ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑ-ΤΩΝ

Μελέτη και Αξιολόγηση Τεχνικών Παραλληλοποίησης Δομών Δεδομένων και Αλγορίθμων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χριστίνα Χρ. Γιαννούλα

Επιβλέπων: Γεώργιος Γκούμας
Λέκτορας Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14η Ιουλίου 2016.

....	....	....
Γ. Γκούμας	Ν. Κοζύρης	Κ. Σαγώνας
Λέκτορας Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016.

....
Χριστίνα Χρ. Γιαννούλα
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Χριστίνα Γιαννούλα, 2016. Εθνικό Μετσόβιο Πολυτεχνείο.
Με επιφύλαξη κάθε δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ' ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπο την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τη συγγραφέα.
Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τη συγγραφέα και δεν πρέπει να ερμηνευτεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Στις μέρες μας, οι πολυπύρηνοι επεξεργαστές έχουν γίνει η κυρίαρχη πλατφόρμα υπολογισμών και έχουν εισαχθεί σε πολλά προγραμματιστικά περιβάλλοντα. Ο παράλληλος προγραμματισμος δεν αφορά πλέον μόνο επιστημονικές εφαρμογές που τρέχουν σε υπερυπολογιστές, αλλα καλύπτει επίσης ένα μεγάλο φάσμα εφαρμογών για προσωπικούς υπολογιστές. Ένα από τα πιο δύσκολα προβλήματα στα συστήματα παράλληλης επεξεργασίας είναι η ανάπτυξη παράλληλου λογισμικού το οποίο κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν μετά από έναν αριθμό επεξεργαστών εξαιτίας του αυξημένου κόστους επικοινωνίας. Προκειμένου να αξιοποιηθούν οι διαθέσιμες αρχιτεκτονικές, οι βασικές δομές δεδομένων και οι σειριακοί αλγόριθμοι πρέπει να επανασχεδιασθούν. Το πρώτο μέρος αυτής της διπλωματικής αφορά τις παράλληλες δομές δεδομένων, με ιδιαίτερη έμφαση στα δυαδικά δέντρα αναζήτησης, εξετάζοντας τον τρόπο συγχρονισμού τους, τα ιδιαίτερα χαρακτηριστικά τους και την κλιμακωσιμότητα που προσφέρουν. Στο δεύτερο μέρος της διπλωματικής παρουσιάζεται μια παραλληλοποίηση του αλγορίθμου του Dijkstra που είναι ένας σειριακός αλγόριθμος. Αυτή η υλοποίηση χρησιμοποιεί Transactional Memory, για να συντονίσει αποτελεσματικά τις ταυτόχρονες προσβάσεις των νημάτων στις κοινές δομές δεδομένων και την έννοια των Helper Threads, για να εξάγει παραλληλισμό. Η αξιολόγηση του αλγορίθμου γίνεται σε ένα σύστημα που υποστηρίζει Hardware Transactional Memory.

Λέξεις-Κλειδία: παράλληλες δομές δεδομένων, δυαδικά δέντρα αναζήτησης, κλιμακωσιμότητα, παράλληλος προγραμματισμός, αλγόριθμος του Dijkstra, Helper Threads, Hardware Transactional Memory

Abstract

Nowadays, multicore processors have become the dominant computing platform and are being used by many programming environments. Parallel programming is no longer about scientific applications run in supercomputers, but covers a wider range of applications on personal computers, too. The most difficult problem is to develop parallel software that scales efficiently. Several applications do not scale further than a number of processors due to communication overhead. To exploit the available architectures basic data structures and sequential algorithms must be redesigned. In the first part of this thesis we study concurrent data structures, particularly focusing on concurrent binary search trees, with respect to the way they are synchronized, their special characteristics and the scalability they provide. The second part of this thesis presents a parallelization of the inherently serial Dijkstra's algorithm. This implementation employs Transactional Memory to efficiently orchestrate the concurrent thread's accesses to shared data structures and the concept of Helper Threads to extract parallelism. We evaluate the execution of the algorithm on a system that supports Hardware Transactional Memory.

Keywords: Concurrent Data Structures, Binary Search Trees, scalability, parallel programming, Dijkstra's algorithm, Helper Threads, Hardware Transactional Memory

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Λέκτορα Ε.Μ.Π. Γεώργιου Γκούμα.

Θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Γεώργιο Γκούμα, για την εποπτεία κατά την εκπόνηση της εργασίας μου, τις γνώσεις που μου προσέφερε με τη διδασκαλία του και την ευκαιρία που μου έδωσε να δουλέψω στο εργαστήριο.

Θα ήθελα επίσης να ευχαριστήσω τον καθηγητή της σχολής κ. Νεκτάριο Κοζύρη για την έμπνευση που μου προσέφερε με τη διδασκαλία του και να τον συγχαρώ για υψηλό επίπεδο σπουδών των μαθημάτων διδασκαλίας του και του εργαστηρίου.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Υποψήφιο Διδάκτωρ Δημήτριο Σιακαβάρα για τη συνεχή καθοδήγησή του κατά τη διάρκεια εκπόνησης αυτής της διπλωματικής εργασίας, καθώς και για την ενθάρρυνσή του, την υπομονή του και το χρόνο που αφιέρωσε. Χωρίς τη συμβολή του η ολοκλήρωση αυτής της εργασίας δε θα ήταν εφικτή.

Επιπλέον, επιβάλλεται να ευχαριστήσω τους φίλους μου και τους συμφοιτητές μου για την πολύτιμη βοήθειά τους σε επιστημονικό και προσωπικό επίπεδο.

Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένειά μου, τους γονείς μου και την αδερφή μου για την αγάπη, την αμέριστη υποστήριξή τους όλα αυτά τα χρόνια και την εμπιστοσύνη τους σε κάθε μου επιλογή.

Χριστίνα Γιαννούλα

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Overview

In 1965, Moore predicted that the number of transistors in an integrated circuit will double approximately every two years, resulting to exponential increase in raw computer power. However, the increasing number of transistors per processor set physical limitations. The dense chips use more electric power and generate more heat limiting the evolution in processors.

As Moore's law was used in the semiconductor industry, more and more transistors integrated in the same silicon chip and the chip performance doubled every 18 months. These high performance microprocessors named as multicore processors. Nowadays, multicore processors is a characteristic of supercomputers, distributed systems and personal computers. Increasing the number of processors can result to high-performance computing. However, multiprocessors systems cause communication and synchronization problems and the scalability of multiprocessors systems remains a challenge as the number of processors increase.

At the same time, in order to take full advantage of these available hardware resources, computer industry developed new architectures techniques. For example, the use of deeper pipelines superscalar architectures increased the operation throughput. As a result, conventional architectures replaced by parallel architectures with a view to maximize performance of applications.

The multicore systems constitute a solution to computation-intensive applications. Scientific applications such as astrophysical and cosmological simulations, weather forecasting, applications in industry and other sectors, user applications such as search engines and web servers are computationally demanding. Contemporary serial algorithms, even if optimized, cannot achieve an optimum performance when executed on multiprocessors architectures. As a consequence, sequential algorithms must be redesigned such that run in parallel and exploit the available architectures.

Parallel programming demands synchronization among parallel processes or threads in order to avoid conflicts and race conditions. Synchronization techniques are implemented in both software and hardware. However, communication and synchronization between different subtasks are some of the greatest obstacles to getting good parallel performance. The challenge is to create hardware and software that will make it easy to develop parallel processing programs such that to achieve good performance and scalability as the number of cores per chip increase.

1.2 Parallel Architecture and Parallel Programming

1.2.1 Memory coherence

Modern processing systems consist of multiple level of cache memory, which reduce the cost of multiple references in the same memory location as there exist copies of recently uses memory locations close to the processor such that to have a quick and low cost access to them. Though, the existence of copies of the same memory location in different levels of cache memory cause many problems. For example, in a single processor system when a device uses Direct Memory Access (DMA) to read data from main memory, any changes to that memory residing in processor caches must be flushed out first.

In multiprocessor systems, it is possible to have many copies of the same memory location in multiple caches when several processors access to this memory location simultaneously. Provided none of the processors changes the data in this location (read only), they can share it without any problem. But as soon as one updates the location, all the other must be notified of the update, otherwise they might work on an out-of-date copy, that resides in their local cache. As a consequence, it must be followed a scheme that ensures that none of the processors is accessing a stale value of a memory location. This scheme is known as memory coherence protocol. A practical multiprocessor invalidate protocol, the most widely used, which attempts to minimize bus usage is the MESI protocol.

In MESI protocol, any cache line can be marked with one of the following states:

Modified: The cache line is present only in the current cache (the only cache copy), and is dirty, it has been modified from the value in main memory. The cache is required to write data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state.

Exclusive: The cache line is present only in the current cache (the only cache copy) and matches the main memory. In a read request from another processor, the cache line will change to the Shared state and in a write request from that processor will change to the Modified state.

Shared: The cache line may be stored in other caches of the machine and matches the main memory.

Invalid: The cache line is not valid.

The MESI protocol uses the above mentioned states for every cache line to permit all processor to have a consistent view on all memory locations. The cache line changes state, according to the state diagram depicted in figure 1.1, as a function of memory access events. An event may be due to local processor activity or due to bus activity.

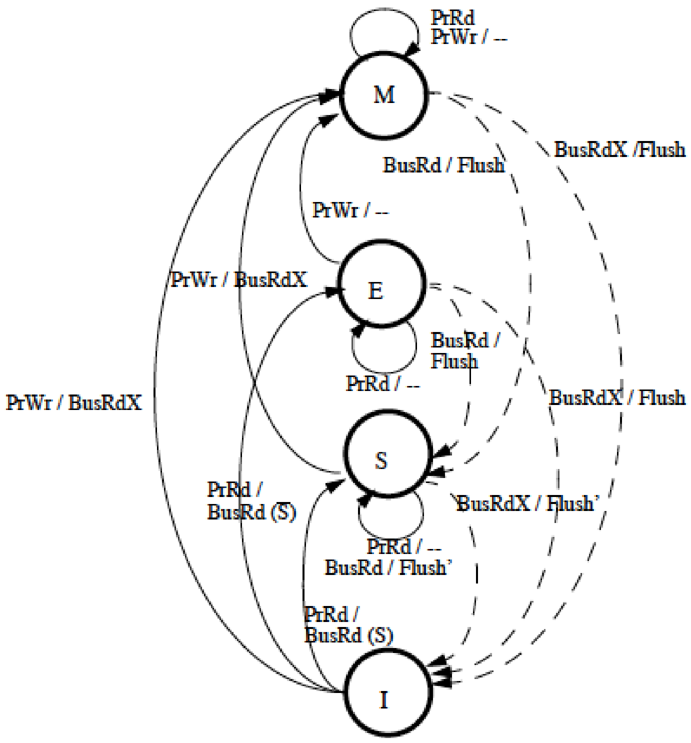


Figure 1.1: MESI state diagram. The transitions are labeled "action observed / action performed".

1.2.2 Memory consistency

As mentioned above, cache coherence means that all the processors see the same value for a particular memory address as they should have if there were no caches in the system. Memory consistency on the other hand, ensures that all the memory instructions appear to execute in the program order, that is consistency refers to the order of accesses to all memory locations. It defines how all the memory instructions in a multiprocessor system will be ordered.

In multiprocessors systems the order between operations of a program is not always guaranteed. Operations issue concurrently by many processors in an order that cannot be determined in advanced. Most real hardware use first-in-first-out (FIFO) write buffers and in this way there is not guaranteed order in parallel programs.

A memory consistent model is a specification of the allowed behaviour of parallel programs executing with shared memory. In a parallel program, unlike to a single-threaded execution, multiple correct behaviours are usually allowed. There are different memory consistencies such as sequential consistency, relaxed consistency and might have an impact on the final program result.

