Pathfinding project

of Advanced Computer Architecture

by Gianluca Andreotti and Aurora Lucrezia Castro

Introduction

Pathfinding algorithms are one of the main topics of graph theory. Their use today is very widespread, from robotics to videogames to navigation systems, networking, and many other fields.

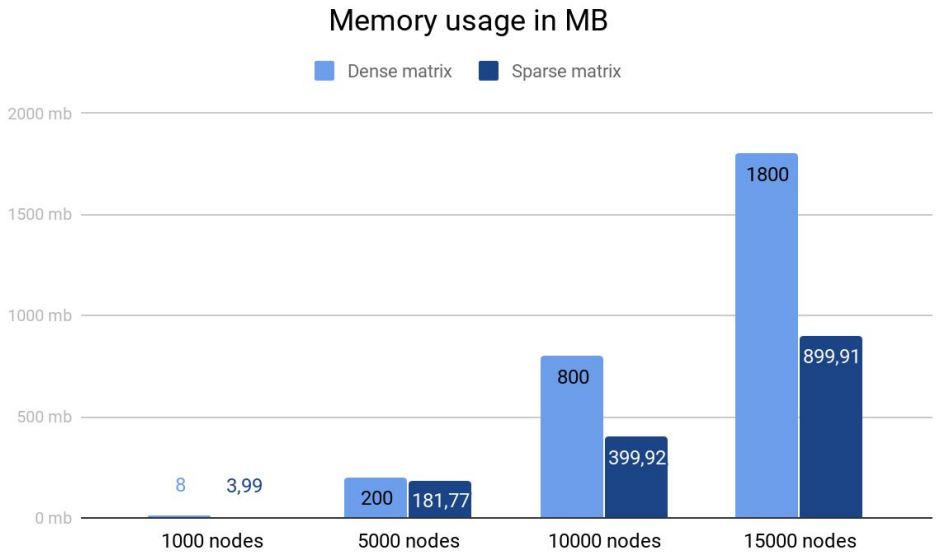
The goal of pathfinding is to find a path given a start and ending point. There are a lot of algorithms available that differ by speed, precision, and applicable. For example, sometimes we want to find the shortest path or find one in the fastest way possible.

In this project, our goal is to implement a concurrent version of some pathfinding algorithms using OpenMP.

Preparing input data

As concurrency introduces some overhead, to appreciate the speedup provided by multicore architectures, we need to test the code with some large input data.

We chose to use an adjacency matrix as graph representation. This comes from the fact that they are easy to work with and share between different programs, even if they might have a large footprint in terms of memory usage, as we can see in the graph below. Moreover, they are able to store edge weight and they are easy to convert in some other representation like sparse matrices.



To make large input data, we created a tool in Python that generates an adjacency matrix of arbitrary size, with random weights between nodes.

Thanks to this tool, we are able to generate graphs in the order of tens of thousand nodes and hundreds of millions of edges.

Considered algorithms

At first glance, we tried to focus on the *Dijkstra algorithm*, which solves the **shortest path problem** on a graph with non-negative weights for edges.

Even if we have achieved a working implementation, when we have parallelized it we have noticed that the execution time was longer than the one of the serial implementation. That is caused by the *well-known sequential bottleneck* [2] of the algorithm and a large number of shared data structures requiring a lot of synchronization between threads.

Unhappy with the final results, we shift to the implementation of another fundamental algorithm of pathfinding: the **depth-first search**.

The depth-first search allows us to find **a path** by keeping a single direction while traversing the graph. This means with DFS, every time a node has a neighbor, the algorithm “restart” from that node before continuing.

A premise: the node level locking

Generics OpenMP critical regions create a big overhead and don't discriminate between different nodes. To avoid this problem, we created a **node-level locking** using some low-level OpenMP functions like `omp_init_lock()`, `omp_set_lock()` and `omp_unset_lock()`. This mechanism allows us to exploit the possibility to **modify different elements of an array in parallel without data races**.

The downside of this is that it increases the memory footprint. If for example, we have a thousand nodes, we will need a thousand locks. A proposed solution to mitigate this disadvantage can be found in the book ***The Art of Concurrency*** where the author suggests what he calls a *modulo lock*, eg. a node for every even/odd node.

