PARELLIZATION OF DIJKSTRA'S ALGORITHM: COMPARISON OF VARIOUS PRIORITY QUEUES

WIKTOR JAKUBIUK, KESHAV PURANMALKA

1. Introduction

Dijkstra's algorithm solves the single-sourced shorest path problem on a weighted graph in $O(m + n \log n)$ time on a single processor using an efficient priority queue (such as Fibonacci heap). Here, m is the number of edges in a graph and n is the number of vertices, and there are $O(m)$ DECREASE-KEY operations and $O(n)$ INSERT and EXTRACT-MIN operations made on the priority queue.

The priority queues that we will test are Fibonacci heaps, Binomial Heaps, and Relaxed heaps. Fibonacci heaps are generally the queues used with Dijkstra's algorithm, but its performance doesn't increase as well as we might like with paralellization because the time bounds for Fibonacci heaps are amortized, and when we split up a task over many processors, one processor can finish much later than the others, leaving the others idle. Binomial heaps offer worse time bounds than Fibonacci heaps but offer guaranteed time bounds, so they parallelize well. Relaxed heaps, a modification of Binomial heaps, offer the same expected time bounds as Fibonacci heaps and also offer better worst-case time bounds, so they also paralellize well (in theory). We plan on putting this theory to test. To the best of our knowledge, this comparison has not been done in parallel settings.

Our goal in this paper is to explore how we can improve Dijkstra's runtime using modern hardware with more than one processor. In particular, we will explore how to parallelize Dijkstra's algorithm for p processors and discuss data structures we can use in the parallel version of Dijkstra's algorithm. Finally, we will compare these data structures in real-life performance tests on modern processors.

2. Description of Data Structures

Before we discuss how to parellalize Dijkstra's algorithm, we will first discuss three implementations of priority queues, a data structure that is required for use in Dijkstra's algorithm. The first such priority

Date: December 14, 2011.

queue we will discuss is a Fibonacci Heap. We will assume that the reader is familiar with with the details of how a Fibonacci Heap work, but we will introduce the data structure at a high level. The second data structure that we will introduce is a Binomial Heap, a data structure that has some similarities to a Fibonacci Heap. Finally, we will introduce Relaxed Heap, a data structure that is a modified version of the Binomial Heap.

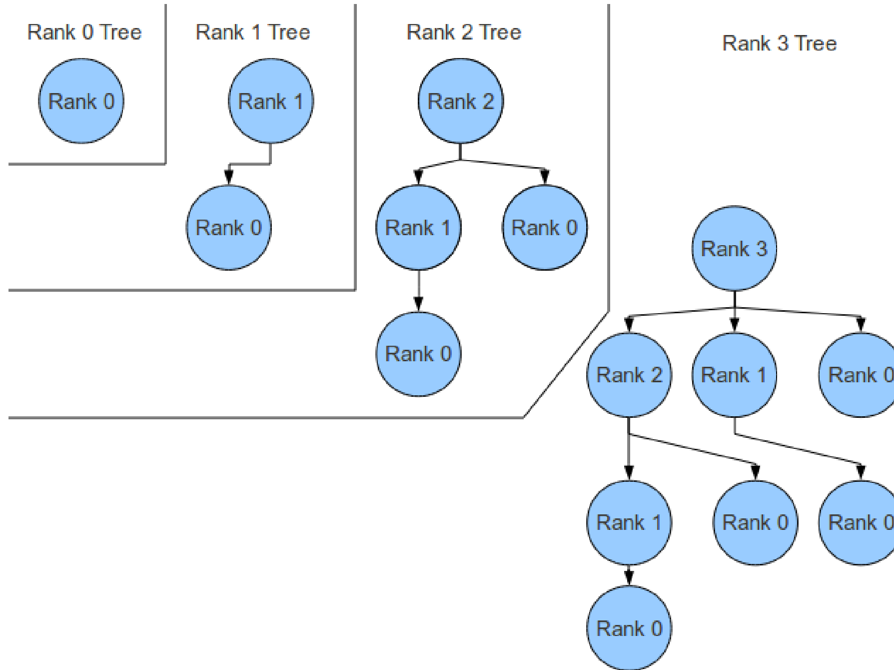
2.1. **Fibonacci Heap.** In this section, we will discuss the key properties of a Fibonacci Heap. A Fibonacci Heap is simply a set of Heap Ordered Trees, as described in [3], such that every node's key is smaller than or equal to it's children's key (heap-order property). Furthermore, Fibonacci Heaps are both lenient and lazy. They are lenient in the sense that they allow some properties (such as a degree r node must have $r-1$ children) to be broken. Furthermore, they are lazy because they avoid current work by delaying the necessary work to the future.

