Parallelization Of Graph Traversal

*CS 535.V Multicore Programming Spring 2021-22 Project

Atalay Pabuşçu
dept. Computer Science
Ozyegin University
Istanbul, Turkey
atalay.pabuscu@ozu.edu.tr

***Abstract*—Graph data type is one of the most used data types in computer science and they consist of vertices and edges layer by layer. Searching algorithms are used to extract the graph structure. In this project, we aim to parallelize a graph traversal algorithm Breadth First Search(BFS). We focus on the BFS algorithm and parallelized this algorithm using OpenMP API.**

***Index Terms*—Multicore Programming; Graph Data Type, Breadth First Search(BFS), Graph Traversal, Parallelization, OpenMP, Intel VTune Profiler**

I. Introduction

In this project, we aim to parallelize the graph traversal process to make it faster. Graph data structures have an important place in computer science. Graphs consist of nodes and edges. Graph traversal refers to visiting each node [1]. This is also known as graph search. In our project, we parallelized one of the searching algorithms Breadth First Search. BFS is a technique to find the shortest path and uses a queue data structure to do that. [2].

We used OpenMP for parallelization of a sequential BFS code. OpenMP is a API that exist independently from the language and control the multiple threads and their works.

II. Traversal Algorithms

Graph traversal is also known as graph search. Traversal means visiting all the nodes in the graph. Graphs are one of the most known algorithms in computer science. Breadth First Search and Depth First Search are the simplest graph traversal algorithms. They have a lot of similarities. The biggest difference is visiting order of the nodes [3]. With both of them, we visit the nodes until visiting all the nodes or find a match. In this project our purpose is visiting all the nodes with avoiding cycles.

A. Breadth First Search

In BFS, visiting order is prioritized of the neighbors of the starting node before the deeper nodes. The algorithm uses a list and a queue data structure. It uses the list for keeping the visited nodes and uses the queue for keeping the neighbors of the visited nodes. Starts with visiting the first node and adding neighbors to the queue. Continues with visiting the node from the queue with the first come first out logic. If there are neighbours of the visited node, add these nodes to the queue and continue this process until all the nodes are visited.

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
```

Fig. 1. BFS Pseudo Code [4]

B. Depth First Search

In DFS, visiting order is prioritized of the deepest node in the frontier. The algorithm uses a list and a stack data structure. It uses the list for keeping the visited nodes and uses the stack for keeping the unvisited adjacent nodes. Unlike BFS, DFS uses a recursive algorithm. Starts with visiting the first node and adding adjacent unvisited nodes to the stack. Continues with visiting the node from the stack with the last in first out logic. If there are unvisited adjacent nodes, add these nodes to the stack and continue this process until all the nodes are visited.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Fig. 2. DFS Pseudo Code [5]

III. OpenMP

OpenMP is a multi platform shared memory multiprocessing API that supports C, C++, Fortran languages. We parallelise BFS Algorithm using OpenMP clauses. Some of OpenMP clauses we used are given below.

- ***# pragma omp parallel num_threads (<thread_count>):*** It allows the programmer to specify the number of threads that should execute the following block

- ***# pragma omp for:*** Takes the following for loop and splits up its iterations between threads:
  - Parallelizes the for loop by dividing the iterations of the loop among the threads
  - Default is equal size chuck (of iterations) for each thread
  - The variable index must have integer or pointer type (no char, float or double type)

  The expressions start, end, and increment statements in for loop must not change during execution of the loop
- ***# pragma omp parallel private(<variable_list>):*** Variables in private context are hidden from other threads. Each thread has its own private copy of the variable, and modifications made by a thread to its copy are not visible to other threads.
  The default context of a variable rules:
  - Variables with static storage duration are shared
  - Dynamically allocated objects are shared
  - Variables in heap allocated memory are shared. There can be only one shared heap
  - All variables defined outside a parallel construct become shared when the parallel region is encountered
  - Loop iteration variables are private within their loops

  Memory allocated within a parallel loop by the allocation function persists only for the duration of one iteration of that loop, and is private for each thread.
- ***# pragma omp parallel default(none) shared(<variable_list>):***
  - shared (list): Declares the scope of data variables in list to be shared across all threads
  - default (shared — none): Defines the default data scope of variables in each thread. With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block Only one default clause can be specified on an omp parallel directive. If default is shared: Stating each variable in a shared(list) clause. If default is none: Each data variable visible to the parallelized statement block must be explicitly listed
- ***# pragma omp parallel schedule(<schedule_type>, <chunk_size>):*** A schedule kind is passed to an OpenMP loop schedule clause: provides a hint for how iterations of the corresponding OpenMP loop should be assigned to threads in the team of the OpenMP region surrounding the loop.

The distribution of chunks between the threads is arbitrary. For schedule(dynamic, 4) and schedule(dynamic, 8) OpenMP divides iterations into chunks of size 4 and 8 respectively. The distribution of chunks to the threads has no pattern. The dynamic scheduling type is appropriate when the iterations require ***different computational costs***. This means that the iterations are poorly balanced between each other.

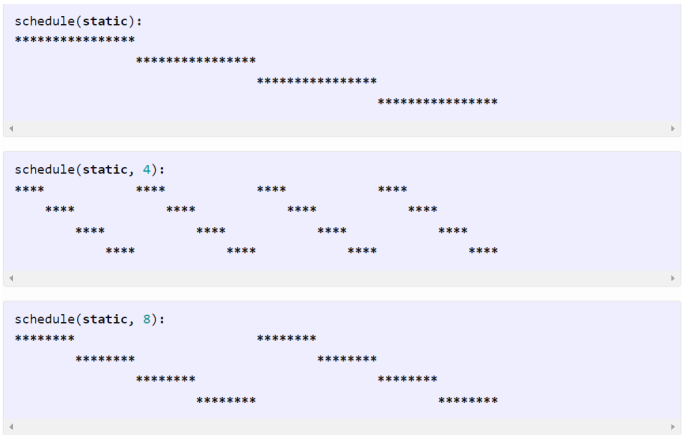

Fig. 3. OpenMP static schedule (chunk size: default, 4, 8) [6]

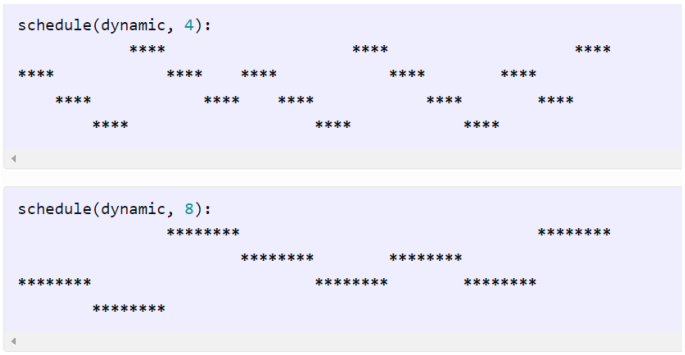

Fig. 4. OpenMP dynamic schedule (chunk size: 4, 8) [6]

IV. Algorithm Implementation

Traditional graph traversal algorithms are generally interested in visitation of nodes relative to each other. There is a pseudocode of sequential BFS algorithm in Fig.1. If we consider this code, "Q" is a queue data structure that will store unvisited graph nodes in order. "v" is the vertex which currently examined in the graph. We stay in an infinite research loop until the graph traversal queue (Q) is empty. For all investigating and marked nodes in order, we dequeue an element from the queue. Then we bring all neighbour nodes of the stated vertex, mark unvisited ones and enqueue them into graph traversal queue (Q) in order.

A. Sequential Breadth First Search Algorithm

A custom graph is given in Fig.5. If we determine the vertex node as node ***0***, we can easily find the breadth first search result of this graph by considering the pseudocode of BFS that is given in Fig.1. The Breadth First Search result of this graph: ***0 5 4 2 7 1 6 3***. We implemented this pseudocode using C programming language and obtained the traversal result as Fig.6.

In order to develop our sequential BFS Algorithm, we used ***C*** programming language in ***Ubuntu*** (v20.04 LTS) operating system. We compiled our programs with ***gcc*** (GNU Compiler Collection). We used the following command in order to

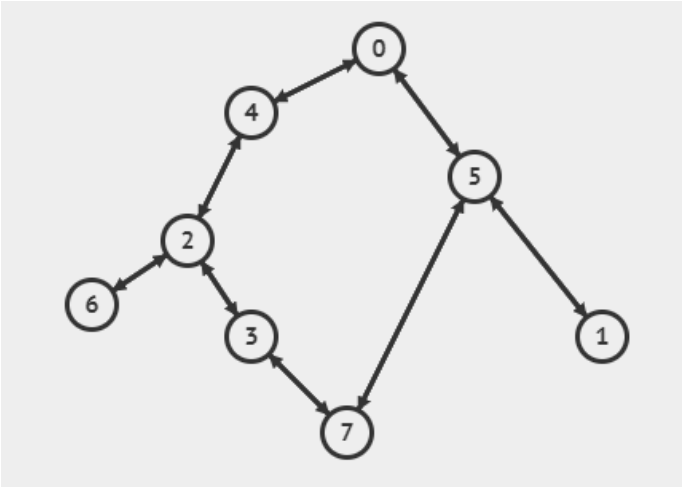

Fig. 5. Custom Graph