Node level locking is a fundamental aspect of our implementation of pathfinding algorithms and we use them extensively.

Implementation

In both implementations of Dijkstra and the DFS, we chose to **eliminate the early exit condition**, meaning that the algorithms don't stop when the destination is encountered, but they continue until there are no more nodes to visits that are reachable from the source. This allows us to measure the total time of the graph traverse and the theoretical worst case.

Dijkstra

This algorithm produces two arrays: the first keeps the **minimum cost between the source node and every other one**, the second tracks the **order of visits** to be able to recreate the shortest route to the source.

Below the sequential implementation of the Dijkstra algorithm.

```
std::pair<std::vector<Node>, std::vector<Node>> dijkstra(Node src)
{
   std::vector<Node> queue;
   queue.push_back(src);

   std::vector<Node> came_from(size(), -1);
   std::vector<Node> cost_so_far(size(), -1);

   came_from[src] = src;
```

```
    cost_so_far[src] = 0;

    while (!queue.empty()) {
        Node current = queue.back();
        queue.pop_back();

        for (int next = 0; next < n_nodes(); next++) {
            if (edge_exists(current, next)) {
                int new_cost = cost_so_far[current] + adj_matrix[current][next];

                if (cost_so_far[next] == -1 || new_cost < cost_so_far[next]) {
                    cost_so_far[next] = new_cost;
                    queue.push_back(next);
                    came_from[next] = current;
                }
            }
        }
    }

    return std::make_pair(came_from, cost_so_far);
}
```

For simplicity reason, this implementation of the algorithm does not use a priority queue, preferring instead a simple vector. A priority queue would have produced the same results with a faster execution time but it would add more complexity to the code.

In this example, we can already see the sequential nature of this algorithm. If we want to create a concurrent version, we need a lot of **critical areas**, one for each of:

- Read and update `queue`
- Read and update `came_from`
- Read and update `cost_so_far`

All of these critical areas will **heavily hinder** the performance of our code. The cost of them in fact, after the tests, is **higher** than the speedup we would get from parallelizing the code.

To try to improve the performance we would need to completely change the algorithm.

Depth-first search (DFS)

Depth-first search can be implemented in two variants:

- Iterative
- Recursive

In general, the iterative version of one algorithm is preferable, because it does not create overhead by allocating multiple stack frames and eliminates the risk of a stack overflow error if the number of recursive calls is excessive. Most compilers, in fact, try to convert recursive algorithms into iterative versions with the **tail call optimization**, but this technique is only applicable under some strict requirements. Moreover, it is typically easier to create a concurrent version of iterative algorithms in respect to recursive ones. [1]

In our project, we implemented and tested **both versions**.

The **iterative version** works with a stack-like queue. The last node is popped from the stack and every neighbor of him is then pushed on top of the stack. This process is repeated until the queue is empty.

The **recursive version**, as cited before, restart the process every time a neighbor is encountered, using it as the origin node.

Both implementations use a vector called `visited` that keeps track of nodes already visited in previous steps. This vector is shared between threads and in the recursive implementation concurrent accesses to modify it need to be synchronized accordingly.

Below the implementation of the iterative version.

```
void dfs(Node src, std::vector<int>& visited)
{
   std::vector<Node> queue { src };

   while (!queue.empty()) {
       Node node = queue.back();
       queue.pop_back();

       if (!visited[node]) {
           visited[node] = true;

           for (int next_node = 0; next_node < n_nodes(); next_node++)
               if (edge_exists(node, next_node) && !visited[next_node])
                   queue.push_back(next_node);
       }
   }
}
```

and the recursive version.

```
void rdfs(Node src, std::vector<int>& visited, int depth = 0)
{
   visited[src] = true;

   for (int node = 0; node < n_nodes(); node++) {
       if (edge_exists(src, node) && !visited[node]) {
           // Limit recursion depth to avoid stack overflow error
           if (depth <= max_depth_rdfs)
               rdfs(node, visited, depth + 1);
           else
               dfs(node, visited);
       }
   }
}
```

The parallel version of the **iterative** DFS works with a single main thread popping the last node from the stack-like queue and other threads in parallel checking what neighbors add to the queue, dividing them in chunks. Each one of these threads has a **private queue** where neighbors still not visited are added. When each chunk of neighbors is checked, the thread adds its private queue on top of the main one.

