Random Maze Generation: 15-418 Final Writeup

Bryce Summers, Brandon Lum
bwsummer, jiajunbl
December 27, 2015

Abstract:

We implemented **lock-free Union find structures** to parallelize an inherently serial random maze construction algorithm and therebye **solve the random maze construction problem faster.**

Using lock-free Union find structures, we achieved a consistent speedup using **8-threads** of around **2x** for the **critical loop** of the maze construction algorithm **relative to** the performance of the **serial algorithm** utilizing a standard serial union find structure. This is sufficient for helping out a professional using a **standard commodity computer.**

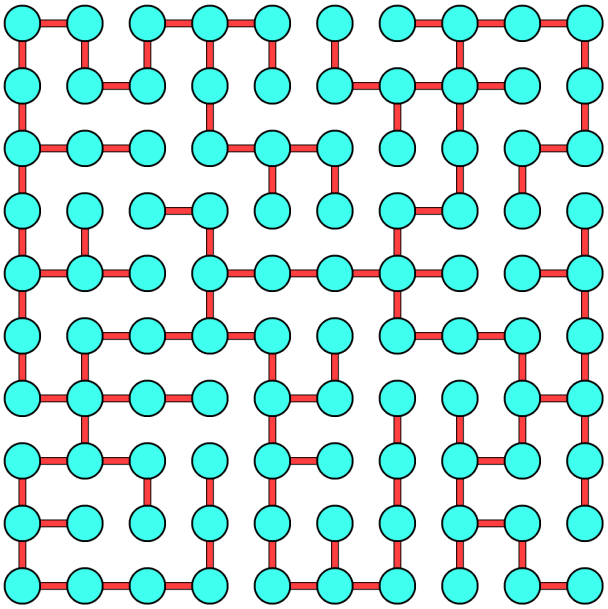

Figure 1: Example Generated Maze

Project Repository:
`https://github.com/Bryce-Summers/Randomized_Acyclic_Connected_Sub_Graphs`

Feel free to peruse the figures in the back of this paper.
Thank you for Reading this paper.

Background:

In this section we present our definitions and representations of the objects used in this project.

Graph

- We Assume the reader has some background knowledge of Graphs.
- Let N be the number of vertices in a graph.
- Vertices are represented by distinct integral indices from 0 to N.
- Edges are represented by pairs of Vertices.
- Graphs are represented by lists of edges.
- We will have the indices of vertices be invariant under the subgraph operation.
- Assume all graphs are undirected, unweighted, and have no self loops.

Maze Space

- A Maze space is a connected graph that specifies all possible edges that could be used in the construction of a maze.
- Please see Figure 2 and Figure 3 on page 14 for the maze spaces used in our testing.

Maze

- A **Maze** is an acyclic connected graph. $\leftrightarrow$
- A **Maze** is a spanning tree of a maze space. $\leftrightarrow$
- A **Maze** is a connected graph with N vertices and $N - 1$ edges.

- A Maze $\mathcal{M}$ is said to be **in** a Maze Space $\mathcal{S}$ if and only if $\mathcal{M}$ is a subgraph of $\mathcal{S}$.
- Please see Figure 4 and Figure 5 on page 15 for example mazes that we have generated.

Random Maze

- A **Random Maze** $\mathcal{M}$ is a Maze picked uniformly from the set of all Mazes in Maze Space $\mathcal{S}$.
- $\mathcal{M}$ is said to be **from** $\mathcal{S}$.

Random Maze Construction Problem

- Input: A Maze Space $\mathcal{S}$.
- Output: A Random Maze from $\mathcal{S}$

Testing System and Parallelization Target:

We envision our work to be used by creative individuals involved in generating mazes, professional network planners, or circuit designers. Our algorithm should mostly be used for proving example graphs for an average user, so we want to achieve a speedup on a commodity machine using cpp threads, which mimics the systems that our users will likely have.

We have tested our code on Andrew Unix using mainly 8 cpp threads. Our timing used the 418 cycle counter to measure the time spent in the critical for loop of the maze construction algorithm.

We have also tried our best to compare executions using the same randomized permutations to try to prevent randomness from overly skewing our readings, but the results do vary depending on the paricular problem size and shuffling permutation.

Why Randomness?:

We care about randomness, because the non random maze construction problem is trivial and would not be useful to a professional looking for varied examples.

Union Find Data Structure Standard ADT:

Union Find Structures, also known as Disjoint-set structures, are an abstract data type that specifies three main operations used in the representation of connected sets of elements.

