5-2018

Towards Scalable Parallel Fibonacci Heap Implementation

Divya Bhattarai
St. Cloud State University, bhattaraidivya@gmail.com

Towards Scalable Parallel Fibonacci Heap Implementation

by

Divya Bhattarai

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in Computer Science

May, 2018

Starred Paper Committee:
Jie Hu Meichsner, Chairperson
Dennis Guster
Mehdi Mekni

Abstract

With the advancement of multiple processors, the sequential algorithms are being investigated and gradually substituted for its concurrent equivalent to effectively exploit the parallel architecture. Parallel algorithms speed up the performance by dividing the task into a number of processes (or threads) that can be scheduled and executed simultaneously in independent processing units. Various well-known basic algorithms and data-structures have been explored for its efficient parallel counterparts and have been published as popular libraries. However, advanced data-structures and algorithms have not seen similar investigation mainly because they have many optimization steps mostly backed by many states and finding safe and efficient parallel implementation isn't an easy endeavor.

Safety concerns for shared-memory parallel implementation are of utmost importance as it provides a basis for consistency of any data structure and algorithm. There are well-known tools like locks, semaphores, atomic operations and so on that assist towards safe parallel implementation but using them effectively and in well-defined synchronization are key factors in the overall performance of any data-structures and algorithms.

This paper explores an advanced data structure, Fibonacci Heap, and its operations to evaluate its implementation using two different synchronization mechanisms: Coarse-grained and Fine-grained. The analysis in this paper shows that a fine-grained synchronized Fibonacci Heap implementation with certainly relaxed semantics is more scalable with growing number of concurrency in comparison to the coarse-grained synchronized Fibonacci Heap implementation.

Acknowledgment

I would like to sincerely thank my advisor Dr. Jie Hu Meichsner, Department of Computer Science and Information Technology for her suggestions and continuous guidance for the successful execution of the research and its implementation.

I would like to express my gratitude towards my committee members Dr. Dennis Guster, Department of Information Systems, for allowing me to use the BCRL lab, where I was able to carry out the experiments for the analysis; and Dr. Mehdi Mekni, Department of Computer Science and Information Technology, for his constructive feedback and support throughout the research.

I would also like to thank my friends and all the faculty members of the Computer Science Department at SCSU who helped me to shape my knowledge throughout my study. Lastly, it wouldn't have been possible without the continuous support of my family and, especially my friend, Suraj Poudel, who helped me on every stage of the research and implementation throughout its completion.

Table of Contents

Page

Chapter Page

List of Tables

List of Figures

List of Algorithms

Chapter I: Introduction

Overview

Modern day computers have seen unprecedented growth in low-cost high-performance computing mainly because of cheap energy efficient multi-core processors. But because multi-core processors are architecturally different from single-core processors, software implementation needs to be re-designed with multiprocessor programming paradigm to be able to maximize performance benefits of such systems. This paradigm, in general, requires implementations to break operations into multiple tasks that can be scheduled and executed simultaneously in different cores. One big challenge in multiprocessor programming stems from the fact that the memory is shared between multiple cores and simultaneous access /modification to the same memory address can lead to an inconsistent state called race condition.

In multiprocessor programming, if a memory address is being written/read simultaneously by multiple processes/threads, then implementations need to ensure guarding operations on this memory address to avoid probable inconsistencies. Applying guards on such operations, e.g., locks, leads to sequentially executing in such regions, called critical paths that restricts implementation to reach its theoretical speedup. Reducing the time taken in executing critical paths has great influence on increasing the efficiency of parallel implementation. The coarse-grained locking mechanism has been popularly used to provide guards in critical paths of programs, which can, however, make the program difficult to scale out as many integrated cores in a single chip becomes more common. This has led the scientific community to research for lock-free algorithms and data structures operations or to explore options to reduce the critical paths, termed as fine-grained locking.

Related Work

Fibonacci Heap is an advanced data structure, introduced by Fredman and Tarjan (1987). It is widely used to implement priority queues. The priority queue is one of the most used data structure to implement various algorithms like Single Source Shortest Path Algorithm, Vertex Problem etc. The sequential Fibonacci heap algorithm is known to be the most efficient algorithm for the implementation of priority queues (Huang & Weihl, 1991). For the Fibonacci heap, the extract minimum operation takes constant, i.e., O(1) amortized time. The insert and decrease key operations also work in constant amortized time.

The primary motivation of the Fibonacci heap was to gain speed up in the performance of Dijkstra's algorithm from O (E log V) to O (E + V log V) (Wayne, 2007). Huang and Weihl (1991) provided a concurrent Fibonacci heap's design and implementation, following closely the sequential algorithm, with a low contention by distributing locks over the entire data structure and showed experimentally to have linearly scalable throughput and speedup up to many processors. The efficiency obtained in the Huang and Weihl studied relied on their assumption that strict semantics on extracting nodes concurrently are mostly undesirable. Shavit and Zemach (1999) addressed problems of designing scalable priority queue structures that support a fixed range of priorities as opposed to an unbounded range of priorities and claim to have better scalability. Shavit and Zemach designed a funnel-based algorithm for priority queue implementation which does not directly correspond with its sequential algorithms. Few other pieces of research have also implemented parallel priority queues based on binomial heap (Das & Pinotti, 2000) and relaxed Fibonacci heap (Boyapati & Rangan, 1995).

Objective

The objective of this paper is to investigate the parallel implementation of the insert and extract-minimum operations in Fibonacci Heap data structure, a computationally best-known priority queue implementation algorithm.

A coarse-grained synchronization mechanism can provide a safe and easy parallel implementation for the Fibonacci heap, but the implementation cannot scale-out well with increasing number of threads which is imminent with growing number of cores in a single chip. However, if Fibonacci heap operations are investigated to find various linearizable sections to implement fine-grained synchronization, then it is more likely to scale out better with increasing parallelism.

In this paper, a coarse-grained synchronized Fibonacci Heap is implemented using a global lock to guard operations in sequential Fibonacci Heap. Algorithm for sequential Fibonacci Heap is provided in Appendix A: Sequential Fibonacci Heap Algorithm and this will be referred as *SEQ_Fibonacci_Heap* hereafter.

This paper focuses on investigating the ways that lead to fine-grained critical regions in the Fibonacci Heap operations to increase the degree of parallelism in the *PARALLEL-FIB-HEAP-INSERT* and *PARALLEL-FIB-HEAP-EXTRACT-MIN* operations. The performance of coarse-grained synchronized Fibonacci Heap is tabulated as a basis for comparison with the fine-grained synchronized Fibonacci Heap performance with increasing parallelism.

Chapter II: Fibonacci Heap

Overview

This chapter of the paper describes the background required to understand Fibonacci heap, associated problems, and the works related to the parallel implementation of the Fibonacci heap.

Overview of Sequential Fibonacci Heap

Fibonacci heap is a collection of heap-ordered trees, where root always contains the minimum element among the trees. The roots of all trees in the Fibonacci heap are connected by the circular, doubly linked list. The circular linked list has advantages in the Fibonacci heap, we can remove an element from the circular, doubly linked list in O (1) time. It has a minimum pointer pointing to the minimum element of the root list (Wayne, 2007). Figure 1 shows the structure of the Fibonacci heap.

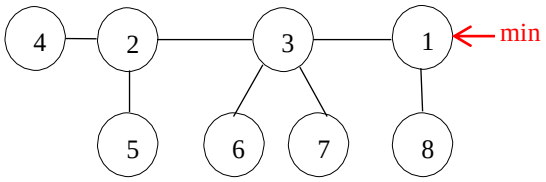


Figure 1. Structure of Fibonacci Heap

Fibonacci heap operations. Amongst many operations possible in the Fibonacci heap, this paper mainly explores insert, extract min, and consolidate operations which are briefly described below.

- Insert: Insert operation refers to the insertion of an element in the root list and can be inserted anywhere. To insert a new element, a new node is created. The position of the newly created node is located by finding the two adjacent nodes, between which it

needs to be inserted. The corresponding pointers of those nodes and newly created nodes are updated such that they are linked to each other. After insert operation, if the new element is smaller than the element pointed by the minimum pointer, the value pointed by the minimum pointer needs to be updated. The insert operation takes constant time to insert an element.

