Parallel Bond Order Assignment

Kevin Shi (kshi) and Patrick Nguyen (phn)

1 Summary

We have implemented a parallel perfect matching algorithm, based on the well known Blossom Algorithm, for chemical bond assignment using OpenMp. We were able to run the algorithm on test cases provided by the client as well as random graphs, and we delivered a speedup plot comparing speedup to the growing number of threads.

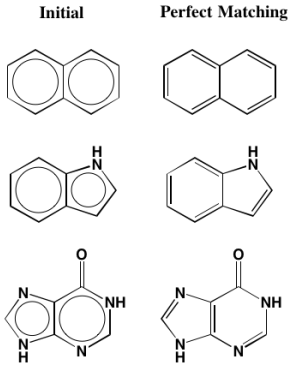
2 Background

2.1 The problem

In nature, atoms form bonds with other atoms to form molecules, but these bonds can be of different types, for example single or double bonds. We can easily perceive these bonds, but assigning types to each bond is harder. This requires assigning bonds such that:

- All carbon atoms have only one double bond
- Pyrrol NH group have no double bonds
- Pyridine N groups have only one double bond
- Hydrogens are ignored
- Edges out of a ring to an oxygen are assumed to be double bonds

One can represent these molecules as an undirected graph with atoms and bonds as vertexes and edges, and then the problem becomes exactly that of finding a perfect matching on the graph, where matched edges are double bonds and the rest are single bonds. A perfect matching on graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that all vertexes in V are incident on exactly one edge in M.



2.2 Preliminary Attempts

As we searched through the literature on parallel perfect matching, we came to a realization: most papers written on parallel perfect matching are not feasible in practice. The very first paper we attempted to understand and implement consisted in assigning random weights to each edge ranging from 0 to $2 * |E|$, but then we had to form an adjacency matrix T where element (i, j) had the value $2^w_{i,j}$ and then take that determinant. Even more, we then had to compute the cofactor matrix as well to compute the determinants $|D_{i,j}|$, where $D_{i,j}$ is T with row i and column j is removed, for all edges (i, j). Needless to say, this algorithm will not scale to a large number of nodes without much tweaking. Other attempts followed in a similar fashion, involving taking determinants of crazy values or requiring a number of processors scaling quadratically or worse with the input graph size.

2.3 Edmond's Blossom Algorithm

We finally settled on taking the canonical sequential algorithm, Edmond's Blossom Algorithm, and parallelizing it. This algorithm relies on the following theorem: a matching M on graph G is maximal if there are no M-augmenting paths P in G. To explain what that means: given a matching M and graph G,

- An **exposed node** v has no edges in M incident on it
- An **alternating path** P has edges alternately in M and not in M.
- An **augmenting path** P is an alternating path with both endpoints distinct and exposed, so the path starts and ends with edges not in M.
- **Augmenting a matching** M along augmenting path P involves taking P and switching the matching and non-matching status of each edge. In other words, it is the operation $M_1 = M \oplus P = (M \setminus P) \cup (P \setminus M)$

Then the strategy is clear: start with a an initial matching M_0 and continuously look for M-augmenting paths in G, augmenting M along the way, until there are no more. Then the final result M is a maximal matching on M.

In order to find augmenting paths, we build forests such that the path from a given root to a leaf is an alternating path starting with an unmatched edge and ending in a matched edge. Starting with all exposed nodes as singletons, we look at edges (v, u) coming out of nodes v in the forest that are even distance from the root. In other words, we only look at v which end the alternating path from the root on a matched edge.

Nodes u will either be in another forest, the same forest, or no forest at all. If u is not in any forest at all, it are matched with edge (u, x) and so we can extend our forest even further with edges $(v, u), (u, x)$, extending our alternating path even further. If u is in another forest and are also of even distance from its root, then we have an augmenting path $root(v), \cdots, v, u, \cdots, root(u)$ because of how we constructed our forest. If it is in the same forest, then we have found a special cycle called a **blossom**, which has the special property that if you contract the blossom to make new graph G' and matching M', then any M' augmenting path on G' can be lifted to a M augmenting path G, and we can simply do the contraction and recurse. Although the original sequential algorithm places much emphasis on blossoms, our parallel variation does not at all.