1.2.3 Amdahl's law

Sequential algorithms need to be redesigned in order to run in parallel and exploit the available hardware and processors. In parallel algorithms, large problems are divided into smaller ones, which are then solved in the same time, as the available cores execute concurrently smaller problems. In comparison with sequential algorithms, it is harder to develop parallel algorithms, however the parallel program have better performance and scalability.

Amdahl's law [1] is a theoretical formula that gives the maximum theoretical speedup achieved. Speedup is a measure of how many times the parallel program is faster than the best sequential algorithm. If T_s is the runtime of the fastest sequential program and T_p is the runtime of the parallel version of the program on p processors, the speedup is defined as:

$$Speedup(S) = \frac{T_s}{T_p}$$

Typically, the speedup S is related with the number of processors p in the inequality: $S \leq p$. If $S = p$, the speedup is *linear*.

Amdahl's law show that the theoretical speedup increases by improving a part of the total program. Consider f the fraction of the problem, which cannot be parallelized and must be executed sequentially, then the runtime of the parallel program is:

$$T_p = fT_s + \frac{(1-f)T_s}{p}$$

and the expression for the speedup is:

$$\text{Total Speedup } S = \frac{1}{f + \frac{1-f}{p}}$$

Amdahl's law is often used to predict the theoretical speedup when using multiple processors. As the number of processors p goes to infinity, the total speedup goes to $1/f$. The theoretical speedup is limited by the part of the program that cannot be executed in parallel. If the parallel part of a program is relatively small, its speedup would be equally small. For instance, the fraction is $f = 90\%$, the parallel program can be 10 times faster in the best case than the serial program, independently of the number of processors. Figure 1.2 depicts the total speedup in parallel executions of different sequential fractions f and number of processors p. According to this figure, the total speedup of a program is bounded by the sequential part of the program and using more processors does not increase the speedup in all cases. Parallelizing more and more of the sequential program is the solution to maximize the performance.

Scalability, another measure for performance, can refer to the capability of a system or a program to increase its performance when more processors are added. Assumed that the program have a constant size, the runtime of the program is expected to scale up as the number of processors increases. However, there are many factors that limit the scalability of a program. First of all, the program has to be divided into small equal pieces, each of

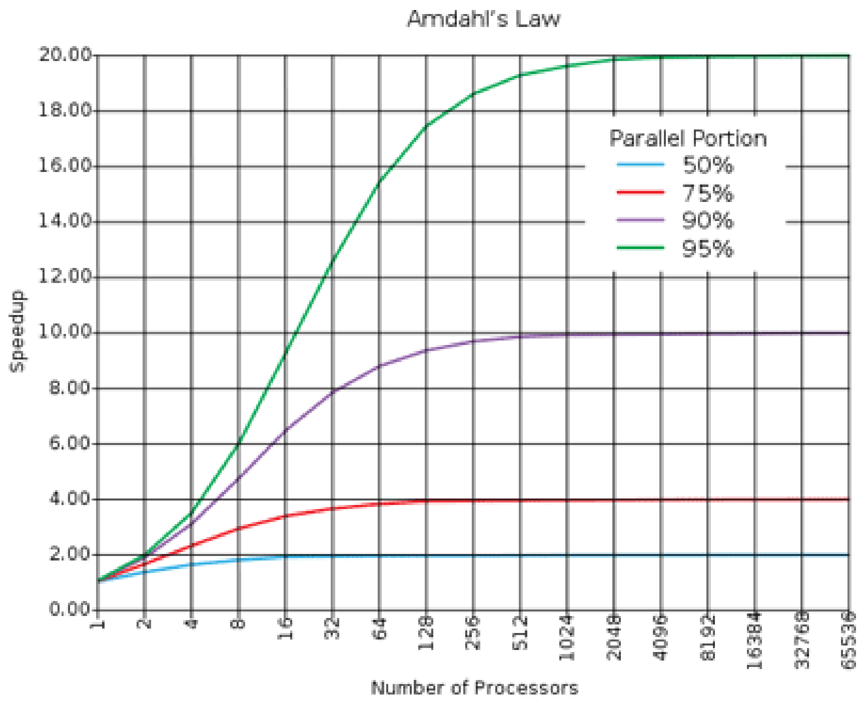


Figure 1.2: Total speedup of a parallel program as parallel fraction and number of processors change.

them is executed in a separate processor. If the pieces are not equal, the processors would wait for other ones with the larger pieces to terminate. Furthermore, the scalability can be limited due to the time spent in communication and synchronization among processors. If this communication and synchronization time is significant compared with the total time, the program does not scale up when the number of processors increases. To conclude, load balancing and time spent for synchronization and communication restrict significantly the scalability of a program and must be taken into consideration in parallel programming.

1.3 Parallel Architectures

Flynn's taxonomy [2] distinguishes computer architectures, according to the level of parallelism they employ to process instructions and data streams. There are four categories:

- **SISD:** Single Instruction, Single Data
  A sequential computer. Sequential computers are incapable of performing parallel operations.
- **SIMD:** Single Instruction, Multiple Data

A parallel computer with a single instruction stream, which performs the same instruction on multiple data.

- **MISD:** Multiple Instruction, Single Data
  Multiple processing units perform tasks-instructions on the same data. MISD is an uncommon and non-commercial architecture.

- **MIMD:** Multiple Instruction, Multiple Data
  A parallel computer, in which each processor executes independent instruction streams on independent data streams. This architecture is the most common and widely used form of parallel architecture. Two examples of this architecture are clusters and systems of multicore processors.

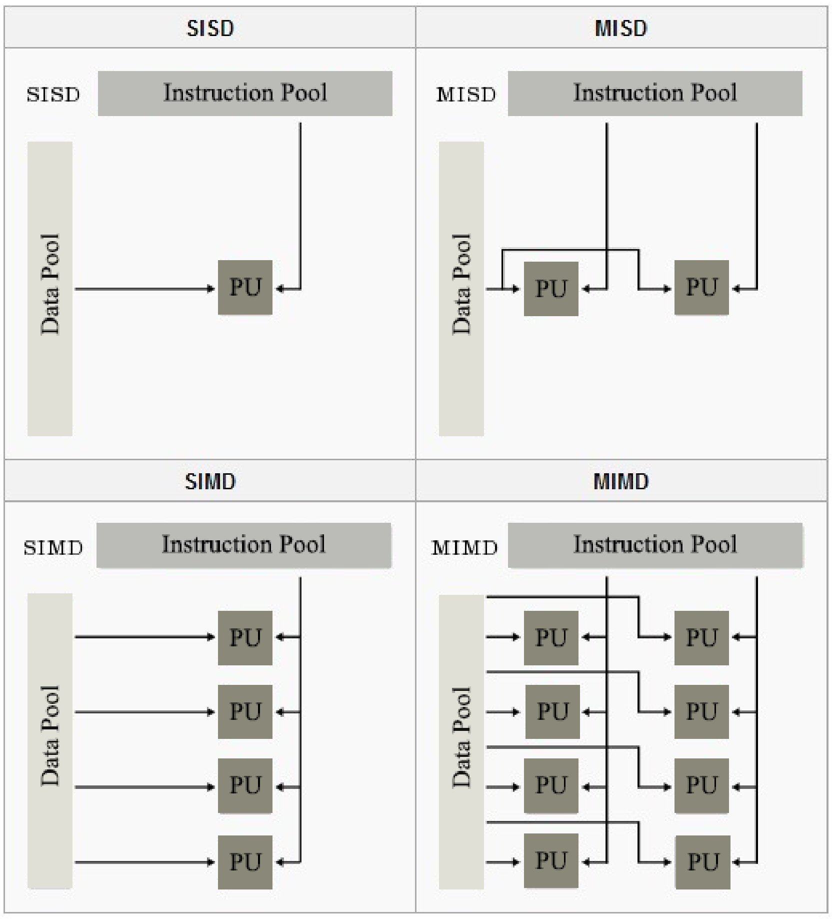


Figure 1.3: Flynn's taxonomy.

As mentioned above, MIMD multiprocessing architecture is the most common parallel architecture and suitable for a wide variety of tasks. MIMD architectures can be categorized based on their memory organization in three categories that will be further analyzed in the next sections: shared memory architectures, distributed memory architectures and hybrid architectures.

1.3.1 Shared memory architecture

In shared memory architecture, each processor has each own private cache memory hierarchy and all processors share a single physical space, known as global memory. A single system bus interconnects all processors. Such systems can execute independent tasks, which have their own virtual address spaces, even if they share a physical address space. Processors communicate by sharing variables stored in the global memory and can access any memory location via loads and stores. If the access to any memory location takes the same amount of time to all processors, the memory organization is called symmetric multiprocessor (SMP) and can be viewed in figure 1.4.

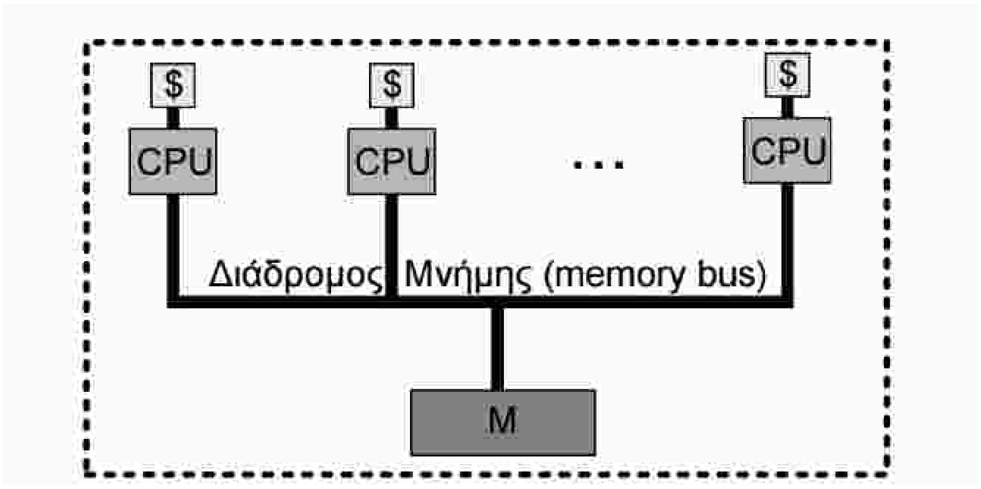


Figure 1.4: Classic organization of a SMP.

Shared memory architecture come in two memory organizations. If the amount of time taken to access any global memory address is equal independently which processor requests the access, the memory organization is called Uniform Memory Access (UMA). If some memory accesses are much faster than the others, depending on which processor requests for which global memory address, the memory organization is called Non-Uniform Memory Access (NUMA). NUMA architectures can have lower latency to nearby memory and higher memory bandwidth.

Processors in shared memory architectures can operate tasks in parallel using the same shared data and race conditions may occur. As a result, processors need to coordinate in order to avoid concurrent accesses to shared data. Thus, synchronization mechanisms like locks and atomic variables, are used in such situations. Furthermore, as analyzed in the previous section, cache coherent protocols like MESI, impose a universal sequence of accesses to the global memory.

The efficient access to all shared data from any processor via simple load and store operations on them make shared memory architectures attractive for parallel programming. However, this memory organization has a single system bus that interconnects all processors and a limited memory bandwidth. As a result, it can be used for no more than 20 or 30 processors because of the limited single bus and the limited memory bandwidth.

1.3.2 Distributed memory architecture

In distributed memory architectures each processor has a local cache hierarchy and a local main memory. The processors are connected in an interconnection network (e.g Ethernet) and are called nodes. They have not shared memory addresses and the only way to communicate each other is via message passing through the interconnection network. The system provides to the programmer routines for send/receive messages to/from any processor. The figure 1.5 depicts the organization of a distributed memory architecture.