This **double-queue** mechanism reduces the waiting of each thread. In fact, **only** the merge with the main queue is a **critical region**. The downside of this approach is that it increases the memory footprint because multiple copies of the same “soon to be visited node” are added to the queue. This also creates a lot of reallocation to grow the capacity of the main queue.

```
void p_dfs(Node src, std::vector<int>& visited)
{
   std::vector<Node> queue { src };

   while (!queue.empty()) {
       Node node = queue.back();
       queue.pop_back();

       if (!visited[node]) {
           visited[node] = true;

           #pragma omp parallel shared(queue, visited)
           {
               // Every thread has a private_queue to avoid continuous lock
               // checking to update the main one
               std::vector<Node> private_queue;

               #pragma omp for nowait schedule(static)
               for (int next_node = 0; next_node < n_nodes(); next_node++)
                   if (edge_exists(node, next_node) && !visited[next_node])
                       private_queue.push_back(next_node);

               // Append at the end of master queue the private queue of the thread
               #pragma omp critical(queue_update)
               queue.insert(queue.end(), private_queue.begin(), private_queue.end());
           }
       }
   }
}
```

We tried various scheduling methods, but we chose to use **static scheduling** because it provides the best performances in our tests. This is caused by the fact that the processing time of neighbors is very similar between different threads.

We also added the `nowait` clause because we do not want an implicit barrier after the *for* loop and we want the threads to be able to continue into the post-processing critical area. [3]

The **recursive** version of the DFS works, again, with a **consumer-producer** mechanism with a single thread processing the origin node and multiple threads checking the neighbors in parallel, repeating the process for each node connected. Differently from the iterative version, we are using **OpenMP Tasks** for each node visit, in fact they are particularly suitable for recursive works. As cited before, we need to share the `visited` vector between threads and function calls.

As we are dealing with large data input, to avoid stack overflow error, we limit the number of tasks **both horizontally and vertically**. This means that multiple paths are followed

from the same origin node and the recursive approach is kept until a maximum defined depth, that we set to 10'000.

If these limits are reached, the algorithm falls back to an iterative approach.

Below is a significant extract of the parallel recursive DFS.

```
void p_rdfs(Node src, std::vector<int>& visited, std::vector<omp_lock_t>& node_locks, int depth = 0)
{
   atomic_set_visited(src, visited, &node_locks[src]);

   // Number of tasks in parallel executing at this level of depth
   int task_count = 0;

   for (int node = 0; node < n_nodes(); node++) {
       if (edge_exists(src, node) && !atomic_test_visited(node, visited, &node_locks[node])) {
           // Limit the number of parallel tasks both horizontally (for
           // checking neighbors) and vertically (between recursive
           // calls).
           //
           // Fallback to iterative version if one of these limits are
           // reached
           if (depth <= max_depth_rdfs && task_count <= task_threshold) {
               task_count++;

               #pragma omp task untied default(shared) firstprivate(node)
               {
                   p_rdfs(node, visited, node_locks, depth + 1);
                   task_count--;
               }

           } else {
               // Fallback to parallel iterative version
               p_dfs_with_locks(node, visited, node_locks);
           }
       }
   }

   #pragma omp taskwait
}
```

Data acquisition

To test the time required by algorithms we used the standard C++ chrono library. Measurements are expressed in **milliseconds** and they do not include initialization of the environment, like loading the graph in memory.

Each algorithm is tested **five times** to reduce the variance and, in the case of the parallel version with a different number of threads, respectively 2, 4, 8, 16, and 32.

Tests are done on virtual machine instances on Google Cloud Platform. We choose to use a high CPU performance virtual machine, with 16 GB of ram and 16 cores.

We used GCC as the compiler of choice, but we also tried LLVM+Clang which gave similar and coherent results.