For this project, we used the integral indices of vertices as our elements, so in the following description we will use **integers** in liu of abstract element types.

Traditional Union Find Structures provide the following three standard operations:

void makeSet(v) : Creates a new set in the structure containing only v. (1)

void Union($v1, v2$) : Combines the sets containing $v1$ and $v2$. (2)

int Find(v) : Returns the canonical representative of the set containing v. (3)

In Addition, users will normally want to check for the connectedness of two elements.

boolean connected($v1$, $v2$): returns true if and only if $v1$ and $v2$ are in the same set. (4)

We will Ignore Make Set.
For applications such the one discussed in this paper that operate on a contiguous sequence of integers, it suffices to provide a constructor that initializes the entire sequence of sets and not worry about implementing makeSet. It may be beneficial to look into parallelizing the instantiation of these contiguous regions of indices, but it is likely an insignificant portion of the entire computation.

Traditional Serial Union Find Structure Implementation:

Canonical Elements

Union find structures keep track of connected components by specifying canonical representatives for every one of the disjoint sets represented. Please see our powerpoint presentation for a visual of the main ideas behind a standard UF implementation. For a comprehensive treatment of the subject, pleas see [Tarjan].

the canonical elements are also called roots, because they are at the root of the convergind set trees.

Parent Pointer Array

Union Find structures create paths from nodes to their canonical elements by maintaining an array A of integers, where the parent of vertice i is at $A[i]$. A vertice v is defined to be the canonical element of its set iff $A[v] = v$.

Find

- Find finds the canonical node element for the set containing a given vertice v.
- ```
  while(v != A[v])
  {
    v = A[v];
  }
  return v;
  ```

Link

We want someway to merge set with different canonical representatives. The solution is the link function that takes two canonical representatives and sets one of them to point to the other.

```
void link(int root1, int root2)
{

  A[root1] = root2;
  // or: A[root2] = root1;

}
```

Since all elements in the set represented by root1 (S) had paths leading to root1 and root1 now points to root2, all elements in S will now correctly return their canonical representative as root2 and the find and connected predicates will behave as expected.
```

Path Compression.

After we have performed many links, it is possible that the paths from vertices to their canonical elements have grown to be long. After a canonical representative is found, we can combat this problem by transversing the path again from the original vertice and update all of the nodes along the path to point directly to the canonical element.

```
void path_compression(int v, int root)
{
  while(v != A[v])
  {
    int parent = A[v];
    A[v] = root;
    v = parent;
  }
}
```

Path compression should be called at the end of every find with the original vertice v and the foundcanonical representative.

Union by Rank.

A standard optimization is to keep track of the longest distance a given root node is away from a vertice that is a member of its set.

When linking, we should link nodes with shorter paths to nodes with longer paths, because then the overall longest path distance has not increased.

Union

Union ensures that after it returns the sets containing v1 and v2 now both have the same canonical element.

```
void union(int v1, int v2)
{
  link(find(v1), find(v2));
}
```

Connected

We can reduce connected to the find as follows:

```
bool connected(int v1, int v2)
{
  return find(v1) == find(v2);
}
```

Concurrent Union Find Specification and Implementation:

Success Return Values

To provide feedback for the possible execution behaviors of concurrent calls to a union find structure, we need to modify the function signatures as follows.

Traditional Union Find Structures provide the following three standard operations:

bool Union($v1, v2$) : Returns true if it succeeded, false if the elements were connected already. (5)

int Find(v) : returns the reprentative for some point in time between the call and return. boolConnect (6)

Relaxed Connected Semantics

In order to enable concurrent access to a union find structure, we needed to relax the semantics of connected to provide a broader view of time.

Return Values:

- Vertices Connected at Beginning of call $\implies$ true.
- Vertices newly connected midway through the call $\implies$ undefined.
- Vertices not connected at end of the call $\implies$ false.

Union by Total Order

Imagine one thread called link(v1, v2) and another called link(v2, v1). This can cause problems in lock based of lock-free implementations.

We want locks to be taken in a consistent order to avoid deadlocking in lock based implementations and to avoid the introduction of cyclic paths in lock free implementations.

Our solution is to always link nodes n to parents p such that $p <= n$, and parents are closer to the canonical element.

Compare and Swap

We are able to gurantee safe concurrent write operations using Compare and swap operations along with some memory fences.

The compare and swap operations allow us to store local variables containing the state of a union or find routine and then atomically only execute the operation if the shared state still matches the local state that the logic has validated.

Locking

We use cpp std:mutex objects for locking. We need to be careful to always take locks in decreasing order at all times to avoid deadlocking. It is also very important to not hold the root for to long, such as well searching for the second element in a union operation. If the root it held, then other threads waiting to get to the root may block the thread that is holding it from reaching the root agin, causing deadlock.

Test and Compare and Swap

There are ample oppurtunities for us improve performance by placing if checks prior to compare and swap operations. Checking the condition of compare and swap operations sends read requests over the bus that can be shared with other threads, whereas if it always went ahead and did the compare and swap operation, all threads would be sending exclusive read requests and invalidating each other, causing high contention and coherence overhead.

Serial Maze Construction Algorithm:

Let $\mathcal{M}$ be the Input Maze Space.
Let UF be a Union find Structure.

1. Initialize an empty list of edges called *Output*.
2. Let E be the list of edges representing $\mathcal{M}$.
3. Randomly Permute E.
4.
```
for each Edge e in E
  {
    if(!UF.connected(e.v1, e.v2))
    {
      UF.union(e.v1, e.v2)
      Output.add(e)
    }
  }
```
5. Return *Output*

Correctness Analysis

Please note that *Output* represents the output maze.
Output is guranteed to be connected because $\mathcal{M}$ is connected.
Output is guranteed to be acyclic because we never introduce an edge that connects two previously connected vertices.

Parallel Maze Construction Algorithm:

Let $\mathcal{M}$ be the Input Maze Space.
Let UF be a concurrency safe Union find Structure with the expanded function signatures.
Let T be the number of Threads we wish to use.

1. Let E be the list of edges representing $\mathcal{M}$.
2. Randomly Permute E. // Parallelization of Shuffling left to the reader.
3. Split E into T roughly equal parts.
4. Spawn T threads.
5. In each thread do the following. Please note that UF is shared with every thread.