Using amortized analysis, we can show that Fibonacci heaps can perform the DECREASE-KEY and INSERT operations in $O(1)$ time, and can perform the EXTRACT-MIN operation in $O(\log n)$ time. It is important to note, however, that the analysis is amortized, and that it can be the case that certain operations take $\Omega(n)$ time in the worst case. In fact, it is even possible to construct a Heap Ordered Tree in a Fibonacci heap of height $\Omega(n)$.

2.2. **Binomial Heap.** In this section, we will describe how Binomial Heap works [1]. Before introducing Binomial Heaps, we will introduce one of its components, a Binomial Tree. A Binomial Tree is similar to the heap-ordered-trees in Fibonacci Heaps in the sense that every node's key is smaller than or equal to its children's key (heap-order property). We can then define a Binomial Tree recursively using ranks and the following rules:

(1) A node in a Binomial Tree cannot have a negative rank.
(2) A node of rank 0 in a Binomial Tree has no children.
(3) A node of rank $r+1$ in a Binomial Tree has exactly one child of rank r, one child of rank $r-1$, one child of rank $r-2$, one child of rank $r-3$ ... and one child of rank 0.
(4) A Binomial Tree of rank r has a node of rank r as its root.

These properties are perhaps best illustrated by a diagram:



We can see that the left-most child of the rank 1 node is an rank 0 Tree, the left most child of an rank 2 node is an rank 1 node, and so on. The second to left most child is simply one rank less, and the one after that is one rank lesser.

Note that because the binomial tree is defined recursively in this manner, it is easy to see that a binomial tree of rank r has exactly 2^r total nodes in the tree. Also note that in the rest of this paper, we will use T_r to denote a binomial tree of rank r and n_r to denote a node of rank r in a binomial tree.

Using binomial trees, we can build a data structure called binomial heap. This data structure ensures that all of our desired operations, INSERT, DECREASE-KEY, and EXTRACT-MIN have a worst-case running time of $O(\log n)$. A binomial heap is just a set of binomial trees with the property that this set can only have one binary tree of any rank. This is unlike the "lazy" Fibonacci Heaps, where this property is not guaranteed until after an EXTRACT-MIN step.

In fact, in a Binomial Heap, if we know how many elements are in this binomial heap, we can determine exactly which binomial trees are present in the binomial heap. This is because of the property that a binomial tree of rank i has exactly 2^i children, so the binary representation of the number of nodes in a binomial heap exactly corresponds

to which binomial trees are present in the heap. It is also easy to see that the largest possible rank of any tree in a binomial heap is $\lceil \log n \rceil$.

Now, we will describe the operations required by Dijkstra's algorithm. Before we do that however, we will first describe an operation called MERGE, which the other operations will require.

The MERGE operation will take two binomial heaps and merge them into a single binomial heap. We proceed in increasing rank of the trees. If two roots have to the same rank r, we will combine the trees by making the tree with the larger key a child of the tree with the smaller key to make a tree of rank $r + 1$. Note that we only do the combining step once per rank (because there are at most originally 2 trees of every rank, and we produce at most 1 additional tree of that rank), and because there can be at most $O(\log n)$ ranks, and because each combining step takes $O(1)$, the MERGE operations runs in $O(\log n)$ time.

The INSERT operation will create a new binomial heap with one node with the value we are inserting. Then, it will MERGE that binomial heap with the binomial heap already present. This operation will take $O(1)$ to create the new binomial heap plus the time for MERGE, so it will take a worst-case $O(\log n)$ time to complete.

The EXTRACT-MIN operation will first go through the list of root nodes to find the one with the minimum key. It will then remove that node from the root list and make all of its children roots in a new binomial heap. Then it will MERGE the two binomial heaps. This operation also takes $O(\log n)$ time because going through the roots and finding the min takes $O(\log n)$ time, removing it and making its children a new heap takes $O(\log n)$ time, and merging the two heaps takes $O(\log n)$ time.

The DECREASE-KEY operation will do the following: if the node that is being decreased is a root node, nothing will happen. If it is not the root, it will check if the node's parent is now greater than the node being changed. If it is, it will swap the two. Then it will check again to see if the node's parent is greater than the node and again swap the two if it is. It will repeat this process until either the node is a root node, or until its parent is smaller than it is. The DECREASE-KEY operation also takes $O(\log n)$ time because there are at most $O(\log n)$ levels in any Binomial Tree with at most n nodes.

Note that the running times for a Binomial Heap for all operations are strictly $O(\log n)$ time; that is, there is no single time that an operation could take more than $O(\log n)$ time in the worst case, which is different from Fibonacci Heaps because some operations in Fibonacci Heaps could take worst-case $\Omega(n)$ time.