```
BFS Queue Traversal:  0  5  4  7  1  0  2  3
BFS elapsed time = 4.482269e-05 seconds
```

Fig. 6. Breadth First Search Algorithm Output for Custom Graph

compile our programs.

gcc -g -Wall -o project_name project_name.c

We used the following command in order to execute our programs.

./ project_name

Before we began to implement any graph traversal algorithms, we firstly declared some auxiliary structs that store our graph, queue and node variables.

```
// Queue Data Structure declaration in C
    programming language
typedef struct queue {
  int items[QUEUE_SIZE];
  int front;
  int rear;
}queue;
```

- ***items:*** It stores integer values that represent node indexes
- ***front:*** First element of the queue (consider FIFO, first added element will go out first)
- ***rear:*** Last element of the queue (new elements are added rear of the queue)

```
// Node variable structure declaration in C
    programming language
struct node {
  int vertex;
  struct node* next;
};
```

- ***vertex:*** It stores its own node index
- ***next:*** It stores next node's start address

```
// Graph variable structure declaration in C
    programming language
struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};
```

- ***numVertices:*** It stores total vertex number of the graph
- ***adjLists:*** It stores adjacent node array start address
- ***visited:*** It stores an integer that express visitation status (***0: unvisited, 1: visited***)

We used the ***AddEdgeToGraph*** method in order to create new custom and random graphs. Adding an edge to the graph is bi-directional (src node to dest note and vice versa).

```
// Add Edge to a Graph
void app_AddEdgeToGraph(struct Graph* graph,
    int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = app_CreateNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = app_CreateNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
```

And finally, our sequential BFS algorithm implementation in C language is shown below.

```
// Sequential Breadth First Search Algorithm
void app_BFS(struct Graph* graph, int
    startVertex)
{
  // Create a queue Q
  queue* search_queue = app_CreateQueue(); //
      Create searching queue to store
      unvisited nodes
  queue* traversal_result = app_CreateQueue();
      // Create traversal result queue to
      print graph traversal

  graph->visited[startVertex] = 0; //
      Determine the root (strat) vertex as 0
  // Mark v as visited and put v into Q
  app_Enqueue(search_queue, startVertex);

  // while Q is non-empty
  while (!app_IsQueueEmpty(search_queue))
  {
    // Remove the head u of Q
    int currentVertex =
        app_Dequeue(search_queue);
    app_Enqueue(traversal_result,
        currentVertex);

    struct node* adjNode =
        graph->adjLists[currentVertex];

    while (adjNode) // Iteration for all
        neighbour nodes
    {
```

```
    int adjVertex = adjNode->vertex;

    if (graph->visited[adjVertex] == 0) // Is
        this node visited
    {
      // Mark and enqueue all (unvisited)
          neighbours of u
      graph->visited[adjVertex] = 1;
      app_Enqueue(search_queue, adjVertex);
    }
    adjNode = adjNode->next;
  }
 }

 app_PrintGraphTraversal(traversal_result);
}
```

B. Parallel Breadth First Search Algorithm

When we started to work for parallelization of the sequential BFS Algorithm, we realized that there were some shortcomings in traditional sequential BFS Algorithm. This algorithm was not containing any iterations for investigated vertices. We needed to find another approach to implement parallelism using for loops. As a result of our research, we found the "Top-Down Approach" for Breadth First Search algorithm. [8]

1) BFS Algorithm Top-Down Approach:

0: **function** BFS
1: $frontier \leftarrow \{source\}$
2: $next \leftarrow [-1, -1, ... - 1]$
2: $parents \leftarrow [-1, -1, ... - 1]$
3: **while** frontier != $\{\}$ **do**
4: top-down-step(frontier, next, parents)
5: $frontier \leftarrow next$
6: $next \leftarrow \{\}$
7: **end while**
8: **return** tree
8: **end function**=0

Top-Down Approach takes the BFS Algorithm in two steps, BFS search, and Top-Down step. In this approach, there is no visitation control among nodes. It cares about ***parent situation*** of nodes in the graph. If we consider the above pseudocode:

- ***frontier:*** It represents the traversal queue of the graph. We enqueue new nodes that provide the conditions to frontier. These new nodes come from the "***next***" queue.
- ***next:*** It represents the next nodes to enqueue the frontier. It is filled in top-down-step function according to neighbourhood and parent situations.
- ***parents:*** It represents the parent status of the node. It stores only integer values (-1 means: None (Empty), Other number means: node indexes)

0: **function** TOP-DOWN-STEP($frontier$, $next$, $parents$)
1: **for** $v \in frontier$ **do**
2: **for** $n \in neighbors[v]$ **do**
3: **if** parents[n] = -1 **then**
4: $parents[n] \leftarrow v$
5: $next \leftarrow nextU\{n\}$
6: **end if**
7: **end for**
8: **end for**
8: **end function**=0

Top-Down-step function gets 3 parameters as "***frontier***", "***next***", "***parents***". In our implementation we also add another parameter to this function: ***Graph***. This parameter provides us to bring adjacent nodes of the stated vertex. In the first step of the Top-Down-step function, all nodes are traversed and checked vertices whether the frontier has this vertex or not. Then all neighbours of this vertex are controlled for the "***parents***" situation. If this node has a parent (set before) continue to control, if it has not any parent add (***enqueue***) this node index to the "next" queue. At the end of the Top-Down Approach, add (enqueue) all nodes of the "***next***" queue to the "***frontier***" queue.

2) Parallel BFS Algorithm Implementation:

In order to develop our parallel BFS Algorithm, we used C programming language in ***Ubuntu*** (v20.04 LTS) operating system. We compiled our programs with gcc (GNU Compiler Collection). We parallelised the BFS algorithm using the OpenMP API. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing ***parallel applications*** for platforms and it supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

We used the following command in order to compile our programs that contain OpenMP instructions.

gcc -g -Wall -fopenmp -o project_name project_name.c

We used the following command in order to execute our programs that contain OpenMP API.

./ project_name number_of_threads

For OpenMP API implementation "***omp.h***" library should be added into project

- # include "omp.h"

We implemented our OpenMP instructions (compiler directives) into the "***top-down-step***" function. We used ***default(none)*** clause in order to determine "***shared***" and "***private***" variables that were used in different threads.

- We determined shared variables as: "***total_nodes***", "***frontier***", "***next***", "***parents***"
- We determined private variables as: "***adjNode***", "***adjVertex***", "***vertex***", "***node***", "***thread_num***"

Our parallelised BFS algorithm implementation in C language is shown below.

