_for (0, sizeofflattenedArray, [&] (int i) {
                    if(flattenedArrFlag[i])
                        packedArray[prefixSumArr[i] - 1] =
                            flattenedArray[i];
                }
    );
    delete[] prefixSumArr;
    delete[] flattenedArrFlag;
    return packedArray;
}
//Prince's Dense Backward function
bool* denseMode(int n, bool* denseFrontier, int* offset, int* E, bool*
     visited, int& nextFrontierSize){
    bool* nextFrontier = new bool[n];
    int count = 0;
    parallel_for (0, n, [&] (int i) {nextFrontier[i] = 0;}); // set
        initial value of new Frontier
    for (int i = 0; i < n; i++) {
        if(!visited[i]){
            //check if its neighbour is in the frontier
            //if yes the add it to the new frontier
            for(int j = offset[i]; j < offset[i+1]; j++){
                if(denseFrontier[E[j]]){
                    visited[i] = true;
                    nextFrontier[i] = 1;
                    count++;
                    break;
                }
            }
        }
    }
    nextFrontierSize = count;
    return nextFrontier;
}
//Prince's BFS function
void BFS(int n, int m, int* offset, int* E, int s, int* dist) {
    bool *visited = new bool[n]; //This is flag to check visited or not.
    int *curFrontier = new int[n];
    int curFrontSize = 0;
    int nextFrontierSize = 0;
    parallel_for(0, n, [&](int i) { dist[i] = -1; }); //initialize all
        the elements in dist[i] to -1
    parallel_for(0, n, [&](int i) { visited[i] = false; }); //make all
        elements not visited.
    int curDistance = 0;
    dist[s] = curDistance; // source
    visited[s] = true; //making source visited
    //This will create 1st frontier
    curFrontier[curFrontSize++] = s;
    //0 means sparse and 1 means dense
    bool mode = 0;
    bool todo = 0;
    int *nextFrontier;
    bool* denseFrontier = new bool[n];
    bool* nextDenseFrontier;
    int* prefixSumArr;
    while (curFrontSize > 0) {
        curDistance++;
        if(curFrontSize <= n/10+1)
            todo = 0;
        else
            todo = 1;
        if(mode != todo){
            //switch mode logic.
            if(todo){
                //switch to dense, make curFrontier of n sized boolean
                parallel_for (0, n, [&] (int i) {denseFrontier[i] = 0;});
                parallel_for (0, curFrontSize, [&] (int i)
                    {denseFrontier[curFrontier[i]] = true;}); // setting
                    the bits for cur dense Frontier
            }
            else{
                //switch to sparse mode, set a packed current frontier.
                //this switch happen only when we are moving from dense
                    to sparse mode
                //Since our current Frontier is in dense Frontier format,
                    we need to obtain a packed cur Frontier
                int* denseFrontierInt = new int[n];
                parallel_for (0, n, [&] (int i) {denseFrontierInt[i] =
                    denseFrontier[i];});
                prefixSumArr = scan(denseFrontierInt, 0, n);
                int ptr = 0;
                if(prefixSumArr[0] == 1)
                   curFrontier[ptr++] = 0;
                for(int i = 1; i < n; i++) {
                   if (prefixSumArr[i] != prefixSumArr[i - 1])
                       curFrontier[ptr++] = i;
                }
            }
            mode = todo;
         }
         if (!mode) {
            //Run in Sparse Mode
            nextFrontier = flattenandPack(curFrontier, curFrontSize,
                offset, E, visited, nextFrontierSize);
            curFrontier = nextFrontier;
            parallel_for(0, nextFrontierSize, [&](int i) {
                dist[curFrontier[i]] = curDistance; }); //Fill the
                distance array
         } else {
            //Run in dense Mode
            nextDenseFrontier = denseMode(n, denseFrontier, offset,
                E,visited, nextFrontierSize);
            denseFrontier = nextDenseFrontier;
            parallel_for (0, n, [&] (int i) {if(denseFrontier[i]) dist[i]
                = curDistance;});
         }
         curFrontSize = nextFrontierSize;
     }
  }
```