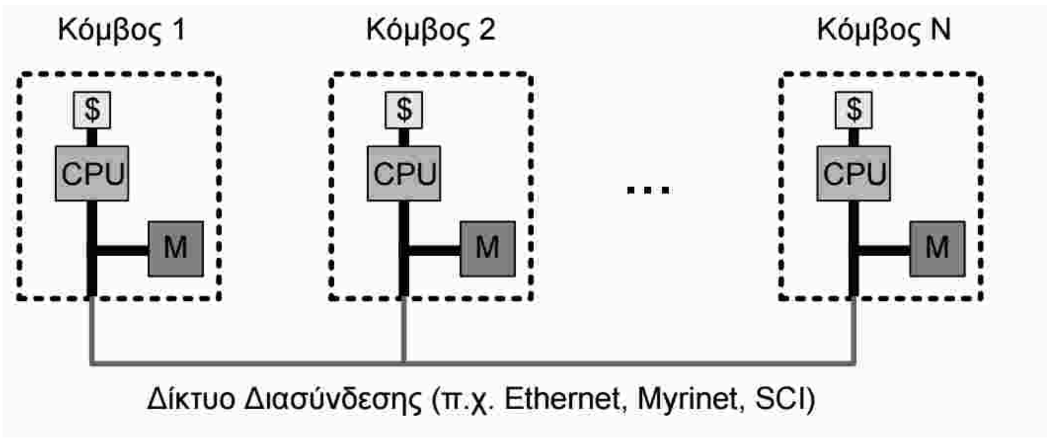


Figure 1.5: Classic organization of a Distributed Memory Architecture.

This architecture is used in clusters that are generally collections of commodity computers that are connected to each other over an I/O interconnect in a network. Each processor has a separate copy of the operating system.

One drawback of clusters is the management cost. The management cost of a cluster with n nodes equals to the management cost of n computers, while the management cost of a multiprocessor with n cores is the same management cost as one single computer. Moreover, another drawback of clusters is the bandwidth and the latency. The processors in a cluster are connected using the I/O interconnection, while the cores in a multiprocessor system are connected via memory interconnect. The memory interconnect has higher bandwidth and lower latency and allow better communication performance.

Programming in a distributed memory architecture is a challenge, since every communication must be identified in advance. Efficient parallelization requires understanding the memory dependencies of the program and an effective distribution of memory in advance in order to eliminate communication between remote processors.

Finally, distributed systems can achieve high scalability because of the absence of shared memory and race conditions. They are constructed of thousand independent nodes that can be dynamically inserted and removed from the network.

1.3.3 Hybrid memory architecture

The hybrid architecture combines the previous two architectures and takes advantages of their benefits. A hybrid system is like a distributed system, in which a symmetric multiprocessor has taken the place of each single processor node. Figure 1.6 depicts the organization of a hybrid architecture. This typical architecture is used in clusters and supercomputers, favors parallel processing within each node and scales up as a distributed memory architecture.

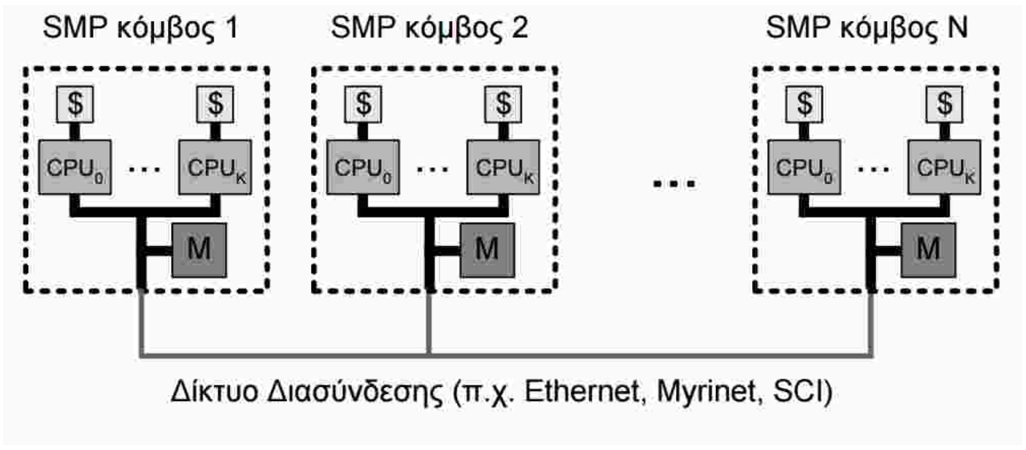


Figure 1.6: Classic organization of a Hybrid Memory Architecture.

1.4 Synchronization

1.4.1 Synchronization definition

In parallel programming, threads or processes [*] need to communicate with each other and execute operations in the same shared data or variables. The operations or the code

[*]We will use the term process to refer to a parallel task, but in parallel programming we can have either processes or threads.

of a process must be executed as if the processes were running in isolation, each with access to its own memory space. However, when processes run in parallel and perform operations on common data structures, they have to be synchronized, otherwise the result will be undefined. The synchronization is extremely important in parallel programming.

Process synchronization is defined as a technique that more concurrent processes do not simultaneously execute some particular segment of the program known as critical section. The critical section is a serialized segment of the program. When one process starts executing the critical section, the other processes should wait until the first completes the critical section. If synchronization techniques are not applied, the values of the variables may be unpredictable and vary depending on the timings of context switches.

A classic problem of synchronization is the Readers-Writers problem, which deals with situations in which many processes try to access the same shared resource at the same time. Some processes may read and some may write, with the constraint that no process may access the share data for either reading or writing, while another process is writing on it. It is only allowed for two or more readers to access the share data concurrently. Another example of how the absence of synchronization among processes or threads leads to inconsistencies can be viewed in listing 1.1. The counter is incremented once by one thread and is decremented once by the other. When the threads execute the critical segment concurrently, the final result of the counter can have either the same initial value or a decreased or increased value.

Listing 1.1: Inconsistencies due to lack of synchronization

```
Thread 1                        Thread 2
var1 = counter;                 var2 = counter;
var1 += 1;                      var2 -= 1;
counter = var1;                 counter = var2;
```

Other than mutual exclusion synchronization can also deals with the following:

- deadlock, is a situation in which many processes are waiting for a shared resource which is being held by another process and there is no progress in the program.
- starvation, which occurs when a process is perpetually denied necessary resources to enter the critical section and the process is forced to wait indefinitely.
- priority inversion, which occurs when a high priority task is in the critical section and it may be interrupted by a medium priority task.
- busy waiting, is a situation in which a process repeatedly checks to determine if it has access to a critical section. This spinning can generate an arbitrary time delay.

Accesses to critical section by different processes are controlled by using synchronization schemes. They can be divided in two categories, blocking synchronization and non-blocking synchronization.

Blocking synchronization

- deadlock-free
  Using mutual exclusion[*], it guarantees that some processes will finish their task in a finite number of steps.
- starvation-free
  Using mutual exclusion, it guarantees that every process will finish their task in a finite number of steps.

Non-blocking synchronization
- lock-free
  It guarantees that some processes will finish their task in a finite number of steps.
- wait-free
  It guarantees that every process will finish their task in a finite number of steps.

Non-blocking schemes allow access by multiple concurrent processes without mutual exclusion. Multiple processes access shared resources and perform operations on them without blocking. The consistency in shared resources is guaranteed using individual operations.

1.4.2 Synchronization techniques

There are different synchronization techniques:

Mutual exclusion

Mutual exclusion is the most usual technique to achieve blocking synchronization. It ensures that two or more concurrent processes are not in their critical section at the same time. Only one will access the critical section of the program at a given time. While one process executes operations in the shared resources, all other should be kept waiting and when that process has finished its work in the shared data, one of the processes waiting will proceed. Mutual exclusion is implemented via mechanisms like semaphores, mutexes and locks. Almost all locks use Test-And-Set (TAS) atomic operation to set a memory location. The process that sets the lock to the LOCKED state is considered to be the lock owner and can proceed to the critical section.

The operation TAS is atomic and only one process can set the memory location at a time. If the shared memory location is in the UNLOCKED state the process can set it as LOCKED, become the lock owner and proceed. Otherwise, if the shared memory location is in the LOCKED state the process continues to loop while checking the state until it becomes UNLOCKED and successfully acquires the lock. This continuously executing TAS on the shared memory location causes heavy bus traffic. In TAS a process sets the memory location of the lock and checks if the previous state of the lock was LOCKED or UNLOCKED in order to proceed to the critical section or wait. According to cache coherence protocols, when a TAS operation sets the memory location (lock), all other copies of lock will be invalidated, including the owner's. This causes heavy cache coherence

[*]It is further analyzed in the next subsection.

protocol traffic. As a result, an improvement of TAS is Test-and-Test-And-Set (TTAS) operation, in which the state of the lock is first read locally and the process tries to set the lock with TAS only if it appears to be in the UNLOCKED state. The process does not write the lock while spinning, but it only reads it locally, so as the cache coherence traffic reduces. This implements a back off mechanism, where a process that found the lock in the LOCKED state will wait some time before checking it again.

Implementing a synchronization with locks is not so easy. Sometimes processes may need to acquire more than one lock in order to access several memory locations in parallel. That scheme is known as fine grained synchronization and a problematic situation in that scheme is deadlock. Deadlock is a situation in which two or more competing processes are each waiting for the other to finish. For example, there are two threads which hold a lock each and each thread tries to acquire the lock held by the other. As a result, none of the two threads has progress and the execution reaches a dead end.

Atomic operations

A problem with mutual exclusion is that if a thread holding a lock is suspended, all other threads are blocked until the suspended thread resumes, as mutual exclusion is a blocking mechanism and is used in blocking algorithms. In order to avoid this problem there are non-blocking algorithms which use atomic operations instead of locking for synchronization. During an atomic operation a processor can simultaneously read a memory location and write on it in the same bus operation. This prevents any other processor from writing or reading memory until the operation is complete. When a process performs an atomic operation the other processes see it as happening instantaneously.

Atomicity implies indivisibility and irreducibility, so an atomic operation must be performed entirely or not performed at all. Moreover, atomicity is a guarantee of isolation from concurrent processes. The system behaves as if each operation occurred instantly. The advantage of atomic operations is that they are quicker that locks, and do not suffer from deadlock and convoying, as they constitute a non-blocking scheme. The disadvantage is that they only perform a limited set of operations, and often there are not enough to synthesize more complicated operations efficiently. However, the programmer should not reject an opportunity to use an atomic operation in place of mutual exclusion and locks.

Correctness of non-blocking algorithms is challenging to prove and it can be done using linearization points. All functions calls have a linearization point at some instant between their invocation and their response. The state of the shared resource in parallel non-blocking algorithms depends from the order by which the linearization points are reached.

Compare And Swap (CAS) is a fundamental atomic operation used to many non-blocking algorithms. It compares the contents of a memory address to a given value and, only if they are the same, changes the contents of that memory address to the given new value. If the value is up-to-date the operation is successful. If not, the value is stale and has been updated in the meantime by another process, so the operation will fail and the current process must restart the operation. To indicate if the value changed, the return result from

CAS is either a simple boolean value or the value read from that memory address (not the value written to it).

Transactional memory

Another non-blocking scheme for synchronization is transactional memory. Transactional memory is a technology of concurrent processes synchronization that simplifies the parallel programming by extracting instruction groups to atomic transactions. The main benefits are that there are not locks and deadlocks, the parallelism level is increased, so performance is boosted as well and it is relatively easy in use for programming. This synchronization scheme will be further analyzed in next chapter.