```
P = thread's partition of E.
Output = new EdgeList.
for each Edge e in P
{
  // Returns true iff the union goes through and makes a connection.
  if(UF.union(e.v1, e.v2))
  {
    Output.add(e)
  }
}
```

6. Gather each thread's output array and combine then.
   // The reader is also welcome to parallelize this step.
7. Return the combined array.

Correctness Analysis

Please note that the same correctness analysis as used in the serial impementation still applies, with the added consideration that the parallel Union-Find Structures do indeed fulfill their semantic promises.

Specific Implementations and Result:

Serial

We wanted to have a baseline state of the art implementation of a Serial Union find structure. Our structure has very tight loops and should be adaquate to the task. It may be prudent to look at alternative Path Compression Schemes, such as the one mentioned in [Tarjan].

We defined the Serial implementation as having a 1x speedup.

The serial implementation is more competitive for smaller problem sizes. Our parallel implementations overtake it more as the problem sizes grow.

Global Locking

We implemented the global lock Union find structure by putting calls to the global lock inside of the publically exposed interface functions and pushing all of the actual worker functions into private functions inside of the class.

It was incredibly simple to implement the global locking struture and we are not surprised that it did not achieve a speedup. It actually achieves a slowdown, which we believe is due to the overhead of taking locks, and the overhead of spawning threads and the generation of cache coherence traffic.

Hand over Hand

We implemented hand over hand locking using the techniques described in the section on Parallel implementations. Hand over hand behaved abysmally. Please see Figure 10 on page 18 to see just how abysmally it performed.