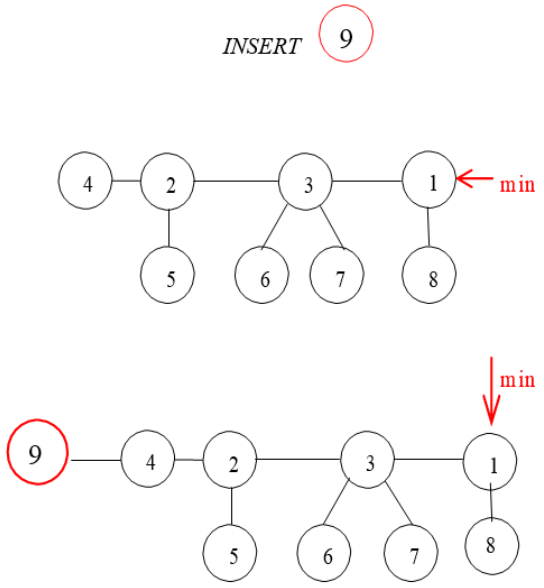


Figure 2. Insert Operation in Sequential Fibonacci Heap

- Extract Min: Extract-min operation extracts the element pointed by the minimum pointer in the heap. Once the value pointed by the minimum pointer is extracted, the child nodes of the root node meld with the root list and the value pointed by the minimum pointer is updated. In Figure 3, the value pointed by minimum pointer 'min' is extracted, i.e., '1' and pointer is updated to next minimum value '2'. The child node of '1' i.e. '8' is melded with the root node (Wayne, 2007). Extract operation takes logarithmic time. The heap needs to be consolidated after the extract-minimum operation.

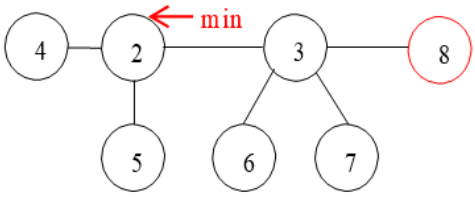
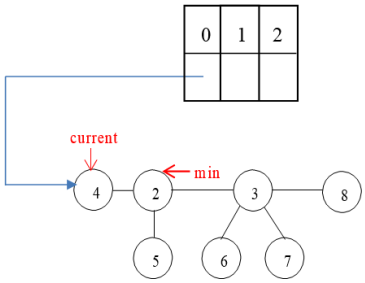
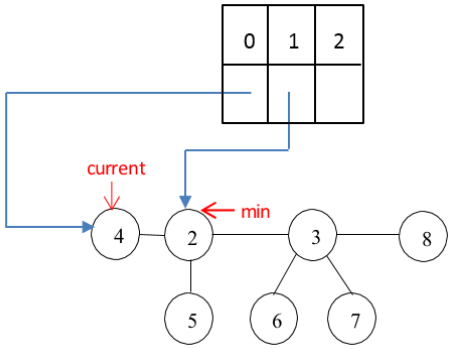


Figure 3. Extract-min Operation in Sequential Fibonacci Heap

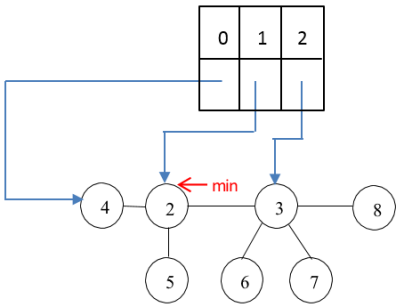
- Consolidate. Consolidation is the process during which, trees with the same degree are merged together, thus reducing the number of trees in the Fibonacci heap. This causes the amortized cost of extracting the minimum node to be O(D(n)) where D(n) is the maximum degree of an n-node Fibonacci heap which has an upper bound of O (log n). The degree of the trees refers to the number of children of its root node.



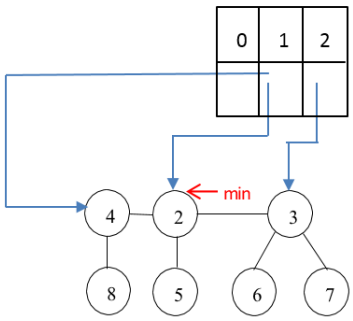
A. Degree of ‘4’ is Zero.



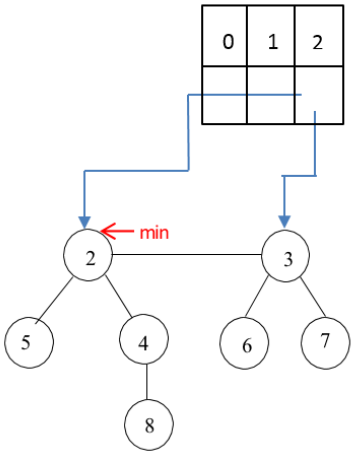
B. Degree of ‘2’ is One.



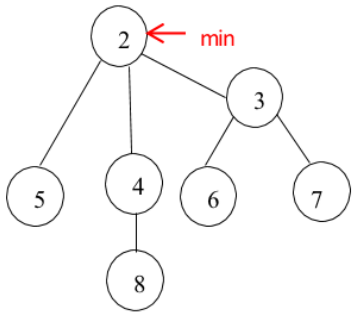
C. Degree of ‘3’ is Two.



D. Degree of ‘8’ is Zero, So Merge It Below ‘4.’



E. Degree of ‘4’ is Equal to the Degree of ‘2’ so Merge Tree 4 Below ‘2.’



F. Degree of ‘2’ is equal to the degree of ‘3’ So Merge Tree 3 Below ‘2’ (Wayne, 2007).

Figure 4. Consolidation Operation in Sequential Fibonacci Heap

Chapter III: Parallel Data Structure

Parallel data structures are the way of storing and organizing data that need concurrent access by multiple threads or processors. This mainly represents data structures that can be accessed by multiple threads executed on multiple processors that can actually be accessing/updating the data and/or internal states of the data structures simultaneously. As parallel data structures can be accessed simultaneously through multiple computing resources, this is also referred to as shared data structures and are generally allocated in a shared storage environment referred to as shared memory.

In a parallel computing environment, data structures need to have additional properties in comparison to sequential environments. Safety and liveness property are two such properties. The liveness property refers to property that specifies data structures to make progress even if the executing multiple processors sometimes might have to wait for certain resources to be available (e.g., wait on locks) in critical sections i.e. part of the program that cannot be executed simultaneously by multiple processors. Because there is no guarantee on how the threads will be scheduled and unscheduled on the multi-processor environment, there are many possibilities of how methods can be interleaved at any threaded execution. So, with safety property, data structures ensure correct execution in various such possibilities. It is, therefore, significantly more difficult to design and verify concurrent data structures than their corresponding sequential data structures.

Issues with Parallel Implementation of a Data Structure

Data structures have operations that modify/accesses its internal and external states. In the parallel implementation, operations on the data structures are called from multiple

threads/processes and there is no guarantee that those calls won't interfere with each other unless synchronized explicitly. If the operations are just reads, then there is no need for any synchronization because all the threads/processes read the same state. However, when there are reads and writes operations called by multiple threads/processes, it is not uncommon for threads to view the inconsistent state of the data structure, which is called race condition.

Race condition. A race condition is a bug in multithreaded programs, which occurs when two or more threads access the same memory location, and the result depends on the order of execution of the threads. Such memory location is called the critical section. However, it only occurs when one of the threads is writing to the memory location. That means we have room to avoid this situation by carefully synchronizing these events as long as the resources do not change (Tsyrklevich & Bennet, 2003).

Example: Let us assume that two threads want to increment the value of a global integer variable by one. Ideally, the following sequence of operations would take place:

Thread 1	Thread 2		Value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

In the above example, the expected final value is 2. However, the end result could be wrong if multiple threads run simultaneously without any locks or synchronization, which is shown below.

Thread 1	Thread 2		Value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

In this case, the final value is 1 instead of the expected result of 2. This is because of the race condition where the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location.

Preventing race conditions. In computing environments, race conditions can be prevented by following methods:

- Thread synchronization: The loading and saving a shared variable are usually implemented as separate operations and are not atomic. This means if we consider the above example, an “increment variable” operation is usually converted into loading, incrementing, and saving operation, so if the variable memory is shared the other process may interfere with the incrementing, easily leading to a race condition. In this method, the race condition can be avoided by the serialization of memory or

storage access. This means if read and write commands are received close together, the read command is executed and completed first by default. This can be achieved by performing atomic actions in the file system and using temporary files (Tsyrklevich & Bennet, 2003).