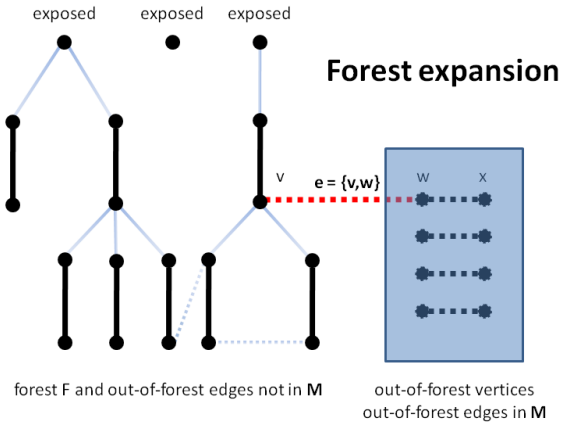


Figure 1: A figure

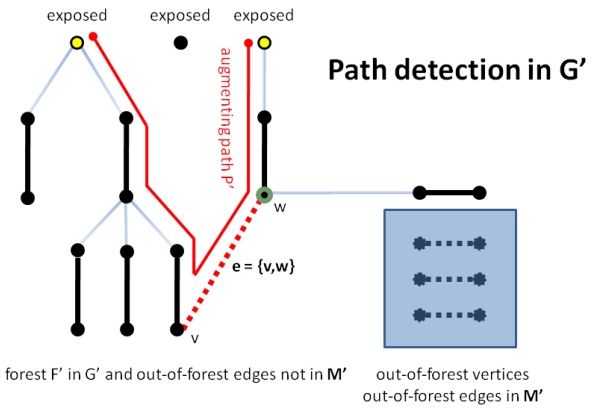


Figure 2: Another figure

3 Approach

Our code base is built off of starter code given to us a University of Pittsburgh professor, who wants this project completed for actual use in research.

For data structures, we need to be able to represent the graph, forests, and matchings, and given a graph we output the corresponding maximal matching on it. Nodes are simply labeled 0 to $n-1$, since although we start with atoms, we can simply order them and assign each of them an index very quickly. We had originally used adjacency lists to represent the forest and graph, but it became too easy to run into races, and furthermore even heavier locking was needed in order to modify the graph. To maximize access to nodes and edges without running to races, an adjacency matrix representation fro the graph and forests are ideal. For the the matchings, they can be viewed as a mapping M where $M(u) = v$ means u and v are matched. To represent this, an array is the best since that is an implicit map already. Then contracting a graph is easy, as each entry (i, j) is independent from each other, and same with the matching.

In looking at the algorithm, one thing to note is that each tree in the forest is independent from other trees. Building the forest by adding to trees happens by looking at edges going out of the forest. In addition, nodes already inside the forest will never change positions or leave, so different threads can add to different parts of the forest in parallel. Thus, our high level strategy is to build the forest in parallel, facilitated by fine grained locking to ensure consistency.

We require fine grained locking in order to add the vertexes in parallel to the forest. The easiest yet most effective way to do this is to assign a lock for each vertex. We cannot add a vertex to the forest or return it as part of an augmenting path without attaining the lock for it. We found that there was problem as with this when iterating through a path you can come from either end. This results in, for example a path A-B-C, T0 taking a lock on A and wanting B, with T1 taking a lock on B and wanting A creating a locking problem. We fixed this by assigning a priority to threads. Essentially the higher thread wins. In this case, T0 would see T1 has a lock on A and gives up and waits for the next iteration to try again. This happens often enough to cause performance issues, up to around 10 percent of the time a lock is taken (average around 2-5 percent) on 24 threads.

Because of the consideration of locks as well as independent threads using those locks, we decided to use OpenMp. This gives us enough control over how threads interact with each other

without having to explicitly schedule them, and in addition we are able to use built in locks in OpenMp.

Our strategy is then as follows: have each thread look at different forest nodes of even distance from the root. By the semantics of OpenMp's for loop, only one thread will be touching the same initial node v. Then we search that node's edges (v, u), seeing if we can attain the lock to u. Assuming we have that lock, that means we're either adding it to the forest (along with its matching node x) or creating an augmenting path with it. The former is making a change to the forest, and the latter is making a change to the matching. In either case, not being able to attain the lock immediately implies that our thread should immediately give up on trying to process u, and it will processing from the first node again since the state of the world is different.

We started with a dynamic scheduling policy as we found during class that it works way faster. Intuitively speaking the workload is very unbalanced with some iterations of the for loop doing 0 work. But we found that static scheduling actually works much faster. We choose to use a static scheduling policy. This is due to the inherent structure of our graph. Atoms in nature are only bonded to 4 other atoms at most. These atoms also tend to be close to the bonded atom. This means that if we use static blocks, each thread owns a "chunk" of the graph and works within it. This reduces contention since generally chunks are mostly independent.Dynamic would allow other threads to steal work from within a thread's chunk and cause contention.