We believe that the reason for its dismal performance is the convergence of the Union Find Structure. Over time, there are fewer and fewer root nodes left, until eventually only 1 root node remains. This means that every one of the threads are contending for the root nodes, even when no useful work will be done. Also, hand over hand locking requires the taking of a large number of locks, because the threads need to lock every node the touch and also take multiple passes to prevent possible deadlock. A major example of this is that hand over hand nodes cannot hold the root for an extended period of time.

Lock-Free, No Path Compression (NPC)

We implemented a Lock-Free version involving no path compression in an attempt to maximize the amount of time that threads were spending finding and linking. UF NPC was a success and achieved a speedup over serial. We were are excited about this, because we were able to gain a parallel speedup for a problem that has seemingly sequential dependencies.

Please see Figure: 6 on page 16 for an image representing an uncompressed path.

Lock-Free, Full Path Compression (FPC)
The goal of full path compression is to compress all paths down to the root nodes after every find operation. This should theoretically save time, because it decreases the distance that sucessive finds need to travel. Once we properly debugged this implementation, it is achieving speeds competitive with the No Path Compression implementation. We even recorded it achieving a 4x speedup on 8 threads one time, but it was probably just a very favorable randomized permutation and has not been consistently replicated.

Please see Figure: 6 on page 16 for a path before compression ad Figure: 7 on page 16 for the same path after compression starting at Vertice 6.

Lock-Free, Constrained Path Compression (CPC)
We wanted to find a middle ground between full path compression and no path compression, we also wanted to reduce cache coherence traffic, so we implemented a scheme where only 1 out of every k nodes is compressed during a find operation.

Please see Figure: 6 on page 16 for a path before compression, Figure: 8 on page 17 for the same path after constrained compression with $k = 2$, and Figure: 9 on page 17 for the original path after constrained path compression starting at Vertice 6 with $k = 3$.

Lock-Free Results