2.3. **Relaxed Heap.** Relaxed heaps were first introduced by Tarjan et. al [2] in 1988 to allow efficient implementation of Dijkstra algorithm on p processors. Relaxed heaps are similar in design to binomial heaps, but, at the cost of "relaxation" of the heap-order property, they achieve a better worst-case running time for DECREASE-KEY, while maintaining the same expected running times as Fibonacci heaps for all operations. On the other hand, they are more structured in terms of internal structures than Fibonacci heaps.

There exist two main variation of Relaxed Heaps: rank-relaxed heaps and run-relaxed heaps. Rank-Delaxed Heaps provide EXTRACT-MIN, and INSERT in $O(\log n)$, while DECREASE-KEY runs in $O(1)$ amortized time and $O(\log n)$ worst-case time. Run-relaxed heaps provide $O(1)$ worst-case running time for DECREASE-KEY. In this paper and the successive practical experiments, we are going to use a Rank-Relaxed Heap, which we will simply refer to as Relaxed Heap.

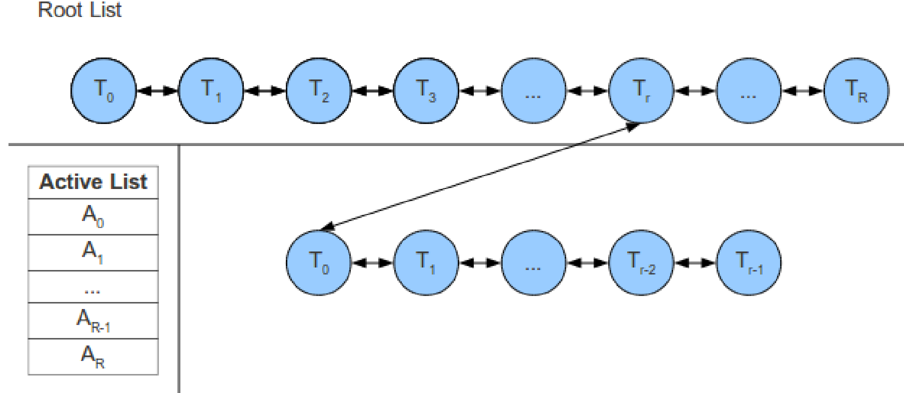
Similarly to Binomial Heap, Relaxed Heap keeps an ordered collection of R relaxed-binomial heaps of ranks $0, 1, ..., R-1$. Additionally each node q has an associated rank, rank(q), which is the same as the rank in a Binomial Heap. Some nodes are distinguished as active. Let c be a node and p be its parent in the collection of binomial heaps. c is active if and only if $key(p) > key(c)$, that is, when the heap-order property is broken.

Similarly to Binomial Trees in Binomial Heaps, each node of rank r in a Binomial Tree in a Relaxed Heap preserves the order of its $r-1$ children. To ensure efficient implementation, children of a node are represented in a child-sibling doubly-linked list, with the last sibling having the highest rank $(r-1)$ and the first child having rank 0. In the following analysis we refer the the right-most child with the highest rank as a last child.

There are two crucial invariants preserved by relaxed-heaps:

(1) For any rank r, there is at most one active node of rank r.
(2) Any active node is a last child.

Since there are at most $\log n$ different ranks, there are at most $\log n$ active nodes. For each rank r, relaxed-heap keeps a pointer to the active node of rank r, visually:



The INSERT operation on relaxed-heap works analogically to the INSERT on Binomial-Heap, by creating a binomial tree T_0 of 1 element, inserting the tree to the root list (and possibly consecutively merging, similarly to Binomial Heaps). Insert runs in $O(\log n)$ worst case.

In order for DECREASE-KEY(q, v) to achieve $O(1)$ expect running time, relaxed-heap may violate the heap-order property by marking q as an active node. Let $p = parent(q)$. If the newly set key $v > key(p)$, then clearly heap-order property is not violated and there is nothing to do. If, $v < key(p)$, then x needs to be marked as active, which might violate invariant (1) or invariant (2) (or both). There are three main transformation procedures which restore the invariants, depending on the structure of node p's immediate neighborhood and the broken invariants.

Lets first define two helper functions used by the transformation. Also, let $p^{(r)}$ indicate that node p has a rank r (that is, p is a root of a binomial heap of rank r).

CLEAN-UP(x):

Let p, x, p' and x' be nodes as in figure 3 (p =parent(x), p' =right-sibling(x), x' =last(p'). If after a series of transformations $x^{(r)}$ becomes active, then due to invariant (1) $x'^{(r)}$ cannot be active, so we can swap x and x' (since $rank(x) = rank(x')$), which (locally) restores invariant (2), that is, the active node x becomes the last child of a' (as will later be shown, due to other constraints, CLEAN-UP does not introduce other invariants violations). Runs in $O(1)$.