Our initial try at paralleling the algorithm also involved finding and contracting blossoms. Blossoms in essence are cycles that are included in the augmenting path. Usually it's an optimization to the augmenting path finding process to contract these blossoms and modify the graph. But we found that removing this actually sped up our program for two reasons. First off, while usually blossoms are rare and large in arbitrary graphs, due to the nature of carbon rings, our graphs are literally constructed from cycles. Meaning that we have tons of small blossoms. Each time we contract we modify the graph which forces all the other threads to abandon their work. Blossoms end up being higher cost than benefit. They also require a much higher level of locking due to contention problems when modifying the graph, which resulted in us not having a full implementation of it.

As a final optimization, instead of starting off with an empty initial matching we can randomly fill in edges to the matching as a first approximation. This cuts off a tremendous amount of work from the threads, since they no longer have to continuously restart after connecting small trees together, which is very common at the start of the algorithm when all or most nodes are exposed. Note that since we are parallelizing over the exposed nodes, by constructing an initial greedy matching we're actually reducing the number of exposed nodes by a significant amount. Meaning that even though overall we get much better performance, our parallel speedup is much worse as a result. Since the graph was fairly sparse and atoms fairly close together (due to the nature of the problem) the greedy matching brings us much closer to the maximal matching than it would on arbitrary graphs.

4 Results

These tests are run on Intel(R) Xeon(R) E5645 CPUs @ 2.40 GHz. With 6 cores per socket, 2 sockets, and 2 threads per core.

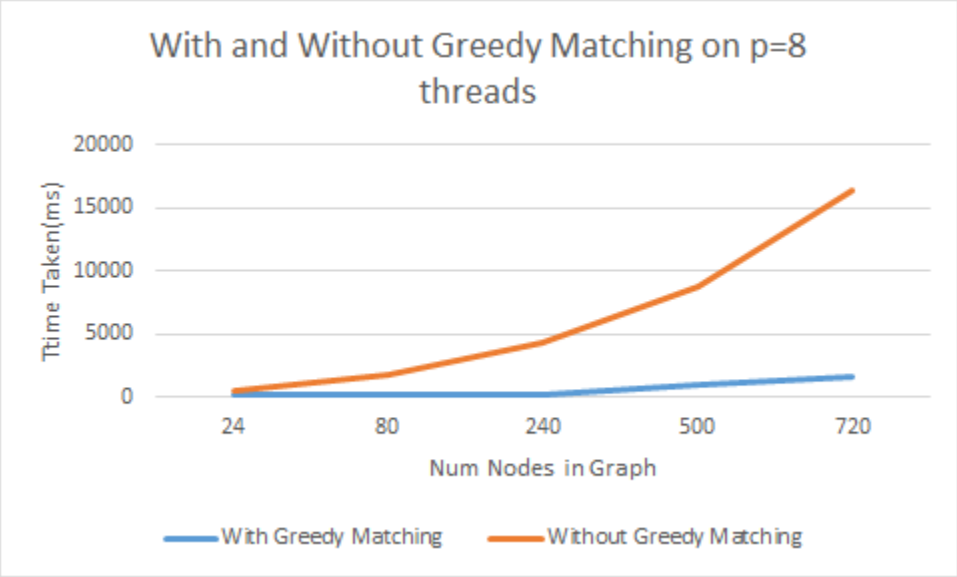
Also note that all tests are run on graphs of chemical molecules. Meaning that the graphs are very sparse with each node only having 2-4 edges. Specifically we use carbon balls (see bonus

section), one of the more common application spaces for the program.