Since the edge orderings are randomized, results varied and all three of the lock free structures were better than the others for some executions. The Lock Free Union find structures consistenty achieved a speed up of around 2x over the serial implementation for the critical loop when run using 8 threads on a standard andrew linux machine and measured by the 418 Cycle Counter. Please see Figure 11 for a visual depiction of canonical performance.

The timings for the parallel code include the time taken to spawn threads and merge data.

No path compression (NPC) and Full path compression (FPC) vary widely depending on the depth of the paths generated during the execution, but Constrained Path Compression (CPC) usually provides performance that is more stable and is a good compromise between NPC and FPC.

There is no significan differense between running our algorithm on the Dense maze space and the lattice maze space. This is probably because of the extreme convergance of the algorithm, which introduces contention far greater then the density of the edges on the input graphs.

We have found that different values for the constrained path compression work well for different permutations, but have not seen any significant universal properties. It seems that the various parameters and lock free implementations each have certain situations where they are ideal.

Scaling
Just as our parallel implementations do better when the problem size is increased, they also obtain gradually higher speedups. Please see Figure 12 on page 20 for a graph that demonstrates scaling up to 20 threads. Their is a drop off around 2.5x at 20 cores, but we conjecture that this could be remedied if the problem size were increased to lattices larger than 4000 by 4000.

Performance Limitations

We think that the convergence or the Union find data structure in addition to the sequential dependancies in this problem were the reasons that it was hard to parallelize.

Final Notes on Optimization Efforts

There were mainly tricky cases to consider when implementing the Union find structures and it took a good bit of effort to figure out how to divide performance bugs from algorithm complexity bugs.