Chapter 2

Concurrent Search Trees

2.1 Concurrent Data Structures

A data structure is a particular way of storing and organizing data in a computer, such that they can be used efficiently. Data structures provide a means to manage large amounts of data efficiently and they are used widely in large databases and internet indexing services. Different kinds of applications demand different kinds of data structures and as projects grow larger, it is vital the use of more sophisticated data structures. In fact, the overall performance of the application is limited by the performance of the underlying data structure. As a result, using efficient data structures is the key to design efficient algorithms.

As multiprocessor computer architecture became the dominant computing platform, these data structures had to be redesigned in order to provide safe and synchronized access to multiple threads (or processes). Multiple threads can access data simultaneously, because they run on different processors that communicate with one another. Thus, parallel programming introduces many difficulties and concurrent data structures are far more difficult to design than sequential ones, because threads executing concurrently may interleave their steps in many ways. This requires developers to understand new design methodologies. Furthermore, concurrent data structures have to ensure consistency against the effects of any operation and provide safety and liveness properties. Safety properties usually state that something bad never happens, while liveness properties state that something good keeps happening and the data structure keeps progressing and serving requests.

Designing concurrent data structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. According to Amdal's law, explained in previous chapter, the sequentially executed parts of the code constitute the most important restricting factor to achieve the maximum gain from parallelization. The operations on a shared data structure belong to that sequential parts of the code. In addition, concurrent data structures are also a restricting factor of application's scalability. It is vital that the speedup of data structures have to grow while the number of processors increases. These data structures are called scalable. In designing scalable data structures, developers must take care that naive approaches to synchronization can severely undermine scalability.

A second problem in parallel programming and concurrent data structures is memory contention. The results of multiple threads executed in different processors, which demand access to the same locations in memory (same shared data structure), are the overhead in cache coherence traffic and the bus congestion and these two constitute also a restriction in application's performance and scalability. Finally, the attempt to reduce the serial parts of a concurrent data structure and increase the work done in parallel, results to synchronization costs among threads.

2.2 Search Trees

A search tree is a data structure for locating specific values from within a set. It consists of a set of key-value pairs and an interface for accessing and manipulating them. The three main operations of this interface are a lookup operation and two update operations (one to insert and one to delete), as shown in figure 2.1. Various search tree data structures exist, several of which also allow efficient insertion and deletion of elements. In order to reduce search time, search trees have to be reasonably balanced (all the leaves are of comparable depths). In this chapter, we will study three different types of search trees according to their balance: binary search trees, AVL trees and Red-Black trees.

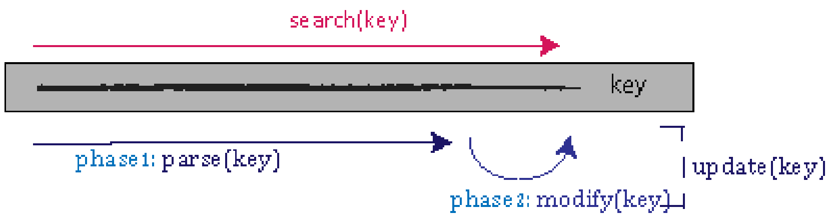


Figure 2.1: Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase.

Search trees are also divided in two categories according to the underlying organization of the key-value pairs in the nodes of the tree. Nodes that have two children are called internal nodes and those that have no children are called external nodes. When the key-value pairs (the useful information) of a search tree are stored only in external nodes (leaves) and the internal nodes of the tree are routing nodes used only as a help for the path to the final external node searched, the search tree is called external tree. When the internal nodes are not only routing nodes, but they also store key-value pairs and are "true" nodes of information, the search tree is called internal tree. The figure 2.2 depicts an external and an internal search tree. We use square shapes to distinguish the leaves, which contain the key-value pairs, from the internal nodes in the external tree. All the other nodes contain only keys and are used for routing to the appropriate leaf.

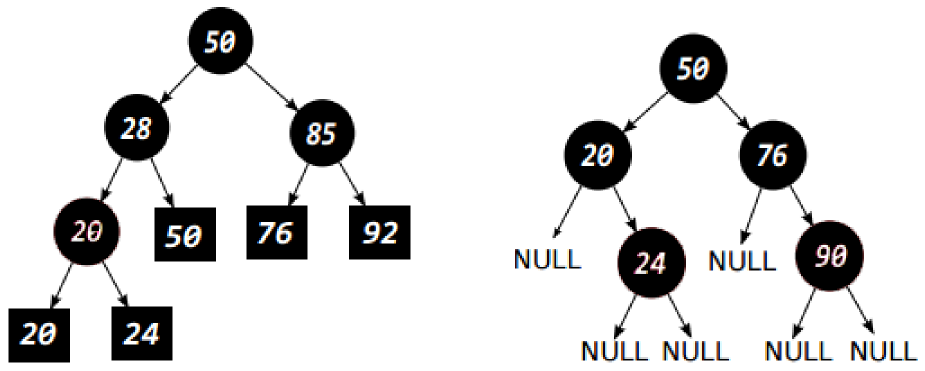


Figure 2.2: An example of a tree in external and internal format.

2.2.1 Binary Search Trees

A binary search tree (BST), also known as an ordered or sorted binary tree, is a node-based data structure in which each node has no more than two child nodes. There is not balance in binary search trees. Each child must either be a leaf node or the root of another binary search tree. Each internal node in BST store a key (and optionally an associated value) and have two distinguished subtrees, commonly denoted left and right. The tree satisfies the binary search property, which states that the key for each node must be greater than all keys in subtree on the left and smaller than any keys in subtree on the right. Duplicate keys are not allowed.

The basic idea behind this structure is to have a storing repository such that the related sorting, searching and retrieving algorithms can be very efficient. Binary search trees are also easy to code and can implement more abstract data structures like dynamic sets of items, multisets, associative arrays and lookup tables that allow finding an item by its key. They have to keep their keys in sorted order, so that lookup and other operations can use the binary search property. While searching for a key in a tree, the traversal begins from the root of the tree to a leaf, the desired key is compared to the keys in BST and deciding, based on the comparison to continue searching in the left or right subtree. If the key is found in BST, the associated value is retrieved. Basic operations (lookup, insertion, deletion) on a BST take time proportional to the height of the tree. On average, each comparison allows the operations to skip about half of the tree, so such operations run in time proportional to the logarithm of the number of nodes n in the tree ($\mathcal{O}(\log n)$). However, in the worst case, where the tree is a linear chain of n nodes, the same operation takes time $\mathcal{O}(n)$.

Although the BST allows fast lookup, addition and removal of items, it has some disadvantages. First of all, the shape of a binary search tree totally depends on the order of insertions and deletions and can become degenerate (e.g a linked list). In this case, the basic operations take a linear time to the number of nodes in the tree. Moreover, when searching a key in a BST, the key of each visited node has to be compared with the key of the element to be searched. And finally, after a long sequence of random insertions and deletions, the expected height of the BST approaches square root of the number of nodes n, $\sqrt{n}$, which grows much faster than $\log n$.

2.2.2 AVL Trees

The AVL tree is a height balanced binary search tree named after its two inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their paper "An algorithm for the organization of information" [3]. It was the first dynamically balanced tree to be proposed. An AVL tree is not perfectly balanced and has two properties. The first AVL property is that every subtree of a node is an AVL tree, too. Assuming that the height of a tree is the number of nodes on the longest path from the root of the tree to a leaf, the second property states that the heights of the two child subtrees of any node differ by at most one. This difference is called **balance factor**. If at any time the balance factor is more than one, rebalancing is required to restore this property. This rebalancing may require the tree to be rebalanced by one or more rotations. Basic operations (lookup, insertion and deletion) take time proportional to the logarithm of the number of nodes n in the AVL tree ($\mathcal{O}(\log n)$) in both average and worst case.

There are many arguments for using AVL trees. First, all the basic operations take time $\mathcal{O}(\log n)$ in the worst case, as the AVL tree is always balanced. Furthermore, the height balancing for the tree adds no more than a constant factor to the speed insertion. And finally, an AVL can never be degenerated in comparison with the BST tree that can be degenerated to a linked list. However, there are also some arguments against using AVL trees. An AVL node has to store the balance factor in order to check violations of the AVL property during insertion or deletion, so an AVL tree demands more space in memory. Programming and debugging an AVL tree is more difficult because of the extra work needed about checking the AVL property and performing rotations. And lastly, although an AVL tree is asymptotically faster than other simple trees, the rebalancing costs time.

Rebalancing in AVL trees

When a thread performs an update operation (insertion, deletion) in an AVL tree, a rebalancing is needed. The rebalancing can be checked through the balance factor of the node. If the balance factor becomes less than -1 or greater than $+1$, the subtree rooted at this node is unbalanced, and a rebalancing is needed. Thus, we have to perform left or right rotations. Figure 2.3 shows a right and a left rotation in a tree without violating the binary search property. An update operation has two phases. The first phase is the standard BST operation (as if the tree were an ordinary binary search tree) and the second includes rotations for rebalancing the tree. When the standard BST operation is performed, there can be four possible cases of the tree that need to be handled. These are depicted in figures 2.4, 2.5, 2.6 and 2.7. The circles represent the nodes being rebalanced. The triangles T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees. The left left case and the right right case are symmetric and need only a single rotation (a right rotation in left left case and a left rotation in the right right case). Similarly, the left right case is symmentric with the right left case and demand two rotations as shown in figures 2.6 and 2.7, respectively.

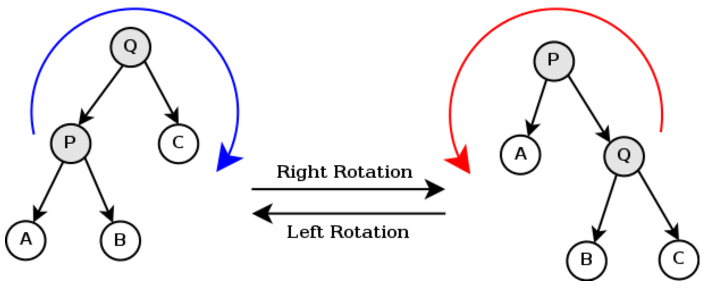


Figure 2.3: A right and a left tree rotation.

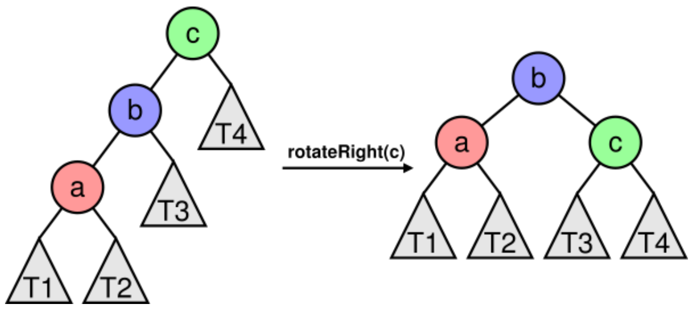


Figure 2.4: Left left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

2.2.3 Red-Black Trees

A Red-Black tree is also a height balanced binary search tree. The Red-Black tree is derived from the symmetric binary B-tree as described in the paper entitled "A Dichromatic Framework for Balanced Trees" [4]. Each node of a Red-Black tree has an extra bit that represents the color of the node. It can be red or black. This painting of each node of the tree preserves the balance of the tree, that is not perfect, as the Red-Black tree is roughly height balanced. Thus, the Red-Black tree satisfies certain properties named coloring properties, which ensure that the tree remains approximately balanced. When the