```
// Parallel Breadth First Search Algorithm
    (OpenMP)
void app_TopDownStep(struct Graph* graph,
    queue* frontier, queue* next, queue*
    parents)
{
  # if (PARALLEL_OPENMP == TRUE)
  # pragma omp prallel
      num_threads(thread_count) default(none)
```

```
    shared(total_nodes, frontier, next,
    parents) private(adjNode, adjVertex,
    vertex, node, thread_num)
    schedule(dynamic, 1)
# endif
for(int vertex = 0; vertex < total_nodes;
    vertex++)
{
  if (app_IsThereAnyNode(vertex, frontier)
      == TRUE) // for v  frontier do
  {
    //printf("%d", vertex);
    struct node* adjNode =
        graph->adjLists[vertex]; // for n
        adjacentList[vertex]
    while(adjNode)
    {
      int adjVertex = adjNode->vertex;
#     if (PARALLEL_OPENMP == TRUE)
      #pragma omp parallel for
#     endif
      for(int node = 0; node < total_nodes;
          node++)
      {
        if (node == adjVertex) // for n
            adjacentList[vertex] do
        {
          if(parents->items[node] == NONE &&
              node != 0) // if parents[n] = -1
              then
          {
#           if (PARALLEL_OPENMP == TRUE)
            int thread_num =
                omp_get_thread_num();
#           else
            int thread_num = 1;
#           endif
            printf("\nAdjacent node of %d
                vertex (OpenMP Thread No: %d):
                %d\n",vertex, adjVertex,
                thread_num);
            printf("Parent of %d adjacent
                vertex (OpenMP Thread No: %d):
                %d\n", adjVertex, vertex,
                thread_num);
            parents->items[node] = vertex; //
                parents[n]  v
            printf("Add node %d to next queue
                (OpenMP Thread No: %d)", node,
                thread_num);
            //next->items[node] = node;
            app_Enqueue(next, node); // next
                next U {n}
          }
        }
      }
      adjNode = adjNode->next; // Traverse
          all adjacent nodes of this vertex
    }
  }
}
```

Then we add a schedule clause for scheduling the threads. We tried "***auto***", "***static***" and "***dynamic***" schedules and decided to use "***dynamic***" schedule due to showing best threading performance. We obtained best parallelism (threading) scores with ***schedule(dynamic, 1)*** clause. Because searching size is unpredictable per each iteration, dynamic schedule can performs better like this inequality situations unlike static schedule. The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are not as balanced as static method between each other. For our parallel Breadth First Search Algorithm implementation, vertices have ***different number of adjacent nodes***, therefore, the search result of each node may take different times from each other. In order to decrease idle time per each thread we need to execute another iteration for free thread when its execution ends. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. We determined ***chunk size*** as ***1*** for most efficiency.

Thread number can be determined by the user. Our system had ***6*** threads. Therefore the maximum number of threads is ***6***. We executed our program with ***2***, ***4*** and ***6*** threads. All custom and random graph outputs, time spent comparisons, thread load balancing (minimizing idle time) and performance analysis will be given in the ***Evaluation and Results*** section.

V. Evaluation and Results

We used the ***Intel VTune Profiler*** analysis tool for evaluating our working results. The VTune Profiler can be used to identify and analysis various aspects in both serial and parallel programs and can be used for both ***OpenMP*** and MPI applications. It can be used with a command line interface (CLI) or a graphical user interface (GUI).The graphical interface of the Intel VTune Profiler can usually be started by using the command ***vtune-gui***. We generally used ***Hotspots*** and ***Parallelism (Threading)*** analysis in order to evaluate our parallelised BFS algorithm. For Hotspots analysis, the elapsed time shows the total runtime of the application including idle times while CPU Time is the sum of the ***CPU times*** of all threads. The Top Hotspots section shows the most time-consuming functions sorted by CPU time. In addition to doing analysis in VTune Profiler, we measured elapsed times in sequential BFS algorithm and our developed parallel BFS algorithms. We compared all of these measurements and analysis in order to determine the best OpenMP schedule for parallelised algorithm (best threading score, effective CPU utilization) and minimum graph node number to provide efficiency of parallelism.

A. Custom Graphs - Graph1 Performance Analysis

We created a custom graph and add 8 nodes to it, then analysis it using VTune Profiler. Let we call this custom graph as Graph1. Graph1 is shown in Fig.7.

We obtained effective CPU Utilization histogram using 6 threads for Graph1 as shown in Fig.9. Our parallelism score is ***66.7%***.

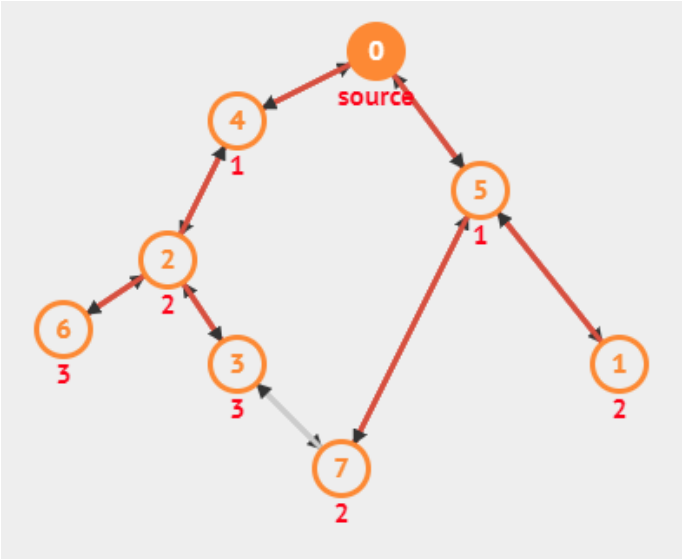

Fig. 7. Custom Graph Generation (8 nodes) - Graph1

```
BFS Queue Traversal:  0  5  4  7  1  0  2  3
BFS elapsed time = 4.482269e-05 seconds
```

Fig. 8. Graph1 - Parallel Breadth First Search Algorithm Output

B. Custom Graphs - Graph2 Performance Analysis

We created a custom graph and add 14 nodes to it, then analysis it using VTune Profiler. Let we call this custom graph as Graph2. Graph2 is shown in Fig.10.

We obtained effective CPU Utilization histogram using 6 threads for Graph2 as shown in Fig.12. Our parallelism score is ***77.8%***.

C. Custom Graphs - Graph3 Performance Analysis

We created a custom graph and add 16 nodes to it, then analysis it using VTune Profiler. Let we call this custom graph as Graph3. Graph3 is shown in Fig.13.

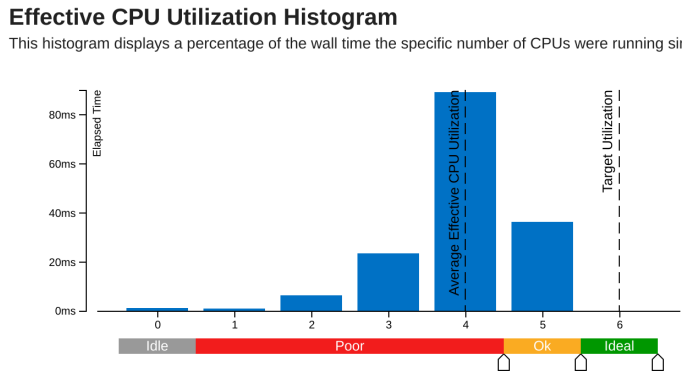

Fig. 9. Graph1 VTune Profiler Hotspots - Effective CPU Utilization Histogram

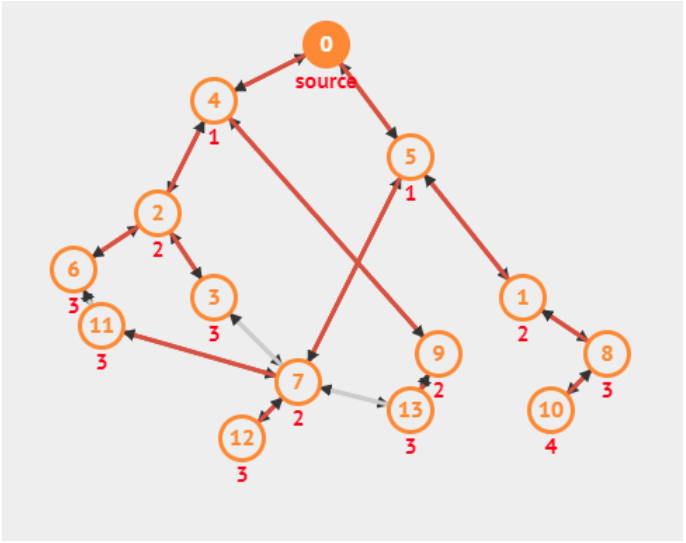

Fig. 10. Custom Graph Generation (14 nodes) - Graph2