I some miraculous results along the way and was able to investigate them and found ways to elliminated overhead in the Serial implementation and gradually get the implementations as finely tuned as possible.

Every time we had good results, I investigated to see if I could find a nefarious reason for it.

I found that correctness checks were not a major bottleneck. Packing multiple arrays into arrays of structures also did not yield a major improvement, although it should theoretically improve the caching locality.

I implemented path compression in hand over hand and shaved off 25 percent of its time.

I was ready to give up on UF Lock Free Full Path compression, when my partner brilliantly noticed that I was checking for parents being less than the node instead of being greater than the node. The Full path compression was able to be competitive after that.

I implemented a graph visualizer and visualized some outputs. That is when I realized that the procedure was not behaving randomly. We then worked on comparing executions on with the same random seeds to get better performance comparisons. We also tried computing average performances for different problems.

We have copious performance data logs for different problems, with different sizes, and different values for the constrained path compression. Please feel free to look at the in our source code's datalogs folder.

`https://github.com/Bryce-Summers/Randomized_Acyclic_Connected_Sub_Graphs/tree/master/Mazes/datalogs`

References:

For a fantastic description and Analysis of Serial Union Find Data structures, please see **Data Structures and Network Algorithms** by Robert Tarjan. As of the writing of this paper, this is probably the author's favorite book.

Although lock free Union Find structures have been researched since at least the 1990's, we have not explicitly consulted with any published literature while making our implementation descisions.

Please look at our official open source project repository:
`https://github.com/Bryce-Summers/Randomized_Acyclic_Connected_Sub_Graphs`

Potential Future Work:

Here are some questions and tasks that future researches might want to investigate.

- Parallelize the shuffling algorithm for edges?
- Parallelize the splitting and gathering of edge lists?