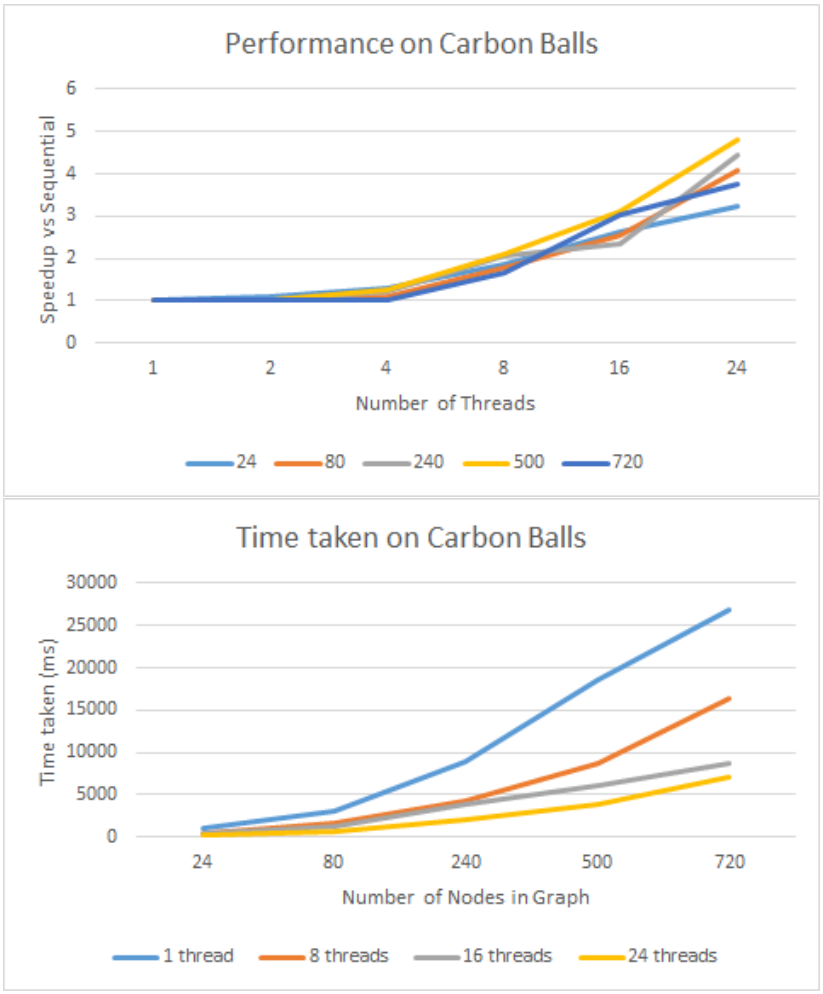
Our results are split by a couple of metrics. We measured performance on chemical structures of various shapes and sizes with various numbers of threads. We also measured performance with and without the final optimization (initial greedy matching) as while this reduces the overall time taken greatly, it also greatly reduces the parallelism allowed. We feel that it's still worthwhile to mention analysis on performance without greedy matching as it showcases some of the effect of some of the parallel strategies and matches the original algorithm better. It also represents performance better on arbitrary graphs as the greedy matching was much more advantageous due to the structure of our graph as atoms.

Also note that while we partition based on whether we do the final optimization or not, we don't actually isolate the time of the final optimization from the main algorithm. This is since we found that the final optimization takes about 0.2-6 ms on even the largest of graphs (hovering at about 0.1 percent of the total time).

The graphs for analysis will be only in terms of speedup vs sequential. The graph below is to give some context around the actual time it takes to find a perfect matching and just how much the greedy matching helps.