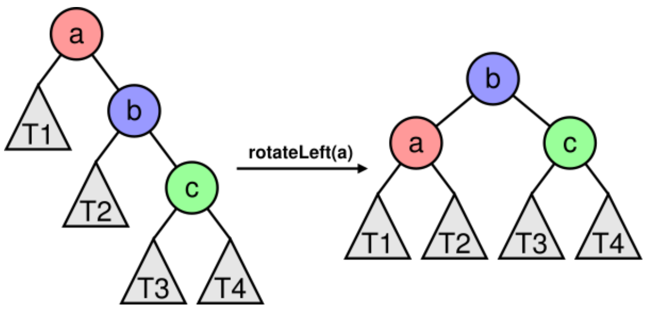


Figure 2.5: Right right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

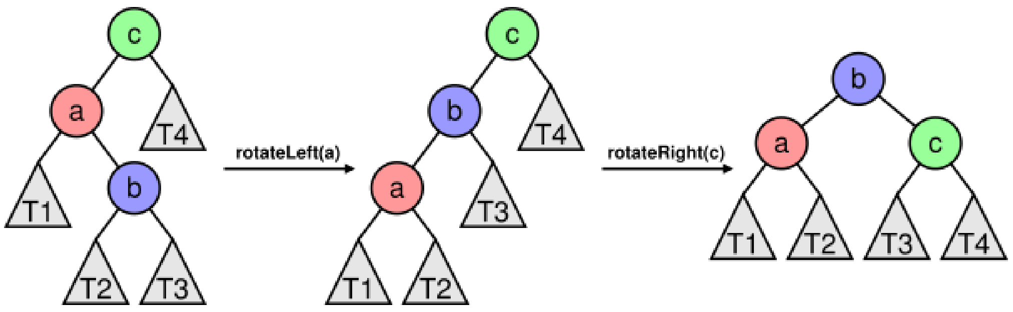


Figure 2.6: Left right case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

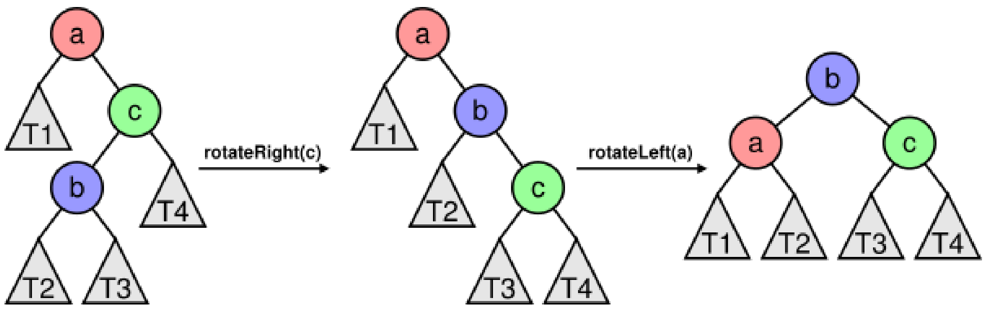


Figure 2.7: Right left case. T1, T2, T3 and T4 represent subtrees which are themselves balanced AVL trees.

tree is modified during operations like insertion or deletion, the tree have to be rearranged and repainted to restore the coloring properties.

A Red-Black tree must satisfy the following coloring properties:

1. A node is either red or black.
2. The root is black (root property).
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black (red property).
5. Every path from a node to a descendent leaf has the same number of black nodes (black property). The number of black nodes from the root to a node is the node's black depth and the uniform number of black nodes in all paths from root to the leaves is called the black-height of the Red–Black tree.

These properties are designed in such a way that the rearranging and the recoloring of the tree can be performed efficiently. Figure 2.8 shows an example of a Red-Black tree. Usually, the leaves of a tree are sentinel nodes as a convenient means of flagging a leaf node and they are black nodes because of the third coloring property. Furthermore, these properties pose another constraint for Red-Black trees: the deepest path in the tree is not longer than twice the shortest one, since all maximal paths have the same number of black nodes according to the last coloring property. More specifically, let B be the number of black nodes of the shortest possible path from the root of the tree to a leaf. The fourth coloring property makes it impossible to insert more than one consecutive red node. Therefore, the longest possible path consists of $2 * B$ nodes, alternating black and red in worst case. Counting the black NIL leaves, the longest possible path consists of $2 * B - 1$ nodes.

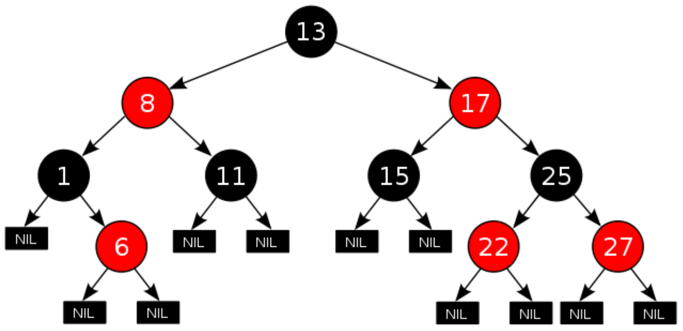


Figure 2.8: An example of a red–black tree.

When inserting or deleting a node, some of the aforementioned properties might be violated and actions must be taken to restore them and rebalance the tree. The two possible violations are: a) *red-red violation*, when a red node acquires a red child (violation of the fourth coloring property) and b) *double black violation*, when a path of a tree contains one less black node than other paths (violation of the fifth coloring property). To deal with these violations a number of node recolors and rotations are applied.

The basic operations (lookup, insertion, deletion) require worst-case time proportional to the height of the tree $\mathcal{O}(\log n)$, where n is the total number of the nodes in the tree. This theoretical upper bound on the height allows Red-Black tree to be efficient in the worst case, unlike ordinary binary search trees. For example, inserting a key in a non-empty Red-Black tree has three steps. In the first step, the BST insert operation is performed, which takes $\mathcal{O}(\log n)$ time, because the tree is balanced. The second step is to color the new node red, which takes $\mathcal{O}(1)$ time, since it just requires setting the value of one node's color field. And in the third step, a restoration of any violated coloring properties is performed. Restoring these properties requires a small number of color changes and no more than three tree rotations (two for insertion). Changing the color of nodes during recoloring is $\mathcal{O}(1)$. However, it might be need to handle a double-red situation further up the path from the added node to the root. In the worst-case, the fixing of a double-red situation along the entire path from the added node to the root is performed. Therefore, in the worst-case, the recoloring that is done during insert is $\mathcal{O}(\log n)$ (= time for one recoloring * max number of recolorings done = $\mathcal{O}(1) * \mathcal{O}(\log n)$). So overall the third step (restoration of coloring properties) is $\mathcal{O}(\log n)$ and the total time for insert is also $\mathcal{O}(\log n)$.

AVL trees are often compared with Red-Black trees because both support the same set of operations and take $\mathcal{O}(\log n)$ time for the basic operations. A Red-Black tree demands less memory space than an AVL tree, since it requires only one bit of information per node for the color, while an AVL tree demands an integer per node for the balance factor. The Red-Black tree does not contain any other specific data, so in many cases the additional bit of information has no additional memory cost and the memory of a Red-Black tree is almost identical to a classic BST tree.

However, AVL trees are more rigidly balanced than Red-Black trees. The height of an AVL tree is bounded by roughly $1.44 * \log_2 n$, while the height of a Red-Black tree may be up to $2 * \log_2 n$. Thus, lookup is slightly slower on the average in Red-Black trees. On the other hand, the AVL trees may cause more rotations during insertions and deletions. So if an application involves many frequent insertions and deletions, the Red-Black trees should be preferred. And if the insertions and deletions are less frequent and lookup is more frequent, then an AVL tree should be preferred. Red-Black trees can be used in data structures for computational geometry and are valuable in time-sensitive applications such as real-time applications and in functional programming to construct associative arrays and sets, which can retain previous versions after mutations, while AVL trees are attractive for data structures that may be built once and loaded without reconstruction, such as language dictionaries.

Rebalancing in Red-Black trees

When a thread inserts a node, it must be colored red. If the new node is black, then inserting it into the tree always introduces a double black violation. The rest of the algorithm would then need to concentrate on fixing the double black violation without introducing a red-red violation. On the other hand, if the new node is red, there is a chance that it could introduce a red-red violation. The rest of the algorithm would then need to work toward fixing the red-red violation without introducing a double black violation. However, if the new node is red, and it is inserted as the child of a black node, then no violations occur at all, whereas if the new node is black, a double black violation always occurs. Therefore, the logical choice is to color the new node red, because there is a possibility that insertion will not violate the rules at all. Red-red violations are also more localized and thus, easier to fix.

Similarly with AVL trees, when a thread runs update operations, the tree is modified and the result may violate the coloring properties. To restore these properties, we have to change the colors of some of the nodes (recoloring) in the tree and perform left and right rotations as appearing in figure 2.3. Upon an insertion of a new red node, there are three possible cases of the tree than need to be handled according to [5]:

- Case 1: the uncle of the inserted node is red
  Figure 2.9 shows the tree in case 1. Both the parent and the uncle of the inserted node are red and their parent has to be black. We can fix the problem by flipping their colors (the parent and the uncle of the inserted node becomes black and the grandparent red). However, if the grandparent's color changes, we risk a violation further up the tree. It is possible that its parent could also be red and there is another red-red violation. Therefore, after this case we have to move up the tree and repeat checking for violations.

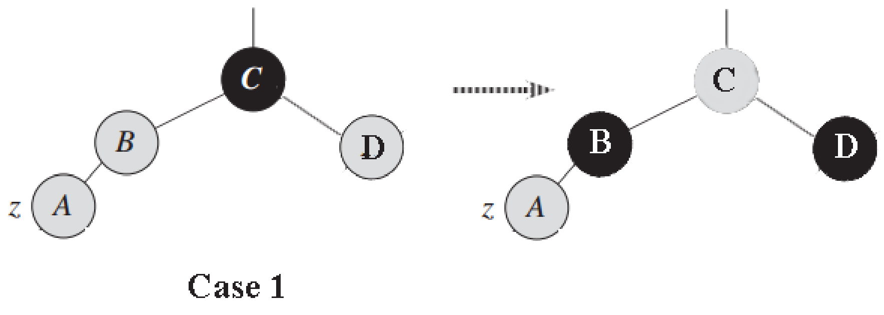


Figure 2.9: Case 1 upon an insetion of a node z in a Red-Black tree. The code for case 1 changes the colors of some node to preserve the coloring properties.

- Case 2: the uncle of the inserted node is black and the inserted node is a left child
  Case 2 is appeared in figure 2.10 In this case, we make the grandparent of the inserted node red and the parent black and we single rotate around grandparent node to the right. This fixes the red-red violation. Nevertheless, the root of this subtree

does not change color. The new root (parent of the inserted node) is black as the previous root (grandparent). In this case, we can be sure that the red-red violation will not propagate upward. Moreover, this rotation does not change the black height of either subtree, so the tree is now balanced and no other rotations or recolors are needed.

- Case 3: the uncle of the inserted node is black and the inserted node is a right child
  Case 3 is also appeared in figure 2.10. In this case a double rotation is needed. We first use a left rotation and the tree becomes the same as in case 2. So, we perform the recolors and a right rotation as they are described in case 2.

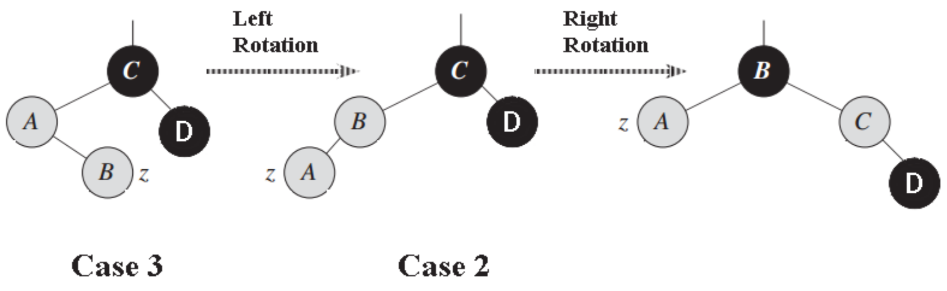


Figure 2.10: Case 2 and case 3 upon an insetion of a node z in a Red-Black tree. We transform case 3 into case 2 by a left rotation. Case 2 causes some color changes and a right rotation to preserve the coloring properties.