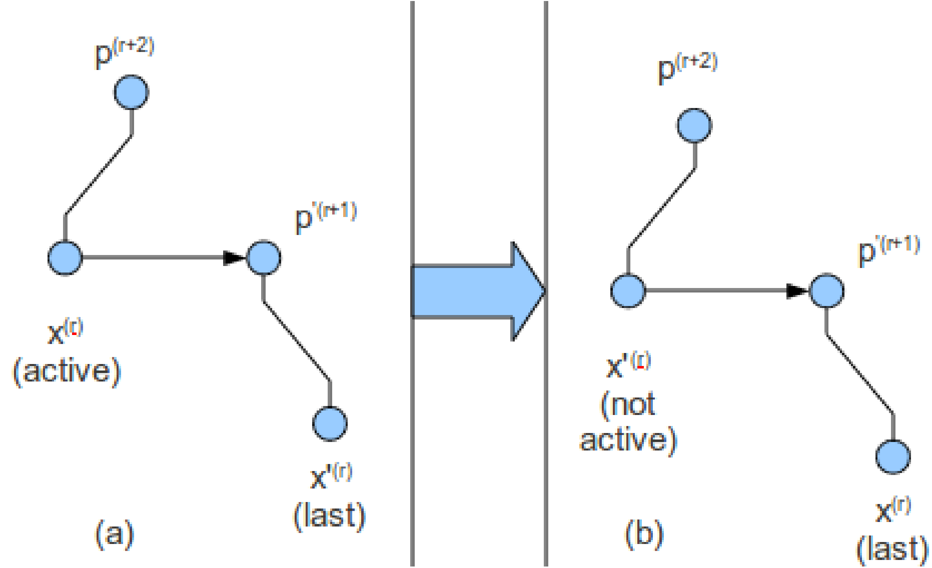


Figure 3a - before CLEAN-UP(x), Figure 3b - after CLEAN-UP(x).

COMBINE(p, q):

Merge two Binomial Trees $p^{(r)}$ and $q^{(r)}$ together (as in regular Binomial-Tree merge) and run CLEAN-UP on the tree merged as a sub-tree of the new root. Also runs in $O(1)$.

Lets now describe the possible invariants violation scenarios and heap transformations reversing them. The transformations are applied recursively after each DECREASE-KEY(q, v), until no further violation exists. Each of these transformations takes $O(1)$ time, as they only operate on pointers to child, parent, sibling. etc.

CASE 1: PAIR TRANSFORMATION

Occurs when $q^{(r)}$ becomes active and there already exists an active node $q'^{(r)}$ of rank r and both $q^{(r)}$ and $q'^{(r)}$ are last children. Let p, p' and g, g' be corresponding parents and grandparents (respectively) as in figure 4. Pair transformation works as follows:

(1) Cut q and q'. Since both are last children, this decreases p and p' ranks by 1 ($p^{(r)}, p'^{(r)}$).
(2) Without loss of generality, assume $key(p) \leq key(p')$ and COMBINE(p, p'), this increases p' rank by 1 ($p'^{(r+1)}$).
(3) Let $Q = $ COMBINE(q, q'). The rank of Q becomes $r + 1$, make it the child of g'.

Because both q and q' were initially active, step 3 decreases the total number of active nodes by at least 1. Node Q might or might not be active at this point. If it is and if any of the invariants are violated, recursively apply this set of transformations on Q.

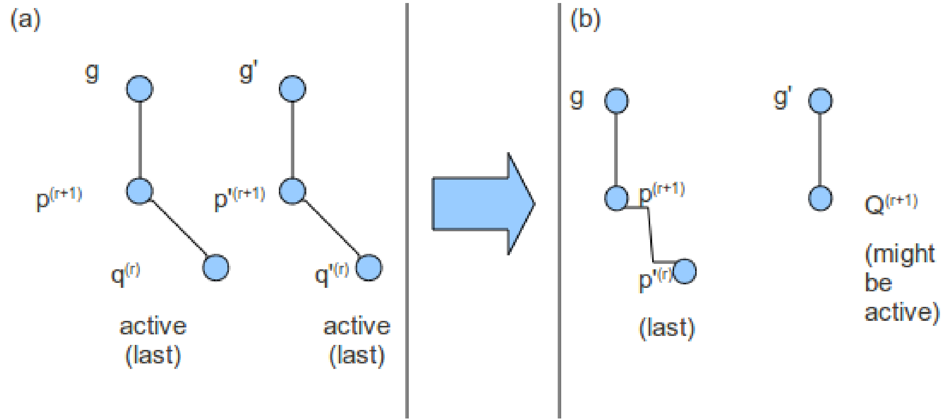


figure 4. pair transformation
a) before pair transformation, b) after pair transformation