```
BFS Queue Traversal:  0  5  4  7  1  0  9  2  13  12  11  3  8  6
BFS elapsed time = 2.717972e-05 seconds
```

Fig. 11. Custom Graph Generation (14 nodes) - Graph2

We obtained effective CPU Utilization histogram using 6 threads for Graph3 as shown in Fig.15. Our parallelism score is ***75.5%***.

We can see the execution moments of each thread instantly using Top-Down Tree Analysis from VTune Profiler. A Top-Down Tree Analysis for Graph3 is shown in Fig.16. We can interpret this analysis as, substantially threads are executing their tasks synchronous (75.5%). Green areas represent live threads, brown areas represent thread's executions.

D. Random Graph Performance Analysis

In our implementation, we added a function that creates nodes randomly with a given maximum node number. Then we analysed our results with ***100***, ***1000*** and ***2000*** nodes in sequential and our parallel BFS Algorithms.

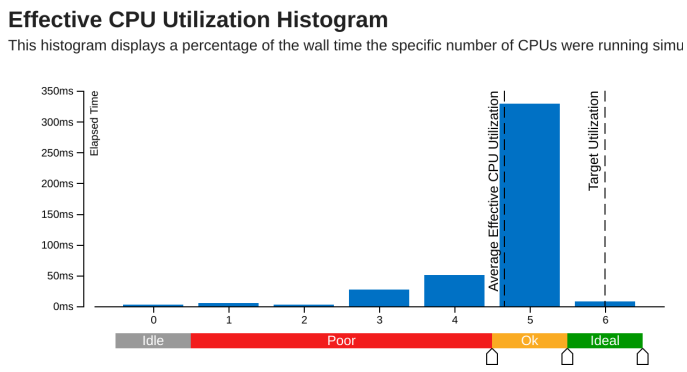

Fig. 12. Graph2 VTune Profiler Hotspots - Effective CPU Utilization Histogram

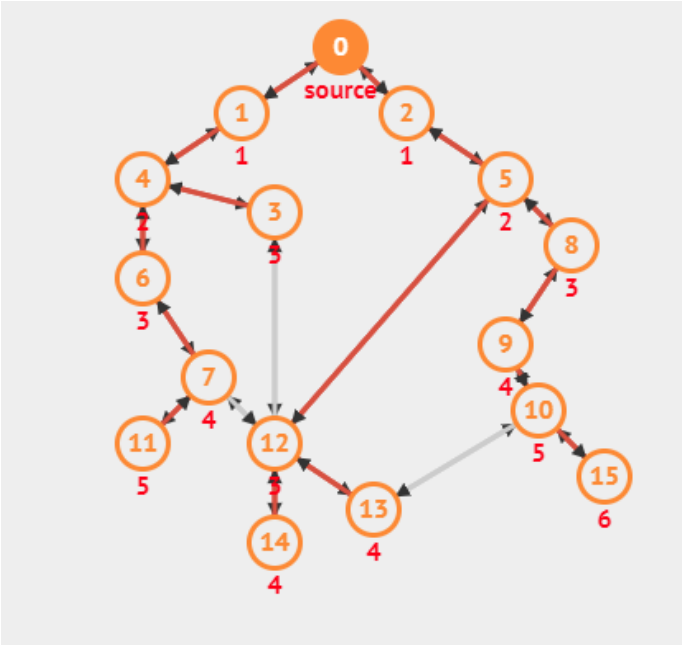

Fig. 13. Custom Graph Generation (16 nodes) - Graph3

```
BFS Queue Traversal:  0  1  2  0  4  5  6  3  12  8  7  14  13  9  11  10
BFS elapsed time = 3.910065e-05 seconds
```

Fig. 14. Graph3 - Parallel Breadth First Search Algorithm Output

We obtained effective CPU Utilization histogram using 6 threads for a random generated graph that has ***2000*** nodes. Histogram is shown in Fig.17. Our parallelism score is ***98.4%***.

Parallelism score ***increases*** with ***total node number*** in the graph linearly. It was a predictable result. Parallel threading becomes inefficient on relatively short tasks when compared to sequential processing due to parallel processing software costs. But when the task number increases, the importance of parallel processing emerges.

In relatively small graphs (roughly less than 200 nodes) the best elapsed time performance (least time spent) belongs

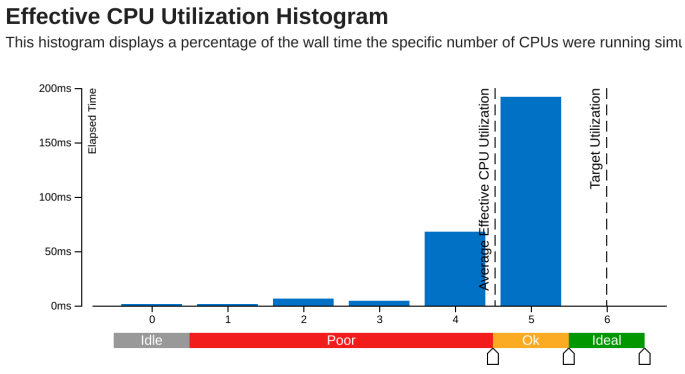

Fig. 15. Graph3 VTune Profiler Hotspots - Effective CPU Utilization Histogram

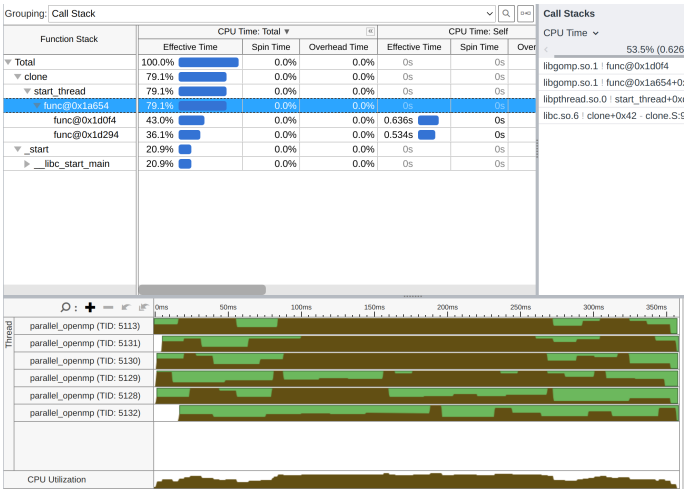

Fig. 16. Graph3 VTune Profiler Hotspots - Top-Down Tree Analysis

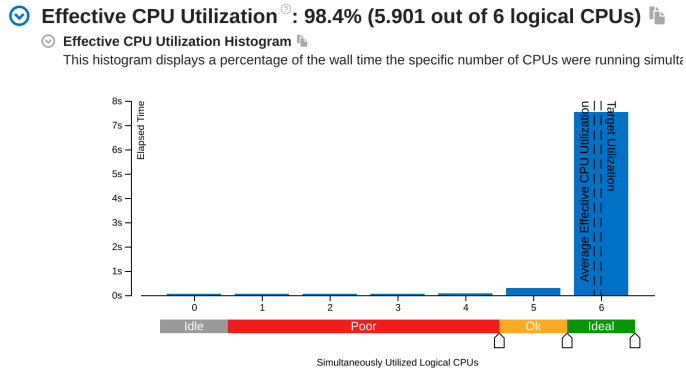

Fig. 17. Random Graph Generation (2000 nodes) - VTune Profiler Hotspots - Effective CPU Utilization Histogram

to sequential algorithm. But on large graphs, things turn around, parallelism begin to become more efficient and spent time dramatically being less when compared to sequential algorithm. All elapsed time measurements are given in table (Fig.18) for sequential and parallel BFS Algorithm.

All given analysis reports have been obtained from machine which has the following specs:

- Used Linux based operation system: ***Ubuntu v20.04.4 (LTS)***

Elapsed Times	*100 Nodes*	*1000 Nodes*	*2000 Nodes*
Sequential BFS	*5.71 ms*	*4.233 sec*	*27.643 sec*
Parallel BFS (OpenMP – 2 Threads)	*55.06 ms*	*3.964 sec*	*9.549 sec*
Parallel BFS (OpenMP – 4 Threads)	*29.80 ms*	*3.613 sec*	*4.561 sec*
Parallel BFS (OpenMP – 6 Threads)	*7.35 ms*	*1.219 sec*	*3.083 sec*

Fig. 18. Elapsed Times for sequential and parallel BFS Algorithm according to total node number

- Used Virtual Machine for Ubuntu: ***Oracle Virtual Machine (VirtualBox 6.1)***
- Used CPU: ***Intel(R) Core i7-9750H CPU @ 2.60GHz 2.60 GHz***
- Processor for Virtual Machine: ***Intel i7-9750H (6 Cores)***
- Base Memory: ***8192MB***

REFERENCES

[1] Graph Data Structure And Algorithms - GeeksforGeeks https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

[2] Difference between BFS and DFS - GeeksforGeeks https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

[3] "Breadth First Search and Depth First Search", Medium, 2022. [Online]. Available: https://medium.com/tebs-lab/breadth-first-search-and-depth-first-search-4310f3bf8416.

[4] "BFS Graph Algorithm(With code in C, C++, Java and Python)", Programiz.com, 2022. [Online]. Available: https://www.programiz.com/dsa/graph-bfs.

[5] "Depth First Search (DFS) Algorithm", Programiz.com, 2022. [Online]. Available: https://www.programiz.com/dsa/graph-dfs.

[6] "OpenMP: For Scheduling", Jakascorner.com, 2022. [Online]. Available: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[7] "Parallel breadth-first search - Wikipedia", En.wikipedia.org, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Parallel_breadth-first_search.

[8] S. Beamer, K. Asanović and D. Patterson, "Direction-Optimizing Breadth-First Search", Scientific Programming, vol. 21, no. 3-4, pp. 137-148, 2013. Available: 10.1155/2013/702694.

APPENDIX A
SEQUENTIAL BFS ALGORITHM SOURCE CODE