The deletion of a node in a Red-Black tree is sure to cause a double black violation, if the deleted node is black. Removing a red node cannot violate any of the coloring properties. Therefore, if we could guarantee that the node to be deleted was red, the deletion would be simplified. When we want to delete node z and z has fewer than two children, then z is removed from the tree, and we want y to be z. When z has two children, then y should be z's successor, and y moves into z's position in the tree. We also remember y’s color before it is removed from or moved within the tree, and we keep track of the node x that moves into y’s original position in the tree, because node x might also cause violations of the coloring properties. We check color of sibling node to decide the appropriate case. There are four cases to be handled [5]:

- Case 1: x's sibling is red
  Case 1 occurs when the sibling node of x is red (Figure 2.11(a)). Since the sibling node has black children, we can switch the colors of the sibling node and the parent of the node x and then perform a left rotation rooted at the parent of node x without violating any of the coloring properties. The new sibling of x, which is one of the previous sibling's children, is now black and thus case 1 has be converted into case 2, 3, or 4.

- Case 2: x's sibling is black and both of its children are black
  In case 2 (Figoure 2.11 (b)) the sibling node is black as well as its children. We recolor the sibling node red. To compensate for removing one black node, we would like an extra black node to the subtree rooted at x. We fix this by repeating the loop for violations and seting the parent of the x as the new node x.

- Case 3: x's sibling is black, sibling's left child is red and sibling's right child is black
  In case 3 (Figure 2.11 (c)) we can switch the colors of the sibling and its left child and perform a right rotation rooted at the sibling node without violating any of the coloring properties. The new sibling node of x is now black with a red right child, and thus case 3 was converted into case 4.

- Case 4: x's sibling is black and its right child is red
  In case 4 (Figure 2.11 (d)) we color the parent of x and the sibling's right child black as well as the sibling node red and then perform a left rotation. Thus, we can remove the black node x without violating any of the coloring properties.

2.3 Techniques for constructing concurrent data structures

As mentioned in the previous chapter, synchronization is vital in parallel programming such that to ensure that two or more parallel tasks like processes or threads do not simultaneously execute the serial segment of the program (critical section). Furthermore, multiple operations are performed simultaneously in concurrent data structures like concurrent search trees. So in order to ensure that correct results are generated in data structures during multiple operations performed and maintain them consistent, a synchronization mechanism is needed. Synchronization techniques like mutual exclusion and atomic operations are performed in the basic operations of concurrent data structures. Programmers' aim is to construct consistent concurrent data structures which result to high performance and scalability in multiprocessor systems. There are at least three common techniques used today to construct concurrent search trees: coarse-grained locking, fine-grained locking and lock-free programming.

2.3.1 Coarse-grained locking

Coarse-grain locking is a technique to construct concurrent data structures using mutual exclusion and locks. An important property of a lock is lock granularity, which is defined as a measure of the amount of data the lock is protecting. There also two useful concepts related with locks, lock overhead and lock contention. Lock overhead is the extra resources for using locks, like the CPU time for lock initialization and destruction, the memory space allocated for locks, and the time for acquiring or releasing locks. An increased usage of locks in a program results to more lock overhead. And lock contention

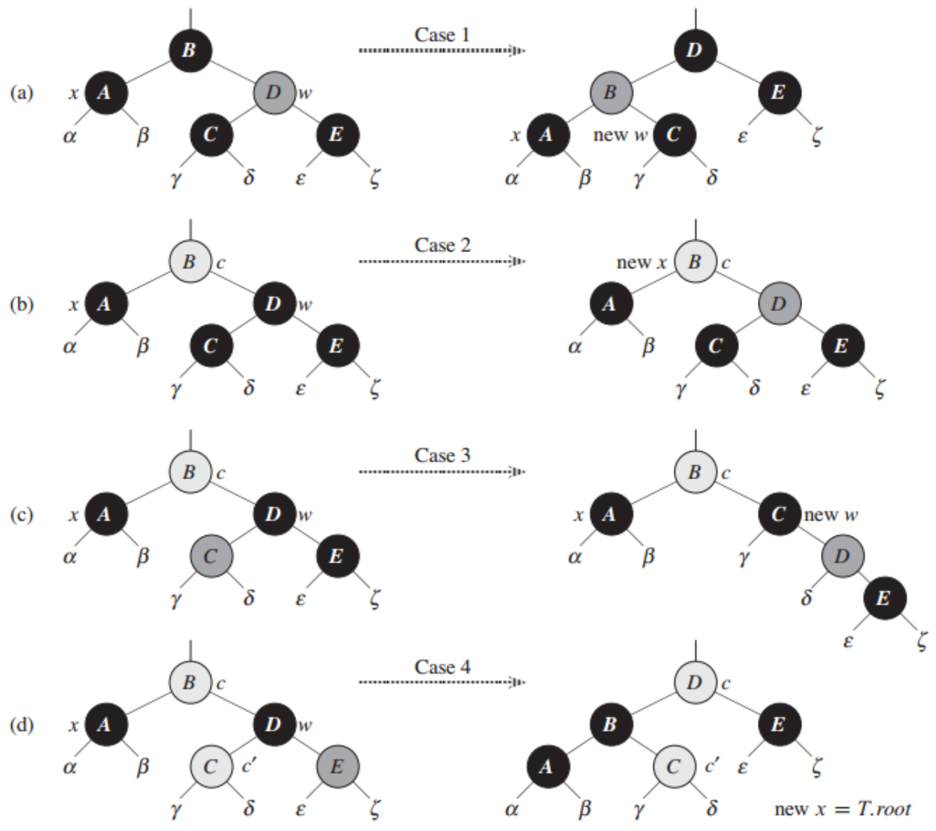


Figure 2.11: The cases upon a deletion of a node in a Red-Black tree. Darkened nodes have color attributes black, heavily shaded nodes have color attributes red, and lightly shaded nodes have color attributes represented by c and c', which may be either red or black . The letters α, β, ..., ζ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black (when a black node is deleted and replaced by a black child, the child is marked as doubly black) or red-and-black. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. (b) In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. (d) Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the coloring properties). This figure has been taken from [5].

appears when a parallel task like a thread attempts to acquire a lock held by another thread. The less granularity the available locks have, the less likely one process/thread will request a lock held by the other.

In coarse-grained locking, when a process/thread needs to access some shared data, the entire shared data are locked via a global lock, read/write operations are performed on them and then the lock is released. The access to the shared data is serialized and only one process/thread can access them, so concurrency is low. Coarse grained locking is relatively simple to implement, easier to use, understand and debug. The only disadvantage is that it is slow and limits the performance in a multiprocessor system. If there are many threads that need access to the shared data, they will have to wait until the thread that holds the global lock finishes its work and releases the lock. Such a situation means a high lock contention and it degrades the performance. Coarse-grained locking is only useful when the threads execute quickly and do not create a lot of lock contention. On the other hand, it results in less lock overhead when a single process/thread is accessing the shared data.

2.3.2 Fine-grained locking

With fine-grained locking, multiple locks of small granularity are used to protect the smallest possible part of the data structure that the current process/thread needs to operate on. This results to an increased lock overhead because more locks are used for the same shared data, more memory allocation is needed and it appears an additional cost of acquiring/releasing locks. On the other hand, fine-grained locking allows high concurrency and exposes more parallelism by reducing the lock contention for the shared data structure. Multiple processes/threads can proceed in parallel when they do not access the same parts of the shared data structure and it can be good for scalability. Although this technique provides more parallelism, it is complex, much more difficult to implement, as it can be very hard to know which locks are needed and in which order, and can create lock dependencies causing problems like race conditions, deadlocks, livelocks. In order to avoid such problems all processes/threads, that perform operations simultaneously on the data structure, have to acquire locks in the same direction (global order).

So overall lock-based mutual exclusions have many disadvantages like high lock contention and lock overhead and the programmer has to find a solution that trades off some parallelism for reduced overhead. Other disadvantages are priority inversion, where a low-priority process/thread holding a lock can prevent high-priority processes/threads from proceeding, and convoying, where all processes/threads have to wait if a process/thread holding a lock is descheduled. Moreover, the debugging is also a challenge, as bugs associated with locks such as deadlocks are time dependent and extremely hard to identify.

2.3.3 Lock-free programming

Lock-free programming is programming without locks. A lock-free program can never be stalled entirely by any single process/thread and can make progress even if individual processes/threads are suspended indefinitely. Lock-free programming can improve system throughput and robustness (by avoiding situations like the failure of a process/thread holding a lock can lead to a system failure) and has desirable liveness properties. The key in lock-free programming is the hardware support. However, it is very hard to design and implement lock-free algorithms properly and programmers choose to design concurrent data structures using non blocking synchronization which is a portable solution and can be used in different kinds of applications. These data structures are called non-blocking data structures.

Non-blocking data structures can be wait-free, if every operation is guaranteed to be finished in a finite number of steps, lock-free, if some operations are guaranteed to be finished in a finite number of steps and obstruction-free, if an operation is guaranteed to be finished in a finite number of steps, unless another operation interferes. Non-blocking data structures do not rely on locks and mutexes to ensure thread-safety, but on techniques like atomic operations and memory barriers. This means that any process/thread either sees the state before or after the operation, but no intermediate state can be observed (atomicity). Most common atomic operations with hardware support in most multiprocessor architectures are compare-and-swap (CAS), test-and-set (TAS), test-and-test-and-set (TTAS), load-linked/store-conditional (LL/SC).

2.4 Basic Interface in Concurrent Search Trees

As mentioned above, a concurrent search tree consists of a set of elements and an interface to access them. This interface includes three main operations: lookup, insetion and deletion. Each element of the set is stored in a node of the tree and consists of a key-value pair. The key uniquely identifies the element in the set. The three main operations have the following semantics:

- **lookup(key)**: searches for a node containing the given key. If it is found, the value that is bound to the key is returned from the operation, otherwise the operation returns NULL.

- **insertion(key, value)**: attempts to insert a new node in the search tree, binding the given key to the given value. The insertion is successful if there is no other node with the same key.

- **deletion(key)**: attempts to delete the node containing the specific key from the tree. The operation is successful if there such a node exists.

The last two operations (insertion and deletion) comprise two distinct phases. First, there is a traversal in the tree until the desired node is reached (always a leaf in case of insertion) and then the actual modification is attempted.

2.5 A naive approach

In this section, we will present some implementations of concurrent search trees which constitute a naive approach for this data structure. Some of these were developed within the context of the papers [6], [7]. Before describing some implementation details we will introduce two concepts. The first that has already mentioned, is a distinction in search trees depending on the way the key-value pairs are being stored in the tree structure. Internal trees store a key-value pair in every node of the tree and external trees store the values only in the leaves while the internal nodes contain only keys and are used solely for routing purposes. The second concept to introduce is about the order in which the necessary modifications for balancing and the update operation (insertion, deletion) related with the given key are performed. Insertion and deletions consist of two phases. The first one traverses the tree in a top-down manner, i.e from the root towards to the leaves for external trees or towards to the appropriate internal node for internal trees, and locate the place where the node with the key is going to be inserted or the node that is going to be removed from the tree. The second phase is performed only in AVL and Red-Black trees. It traverses the tree in a bottom-up manner, i.e from the lead towards the root, modifying parts of the tree and rebalancing the tree in order to restore the tree properties. Whereas the top-down phase always reaches a leaf, the bottom-up phase backtracks a number of times depending on the violation. In the worst case it shall reach the root of the tree. Thus, when the update operations have these two phases, the implementation is called bottom-up.