CASE 2: ACTIVE SIBLING TRANSFORMATION

Occurs when $q^{(r)}$ becomes active while its right sibling $s^{(r+1)}$ is already active. Due to invariant (2), s must be the last child of p, so p must have a rank of $r+2$ (see figure 5). The steps taken in active sibling transformation are as follows:

(1) Cut $q^{(r)}$ and $s^{(r+1)}$, p has now rank r.
(2) R = COMBINE($q^{(r)}, p^{(r)}$), R has rank $r+1$. R is not active.
(3) W = COMBINE($R^{(r+1)}, s^{(r+1)}$). W has rank $r+2$, might be active.
(4) Make W the child of g (replaces the previous $p^{(r+2)}$).

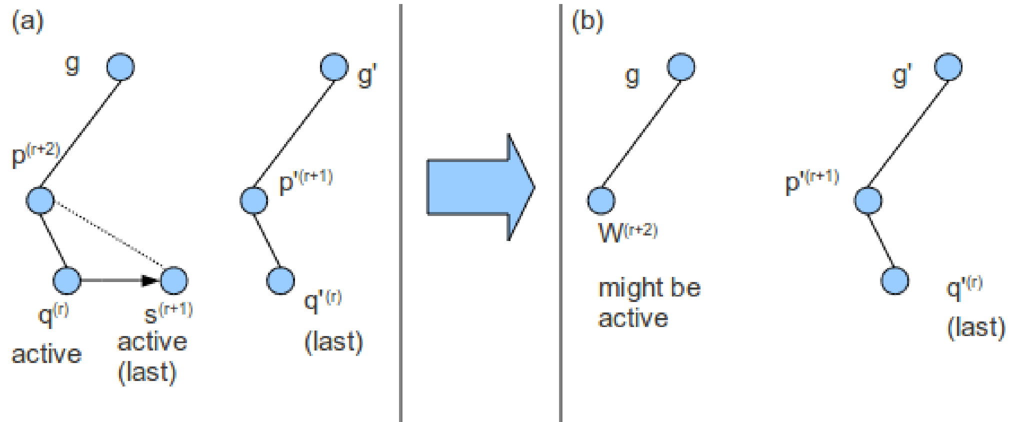


figure 5 - active sibling transformation
a) before b) after active sibling transformation.

Notice that in active-sibling transformation, $q'^{(r)}$, a node that had been active before $p^{(r)}$ became active, is not affected at all (it does even have to exist!). The transformation decreases the number of active nodes by at least 1.

CASE 3: INACTIVE SIBLING TRANSFORMATION

Let $q^{(r)}$ be the just activated node, $s^{(r+1)}$ be its right sibling and $c^{(r)}$ be the last child of s. If s is not active we cannot apply active-sibling transformation. Depending if c is active, there are two cases:

Case 1 - c is active (see figure 6):

(1) Because of q and c are active and have the same rank, apply PAIR transformation on q and c. This will in effect merge q and c together into $R^{(r+1)}$ and make it a right sibling of p.

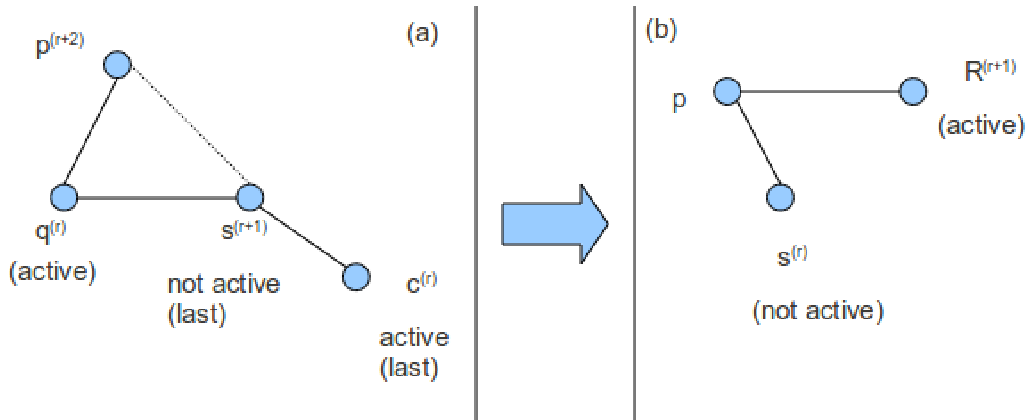


figure 6 - inactive sibling transformation - case 1

Case 2 - c is inactive (see figure 7):

(1) Do CLEAN-UP(q). This effectively swaps s, c and q. Notice that because c was inactive (that is, $key(c) \geq key(s)$), both s and c after transformation are inactive.
(2) If q is still active after step 1, because it is now the last child of c, a regular PAIR transformation can be used on q to restore the invariants.