Results obtained

Before comparing and analyzing the data we collected, we calculated the theoretical speedup using Amdahl's law. This allowed us to compare the theoretical speedup with the real one.

The Amdahl's law is specified as:

$$Amdahl's\ law:\ S(f,\ n) \leq \frac{1}{(1-f) + (\frac{f}{n})}$$

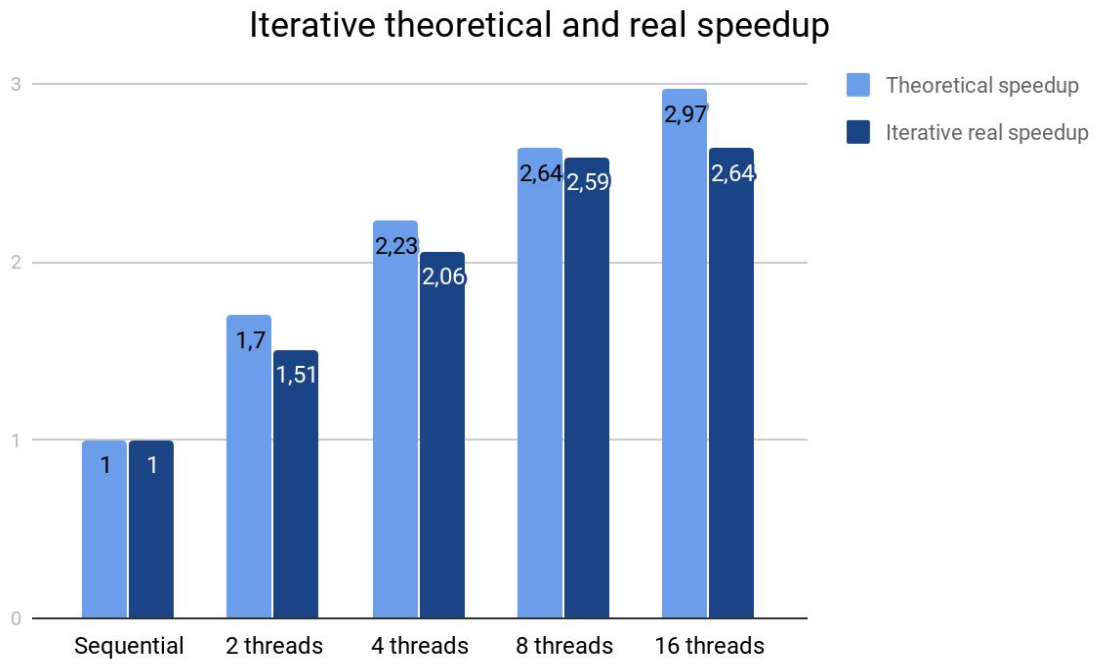
where S is the speedup in relation to f and n, where f ($0 < f < 1$) is the percentage of execution time that runs in parallel and n is the number of processors we used in the parallel application.

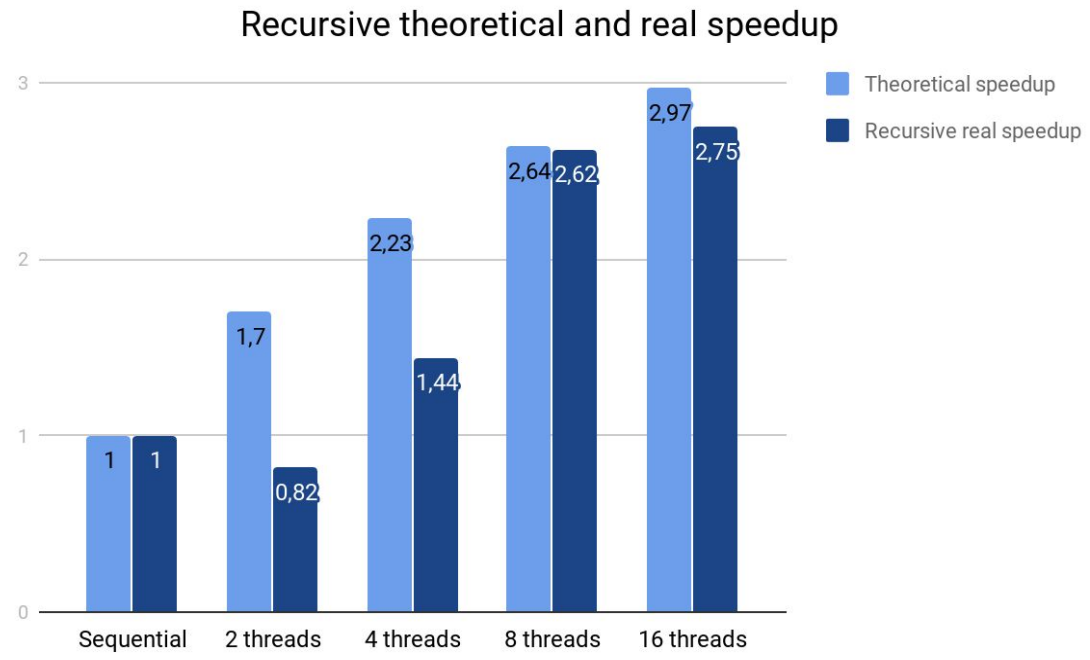
To calculate the real speed up, we used the formula:

$$real\ speed\ up:\ S = \frac{execution\ time\ without\ enhancement}{execution\ time\ with\ enhancement}$$

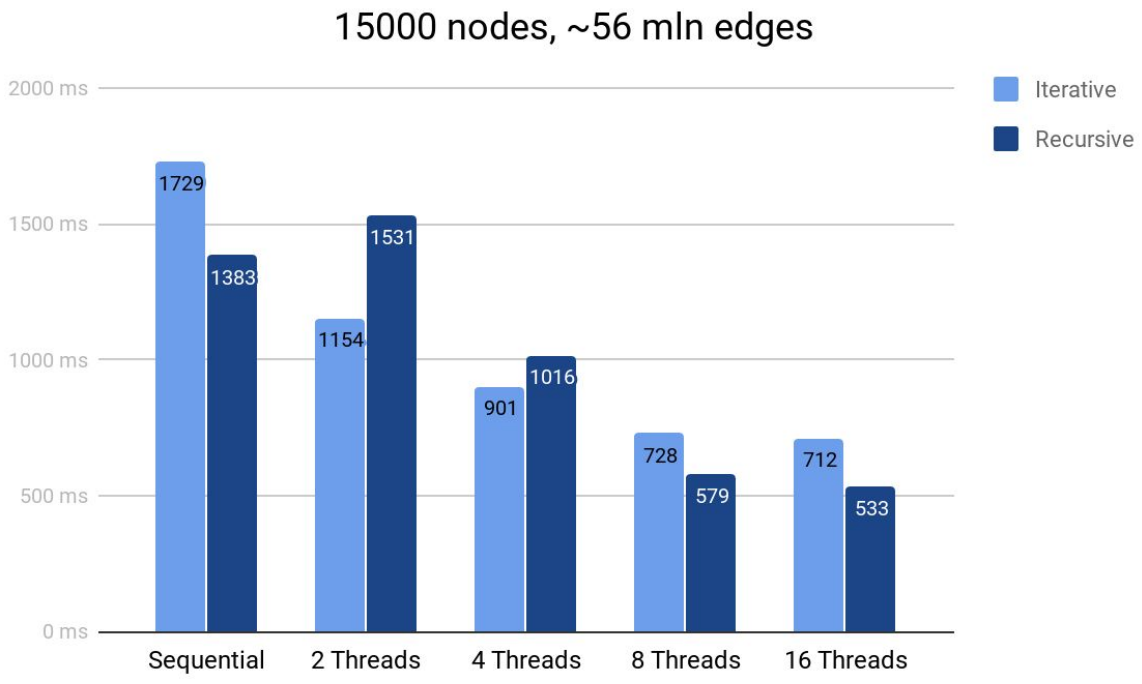
The speedup we obtain are in the following table