Locking. If we grant an exclusive right to perform a certain operation, it helps to avoid the race condition. However, several other problems get introduced along with the locks, namely, deadlocks, livelocks, and releasing "stuck" locks if a program does not clean up its locks. A deadlock can occur if programs cannot proceed forward because of waiting for each other to release resources (Tsyrklevich & Bennet, 2003). For example, a deadlock would occur if Process 1 locks Resource A and wait for Resource B, while Process 2 locks Resource B and waits for Resource A. Many deadlocks can be prevented by simply requiring all processes that lock multiple resources to lock them in the same order (e.g., alphabetically by lock name). The locking mechanism can be implemented in following ways: Using Files as Locks: Whenever process wants to access the file, lock that file so that other process cannot request for the file access.

Implementation techniques for parallel algorithms. As discussed earlier, the implementation and design of parallel algorithms is a difficult endeavor. Although the tools and constructs required to assist safe and live implementation are prevalent, putting them together to get the complete implementation of any parallel algorithms and data structures require more consideration. One of the key issue to be addressed is performance.

The speedup of any algorithms is the ratio of its execution time in a single processor to its execution time in multiple processors. The ideal speedup is to be linear, i.e., with P processors

the speedup should be P. Data structures and algorithms that have a linear speedup are called scalable. However, using tools for e.g. locks can severely undermine the scalability if used naively. Techniques for implementing various parallel algorithms fall into this broad category of how locks are used to implement various synchronization points to provide the safe implementation and those techniques have the different impact on the performance of the algorithms which are discussed below:

Blocking concurrency algorithms: A blocking concurrency algorithm is an algorithm which either performs the action requested by the thread or blocks the thread until the action can be performed safely.

There are several algorithms and concurrent data structures which are blocking. If we consider the concurrent BlockingQueue in Java, if a thread attempts to insert an element into a BlockingQueue and the queue does not have space, the inserting thread is blocked until the BlockingQueue has space for the new element (Cao & Singhal, 1998).

The following diagram illustrates the behavior of a blocking algorithm guarding a shared data structure:

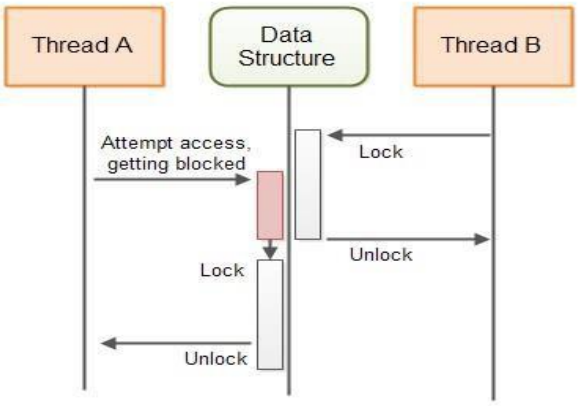


Figure 5. Blocking Algorithm Illustration (Jenkov, 2015)

While implementing concurrent programs, the only bottleneck might be the lock contention. When multiple threads run at the same time they might compete for the same lock. If one thread holds a lock on a resource for a while and the other thread waits for the same resource, it turns into a competition. This introduces widely used terms such as "coarse-grained" locking and "fine-grained" locking mechanism.

In the coarse-grained locking, a larger portion of data is locked by a single lock, which makes it easier to implement. Hence, the coarse-grained locking mechanism can easily make algorithms safe since large portions of the data are guarded with very few locks. However, in fine-grained locking, we guard individual data elements with different locks as opposed to a single lock guarding most of the data elements. This highly reduces the lock contention and improves performance in terms of speed up. But, fine-grained locking can easily deadlock/livelock if not carefully considered of various scenarios, which makes it difficult to implement. This paper focuses on fine-grained locking algorithms and its implementation.

Non-blocking concurrency algorithms. A non-blocking concurrency algorithm is an algorithm which either: performs the action requested by the thread or notifies the requesting thread that the action could not be performed (Cao & Singhal, 1998).

If we consider the Java again, it contains several such non-blocking data structures. The *AtomicBoolean, AtomicInteger, etc* are some non-blocking data structures. This diagram illustrates the behavior of a non-blocking algorithm guarding a shared data structure.

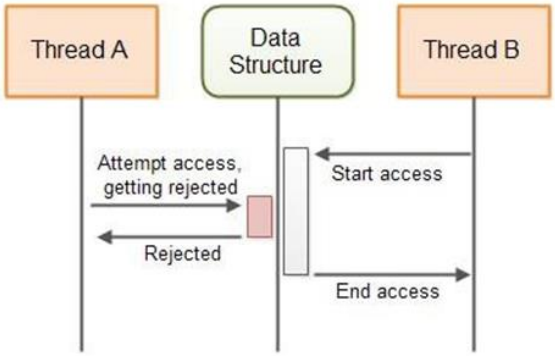


Figure 6. Non-Blocking Algorithm Illustration (Jenkov, 2015)

Chapter IV: Parallel Implementation Details for Fibonacci Heap

This chapter describes the tools, libraries, and system used to implement both coarse-grained and fine-grained synchronized implementation of Fibonacci Heap.

System and Libraries

Parallel Fibonacci Heap is implemented in C++ and different libraries in C/C++ is used to make low-level system calls e.g. creating a thread. Also, the implementation was tested on Linux system with multi-core processors. The following section describes them in detail.

POSIX threads. *POSIX threads library is a s*tandardized C language threads programming interface designed to develop portable threaded applications for UNIX systems. It has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads or Pthreads (Blaise, 2017). Pthreads are defined as a set of C language programming types and procedure calls, implemented with a "*pthread.h"* header/include file and a thread library - although this library may be part of another library, such as libc, in some implementations. Pthreads library was considered for the implementation because of the following reasons:

Lightweight:

- A thread can be created with less OS overhead as compared to the process.
- Managing threads requires fewer system resources than managing processes.

Table 1

Comparison of Timing Results between fork and pthread_create Method

Platform	fork ()			pthread_create ()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Efficient Communications/Data Exchange:

- For Pthreads, there is no intermediate memory copy required because threads share the same address space within a single process.
- There is no data transfer and it can be as efficient as simply passing a pointer (Blaise, 2017).

Table 2

Comparison of MPI Shared Memory Bandwidth to Pthreads Worst-Case Memory-to-CPU Bandwidth

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Other common reasons: Threaded applications offer potential performance gains and practical advantages over non- threaded applications in several other ways:

- Overlapping CPU work with I/O: For example, a program may have sections where it is performing a lengthy I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

System details. The experiments shown in this paper were performed on the system with the following configuration:

Hostname	Csci606
Number of Cores	8
Processor Model	Intel Xeon CPU E5-2680 v2 @ 2.8 GHz
Memory	16 GB
Compiler	g++ 5.4.0
Profiler	gprof

Algorithm Details

The following section describes sequential, coarse-grained, and fine-grained algorithms in detail.

Sequential fibonacci heap. Sequential Fibonacci Heap was implemented according to Appendix A using C++.

Coarse-grained fibonacci heap. The coarse-grained implementation shown below maintains a global lock to guard individual operations to synchronize between various operations performed on the heap.

The sequential algorithm for each Fibonacci Heap operations is guarded by a global lock to avoid two or more threads from simultaneously updating the heap state, thus creating a coarse-grained synchronized parallel Fibonacci Heap implementation. Fibonacci Heap has various other internal operations like CONSOLIDATE which do not explicitly need guards as guarding FIB-HEAP-EXTRACT-MIN and FIB-HEAP-INSERT operation automatically avoids race

conditions. Such implementation of parallel Fibonacci Heap is a naive approach towards guaranteeing safety.

Algorithm 1

Coarse-Grained Synchronized Fibonacci Heap

```
state:
    std::mutex global_lock
    SEQ_Fibonacci_Heap heap


operations:


PARALLEL-FIB-HEAP-EXTRACT-MIN ():
    global_lock.lock()
    min = FIB-HEAP-EXTRACT-MIN(heap)
    global_lock.unlock()
    return min


PARALLEL-FIB-HEAP-INSERT (x):
    global_lock.lock()
    FIB-HEAP-INSERT (heap, x)
    global_lock.unlock()
```

There is a subtle assumption in the above algorithm, that is simultaneous operations on the Fibonacci Heap are always unsafe regardless of any input and it is safe to always avoid having two operations to occur together with no matter what. However, with this approach the cases where *FIB-HEAP-EXTRACT-MIN* and *FIB-HEAP-INSERT* operations that might not have race conditions are not considered. For example, if we perform *FIB-HEAP-INSERT(x)* in one of the trees in one part of heap whose minimum is far from being the current minimum, then

performing *FIB-HEAP-EXTRACT-MIN* can happen simultaneously without racing the *FIB-HEAP-INSERT(x)* operation.