```
/*
 * Atalay PABUSCU - Mustafa PALA
 *
 * Multicore Programming Project
 *
 * Parallelization Of Graph Bredth First
    Search Algorithm
 *
 * OZYEGIN UNIVERSITY
 *
 * With OpenMP
 * Compile: gcc -g -Wall -fopenmp -o
    parallel_openmp_bfs_project
    parallel_openmp_bfs_project.c
 * Run: ./parallel_openmp_bfs_project
    <number_of_threads>
 * Without OpenMP
 * Compile: gcc -g -Wall -o
    parallel_openmp_bfs_project
    parallel_openmp_bfs_project.c
 * Run: ./parallel_openmp_bfs_project
 *
 *
 * Pseudocode of Sequential BFS Algorithm:
 *
 *  create a queue Q
 *  mark v as visited and put v into Q
 *  while Q is non-empty
 *    remove the head u of Q
 *    mark and enqueue all (unvisited)
    neighbours of u
 *
 *
 * queue = new Queue();
 * queue.enqueue(r); // initialize the
    queue to contain only the root vertex
    r
 * distance of r = 0;
 * while (!queue.isEmpty()) {
 * x = queue.dequeue(); { // remove vertex
    x from the queue
 *  for (each vertex y that is adjacent to
    x) {
 *   if (y has not been visited yet) {
 *     y's distance = x's distance + 1;
 *     y's back-pointer = x;
 *     queue.enqueue(y); // insert y into
    the queue
 *    }
 *  }
 * }
 *
 * Reference: Direction-Optimizing Breadth
    First Search, Scott Beamer, Krste
    Asanovic, David Patterson, Electrical
    Engineering and Computer Science
    Department University of California,
    Berkeley
 * https://www.programiz.com/dsa/graph-bfs
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define FALSE 0
#define TRUE 1
#define NONE -1

#define GRAPH1 10
#define GRAPH2 11
#define GRAPH3 12

#define CUSTOM_GRAPHS GRAPH1

#define GET_TIME(now) { \
  struct timeval t; \
  gettimeofday(&t, NULL); \
  now = t.tv_sec + t.tv_usec/1000000.0; \
}

#if(CUSTOM_GRAPHS == GRAPH1)
#define QUEUE_SIZE 8
#define NUMBER_OF_NODES 8
#elif(CUSTOM_GRAPHS == GRAPH2)
#define QUEUE_SIZE 14
#define NUMBER_OF_NODES 14
#elif(CUSTOM_GRAPHS == GRAPH3)
#define QUEUE_SIZE 16
#define NUMBER_OF_NODES 16
#else
#define QUEUE_SIZE 2000
#define NUMBER_OF_NODES 2000
#endif

int total_nodes = NUMBER_OF_NODES; //
    Numbor of nodes in Graph

typedef struct queue {
  int items[QUEUE_SIZE];
```

```
  int front;
  int rear;
}queue;

struct node {
  int vertex;
  struct node* next;
};

struct node* app_CreateNode(int);

struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};

queue* app_CreateQueue();
void app_Enqueue(queue* q, int);
int app_Dequeue(queue* q);
int app_IsQueueEmpty(queue* q);
int app_IsThereAnyNode(int vertex, queue*
    q);
void app_PrintGraphTraversal(queue*
    traversal);
void app_CreateCustomGraph(struct Graph*
    graph);
int app_GetUniqueRnd(int node, int cnt);
void app_CreateRndGraph(struct Graph*
    graph, int node_num);


/*
 * Traditional Sequential BFS Algorithm
 *
 * create a queue Q
 * mark v as visited and put v into Q
 * while Q is non-empty
 *   remove the head u of Q
 *   mark and enqueue all (unvisited)
     neighbours of u

*/
void app_BFS(struct Graph* graph, int
    startVertex)
{
  // Create a queue Q
  queue* search_queue = app_CreateQueue();
      // Create searching queue to store
      unvisited nodes
  queue* traversal_result =
      app_CreateQueue(); // Create
      traversal result queue to print graph
      traversal

  graph->visited[startVertex] = 0; //
      Determine the root (strat) vertex as 0
  // Mark v as visited and put v into Q
  app_Enqueue(search_queue, startVertex);

  // while Q is non-empty
  while (!app_IsQueueEmpty(search_queue))
  {
    // Remove the head u of Q
    int currentVertex =
        app_Dequeue(search_queue);
    app_Enqueue(traversal_result,
        currentVertex);

    struct node* adjNode =
        graph->adjLists[currentVertex];

    while (adjNode) // Iteration for all
        neighbour nodes
    {
      int adjVertex = adjNode->vertex;

      if (graph->visited[adjVertex] == 0)
          // Is this node visited
      {
        // Mark and enqueue all (unvisited)
            neighbours of u
        graph->visited[adjVertex] = 1;
        app_Enqueue(search_queue, adjVertex);
      }
      adjNode = adjNode->next;
    }
  }

  app_PrintGraphTraversal(traversal_result);
}

// Creating a node
struct node* app_CreateNode(int v) {
  struct node* newNode =
      malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

void app_PrintGraphTraversal(queue*
    traversal)
{
  printf("\n\nBFS Queue Traversal:\n");
  while (!app_IsQueueEmpty(traversal))
    printf("Visited Vertex: %d\n",
        app_Dequeue(traversal));
}

int app_IsThereAnyNode(int vertex, queue*
    q)
{
  for(int node = 0; node < QUEUE_SIZE;
      node++)
  {
    if(vertex == q->items[node])
    {
      return TRUE;
    }
  }
  return FALSE;
}