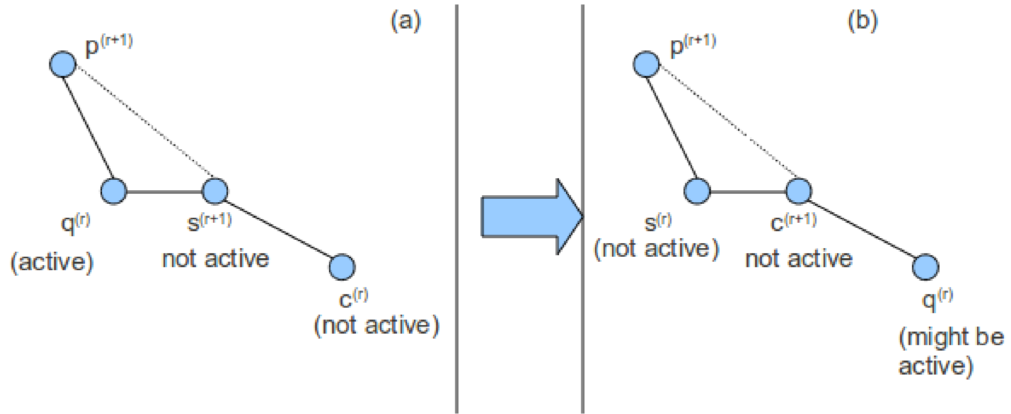


figure 7 - inactive sibling transformation - case 2

Case 1 of Inactive-sibling transformation decreases the total number of active nodes by 1. Case 2 does not, however both cases restores invariant (2) for rank r.

Let α be the total number of active nodes. Each DECREASE-KEY operation can increase α by at most 1. However, after each DECREASE-KEY comes a series of transformation. Each transformation either decreases α, or, does not decrease α, which means it is the final transformation in the series. α can never go below 0, so the series of m DECREASE-KEY runs in $O(m)$ time, therefore a single DECREASE-KEY runs in $O(1)$ amortized time. Since there are at most $\log n$ different ranks, in the worst case this takes $O(\log n)$.

3. Parallel Dijkstra's

In this section, we will discuss how to make Dijkstra's algorithm into a parallel algorithm and also how the priority queues mentioned in Section II relate to the parallel algorithm.

We will assume that the reader is familiar with the traditional version of Dijkstra's algorithm run on a single processor. In particular, we will assume that the reader is familiar with the basic greedy steps of

Dijkstra's algorithm. Dijkstra's algorithm initially assigns a distance of infinity to all vertices from the source and a distance of 0 to the source. It then picks the best vertex not yet "finalized" and finalizes the next best vertex, and updates the best known path to the adjacent vertices of the last finalized vertex, and then finalizes the next vertex, and so on. We will formalize this as follows:

Suppose we are given a graph G with vertices V and edges E, with a source s, and we want to find the distance of the shortest path from s to every other vertex. Then, the algorithm proceeds as follows:

(1) Initialize a set S to store finalized vertices. Let S be initially empty.
(2) Initialize a distance matrix D, where $D[v]$ represents the length of the shortest path from s to v. Let $D[s] = 0$ and $D[v] = \infty$ for v not equal to s initially.
(3) Pick vertex v with smallest distance in D not in S. Look at v's adjacent vertices and update their distances in D. Add v to S and repeat until every vertex is in S.

The crucial step in Dijkstra's algorithm is step 3, where we are picking the next best vertex, updating a distance matrix, and then again picking the next best vertex. It is, in fact, this step, that gains the most from parallelization. It is exactly here where Fibonacci heaps fail in parallel efficiency, and the introduction of new data structures are necessary.

We split up the algorithm by splitting up the priority queue in step 3 into p processors, so each of the processors holds a priority queue of size n/p. Because every processor must be synchronized (they must be at the same iteration), and because Fibonacci Heaps only obtain amortized time bounds, in many cases, some processors are left waiting for other processors to finish, and a lot of processor time is left idling. However, with Relaxed Heaps, because the worst case time bounds are lower, this has smaller impact on the performance, and much less time is left waiting around for all processors to finish an iteration of step 3.

4. Setup of the experiment

We have implemented Fibonacci, Binomial and Relaxed heaps in Java using the Oracle's 64-bit JVM 1.6.0.29 with the original Java's threading library. We run our programs on a 64-bit, 4-core, 1.7GHz Intel i5 processor with 3MB of L3 cache and 8GB of RAM. We have implemented a serial Dijkstra algorithm based on our Fibonacci heap and three parallel Dijkstra algorithms with Fibonacci, Binomial and Relaxed heaps as their internal priority queues.