Fine-grained fibonacci heap. This paper primarily focuses on implementing the basic relaxed semantics proposed by Huang and Weihl (1991) and improved it using various optimization techniques. The following sections describe the proposed insert, extract-min, consolidate and spill over buffer implementation in details.

Insert operation. Insert semantics for fine-grained Fibonacci Heap is similar to the insert operation of a sequential Fibonacci heap. However, if the same sequential insert implementation is used for the parallel counterpart, then a thread trying to insert an element in the root list needs to first lock the complete list. If another thread tries to insert a new element in the heap subsequently, it will have to wait till the first thread completes its insert operation and releases the lock. This makes multiple threads contend on a single lock for insert, thus creating a bottleneck for scaling insert operation. Hence, modifications are required to avoid this bottleneck. This paper proposes spreading out insert operations across root list by placing many dummy nodes, which owns a unique lock to guard smaller sections of the root list as shown in Figure 7.

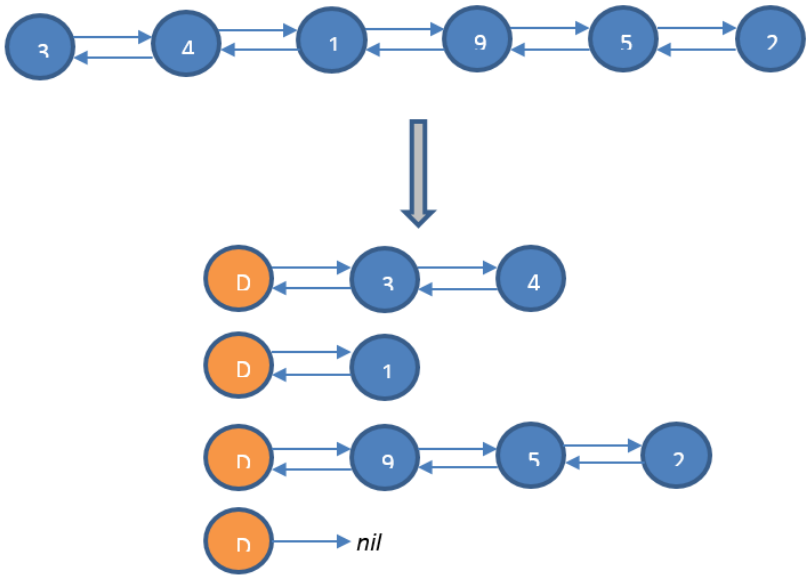
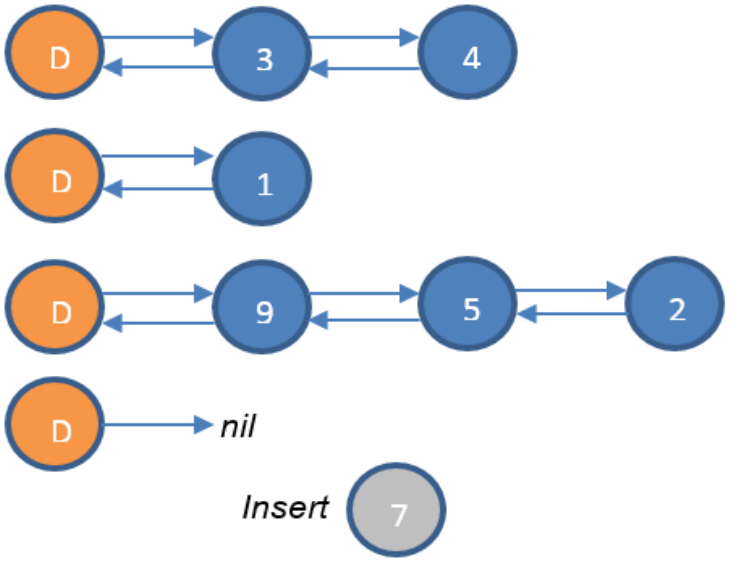


Figure 7. Breaks Root List into Multiple Sections to Scale Multiple Inserts

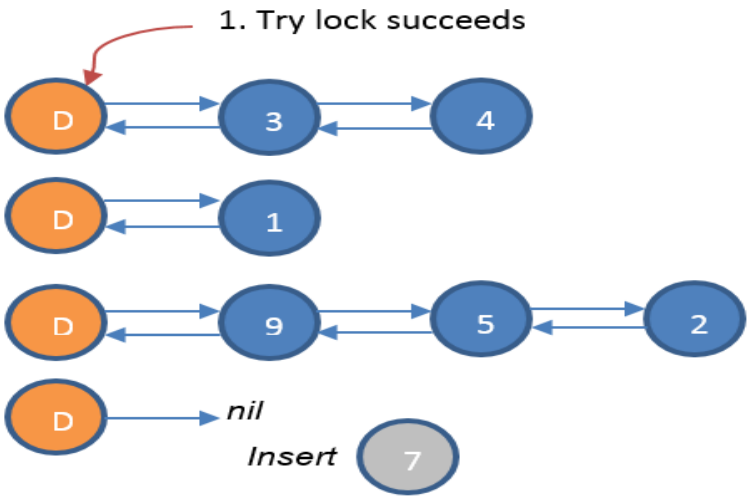
Dummy nodes are like normal nodes of the doubly linked list, except they do not represent node with a valid value for the heap. On a request for insert operation by a thread, a dummy node is randomly chosen to try to acquire its lock. If the lock in the dummy node is successfully acquired, the element is inserted into the right link of the dummy node. If the lock couldn't be acquired, the dummy node was already locked by another thread. So, the thread retries the entire process i.e. another dummy node is randomly chosen to be acquired and the new node is inserted on its right side, if successful (Refer to Algorithm 2 for an algorithm and Figure 8 for an example).

After inserting a new node, the minimum pointer might need to be updated. This paper is based on the design, which relaxes the idea of only using a single minimum pointer and substitutes it with a list of pointers to potential minimum elements referred as a promising list, similar to the Huang and Weihl (1991) implementation. However, this paper proposes that this method needs to be separated from each insert operations, instead amortized with extract

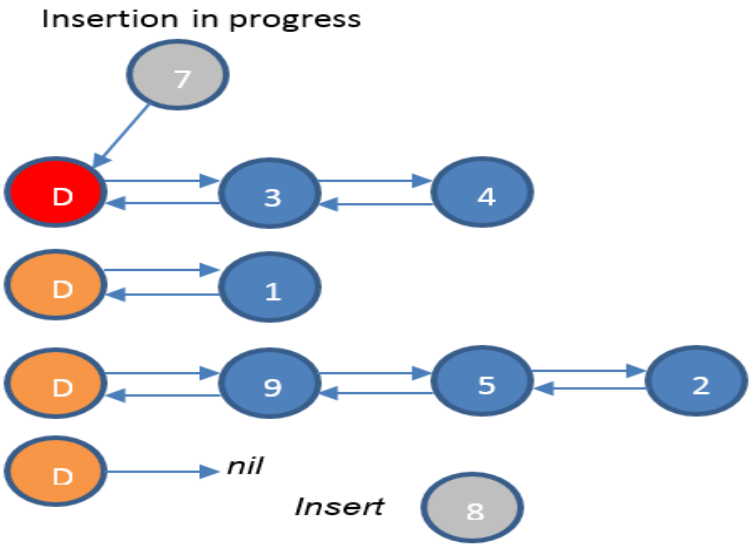
operation which will be discussed later in the section on Spill-Over Operation as spill-over buffer operation. This was proposed because of a subtle flaw in Huang et al implementation. According to Huang and Weihl, the promising list is updated if any element in the promising list is greater than the new element, or if any node in the promising list is dead (dead refers to the node which is extracted from the promising list) or if the node is nil (empty node). However, there is a problem with this logic of updating the promising list in an insert operation. Consider a scenario when 100 elements are inserted in the Fibonacci Heap in ascending order and the promising list is of size, say three. Then the first three elements are inserted in the promising list. No thread extracts an element from the promising list, until the 99^{th} insertion. Before 100^{th} insertion, another thread extracts an element from promising list and marks it as dead. Now in the 100^{th} insertion, on checking for its eligibility in the promising list, it finds a dead node existing and replaces it. If another thread performs extract min, then 100 might be returned, despite the presence of several other minimum elements in the heap. This paper proposes to improve such cases which will be discussed later.