// Creating a Graph
struct Graph* app_CreateGraph(int
    vertices) {
  struct Graph* graph =
      malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices *
      sizeof(struct node*));
```

```
  graph->visited = malloc(vertices *
      sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}

// Add Edge to a Graph
void app_AddEdgeToGraph(struct Graph*
    graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode =
      app_CreateNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = app_CreateNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Create a queue
queue* app_CreateQueue() {
  queue* q = malloc(sizeof(queue));
  q->front = NONE;
  q->rear = NONE;
  for (int node = 0; node <
      NUMBER_OF_NODES; node++)
  {
    q->items[node] = NONE;
  }
  return q;
}

// Check if the queue is empty
int app_IsQueueEmpty(queue* q) {
  if (q->rear == NONE)
    return 1;
  else
    return 0;
}

// Adding elements into queue
void app_Enqueue(queue* q, int value) {
  if (q->rear == QUEUE_SIZE - 1)
  {
    //printf("\nQueue is Full!!");
  }
  else {
    if (q->front == NONE)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}

// Removing elements from queue
int app_Dequeue(queue* q) {
  int item;
  if (app_IsQueueEmpty(q)) {
    //printf("Queue is empty");
    item = NONE;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      //printf("Resetting queue ");
      q->front = q->rear = NONE;
    }
  }
  return item;
}

void app_CreateRndGraph(struct Graph*
    graph, int node_num)
{
  srand(time(NULL));
  // When you do srand(<number>) you
      select the book rand() will use from
      that point forward.
  // If you don't select a book, the
      rand() function takes numbers from
      book #1 (same as srand(1)).
  // It always returns same value

  for(int node = 0; node < node_num;
      node++)
  {
    int rnd_node = app_GetUniqueRnd(node,
        node_num);
    app_AddEdgeToGraph(graph, node,
        rnd_node);
    printf("\nNode.%d => Rnd_Dest_Node:
        %d", node, rnd_node);
  }
  printf("\n\n");
}

int app_GetUniqueRnd(int node, int cnt)
{
  int val = rand() % cnt;

  if(node != val)
  {
    return val;
  }
  else
    app_GetUniqueRnd(node, cnt);
}

void app_CreateCustomGraph(struct Graph*
    graph)
{
  #if(CUSTOM_GRAPHS == GRAPH1)
  app_AddEdgeToGraph(graph, 0, 4);
  app_AddEdgeToGraph(graph, 0, 5);

  app_AddEdgeToGraph(graph, 1, 5);

  app_AddEdgeToGraph(graph, 2, 3);
  app_AddEdgeToGraph(graph, 2, 4);
  app_AddEdgeToGraph(graph, 2, 6);

  app_AddEdgeToGraph(graph, 3, 2);
  app_AddEdgeToGraph(graph, 3, 7);

  app_AddEdgeToGraph(graph, 4, 0);
  app_AddEdgeToGraph(graph, 4, 2);
```

```
app_AddEdgeToGraph(graph, 5, 0);
app_AddEdgeToGraph(graph, 5, 1);
app_AddEdgeToGraph(graph, 5, 7);

app_AddEdgeToGraph(graph, 6, 2);

app_AddEdgeToGraph(graph, 7, 3);
app_AddEdgeToGraph(graph, 7, 5);
#elif(CUSTOM_GRAPHS == GRAPH2)
app_AddEdgeToGraph(graph, 0, 4);
app_AddEdgeToGraph(graph, 0, 5);

app_AddEdgeToGraph(graph, 1, 5);
app_AddEdgeToGraph(graph, 1, 8);

app_AddEdgeToGraph(graph, 2, 3);
app_AddEdgeToGraph(graph, 2, 4);
app_AddEdgeToGraph(graph, 2, 6);

app_AddEdgeToGraph(graph, 3, 2);
app_AddEdgeToGraph(graph, 3, 7);

app_AddEdgeToGraph(graph, 4, 0);
app_AddEdgeToGraph(graph, 4, 2);
app_AddEdgeToGraph(graph, 4, 9);

app_AddEdgeToGraph(graph, 5, 0);
app_AddEdgeToGraph(graph, 5, 1);
app_AddEdgeToGraph(graph, 5, 7);

app_AddEdgeToGraph(graph, 6, 2);
app_AddEdgeToGraph(graph, 6, 11);

app_AddEdgeToGraph(graph, 7, 3);
app_AddEdgeToGraph(graph, 7, 5);
app_AddEdgeToGraph(graph, 7, 11);
app_AddEdgeToGraph(graph, 7, 12);
app_AddEdgeToGraph(graph, 7, 13);

app_AddEdgeToGraph(graph, 8, 1);
app_AddEdgeToGraph(graph, 8, 10);

app_AddEdgeToGraph(graph, 9, 4);
app_AddEdgeToGraph(graph, 9, 13);

app_AddEdgeToGraph(graph, 10, 8);

app_AddEdgeToGraph(graph, 11, 6);
app_AddEdgeToGraph(graph, 11, 7);

app_AddEdgeToGraph(graph, 12, 7);

app_AddEdgeToGraph(graph, 13, 9);
app_AddEdgeToGraph(graph, 13, 7);
#elif(CUSTOM_GRAPHS == GRAPH3)
app_AddEdgeToGraph(graph, 0, 1);
app_AddEdgeToGraph(graph, 0, 2);

app_AddEdgeToGraph(graph, 1, 4);
app_AddEdgeToGraph(graph, 1, 0);

app_AddEdgeToGraph(graph, 2, 5);

app_AddEdgeToGraph(graph, 3, 12);
app_AddEdgeToGraph(graph, 3, 4);
app_AddEdgeToGraph(graph, 4, 6);
app_AddEdgeToGraph(graph, 4, 3);

app_AddEdgeToGraph(graph, 5, 8);
app_AddEdgeToGraph(graph, 5, 12);

app_AddEdgeToGraph(graph, 6, 7);
app_AddEdgeToGraph(graph, 6, 4);

app_AddEdgeToGraph(graph, 7, 6);
app_AddEdgeToGraph(graph, 7, 11);
app_AddEdgeToGraph(graph, 7, 12);

app_AddEdgeToGraph(graph, 8, 9);

app_AddEdgeToGraph(graph, 9, 10);

app_AddEdgeToGraph(graph, 10, 15);
app_AddEdgeToGraph(graph, 10, 13);

app_AddEdgeToGraph(graph, 11, 7);

app_AddEdgeToGraph(graph, 12, 7);
app_AddEdgeToGraph(graph, 12, 13);
app_AddEdgeToGraph(graph, 12, 14);

app_AddEdgeToGraph(graph, 13, 10);
app_AddEdgeToGraph(graph, 13, 12);

app_AddEdgeToGraph(graph, 14, 12);

app_AddEdgeToGraph(graph, 15, 10);
#else

#endif
}

int main(int argc, char* argv[]) {
  double t_start, t_finish;
  struct Graph* graph =
      app_CreateGraph(NUMBER_OF_NODES);

  #if(CUSTOM_GRAPHS == FALSE)
  app_CreateRndGraph(graph,
      NUMBER_OF_NODES);
  #else
  app_CreateCustomGraph(graph);
  #endif