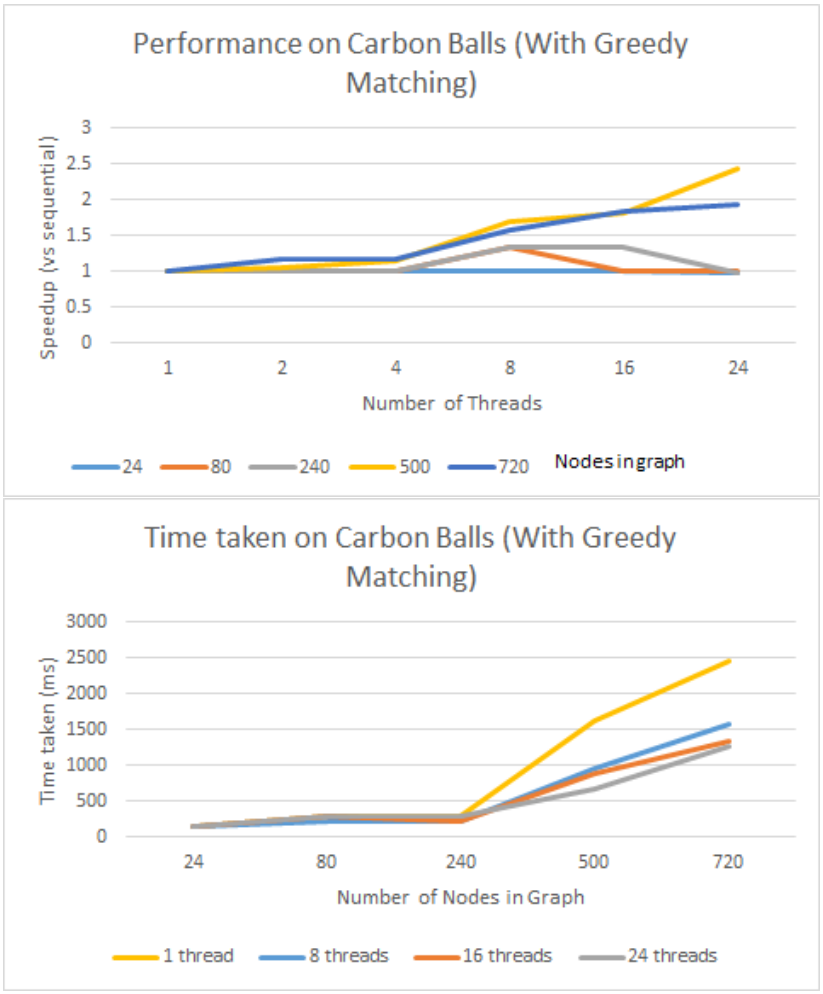


4.1 Without Greedy Matching



As you can see by the graphs, generally the speedup increases with number of threads up to about 4.5-5x speedup over the sequential version. From the second graph, with more threads, the algorithm also does significantly better as number of nodes increases. More analysis of how we got these speedups can be seen in the "Approach" section.

4.2 With Greedy Matching



As you can see by the graphs, although generally the speedup does increase with number of threads on larger graphs, with the greedy matching in place our speedup is a lot worse, hovering below 1.5x sequential for small graphs and capping at 2.5x speedup. From the second graph we can see the time taken closely follows the sequential version. The algorithm across the board does significantly better than without the greedy matching though. More analysis of how we got these speedups can be seen in the "Approach" section.

This performance might not seem amazing, but this is reasonably good for this problem. The problem itself doesn't lend itself well to parallelism. A recent paper's (Patwary 10) performance is here.

	Seq	$p = 2$		$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 64$	
	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql
er1	99.99	1.32	98.87	1.66	98.66	1.82	97.81	2.35	86.87	3.27	63.63	**5.23**	46.66
er2	99.99	1.48	99.16	2.26	99.18	2.60	99.17	2.90	96.63	3.75	71.99	**5.31**	53.75
er3	100.00	1.47	99.33	2.48	99.39	2.88	99.33	3.38	95.60	3.99	89.41	**5.05**	60.85