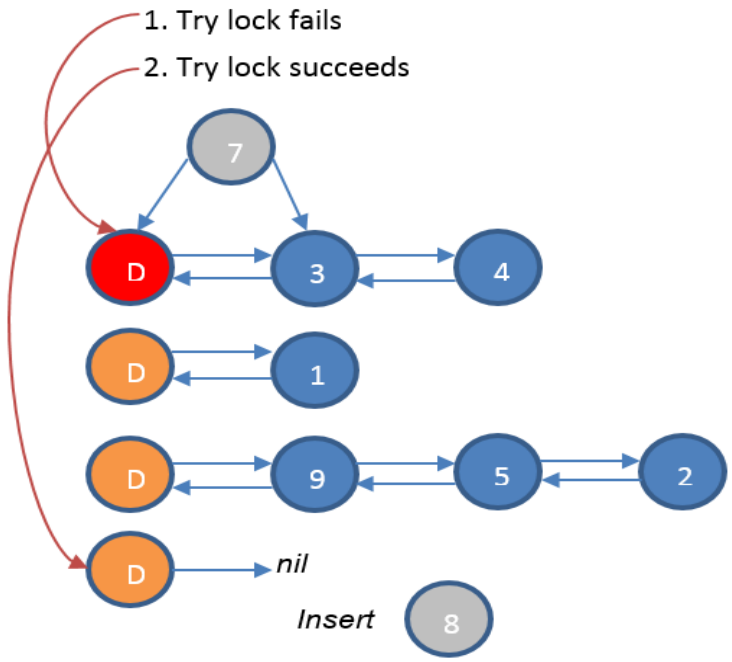
A. Insert '7' into the Fibonacci Heap



B. Try lock at a random dummy node. Locking succeeds so insertion for 7 begins to the right of the dummy node.



C. While insertion for 7 is in progress in a thread A, another ‘insert 8’ operation is performed in another thread B.



D. Try lock at a random dummy node. Here, the first attempted random dummy node already had lock acquired because of ongoing 'insert 7' operation. So, another dummy node is attempted for trying lock. It then finds an unlocked dummy node to insert into.

Figure 8. Various Insert Scenarios in Fine-Grained Fibonacci Heap Implementation

Primarily, much of the insertion logic remains same as Huang and Weihl (1991). The only but major difference is the inserted value does not compete to be on the promising list. The inserted node is just added to the doubly linked list at one Fibonacci heap section that the thread successfully acquires the lock to. The inserted node is only allowed to compete for the promising list when the *EXTRACT* thread does a spill over.

As an example, consider that the first insert does not update the minimum value immediately. Only the first extract operation goes ahead to attempt to consolidate on one node and then reattempts *EXTRACT* to do a spillover from that node to the promising list.

Algorithm 2

Fine-Grained Synchronized Fibonacci Heap Insert Operation

```
1.  void insert(FibonacciHeapNode < T > * to_add_node) {
2.      int dummy_list_index = -1;
3.      # Try to acquire a section of the Fibonacci heap root list
4.      do {
5.          dummy_list_index = get_random_index();
6.      } while (!dummy_list_locks[dummy_list_index].try_lock());
7.
8.      # Adjust the pointers of doubly linked list
9.      FibonacciHeapNode <T> * dummy_node = dummy_list[dummy_list_index];
10.     to_add_node -> right = dummy_node -> right;
11.     to_add_node -> left = dummy_node;
12.     if (dummy_node -> right != nullptr) {
13.         dummy_node -> right -> left = to_add_node;
14.     }
15.     dummy_node -> right = to_add_node;
16.
17.     # Unlock this portion
18.     dummy_list_locks[dummy_list_index].unlock();
19. }
```

Extract min operation. The semantics of the insert operation is less strict as compared to the extract operation of a sequential Fibonacci heap. The sequential Fibonacci heap always gives the minimum element of the heap on extract operation. However, on parallel implementation of the Fibonacci heap, if many processes are extracting nodes concurrently, then contention occurs. The minimum value needs to be updated and consolidation of the tree must happen after each extraction. So, the next thread must wait till the first thread is done, i.e., extraction will happen sequentially. This will be the bottleneck for extract operation. Hence, in this paper, the extraction operation is relaxed, such that, the extract operation returns one of the values from the promising list and marks the node dead.

Performing multiple extractions will result in an empty promising list. To facilitate the early warning that the promising list might be empty in near future, few modifications are proposed in this paper. Firstly, the promising list size(*PL_SZ*) is tripled (3 * *PL_SZ*) and a pointer *EXTRACT_PTR* moves along the list in a cyclic fashion per call to the extract-min. When the *EXTRACT_PTR* has moved by *PL_SZ* value from the last *CONSOLIDATE*, the *CONSOLIDATE* is called. If the last call to extract-min made a *CONSOLIDATE*, it would result in calling spill-over. If neither of this happens, it just goes through the promising list and tries to lock a node. The first successful node at which lock is attainable is returned after the node is marked as dead and unlocked. The *EXTRACT_PTR* has incremented atomically.

Algorithm 3

Fine-Grained Synchronized Fibonacci Heap Extract Min Operation

```
1.  const FibonacciHeapNode < T > * extractMin() {
2.      int offset = pointing++;
3.      if (offset % (3 * PROMISING_LIST_SZ) == PROMISING_LIST_SZ - 2) {
4.          consolidate();
5.      } else if (offset % (3 * PROMISING_LIST_SZ) == PROMISING_LIST_SZ - 1) {
6.          spill_from_all_section();
7.      } else if (offset % (PROMISING_LIST_SZ) == 0) {
8.          count += PROMISING_LIST_SZ;
9.      }
10.
11.     for (int j = (offset % (3 * PROMISING_LIST_SZ));
12.              j != count % (3 * PROMISING_LIST_SZ);
13.              j = (j + 1) % (3 * PROMISING_LIST_SZ)) {
14.         int k = j;
15.         if (promising_list_locks[k].try_lock()) {
16.             if (promising_list[k] != nullptr
17.                   && promising_list[k]-> state == PROMISING) {
18.                 T value = promising_list[k]->value;
19.                 promising_list[k]-> state = DEAD;
20.                 promising_list_locks[k].unlock();
21.                 return new FibonacciHeapNode<T>(value);
22.             }
23.             promising_list_locks[k].unlock();
24.         }
25.     }
26.     return extractMin();
27. }
```

Spill-over operation. During this operation, the thread starts attempting to lock all the consolidate locks from index 0 to *CONSOLIDATE_LOCKS_SIZE*. It blocks on waiting for the locks to be acquired. After acquiring the lock for a section, it scans through the nodes in that section (only the root list) and stores the reference of top *PL_SZ* minimum number in the list. As the number is being added, it goes through an *IN_PROGRESS* list, that maintains the sorted list across all the section, doing an insertion sort like sequence. Thus, at the end of the spill-over operation, *PL_SZ* number of minimum values are obtained which are ready to be spilled over the main promising list in the spill-over section.

This refills the promising list thus making more extract-min to return the value immediately without undergoing the relatively expensive consolidate and spill-over operation, thus resulting in non-contending parallel extraction.

Algorithm 4

Fine-Grained Synchronized Fibonacci Heap Spill Over Operation

```
1.  void spill_over_buffer(int section_index) {
2.      dummy_list_locks[section_index].lock();
3.      FibonacciHeapNode<T> *node = dummy_list[section_index]->right;
4.      dummy_list_locks[section_index].unlock();
5.      int position = in_progress_size;
6.      while (node != nullptr) {
7.          if (node->state == UNMARKED) {
8.              in_progress[position] = node;
9.              int i = position - 1;
10.             for (;i >= 0
11.                  && comparator(in_progress[i+1]->value,in_progress[i]-
>value); i--) {
12.                 FibonacciHeapNode<T> * temp    = in_progress[i];
13.                 in_progress[i]                 = in_progress[i + 1];
14.                 in_progress[i + 1]             = temp;
15.             }
16.             if (position < PROMISING_LIST_SZ * DUMMY_MAX_SZ) position++;
17.         }
18.         node = node->right;
19.     }
20.     in_progress_size = position;
21. }
```