- Investigate having the algorithm terminate one $N - 1$ edges have been found?
- Do some more detailed investigating of the cache coherence properties of the system?
- What are the ideal k values of CPC for ideal results?
- When do FPC, NPC, and CPC work best?

Distribution of work done by Partners: .

Both partners made worthwhile contributions to this project. This paper was written by Bryce.

Figures:

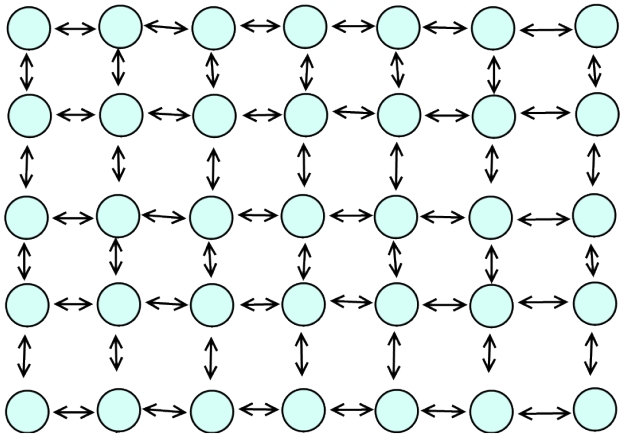

Figure 2: Lattice Maze Space of size 7 by 5. Edges $\sim 2 * size^2$, where the Lattice has $size^2$ vertices. All Lattice spaces used in this project are squares where size equals the length of both of their sides.

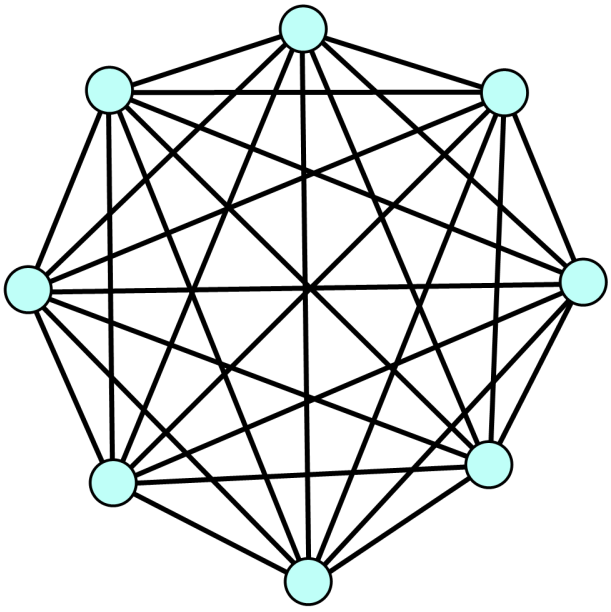

Figure 3: Dense Maze Space of size 8. Edges $\sim size^2$

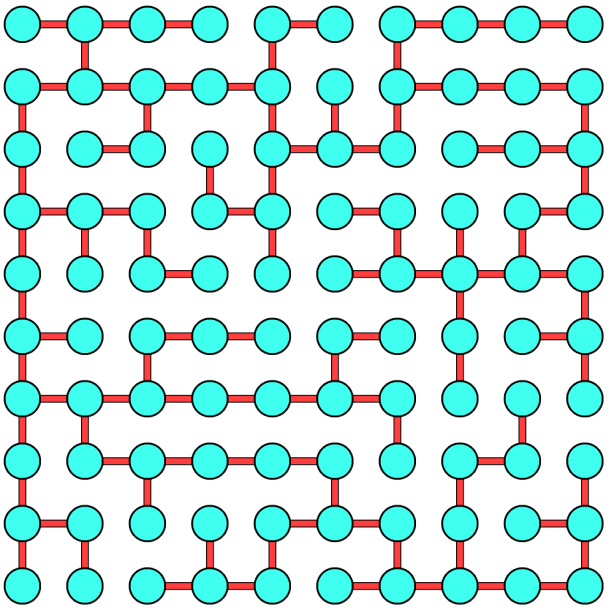

Figure 4: Example Lattice Maze result generated by our system.

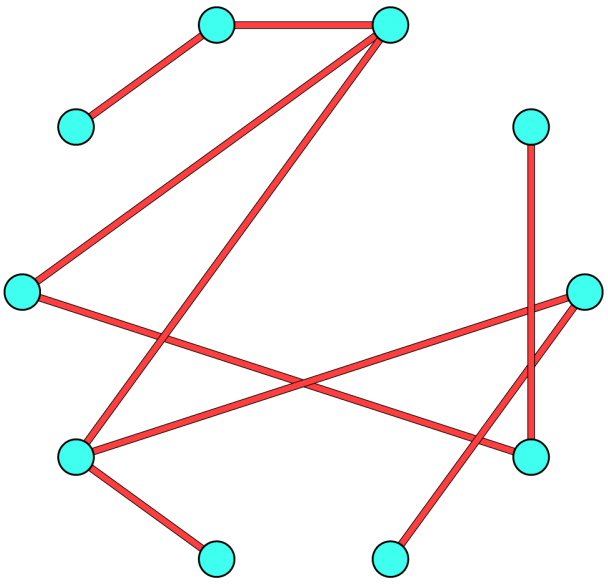

Figure 5: Example Dense Maze result generated by our system.

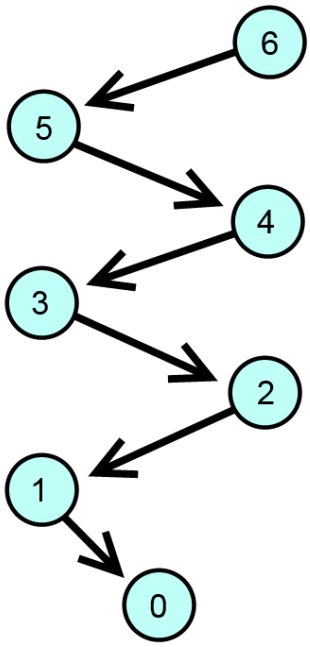

Figure 6: Uncompressed path of length 6.

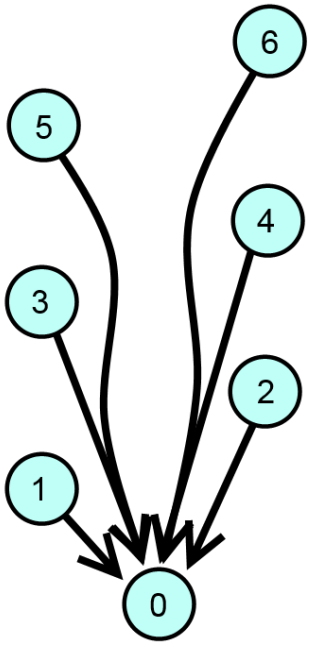

Figure 7: Structure after Figure 6 has been fully path compressed after a find at node 6.

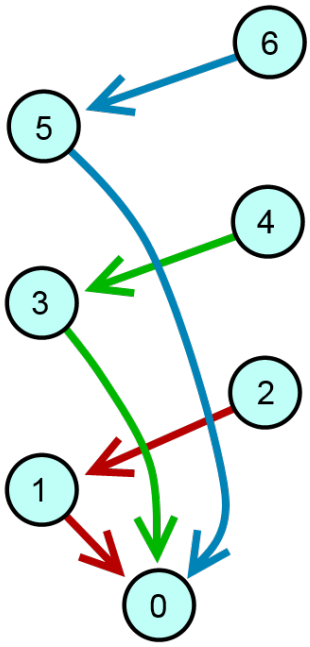

Figure 8: Structure after Figure 6 has been constrained path compressed with k = 2 after a find at node 6.

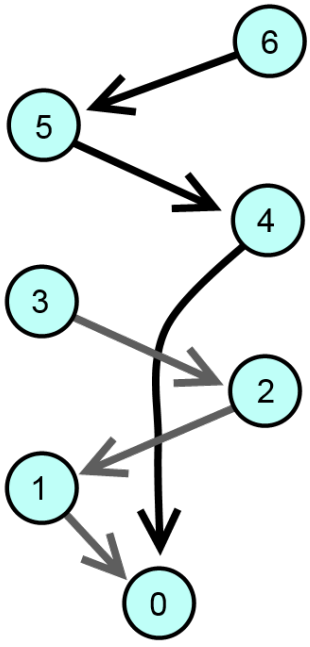

Figure 9: Structure after Figure 6 has been constrained path compressed with k = 3 after a find at node 6.

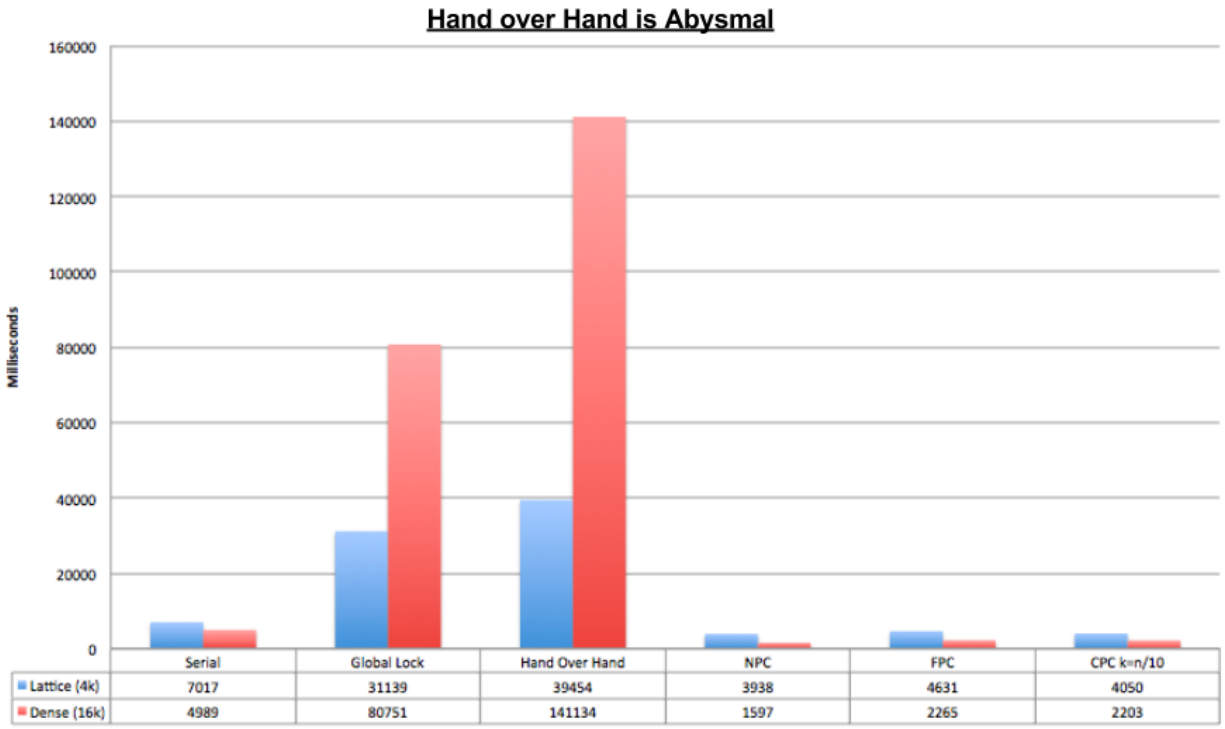

Figure 10: Implementations using locks behaved abysmally, most likely due to the the convergence of the operations over time. In other words, at the end of the algorithm every union find operation goes to the same root node and the locking implementations need to take and release locks prolifically so as to prevent dead lock.

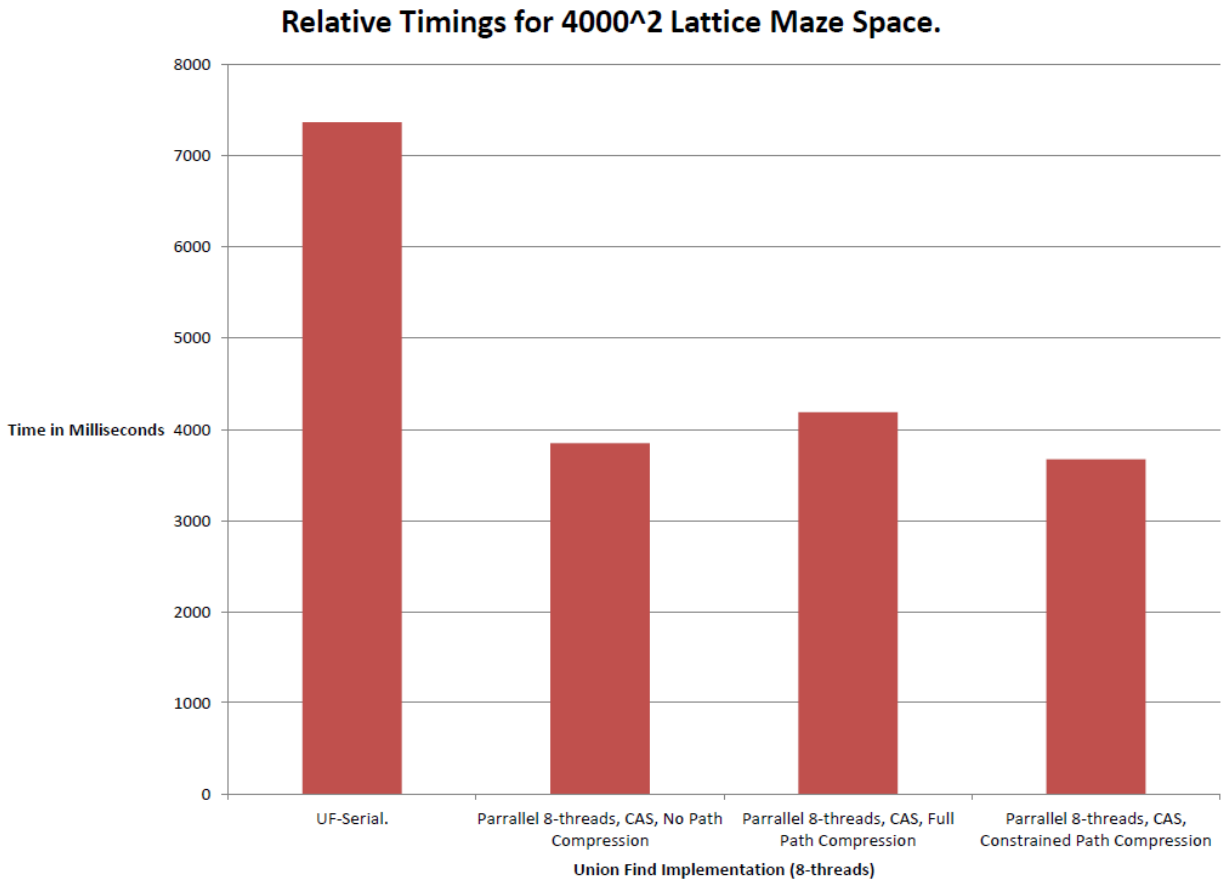

Figure 11: Lock Free Implementations consistently achieve a speedup of around 2x for the critical loop of the algorithm when run on 8 threads.

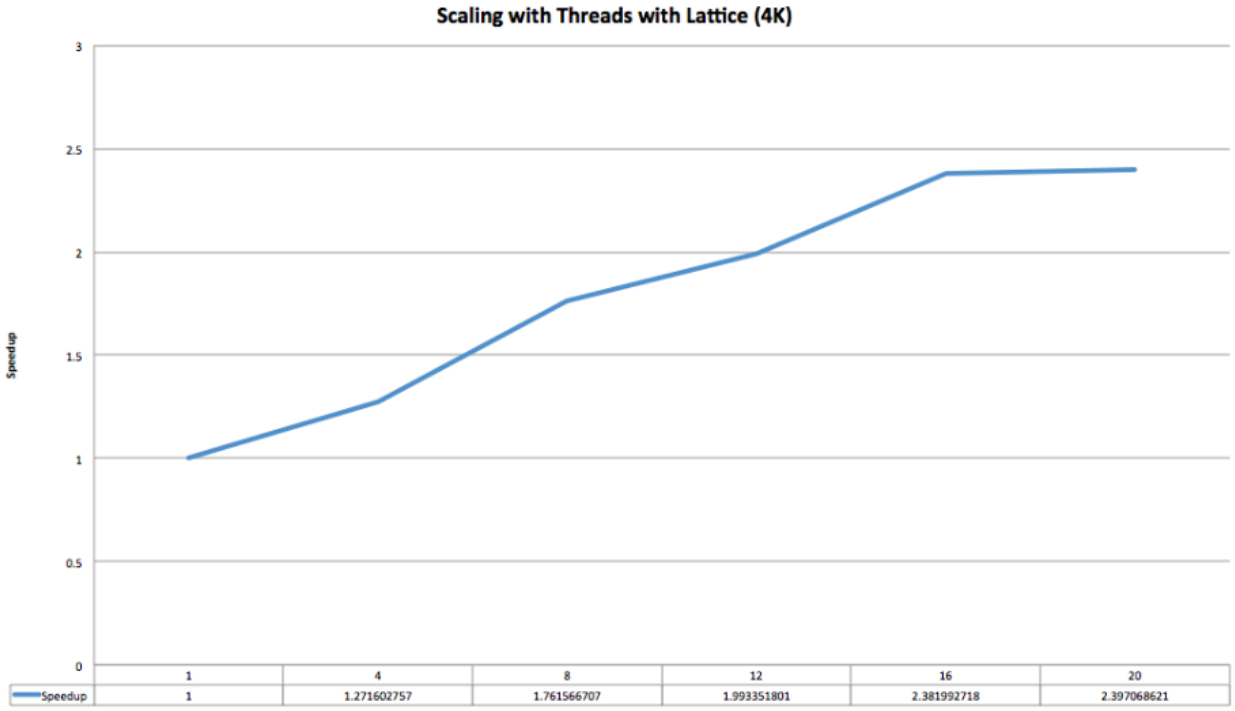

Figure 12: Lock-Free performance scales with higher thread counts for 4000 by 4000 lattices.