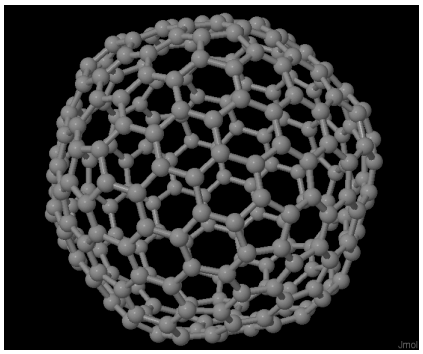


Figure 3: Carbon Ball

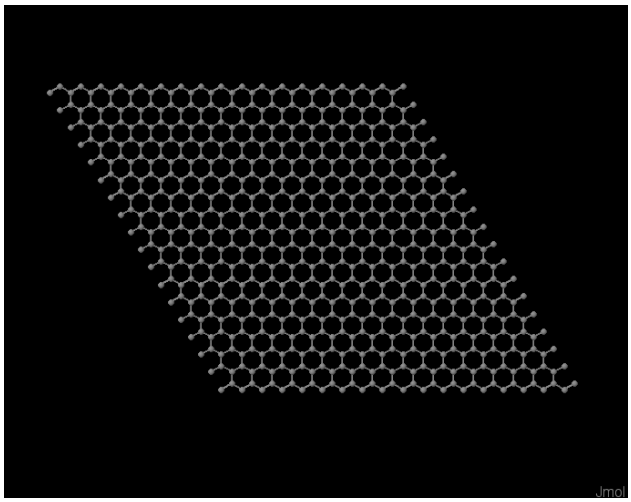


Figure 4: Carbon Sheet

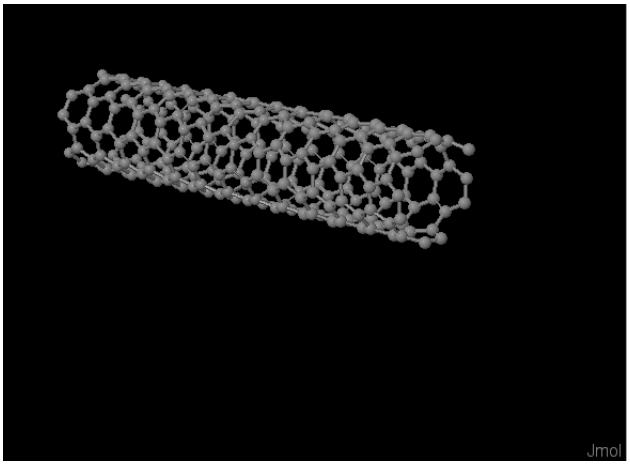
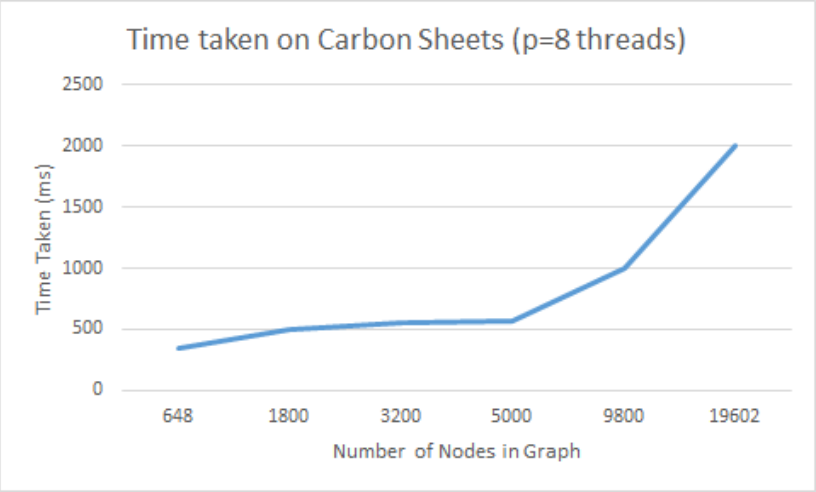
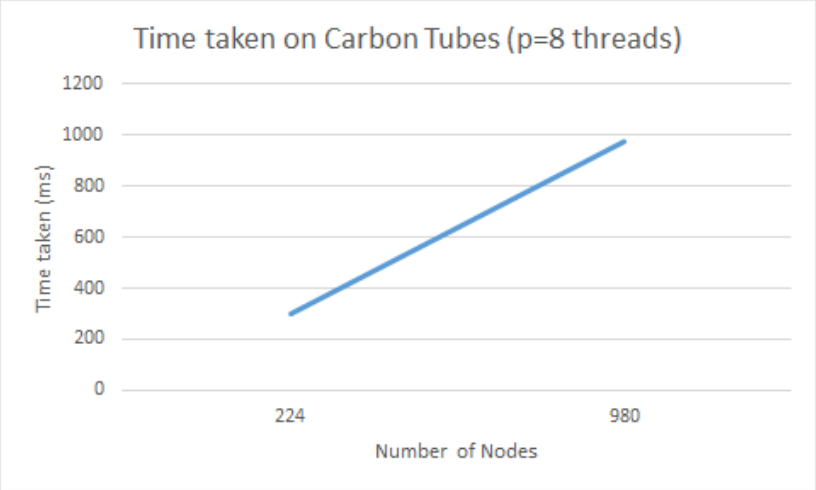


Figure 5: Carbon Tube

4.3 Bonus! Different Structures

All the graphs above are done on Carbon Balls [fig 3] but chemical molecules can take on a variety of other structures. Other common large molecules to analyze are sheets [fig 4] like Graphene sheets and tubes like carbon nanotubes [fig 5]





The performance gains are quite similar to the carbon balls but these are just here to show that our program run fine on different chemical macro-structures as well. A lot of our gains are at the atom level (atoms can only have 4 bonds etc.) rather than the macro level, so our performance is macro-structure independent. With carbon sheets we also show that our program runs relatively fast even on really large graphs.

Note that the performance within the carbon sheets is faster than the carbon balls. This is mainly due to the greedy matching. Since we construct a greedy matching, if we were to start tracing a path on a carbon ball [fig 3] then we would loop around back to the start pretty fast, where as with a carbon sheet, we would reach the end and just have the majority of the work done.

5 References

Starter Code and Problem Spec

Wikipedia Page with lots of images, pseudocode, and overall algorithm used.

6 Work by Student

Equal work was performed by both project members

We pair programmed on the vast majority of the code base and for the reports. Splitting only for minor/modular tasks.

7 Github

Link