  GET_TIME(t_start);
  app_BFS(graph, 0);
  GET_TIME(t_finish);

  printf("BFS elapsed time = %e
      seconds\n", t_finish - t_start);

  return 0;
}
```

APPENDIX B
PARALLEL BFS ALGORITHM SOURCE CODE

```
/*
 * Atalay PABUSCU - Mustafa PALA
 *
```

```
 * Multicore Programming Project
 *
 * Parallelization Of Graph Bredth First
    Search Algorithm
 *
 * OZYEGIN UNIVERSITY
 *
 * With OpenMP
 * Compile: gcc -g -Wall -fopenmp -o
    parallel_openmp_bfs_project
    parallel_openmp_bfs_project.c
 * Run: ./parallel_openmp_bfs_project
    <number_of_threads>
 * Without OpenMP
 * Compile: gcc -g -Wall -o
    parallel_openmp_bfs_project
    parallel_openmp_bfs_project.c
 * Run: ./parallel_openmp_bfs_project
 *
 *
 * Pseudocode of Sequential BFS Algorithm:
 *
 *  create a queue Q
 *  mark v as visited and put v into Q
 *  while Q is non-empty
 *    remove the head u of Q
 *    mark and enqueue all (unvisited)
    neighbours of u
 *
 *
 * queue = new Queue();
 * queue.enqueue(r); // initialize the
    queue to contain only the root vertex
    r
 * distance of r = 0;
 * while (!queue.isEmpty()) {
 * x = queue.dequeue(); { // remove vertex
    x from the queue
 *  for (each vertex y that is adjacent to
    x) {
 *    if (y has not been visited yet) {
 *      y's distance = x's distance + 1;
 *      y's back-pointer = x;
 *      queue.enqueue(y); // insert y into
    the queue
 *    }
 *  }
 * }
 *
 * Reference: Direction-Optimizing Breadth
    First Search, Scott Beamer, Krste
    Asanovic, David Patterson, Electrical
    Engineering and Computer Science
    Department University of California,
    Berkeley
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define FALSE 0
#define TRUE 1
#define NONE -1

#define GRAPH1 10
#define GRAPH2 11
#define GRAPH3 12

#define PARALLEL_OPENMP TRUE
#define CUSTOM_GRAPHS GRAPH3

#if (PARALLEL_OPENMP == TRUE) // Compiler
    switch for BFS Parallelization using
    OPENMP
#include <omp.h>
#endif

#define GET_TIME(now) { \
  struct timeval t; \
  gettimeofday(&t, NULL); \
  now = t.tv_sec + t.tv_usec/1000000.0; \
}

#if(CUSTOM_GRAPHS == GRAPH1)
#define QUEUE_SIZE 8
#define NUMBER_OF_NODES 8
#elif(CUSTOM_GRAPHS == GRAPH2)
#define QUEUE_SIZE 14
#define NUMBER_OF_NODES 14
#elif(CUSTOM_GRAPHS == GRAPH3)
#define QUEUE_SIZE 16
#define NUMBER_OF_NODES 16
#else
#define QUEUE_SIZE 2000
#define NUMBER_OF_NODES 2000
#endif

int thread_count = 1; // Default thread
    number (No parallelization-just main
    thread)
int total_nodes = NUMBER_OF_NODES; //
    Numbor of nodes in Graph

typedef struct queue {
  int items[QUEUE_SIZE];
  int front;
  int rear;
}queue;

struct node {
  int vertex;
  struct node* next;
};

struct node* app_CreateNode(int);

struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};

queue* app_CreateQueue();
void app_Enqueue(queue* q, int);
int app_Dequeue(queue* q);
int app_IsQueueEmpty(queue* q);
int app_IsThereAnyNode(int vertex, queue*
    q);
void app_TopDownStep(struct Graph* graph,
    queue* frontier, queue* next, queue*
    parents);
```

```
void app_PrintGraphTraversal(queue*
    traversal);
void app_CreateCustomGraph(struct Graph*
    graph);
int app_GetUniqueRnd(int node, int cnt);
void app_CreateRndGraph(struct Graph*
    graph, int node_num);


/*
 * function top-down-step(frontier, next,
     parents)
 * for v   frontier do
 *   for n   neighbors[v] do
 *    if parents[n] = -1 then
 *       parents[n]  v
 *       next   next   {n}
 *    end if
 *   end for
 * end for
*/
void app_TopDownStep(struct Graph* graph,
    queue* frontier, queue* next, queue*
    parents)
{
  # if (PARALLEL_OPENMP == TRUE)
  # pragma omp prallel
      num_threads(thread_count)
      default(none) shared(total_nodes,
      frontier, next, parents)
      private(adjNode, adjVertex, vertex,
      node, thread_num) schedule(dynamic, 1)
  # endif
  for(int vertex = 0; vertex <
      total_nodes; vertex++)
  {
    if (app_IsThereAnyNode(vertex,
        frontier) == TRUE) // for v
        frontier do
    {
      //printf("%d", vertex);
      struct node* adjNode =
          graph->adjLists[vertex]; // for n
          adjacentList[vertex]
      while(adjNode)
      {
        int adjVertex = adjNode->vertex;
  #     if (PARALLEL_OPENMP == TRUE)
        #pragma omp parallel for
  #     endif
        for(int node = 0; node <
            total_nodes; node++)
        {
          if (node == adjVertex) // for n
              adjacentList[vertex] do
          {
            if(parents->items[node] == NONE
                && node != 0) // if
                parents[n] = -1 then
            {
  #           if (PARALLEL_OPENMP == TRUE)
              int thread_num =
                  omp_get_thread_num();
  #           else
              int thread_num = 1;
  #           endif
              printf("\nAdjacent node of %d
                  vertex (OpenMP Thread No:
                  %d): %d\n",vertex,
                  adjVertex, thread_num);
              printf("Parent of %d adjacent
                  vertex (OpenMP Thread No:
                  %d): %d\n", adjVertex,
                  vertex, thread_num);
              parents->items[node] = vertex;
                  // parents[n]  v
              printf("Add node %d to next
                  queue (OpenMP Thread No:
                  %d)", node, thread_num);
              //next->items[node] = node;
              app_Enqueue(next, node); //
                  next  next U {n}
            }
          }
        }
        adjNode = adjNode->next; // Traverse
            all adjacent nodes of this vertex
      }
    }
  }
}

/*
 * BFS ALGORITHM
 * function breadth-first-search(graph,
     source)
 * frontier  {source}
 * next   {}
 * parents  [-1,-1,. . . -1]
 * while frontier 6= {} do
 *   top-down-step(frontier, next, parents)
 *   frontier  next
 *   next   {}
 * end while
 * return tree
*/
void app_BFS(struct Graph* graph, int
    startVertex) {
  queue* frontier = app_CreateQueue(); //
      frontier  {}
  queue* next = app_CreateQueue(); // next
      {}
  queue* parents = app_CreateQueue(); //
      parents  {}
  queue* traversal_result =
      app_CreateQueue();

  graph->visited[startVertex] = 1;
  app_Enqueue(frontier, startVertex); //
      frontier  {source}

  while (!app_IsQueueEmpty(frontier)) //
      while frontier != {} do: If frontier
      is not empty
  {
    app_TopDownStep(graph, frontier, next,
        parents);

    int currentVertex =
        app_Dequeue(frontier);
    app_Enqueue(traversal_result,
        currentVertex);
```

```
    while(!app_IsQueueEmpty(next))
    {
      // frontier  next
      int currentVertex = app_Dequeue(next);
      app_Enqueue(frontier, currentVertex);
    }

    next = app_CreateQueue(); // next  {}
  }

  app_PrintGraphTraversal(traversal_result);
}

// Creating a node
struct node* app_CreateNode(int v) {
  struct node* newNode =
      malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

void app_PrintGraphTraversal(queue*
    traversal)
{
  printf("\n\nBFS Queue Traversal:\n");
  while (!app_IsQueueEmpty(traversal))
    printf("Visited Vertex: %d\n",
        app_Dequeue(traversal));
}

int app_IsThereAnyNode(int vertex, queue*
    q)
{
  for(int node = 0; node < QUEUE_SIZE;
      node++)
  {
    if(vertex == q->items[node])
    {
      return TRUE;
    }
  }
  return FALSE;
}