In serial implementations, bottom-up trees are very efficient, but in concurrent implementations parallel processes/threads might traverse the tree in opposite directions and in case of fine-grained locking a bottom-up implementation is very complicated. Parallel processes/threads acquire locks in their way and there is no global order for the locks. This may lead to a deadlock. To enable fine-grained synchronization, top-down approaches have been proposed [4], [8], where insertion and deletion are performed in a single top-down pass. For balanced trees (AVL, Red-Black trees) in order to achieve this, while traversing the tree from the root to the appropriate leaf, the necessary modifications (i.e recolors in Red-Black trees) and rotations are applied, ensuring that no bottom-up traversal of the tree is required. In this case, in a concurrent execution, all processes/threads acquire locks in the same direction usually using the well known hand-over-hand technique [9] and avoiding the possibility of deadlock. At each step the nodes that are locked, are released only after the next nodes, that appear lower in the tree are locked. However, as top-down implementations perform generally more tree modifications compared to bottom-up implementations, they impose more overhead and result to worse performance in serial executions.

The lookup operation is a bit more simple and is the same for all three types of search trees (BST, AVL, Red-Black tree). Starting from the root of the tree, a path of nodes is traversed until the node associated with the given key is reached. In a concurrent implementation, in case of fine-grained locking the synchronization in the lookup operations is achieved with the hand-over-hand locking techinque [9], too. Locking is performed at each distinct step of traversal and as in update operations (insertion, deletion), the lock of the next node to be visited is acquired before the lock of the current node is released.

2.5.1 Description

We have developed nine different implementations of concurrent search trees, internal and external trees, using the aforementioned techniques for synchronization and both bottom-up and top-down approaches for rebalancing. Furthermore, there are no duplicates in the trees. The insertion of a key is successful, only if the key does not exist in the tree, and in this case a new node with a key-value pair is inserted, otherwise it returns false as a sign for unsuccessful insertion.

Our concurrent search trees are:

- avl-bu-cg-ext-lock tree
  - AVL tree
  - external tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation
- bst-td-fg-int-lock tree
  - BST tree
  - internal tree
  - fine-grained locking
  - top-down (no rebalance needed)
- rbt-bu-cg-ext-iter-lock tree
  - Red-Black tree
  - external tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation
- rbt-bu-cg-int-iter-lock tree
  - Red-Black tree
  - internal tree
  - coarse-grained locking
  - bottom-up approach for rebalance
  - iterative implementation
- rbt-bu-cg-ext-rec-lock tree
  - Red-Black tree
  - external tree
  - coarse-grained locking

  - bottom-up approach for rebalance
  - recursive implementation

- rbt-td-cg-ext-lock tree
  - Red-Black tree
  - external tree
  - coarse grained locking
  - top-down approach for rebalance

- rbt-td-fg-ext-lock tree
  - Red-Black tree
  - external tree
  - fine-grained locking
  - top-down approach for rebalance

- rbt-td-cg-int-lock tree
  - Red-Black tree
  - internal tree
  - coarse grained locking
  - top-down approach for rebalance

- rbt-td-fg-int-lock tree
  - Red-Black tree
  - internal tree
  - fine-grained locking
  - top-down approach for rebalance

2.5.2 Implementation details

- All these implementations were developed in C programming language.

- In order to implement parallelism we used POSIX threads or Pthreads as a parallel execution model. POSIX threads is an API defined by the standard IEEE POSIX 1003.1c.

- False sharing is the most limiting factor on achieving scalability for parallel threads of execution in a symmetric multiprocessor system (SMP), where each processor has a local cache. It occurs when threads on different processors modify independent variables that share the same cache line. This situation invalidates the cache line and forces an update, which degrades performance. Thus, in order to avoid false sharing and align the node structure of the tree in memory, we applied structure padding. This is a technique in which one or more bytes are inserted between memory addresses. As a result, nodes of the tree that were previously allocated in consecutive addresses in memory, now reside on different cache lines. Listing

2.1 and figure 2.12 show a typical bst node structure and a representation of it in memory, respectively. Supposing that a cache line is 64 bytes, the padding is 64 bytes minus the total 'real' bytes of information of a node, such that each node will be allocated in exactly one cache line.

Listing 2.1: A typical bst node structure

```c
typedef struct bst_node{
        int key;
        struct bst_node *lchild;
        struct bst_node * rchild;
        void *value;

        char padding[CACHE_LINE_SIZE - sizeof(int) -
               2 * sizeof(struct bst_node *) -
               sizeof(void *)];
} bst_node_t;
```

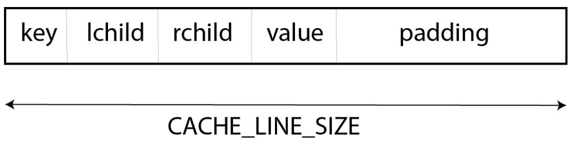


Figure 2.12: A typical bst node structure in memory using padding, such that to be the same bytes as one cache line.

- Iteration is a loop based imperative repetition of a process that repeats some part of the code and recursion a method where the solution to a problem depends on solutions to smaller instances of the same problem. A recursive function calls itself again to repeat some code for a smaller piece of a complicated task and then it combines the results. We have implemented iterative versions of concurrent Red-Black trees as well as a recursive and an iterative version of an AVL tree. In an attempt to implement a bottom-up and iterative approach, we used stack structure. While a thread traverses the tree with a top-down manner until the desired node is reached, we store the path of nodes in the stack such that to access the reverse path with a bottom-up manner and rebalance the tree. As concerning the recursive version, the basic operations of the tree (lookup, insertion, deletion) are recursive functions which call themselves within the program text and when they return, they rebalance the tree if needed. The disadvantage of these recursive functions is that they traverse the whole path from the root to the appropriate for modifications node (always a leaf in external trees) two times, while the iterative, bottom-up approach does not always traverse the whole reverse path to the root in the bottom-up phase, but it stops when no rebalancing is needed.

- In order to implement a coarse-grained locking there is a single global lock (pthread spinlock) shared for all threads. Every thread that performs one of the three main operations waits until the lock is released, acquires the lock, executes the operation and then releases the lock. Only one thread acquires the shared lock at a given time. Coarse-grained locking is easy to implement, but leads to serialization of accesses.
- In fine-grained locking version of concurrent search trees, each node structure has its own lock. Thus, we have added an additional field in the node structure for a pthread spinlock. We only implement top-down approaches because, as explained before, a bottom-up approach is hard, if possible at all, to be implemented. We also keep a global order in locks (acquire locks always in the same direction) to avoid deadlocks. More specifically, while a thread traverses the tree from the root to the leaves, it acquires locks with a top-down manner (lock the node which locates in a higher level of the tree first and then the node in the lower level) via hand-over-hand locking technique [9]. However, internal and external trees entail different requirements and challenges in fine-grained locking. An important one is that in order to delete an internal node (node with two children) from an internal tree, we have first to find its successor (the leaf node with the greatest key that is less than the key of the node to be deleted), swap their key-value pairs and remove the successor leaf node. This operation requires exclusive access to the every node between these two nodes. To achieve this, we keep locked the internal node that we want to delete, until its successor (leaf) node is found. As shown in figure 2.13 the whole subtree rooted at this internal node is locked. If we do not keep locked the internal node, that we want to delete, unlock it and acquire again its lock after we find its successor, it may occur deadlock because of acquiring locks with the opposite direction (first the lower level leaf-successor and then the higher level internal node to be deleted). On the other hand, a deletion in an external tree involves only leaf nodes and there is no such problems.

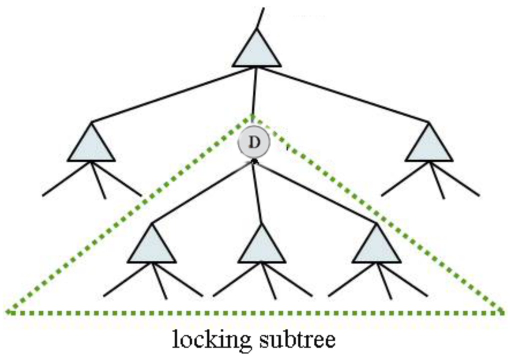


Figure 2.13: Deletion of node D in an internal tree. While searching D's successor to swap their key-value pairs, the subtree rooted at D node must be locked.

- In avl implementation we have also added an additional field in the node structure which represents the height of the node, such that to compute the balance factor and perform rebalancing to restore the AVL property. Similarly, in Red-Black trees each node has a color in order to restore the coloring properties. So, the structure of a Red-Black node has also an additional field for the color.

- Finally, in internal implementations the leaves of the tree are sentinel nodes. Sentinel nodes are designated nodes used as traversal path terminators instead of NULL pointers. Using sentinel nodes, we reduce the code size by avoiding additional checks for NULL pointers.

2.6 A more sophisticated approach

This section includes more sophisticated approaches for concurrent search trees. These implementations are complex algorithms (lock-based or lock-free) taken from the paper [10] and use a more complicated synchronization mechanism than naive concurrent search trees. Thus, we expect to have higher performance and scale better. We will briefly describe the concept of each implementation.

2.6.1 Description

Bronson

This is a concurrent relaxed balance AVL tree proposed at [11], that delivers high performance, good scalability and tolerates contention by controlling the size of critical sections (all updates have fixed size critical sections) and taking advantage of validation logic. A lookup can block until a concurrent update is completed. In an attempt to check concurrent updates, this algorithm uses version numbers. In version number there is a "changing" bit to indicate if a write is in progress and the remainder of the bits form a counter. Each node has a version number, such that to verify if a read is still valid. For example, at time t_1 a thread executes a read and the associated version number is v_1. The thread must block until the change bit in version number is not set (a concurrent write to be completed), read the protected value x and then at time t_2 rereads the version number v_2. If $v_1 = v_2$, then the read is still valid at t_2.

Moreover, as a concurrency control mechanism for searching and traversing the binary search tree the algorithm performs hand-over-hand optimistic validation. Hand-over-hand locking [9] reduces the duration over which locks are acquired by releasing locks on nodes whose rotation no longer affect the correctness of the search and optimistic validation is used to protect critical sections and is chained with hand-over-hand approach. If a key is present in the set of elements, then a thread must traverse the tree from the root to the node associated with the key. Similarly, if a key k is not present in the tree, then a thread must reach the node that would be the k's parent, if it were inserted. Through hand-over-hand optimistic validation, in a lookup, which consists of an interval of keys, the algorithm attempts to check whether a key is absent from the entire tree or present in the current subtree. Each time a lookup operation navigates downward to the tree after performing a comparison between keys the interval is decreased. At all times the interval includes the target key, so if the subtree ever becomes empty, it means that there is no node with that key present in the entire tree. The optimistic validation scheme only needs to invalidate lookups whose state is no longer valid.

Finally, the described tree is referred as partially external tree. In internal trees with no routing nodes the deletion of a node with two children requires that the node's successor must be unlinked from the tree and linked in node's position in the tree. This unlink and relink of node's successor in concurrent trees must be done atomically and every node along the path from the node to be deleted to its successor must be locked. This excessive locking limits scalability and performance. On the other hand, in external trees there is no such problems in a deletion of a node with two children. However, external trees with N nodes require $N-1$ routing nodes and it increases the storage overhead and the average search path. As a result, this algorithm uses a simple scheme referred in [11] as partially external tree that simplifies deletions by leaving a routing node in the tree when deleting a node has two children. When rebalancing is performed, routing nodes with fewer than two children are unlinked from the tree. A deletion of a node with fewer than two children is handled immediately unlinking the node. Partially external trees require fewer routing nodes in most cases than an external tree after a sequence of update operations, but in the worst case they may have exactly the same number of routing nodes. As concerning the node structure, this implementation has the same node structure for both key-value associations and routing nodes and this permits a value node to be converted to a routing node (or the reverse) by changing a field in the node structure and without modifying other inter-node links.