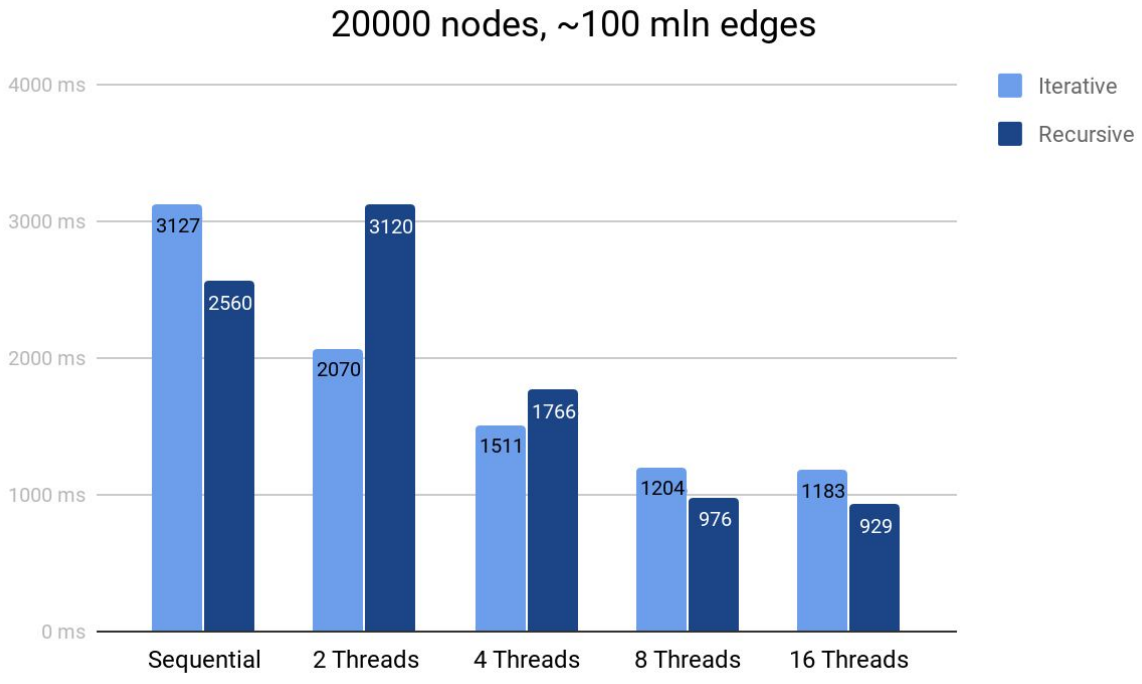
	Iterative real speedup	Recursive real speedup
Sequential	1	1
2 threads	1.51	0.82
4 threads	2.06	1.44
8 threads	2.59	2.62
16 threads	2.64	2.75





And here the times required





As expected, increasing the number of parallel threads improves the execution time and we reach a *plateau* between 8 and 16 threads in parallel. With more threads, the execution time increases.

It is important to underline the fact that with two recursive threads we get slower times than with the sequential implementation. That is because the **overhead caused by the**

synchronization cost is not present in the sequential implementation. This overhead is in fact **higher than the speedup** obtained by using multiple parallel threads. When we are using four or more threads, the speedup overcomes this overhead.

Conclusions

What we wanted to achieve in this project was to obtain a good speedup by parallelizing the depth-first search algorithm. We tried to implement both iterative and recursive DFS.

What we obtained were working algorithms that gave us significant data for our evaluation. From these data, we were able to confirm the goodness of our parallelization and the consequent speedup. Having a theoretical speedup, allowed us to understand if our real speedup was acceptable or not.

Differently, we do not consider ourselves satisfied with our Dijkstra algorithm implementation. We tried to obtain comparable results like with the DFS algorithm, but we found difficulties with the parallelization outcome. We have seen some research in this field that tries to overcome this problem by significantly changing the algorithm, but we preferred to focus on a better implementation of the DFS algorithm, as we have shown in this report.

References

[1] Clay Breshears - The Art of Concurrency - page. 211, chapter 10

[2] Vijay K. Garg - Removing Sequential Bottleneck of Dijkstra’s Algorithm for the Shortest Path Problem - The University of Texas at Austin - 2018

[3] Programming Parallel Computers - http://ppc.cs.aalto.fi/ch3/nowait/