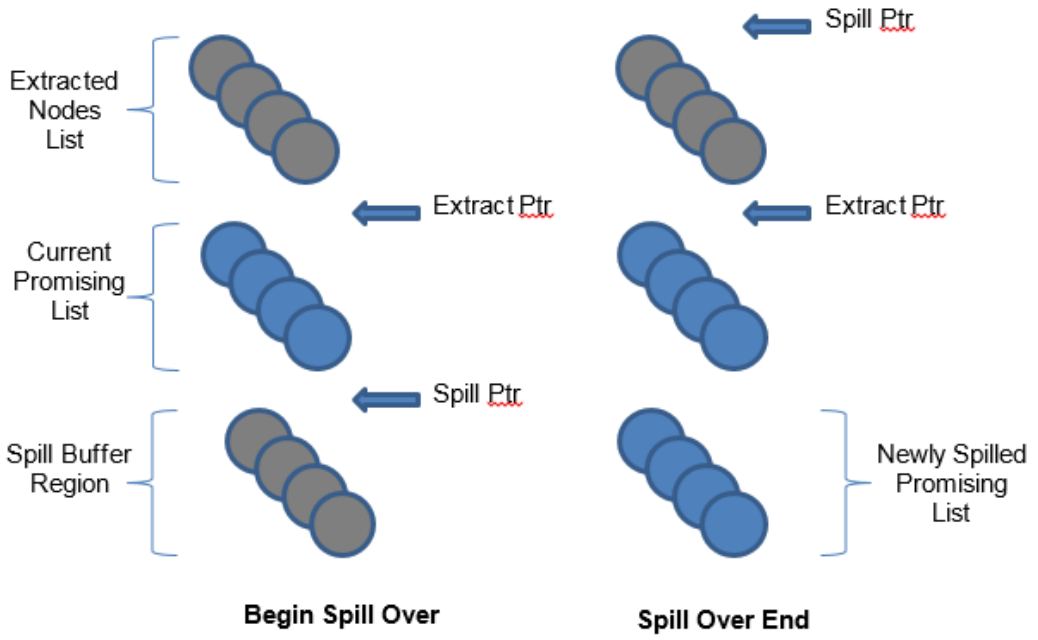


Figure 9. Spill Over Operation in Fine-Grained Fibonacci Heap

Consolidation operation. There are two conditions under which consolidation operation will be performed. First, when the promising list has no elements and a thread request for the minimum element. During this process, the consolidation operation will be performed, and the promising minimum elements are filled in the promising list. The thread requesting extract-min then retries its extract operation. Second, consolidation is performed after the extract operation. A thread performing consolidate operation randomly chooses a section (between two dummy nodes). If the section is not already in consolidation process by other thread, the thread locks the section and walks through the nodes. The consolidation process merges trees of the same degree to reduce the number of trees in the list.

In this paper, most of the logic in the consolidate operation is similar to Huang and Weihl (1991) except that the promising list is not updated during this operation. It moves through each

Fibonacci heap section attempting to lock the consolidation lock without blocking on an already acquired lock. This is because of the following reasons.

- If some other thread is consolidating then waiting on this section does not make sense, as not much change is expected when the thread waits and attempts to consolidate on the recently consolidated section as the degree of all the nodes in that will be almost different for a recently consolidated section.
- Similarly, when some thread is doing spill-over, all the values are being checked in that list for filling in next set of values in the promising list, waiting to consolidate it isn't essential. This will eventually get consolidated in the next consolidate cycle. After a thread performing consolidate operation acquires a lock, it goes through the logic similar to the sequential consolidate to merge the nodes. One difference though is that if the consolidate sees a dead node it attaches its child list in its place.

Algorithm 5

Fine-Grained Synchronized Fibonacci Heap Consolidate Operation

```
1.  void consolidate() {
2.      int section_index = -1;
3.      do {
4.          section_index++;
5.          FibonacciHeapNode<T> * fibonacci_list[DEGREE_LIMIT];
6.          for (int i = 0; i < DEGREE_LIMIT; i++)
7.              fibonacci_list[i] = nullptr;
8.
9.          if (consolidate_locks[section_index].try_lock()) {
10.             FibonacciHeapNode<T> * head = dummy_list[section_index];
11.             FibonacciHeapNode<T> * curr = head;
12.
13.             dummy_list_locks[section_index].lock();
14.             curr = head->right;
15.             dummy_list_locks[section_index].unlock();
16.
17.             if (curr != nullptr) {
18.                 do {
19.                     if (curr-> state == UNMARKED) {
20.                         if (fibonacci_list[curr-> degree] == nullptr) {
21.                             fibonacci_list[curr-> degree] = curr;
22.                         } else {
23.                             FibonacciHeapNode<T> * current_in_list = nullptr;
24.                             do {
25.                                 current_in_list = fibonacci_list[curr-
> degree];
26.                                 fibonacci_list[curr->degree] = nullptr;
27.                                 curr = link(curr, current_in_list);
```

```
28.                             } while (fibonacci_list[curr-
>degree] != nullptr);
29.                             fibonacci_list[curr-> degree] = curr;
30.                         }
31.                     } else if (curr->state == DEAD) {
32.                         remove_and_merge_child(curr);
33.                         curr = curr->left;
34.                     }
35.                     if (curr != nullptr) curr = curr->right;
36.                 } while (curr != nullptr);
37.             }
38.             consolidate_locks[section_index].unlock();
39.         }
40.     } while (section_index < DUMMY_MAX_SZ - 1);
41. }
```

Chapter V: Experiments and Results

Various experiments were performed to compare the behavior of coarse-grained synchronized Fibonacci Heap with fine-grained synchronized Fibonacci Heap. It is a difficult challenge in multi-threaded application to design experiments where results can be comparable to one another. The random nature of the timing of scheduling and execution for different threads make any two executions nearly impossible to match. So, the approach followed here to experiment with some degree of comparable results was to make sure that same set of operations were queued in the system in the same order. There was no restriction on which threads would pick on what operation mainly because of the amortized nature of the algorithm; i.e. all long running operations might be scheduled to run in the same thread.

Experiments were chosen with 10K, 100K, and 1M operations as variations of workload chosen at random but using the same seed for random number generation for execution in each of coarse-grained and fine-grained synchronized Fibonacci Heap. The following table summarizes the results collected for various settings of a number of operations and number of executing threads. Table 3 compares CPU utilization and execution time results for the varied number of operations and number of threads settings for coarse-grained vs. fine-grained synchronized Fibonacci Heap.

Table 3

Comparison of CPU Utilization and Execution Time between Coarse-Grained and Fine-Grained Synchronized Fibonacci Heap

Operations	Threads	CPU Utilization (%)		Execution Time (sec)	
		Coarse Grained	*Fine Grained*	*Coarse Grained*	*Fine Grained*
10K	2	150	177	0.05	0.02
	4	104	266	0.04	0.02
	8	133	336	0.05	0.02
	16	128	352	0.05	0.02
100K	2	152	187	0.55	0.28
	4	121	311	0.45	0.25
	8	113	370	0.44	0.24
	16	109	511	0.43	0.25
1M	2	153	197	6.08	10.47
	4	135	381	5.37	6.44
	8	126	558	5.4	5.14
	16	125	604	5.81	5.1

The following sections explain various metrics of Table 3 in further detail.

CPU Utilization

CPU utilization metric denotes the non-idle time, i.e., time CPU was not running the idle thread. CPU utilization in a parallel execution is usually a strong indicator that CPU is constantly executing instructions and not staying idle waiting for other activities like wait on acquiring the lock. Ideally, multiple threads in a system with multiple physical cores can be scheduled independently in different cores and thus utilization of CPU would increase with increasing

number of threads as long as there is enough independent processing unit available for those threads.

Based on the experiments executed on an 8-core system, Figure 10 shows that for coarse-grained synchronized Fibonacci Heap, increasing the number of threads does not have much effect on the CPU utilization. In fact, on multiple workloads, the performance degrades with increasing number of threads. While on the other hand, fine-grained synchronized Fibonacci Heap seems to much better use the available cores as the threads increase. Also, with increasing workload, the CPU utilization is increasing which is likely because there is more work being done per thread. The increasing trend seems to flatten while going from 8 to 16 threads, but that can be attributed to the fact that processing resources are shared in executing 16 threads on an 8-core system.