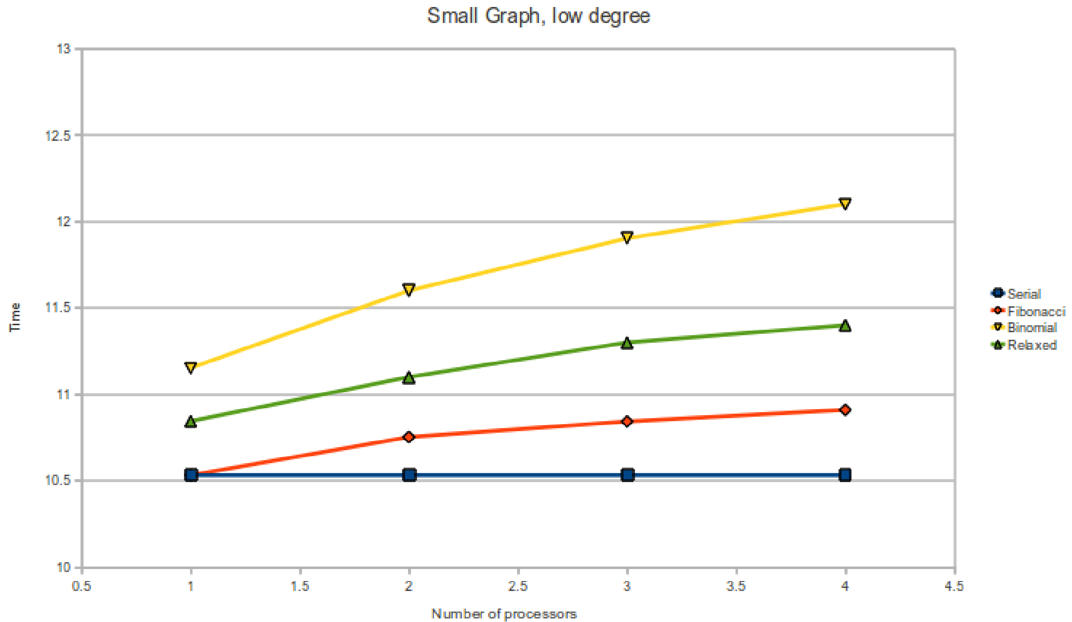
There are three input test cases on which we have tested our implementations and which we consider be be representative and cover multiple use cases. n is the number of nodes in a graph and d is average degree of a node:

(1) Small n ($n = 100,000$), small d ($d = \log n = 16$).
(2) Big n ($n = 10^8$), small d ($d = \log n$ 27).
(3) Big n ($n = 10^8$), big d ($d = 10\sqrt{n} = 10^5$).

The graphs were generated randomly, with small variation of edges' lengths. All of our test cases fit into the test computer's RAM and we have made our best effort to implement the data structures in the most efficient way.

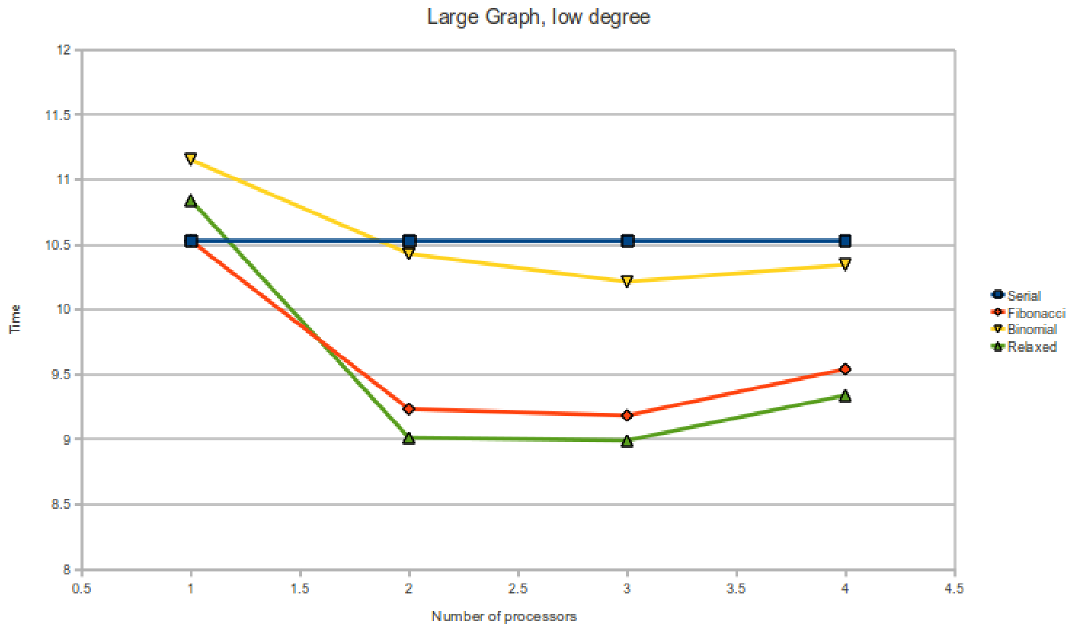
5. Results

With small n and small d, we got the following results:



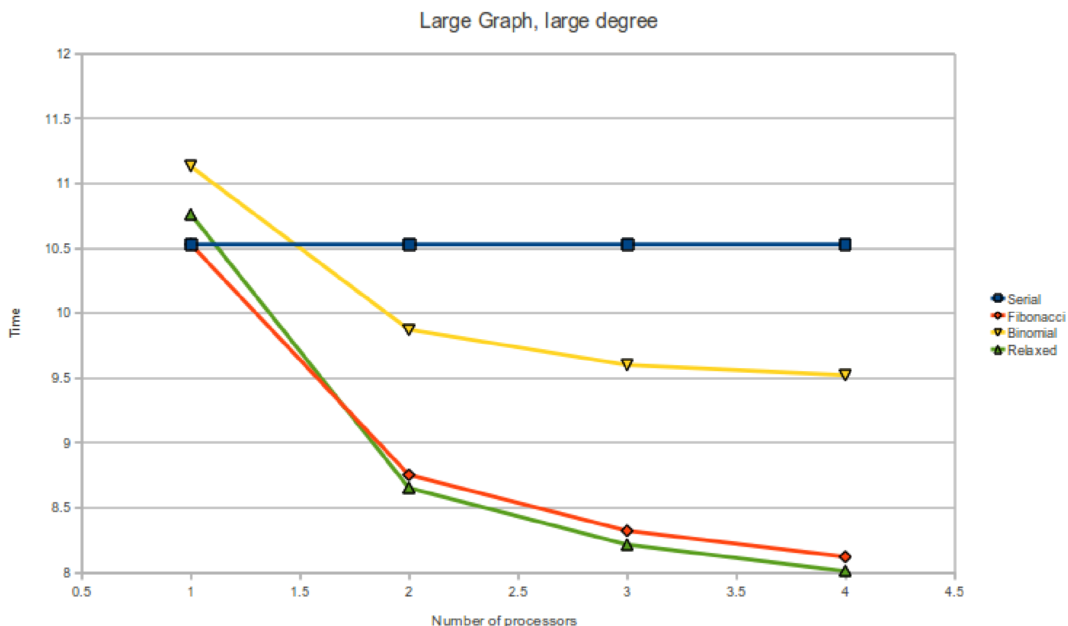
Note that for the serial case, we actually did not increase the number of processors. The results show that as we increase the number of processors, the parallel implementations actually get **worse**. This is probably because the overhead to maintain parallelization is much larger than the benefits we receive for a graph this small.

With large n and small d, we got the following results:



Note that we scaled the time so that the time taken for the serial case is the same as in the small graph so its easier to compare results. The results show that initially, with no parallization, Fibonacci heaps outperform relaxed heaps, probably due to a higher overhead for relaxed heaps. However, as we increase p, relaxed heaps start to outperform Fibonacci heaps as expected. Eventually, the overhead for maintaining higher p's take over and as we increase p, the performance actually gets worse. This is likely because we only have 4 cores, and we need at least one thread to run the main algorithm and one core for each of the paralellizations.

With large n and large d, we got the following results:



In this case, the gains from parallelization is predictably greater, and we don't yet see the deterioration as we increase p.

6. Summary

Due to an increase in parallelization of modern hardware, it is expected that parallel algorithms will play increasingly more important role in modern computing. Shortest path algorithms, such as the Dijkstra algorithm, play an important role in many practical applications and optimizing it for multiple cores should bring increasingly more benefits. We have shown how to transform the original Dijkstra's algorithm to a parallel version. Furthermore, as our experiment have demonstrated, using Relaxed heaps as priority queues in the parallel version offers an improvement over the traditional Fibonacci heaps.

References

[1] Vuillemin, Jean. "A data structure for manipulating priority queues", *Communications of the ACM.* Vol. 21 Issue 4 (1978): pp. 309-315.
[2] Driscoll, Harold N. Gabow, Ruth Shrairman and Robert E. Tarjan. "Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation", *ACM Transactions on Graphics.* Vol. 31 Issue 11 (1988): pp. 1343-1354.
[3] Cormen, Thomas H., Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to Algorithms.* Cambridge, MA: MIT, 2001. Print.