// Creating a Graph
struct Graph* app_CreateGraph(int
    vertices) {
  struct Graph* graph =
      malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices *
      sizeof(struct node*));
  graph->visited = malloc(vertices *
      sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}

// Add Edge to a Graph
void app_AddEdgeToGraph(struct Graph*
    graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode =
      app_CreateNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = app_CreateNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Create a queue
queue* app_CreateQueue() {
  queue* q = malloc(sizeof(queue));
  q->front = NONE;
  q->rear = NONE;
  for (int node = 0; node <
      NUMBER_OF_NODES; node++)
  {
    q->items[node] = NONE;
  }
  return q;
}

// Check if the queue is empty
int app_IsQueueEmpty(queue* q) {
  if (q->rear == NONE)
    return 1;
  else
    return 0;
}

// Adding elements into queue
void app_Enqueue(queue* q, int value) {
  if (q->rear == QUEUE_SIZE - 1)
  {
    //printf("\nQueue is Full!!");
  }
  else {
    if (q->front == NONE)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}

// Removing elements from queue
int app_Dequeue(queue* q) {
  int item;
  if (app_IsQueueEmpty(q)) {
    //printf("Queue is empty");
    item = NONE;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      //printf("Resetting queue ");
      q->front = q->rear = NONE;
    }
  }
  return item;
}
```

```
void app_CreateRndGraph(struct Graph*
    graph, int node_num)
{
  srand(time(NULL));
  // When you do srand(<number>) you
      select the book rand() will use from
      that point forward.
  // If you don't select a book, the
      rand() function takes numbers from
      book #1 (same as srand(1)).
  // It always returns same value

  for(int node = 0; node < node_num;
      node++)
  {
    int rnd_node = app_GetUniqueRnd(node,
        node_num);
    app_AddEdgeToGraph(graph, node,
        rnd_node);
    printf("\nNode.%d => Rnd_Dest_Node:
        %d", node, rnd_node);
  }
  printf("\n\n");
}

int app_GetUniqueRnd(int node, int cnt)
{
  int val = rand() % cnt;

  if(node != val)
  {
    return val;
  }
  else
    app_GetUniqueRnd(node, cnt);
}

void app_CreateCustomGraph(struct Graph*
    graph)
{
  #if(CUSTOM_GRAPHS == GRAPH1)
  app_AddEdgeToGraph(graph, 0, 4);
  app_AddEdgeToGraph(graph, 0, 5);

  app_AddEdgeToGraph(graph, 1, 5);

  app_AddEdgeToGraph(graph, 2, 3);
  app_AddEdgeToGraph(graph, 2, 4);
  app_AddEdgeToGraph(graph, 2, 6);

  app_AddEdgeToGraph(graph, 3, 2);
  app_AddEdgeToGraph(graph, 3, 7);

  app_AddEdgeToGraph(graph, 4, 0);
  app_AddEdgeToGraph(graph, 4, 2);

  app_AddEdgeToGraph(graph, 5, 0);
  app_AddEdgeToGraph(graph, 5, 1);
  app_AddEdgeToGraph(graph, 5, 7);

  app_AddEdgeToGraph(graph, 6, 2);

  app_AddEdgeToGraph(graph, 7, 3);
  app_AddEdgeToGraph(graph, 7, 5);
  #elif(CUSTOM_GRAPHS == GRAPH2)
  app_AddEdgeToGraph(graph, 0, 4);
  app_AddEdgeToGraph(graph, 0, 5);
  app_AddEdgeToGraph(graph, 1, 5);
  app_AddEdgeToGraph(graph, 1, 8);

  app_AddEdgeToGraph(graph, 2, 3);
  app_AddEdgeToGraph(graph, 2, 4);
  app_AddEdgeToGraph(graph, 2, 6);

  app_AddEdgeToGraph(graph, 3, 2);
  app_AddEdgeToGraph(graph, 3, 7);

  app_AddEdgeToGraph(graph, 4, 0);
  app_AddEdgeToGraph(graph, 4, 2);
  app_AddEdgeToGraph(graph, 4, 9);

  app_AddEdgeToGraph(graph, 5, 0);
  app_AddEdgeToGraph(graph, 5, 1);
  app_AddEdgeToGraph(graph, 5, 7);

  app_AddEdgeToGraph(graph, 6, 2);
  app_AddEdgeToGraph(graph, 6, 11);

  app_AddEdgeToGraph(graph, 7, 3);
  app_AddEdgeToGraph(graph, 7, 5);
  app_AddEdgeToGraph(graph, 7, 11);
  app_AddEdgeToGraph(graph, 7, 12);
  app_AddEdgeToGraph(graph, 7, 13);

  app_AddEdgeToGraph(graph, 8, 1);
  app_AddEdgeToGraph(graph, 8, 10);

  app_AddEdgeToGraph(graph, 9, 4);
  app_AddEdgeToGraph(graph, 9, 13);

  app_AddEdgeToGraph(graph, 10, 8);

  app_AddEdgeToGraph(graph, 11, 6);
  app_AddEdgeToGraph(graph, 11, 7);

  app_AddEdgeToGraph(graph, 12, 7);

  app_AddEdgeToGraph(graph, 13, 9);
  app_AddEdgeToGraph(graph, 13, 7);
  #elif(CUSTOM_GRAPHS == GRAPH3)
  app_AddEdgeToGraph(graph, 0, 1);
  app_AddEdgeToGraph(graph, 0, 2);

  app_AddEdgeToGraph(graph, 1, 4);
  app_AddEdgeToGraph(graph, 1, 0);

  app_AddEdgeToGraph(graph, 2, 5);

  app_AddEdgeToGraph(graph, 3, 12);
  app_AddEdgeToGraph(graph, 3, 4);

  app_AddEdgeToGraph(graph, 4, 6);
  app_AddEdgeToGraph(graph, 4, 3);

  app_AddEdgeToGraph(graph, 5, 8);
  app_AddEdgeToGraph(graph, 5, 12);

  app_AddEdgeToGraph(graph, 6, 7);
  app_AddEdgeToGraph(graph, 6, 4);

  app_AddEdgeToGraph(graph, 7, 6);
  app_AddEdgeToGraph(graph, 7, 11);
  app_AddEdgeToGraph(graph, 7, 12);
```

```
  app_AddEdgeToGraph(graph, 8, 9);

  app_AddEdgeToGraph(graph, 9, 10);

  app_AddEdgeToGraph(graph, 10, 15);
  app_AddEdgeToGraph(graph, 10, 13);

  app_AddEdgeToGraph(graph, 11, 7);

  app_AddEdgeToGraph(graph, 12, 7);
  app_AddEdgeToGraph(graph, 12, 13);
  app_AddEdgeToGraph(graph, 12, 14);

  app_AddEdgeToGraph(graph, 13, 10);
  app_AddEdgeToGraph(graph, 13, 12);

  app_AddEdgeToGraph(graph, 14, 12);

  app_AddEdgeToGraph(graph, 15, 10);
  #else

  #endif
}

int main(int argc, char* argv[]) {
  double t_start, t_finish;
  struct Graph* graph =
      app_CreateGraph(NUMBER_OF_NODES);
  thread_count = strtol(argv[1], NULL, 10);

  #if(CUSTOM_GRAPHS == FALSE)
  app_CreateRndGraph(graph,
      NUMBER_OF_NODES);
  #else
  app_CreateCustomGraph(graph);
  #endif

  GET_TIME(t_start);
  app_BFS(graph, 0);
  GET_TIME(t_finish);

  printf("BFS elapsed time = %e
      seconds\n", t_finish - t_start);

  return 0;
}
```