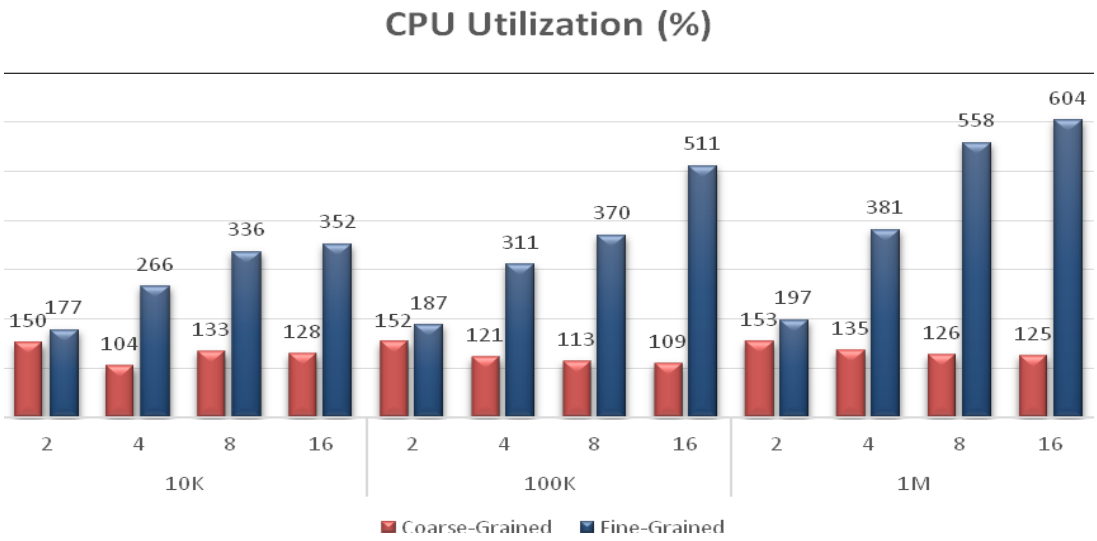


Figure 10. Comparison of CPU Utilization (%) for Coarse-Grained and Fine-Grained Synchronized Fibonacci Heap Implementations Executing 10K, 100K, and 1M Operations in 2, 4, 8, and 16 Threads Settings

Strong Scaling

Strong scaling refers to how the execution time varies with increasing number of processing units for fixed problem size. This is an indicator if the system can reduce the execution time in proportion to the amount of resource added to the system. Ideally, the execution time should decrease linearly with increasing processing unit. To compare the strong scaling aspect of coarse-grained synchronized Fibonacci Heap with the fine-grained Fibonacci Heap, 1M operations (constant problem size) were performed on the heaps for various numbers of threads on an 8-core system.

Figure 11 shows that coarse-grained synchronized Fibonacci Heap does not scale at all with increasing number of threads. Moreover, with a system with 8 physical cores, coarse-grained synchronized Fibonacci Heap performed worse with 8 threads and beyond. However, it is not the case for fine-grained synchronized Fibonacci Heap. Even though the scaling here isn't

perfectly linear with respect to increasing threads, it can be observed that the fine-grained synchronized Fibonacci Heap is scaling much better than coarse-grained synchronized Fibonacci Heap. Although the absolute execution time for 2-threads is much worse for fine-grained synchronization, the strong scaling aspect for it is much more desirable as it has the capability to add more cores (and threads) to reduce the execution time in total. The flat lines after 8 threads are again attributed to the fact that since this experiment was done on an 8-core system, after 8 threads the processing resource is shared amongst threads.

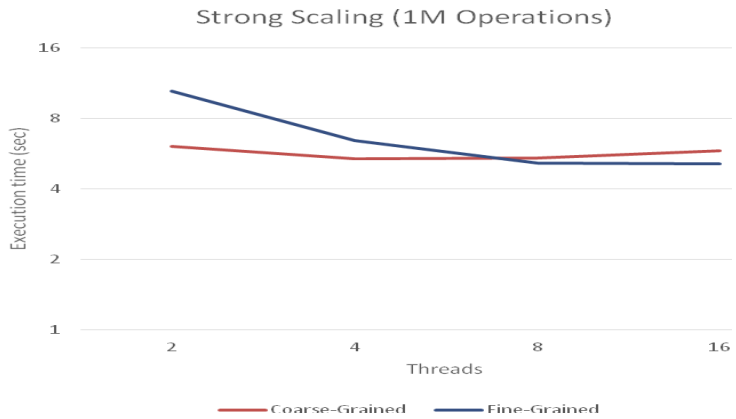


Figure 11. Comparison of Strong Scaling Results for Coarse-Grained and Fine-Grained Synchronized Fibonacci Heap for 1M Operations. The x-axis represents the number of threads (processing unit) and the y-axis represents execution time in seconds. Fine-grained implementation scales are better here.

Weak Scaling

Weak scaling refers to how the execution time varies over the number of processing units for the fixed amount of work per processing unit. Ideally, the execution time should remain constant because each processing unit has the same amount of work to do. The weak scaling describes whether the overhead in the parallel execution varies faster or slower than the amount of work.

As shown in Figure 12, the experiment to compare weak scaling for two implementations of Fibonacci Heap was done by assigning the workload of 100K operations per threads in four different operations and threads settings. It is observed that the thread overhead grows almost in the same manner for both the implementations for increasing number of a workload in increased concurrency settings. This can potentially be attributed to the distribution of various operations amongst threads in different threads settings which is a difficult thing to control.

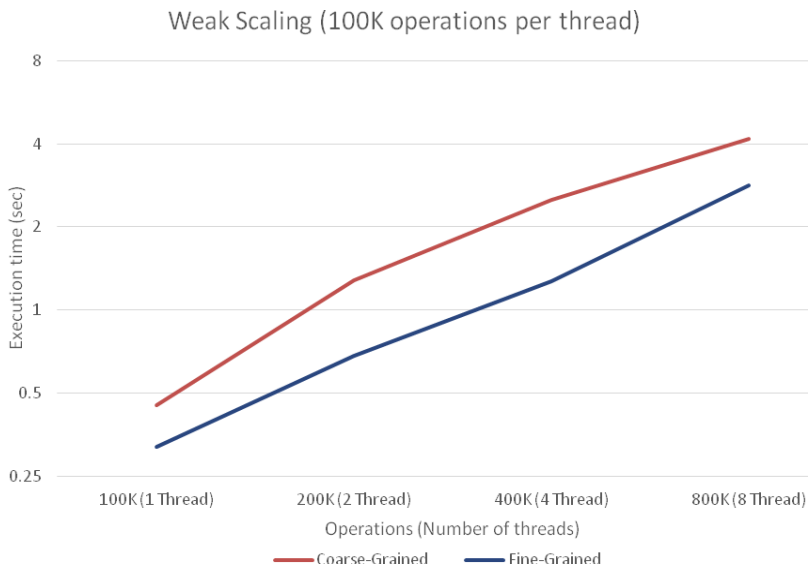


Figure 12. Comparison of Weak Scaling Results for Coarse-Grained and Fine-Grained Synchronized Fibonacci Heap for 100K Operations Per Thread. The x-axis represents a number of threads (processing unit) with total operations performed and the y-axis represents execution time in seconds. Both implementations show a similar trend in parallel overhead.

Chapter VI: Conclusions and Future Works

Conclusions

Based on various experiments conducted and the metrics evaluated, it can be seen that the fine-grained synchronized Fibonacci Heap operations scale much stronger than a coarse-grained synchronized Fibonacci Heap. This means that increasing the resources can increase the performance of fine-grained synchronized Fibonacci Heap. However, the strong scaling aspect of fine-grained synchronized Fibonacci Heap is more advantageous when the workloads are huge, and system includes many processing units. As evident from the strong scaling metric for the coarse-grained Fibonacci Heap, it is only an option for a safe implementation that produces an outcome which is easy to reason about. But, with a certain degree of relaxed semantics, strong scaling can be achieved by putting more careful and thoughtful consideration about reducing the critical sections in operations of such data structure. The proposed fine-grained Fibonacci Heap data structure is one such example.

Future Works

Even though this paper provides a more scalable Fibonacci heap implementation using fine-grained implementation, there are various further improvements and optimization possibilities that can be looked in the future. Some of those are discussed below.

- For insertion purpose, this paper proposes a way to distribute the insertions points by distributing the data structure into multiple root lists each of which has blocking insertions to some degree even though thread does not wait on any locks for more than the lock attempt period. In the future, the insert operation can further be scaled

by using a lock-free queue for each of those multiple root lists (Laden-Mozes & Shavit, 2004).

- The current implementation of fine-grained implementation only implements extract min and insert operation. The future version of this implementation can add other operations thus providing a complete implementation.
- Distributed and parallel (Hybrid) priority queues can be an important data structure for various frameworks (e.g., Pearce, Gokhale, & Amato, 2014). So, the future implementation can include message passing to deploy in such systems out of the box.
- Currently, due to resource constraints, the scalability tests for this implementation is only done with 16 threads. Tests and robustness for scalability can further be investigated on resources with more performance capability and capacity.

References

Blaise, B. (2017). *POSIX threads programming*. Livermore, CA: Lawrence Livermore National Laboratory. Retrieved from https://computing.llnl.gov/tutorials/pthreads/

Boyapati, C., & Rangan, C. P. (1995). *Relaxed fibonacci heaps: An alternative to fibonacci heaps with worst-case rather than amortized time bounds* (Technical Report TRTSC-95-07). Chennai, Tamil Nadu: Indian Institute of Technology, Madras.

Cao, G., & Singhal, M. (1998). On the impossibility of min-process non-blocking checkpointing and an efficient checkpointing algorithm for mobile computing systems. In *Parallel Processing, 1998. Proceedings. 1998 International Conference on* (pp. 37-44). IEEE.

Das, S. K., & Pinotti, M. C. (2000). Parallel priority queues based on binomial heaps. *Parallel Computing, 26*(11), 1411-1428.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM), 34*(3), 596-615.

Huang, Q., & Weihl, W. W. (1991). An evaluation of concurrent priority queue algorithms. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on* (pp. 518-525). IEEE.

Jenkov, J. (2015). *A non-blocking algorithm*. Retrieved from http://tutorials.jenkov.com/javaconcurrency/non-blocking-algorithms.html

Laden-Mozes E., & Shavit N. (2004). An optimistic approach to lock-free fifo queues. In R. Guerraoui (Ed.), *Distributed Computing. DISC 2004. Lecture Notes in Computer Science*, 3274. Springer, Berlin, Heidelberg.

Pearce, R., Gokhale, M., & Amato, N. M. (2014). Faster parallel traversal of scale-free graphs at extreme scale with vertex delegates. In *Proceedings Supercomputing, New Orleans, LA*.

Shavit, N., & Zemach, A. (1999). Scalable concurrent priority queue algorithms. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing* (pp. 113-122).

Tsyrklevich, E., & Bennet Y. (2003). *Dynamic detection and prevention of race conditions in file accesses* (Doctoral Dissertation, University of California, San Diego).

Wayne, K. (2007). *Fibonacci heaps*. Retrieved from https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf

Appendix

Sequential Fibonacci Heap Data Structure

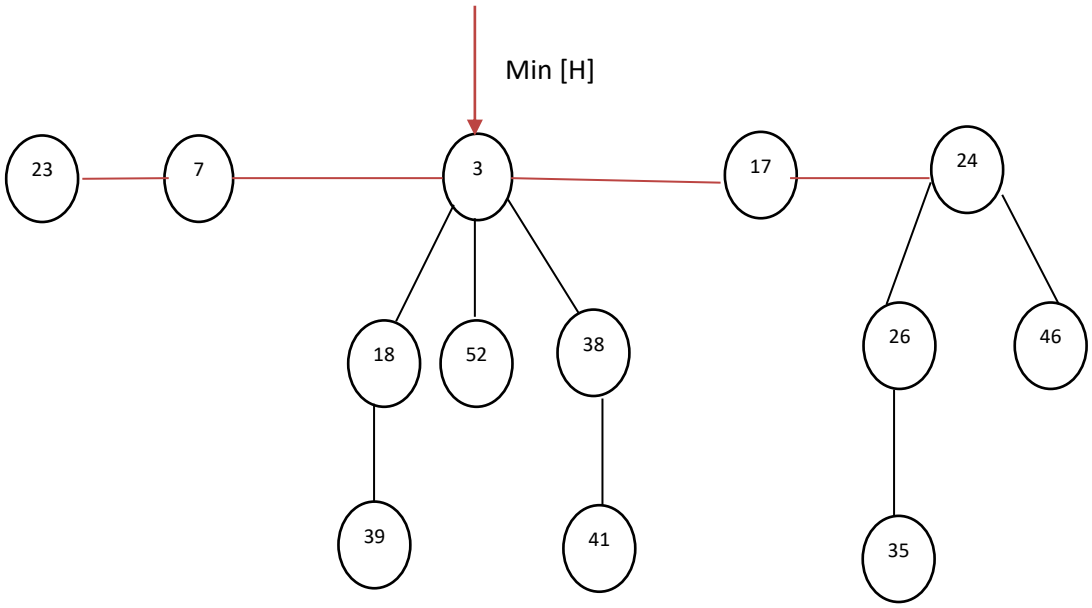


Figure A.1. A Fibonacci heap consisting of five heap-ordered trees and 14 nodes. The red line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this Fibonacci heap is 5 + 2 * 3 = 11

Each node x contains a pointer *p[x]* to its parent and a pointer *child[x]* to any one of its children.

The children of x are linked together in a circular, doubly linked list, which is referred as the *child list of x*.

Each child y in a child list has pointers *left[y]* and *right[y]* that point to y's left and right siblings, respectively. If node y is an only child, then *left[y] = right[y] = y*. *degree[x]* is the number of children in the child list of node x. *mark[x]*, Boolean-valued field indicates whether node x has lost a child since the last time x was made the child of another node.

Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. *min[H]* is called the minimum node of the Fibonacci heap containing *a minimum key*. If a Fibonacci heap H is empty, then *min[H] = NIL*.

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called *the root list* of the Fibonacci heap. The pointer *min[H]* thus points to the node in the root list whose key is minimum. *n[H]* is the number of nodes currently in Fibonacci heap H.

If the number of trees in the root list of H is indicated by *t(H)* and the number of marked nodes in H is indicated by *m(H),* The *potential of Fibonacci heap H* is then defined by,

(H) = t(H) + 2m(H)

Inserting a Node

The following algorithm inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $key[x]$ has already been filled in.

FIB-HEAP-INSERT (H, x)

$degree[x] \leftarrow 0$

$p[x] \leftarrow$ NIL

$child[x] \leftarrow$ NIL

$left[x] \leftarrow x$

$right[x] \leftarrow x$

$mark[x] \leftarrow$ FALSE

concatenate the root list containing x with root list H

if min[H] = NIL or $key[x] < key[min[H]]$

then $min[H] \leftarrow x$

$n[H] \leftarrow n[H] + 1$

Extracting the Minimum Node

The following algorithm extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary CONSOLIDATE operation, which is presented below.

FIB-HEAP-EXTRACT-MIN (*H*)

$z \leftarrow min[H]$

if $z \neq$ NIL

then for each child *x* of *z*

do add *x* to the root list of *H*

$p[x] \leftarrow$ NIL

remove *z* from the root list of *H*

if $z = right[z]$

then $min[H] \leftarrow$ NIL

else $min[H] \leftarrow right[z]$

CONSOLIDATE(*H*)

$n[H] \leftarrow n[H] - 1$

return *z*

Consolidation

In consolidation, the number of root lists (the number of trees) in the Fibonacci heap is reduced; this is performed by the call CONSOLIDATE(*H*). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value.

Find two nodes *x* and *y* in the root list with the same degree, where $key[x] \leq key[y]$.

Link y to *x*: remove *y* from the root list, and make *y* a child of *x*. This operation is performed by the FIB-HEAP-LINK algorithm. The field *degree*[*x*] is incremented, and the mark on *y*, if any, is cleared.

The CONSOLIDATE operation uses an auxiliary array *A* [0... *D*(*n*[*H*])]. if *A*[*i*] = *y*, then *y* is currently a root with *degree*[*y*] = *i*.

```
CONSOLIDATE (H)
for i ← 0 to D(n[H])
   do A[i] ← NIL
for each node w in the root list of H
   do x ← w
     d ← degree[x]
     while A[d] ≠ NIL
       do y ← A[d]
         if key[x] > key[y]
           then exchange x ↔ y
          FIB-HEAP-LINK (H,Y,x)
          A[d] ← NIL
```

$d \leftarrow d + 1$

$A[d] \leftarrow x$

$min[H] \leftarrow$ NIL

for $i \leftarrow 0$ to $D(n[H])$

do if $A[i] \neq$ NIL

then add $A[i]$ to the root list of H

if $min[H]$ = NIL or $key[A[i]] < key[min[H]]$

then $min[H] \leftarrow A[i]$

FIB-HEAP-LINK (H, y, x)

remove y from the root list of H

make y a child of x, incrementing $degree[x]$

$mark[y] \leftarrow$ FALSE