Drachsler

This tree presented in [12] is a BST internal tree that uses logical ordering among nodes. The key to design correct and efficient concurrent binary search trees is to implement a scalable design for the lookup operation. The tree ordering layout is separated from the tree physical layout and thus, lookup operations can proceed concurrently with operations that modify the physical tree layout without synchronization. This approach allows fast lookup operations. The authors of [12] exploited this idea to obtain an intuitive, simple and robust lookup operation, that also provide strong progress guarantees.

The logical ordering among elements can be viewed as consecutive intervals. For instance, the logical ordering for the elements $1 < 3 < 5 < 7 < 9$ can be viewed as intervals $(-\infty, 1), (1, 3), (3, 5), (5, 7), (7, 9), (9, +\infty)$. An element belongs to the tree if and only if it is an endpoint of some interval and does not belong otherwise. The algorithm for these concurrent binary search trees uses intervals to answer lookup requests and to synchronize operations. Each node of the tree keeps its successor endpoint (succ field in the node structure) and its predecessor endpoint (pred field in the node structure), which are unique. The synchronization may also be performed on these endpoints. As this tree is a fine-grained locking approach, two locks are needed, a treeLock, which protects the tree's physical layout fields (left, right, parent), and a succLock which protects the logical layout fields (the succ field and the pred field of the node). That is, for a node n, n's succLock protects the interval $(n, succ(n))$.

A key k is present in the tree if it is the endpoint of an interval ($k \in [k_1, k_2]$). Thus, in a lookup operation, we have to find if there is such an interval. Lookup operation has two phases. Firstly, there is a traversal of the physical tree layout until a leaf is reached and if the key k was found during traversal, then the key k is in the tree. And secondly, if the key k was not found, it must be found an interval with keys k_1 and k_2, that are in the tree and such that $k \in (k1, k2)$. This search for key k terminates when it reaches a node of value $\tilde{}k$ where: (i) $k \in (pred(\tilde{}k), \tilde{}k)$, or (ii) $k \in (\tilde{}k, succ(\tilde{}k)$. This will be done via the logical ordering pointers, the predecessor (pred) and the successor (succ).

Eventually, as in all concurrent search trees a synchronization mechanism is needed. According to [12], each update operation is performed in four steps:

1. Acquire logical ordering layout locks.
2. Acquire physical layout locks.
3. Update the logical ordering layout and release logical ordering locks.
4. Update the physical layout and release physical locks.

As mentioned, acquiring the tree's physical layout locks (treeLocks) prevents simultaneous updates to the node's physical layout information like node's children, parent, e.t.c. and acquiring the tree's logical ordering layout locks (succLocks) prevents simultaneous updates to the intervals. Each interval is associated with a lock, the succLock of the node with the beginning number of the interval as its key. When an operation updates two intervals (like merging two intervals in deletion), it must acquire two succLocks for the two intervals. Therefore, there is synchronization via locks for both the tree physical and logical ordering layout. In an attempt to avoid deadlocks, succLocks, which are used for the tree logical ordering layout should be acquired before treeLocks, which are used for the tree physical layout. Between two succLocks the lock of the node with the smaller key should be acquired first to keep a global order and between two treeLocks the lock of the node that appears lower in the tree should be acquired first. However, deletion operation must acquire treeLocks against the locking order. As a result, when locking treeLocks against the locking order is required, threads optimistically attempt to acquire them without blocking on it (using tryLock()) and if they fail, they release all locks and the operation is restarted. In this way, deadlock cannot occur.

BST Ticket

This concurrent binary search tree proposed in [10] is a lock-based BST implementation. It attempts to reduce the number of acquired lock per update operation and as a consequence the number of cache lines transfers. More specifically, BST Ticket is an external tree, where every internal node used for routing purposes is protected by a lock and contains a version number. As referred in [10] bst-tk stands for BST Ticket and has ticket locks for locking and keeping track of version numbers of nodes. The version numbers are used in order to be able to optimistically traverse the tree and later detect concurrency. It can be validated in order to avoid concurrent conflicting updates. Based on the observation that a ticket lock already contains a version field, the algorithm integrates the version validation and increment, with locking and unlocking, respectively. The interface of ticket locks is modified so that the lock acquisition involves the version number of the node and as a result performing a locking and validating the version can be done in a single step. Briefly, the lookup operation is executed in a wait-free manner and an update operation traverses the tree until the appropriate for modifications node is found, acquires a number of locks and executes the update. If the lock acquisition fails, the version of the lock has been incremented by another concurrent update and the operation has to be restarted. Furthermore, the tree is optimized by allocating two small ticket locks for each node, such that the left and the right child pointer of a node can be locked separately.

In insertion operation the first phase is a lookup operation that keeps track of the predecessor node apart from the current one. If the update is possible, two nodes are allocated, an external node that stores the key-value pair and a routing node, and then a lock is acquired, protecting either the left or the right child pointer of the predecessor. Once the locking succeed, the update is performed, otherwise the operation is restarted. Similarly, the lookup phase of a deletion operation keeps tracks of both predecessor and the predecessor of the predecessor, as a deletion influences both nodes. If the deletion is possible, the appropriate child pointer (left or right) of the predecessor of the predecessor is locked, as well as both pointers of the predecessor (this is done in a single step). If both acquisitions are successful, then the deletion is performed, otherwise the operation has to be restarted. Overall, this implementation demands one lock for a successful insertion and two locks for a successful deletion.

Aravind

This concurrent binary search tree is presented in the paper [13]. It is a lock-free algorithm that supports the three basic operations for binary search trees. Lock-freedom requires that some process be able to complete its operation in a finite number of steps. Thus, this lock-free approach uses two atomic operations for reads and writes, compare-and-swap (CAS) and bit-test-and-set (BST). As in previous concurrent search tree, an external representation of search trees has been selected. In order to limit the conflicts among update operations and reduce the overhead of update operations there are some optimizations. As mentioned in [13] (i) the algorithm is based on marking edges as deleted rather than nodes, (ii) it does not use explicit objects for coordination between conflict operations and finally, (iii) it permits multiple keys (nodes) being removed from the tree in a single step. As a result, update operations in this algorithm work on a smaller portion of the tree (smaller contention window), allocate fewer objects and execute fewer atomic operations (one for insertion and three for deletion).

In contrast to previous implementations this algorithm marks the edges as deleted instead of the nodes. Each update operation becomes the "owner" of some edges that it needs to work on. Note that every edge has a tail and a head node. Marking an edge means that either both tail and head nodes or only its tail node will be deleted from the tree. Thus, it is vital to distinguish between these two cases. In paper [13], the first type of marking is referred as *flagging* and the second type as *tagging* and as for the implementation, to enable flagging or tagging they exploit two bits (denoted flag and tag) from each child address (each child pointer in the node structure). If one of the two bits in a child address has value 1, then the corresponding outgoing edge has been marked (flagged or tagged), otherwise the edge is not marked. For example, in a deletion operation of a node (leaf) n, marking an edge means setting the flag bit in the child field of n→parent, that points to n→leaf to 1 (set a bit in the pointer that is associated with the left or the right child). Additionally, as explained in section 3.2.4 [13], if there are multiple edges marked in the tree, this will cause multiple leaf nodes to be removed from the tree in a single step during deletion operation.

Finally, in this algorithm there is also a helping strategy that is performed only in deletion operations. There is no helping strategy for insertions. This is why a helping strategy increases the overhead of an operation and may provoke duplication of work. Moreover, this algorithm does not use explicit objects for coordination, but steals two bits from the child address of the node structure. In an insertion operation of a node n, a helping strategy needs to be performed when it is discovered that the the edge from n→parent to n→leaf exists and has been marked (flagged or tagged). It means that a concurrent deletion operations is attempting to delete n→parent from the tree. As a result, the insertion operations helps the concurrent deletion operation to complete (n→parent and one of its children to be removed from the tree). Subsequently, the insertion operation restarts from the beginning (lookup phase). Similarly, the deletion operation of a leaf node n performs a flagging of the edge from n→parent to n→leaf using CAS atomic operation. If the CAS operation fails, the deletion operations executes a helping strategy in the same way as in insertion operation and then it tries again by retrying the lookup phase.

Ellen

The last implementation is presented in [14] and describes a non-blocking and linearizable binary search tree (BST). This algorithm is a lock-free version of a BST that uses non-blocking synchronization and more specifically the atomic operation compare-and-swap (CAS). Therefore, it can tolerate any number of crash failures. Furthermore, as mentioned in [14] they use a leaf-oriented BST that has already presented as external tree. All keys of the set of elements for the tree are stored in leaves and every internal node that has exactly two children is used to find the path to the correct leaf and its key may or may not be in the set of elements. Update operations (insertions and deletions) that alter different parts of the tree and do not interfere with one another can proceed concurrently. Lookups only perform reads of shared memory and traverse the tree from its root to a leaf (as this is an external tree), so they do not interfere with updates, too.

In update operations the appropriate modifications for the tree are performed using the atomic operation CAS. However, in concurrent updates this can lead to problems and in an inconsistent state of the tree. To avoid analogous problems there is a mark field in the node structure named "state", so that in a deletion operation of a leaf, the parent's state field is set before unlinking the parent from the tree. Setting the node's state field through CAS steps ensures that its child pointers cannot change. This field is also used to flag the node to indicate that an update is attempting to change a child pointer of the node. Before an update operation, that changes either of node's child pointers, the state field is changed to a flag value according to the update operation (insertion or deletion) to be performed. After the termination of the update operation the state changes back to a "clean" state. Generally, setting the state field of a node is similar to locking its child pointers. In concurrent search trees an operation must successfully acquire the lock in order to alter node's child pointers, because this ensures that they never can change until releasing the lock. A lookup operation does not change any child pointer and thus, it does not acquire any locks. On the other hand, insertion operation is guaranteed to complete when it acquires a lock (sets the state field) of a single node and a deletion operation after acquiring locks of two nodes. Since only update operations need to acquire locks of one or two nodes near a leaf of the tree, this locking scheme does not provoke serious contention problems and concurrent updates which do not interfere with one another (perform modifications on different parts of the tree) can proceed simultaneously.

In this implementation there is also a helping strategy. More specifically, a process helps another process's operation to finish only if the other operation is preventing its own progress. As a lookup operation does not modify the tree and cannot never be blocked, it never helps any other operation. Nevertheless, an update operation that must lock a node (setting the state field in node structure) that is already locked helps complete the operation that locked the node first and then in retries its own update operation. In an attempt to achieve this helping strategy, there is an Info record. When an operation locks a node, it stores enough information (in an Info record), so that another process that requests the locked node, to help complete the operation. This mechanism suffices to achieve a non-blocking synchronization. According to the update operation to be performed there are two types of Info record, as an insertion and a deletion operation demand different information to be stored. As described in [14], to complete an insertion a process must have a pointer to the leaf which is to be replaced, that leaf's parent and the newly created subtree that will be used to replace the leaf. And these are the information to be stored to an Info record for an insertion. Similarly, to complete a deletion operation a process must have a pointer to leaf to be deleted, its parent, its grandparent and a copy of the state and info fields of the parent. So, these information are stored to an Info record in a deletion operation. Therefore, if a insertion finds that some other operation has locked (has set the state field of the node structure) the parent of the node that is to be inserted, it helps the other operation to complete and then retries. If a deletion operation finds that the parent or the grandparent of the node to be deleted has already locked, it helps that operation to complete and then starts over with a new attempt. However, a deletion operation demands two locks, one for node's grandparent (acquiring it first) and one for node's parent. Thus, it is possible that a deletion will fail to complete after the grandparent is locked, because the parent is already locked by another process. In this case, it helps the operation that locked the parent to complete and performs a backtrack CAS to release the grandparent.

2.6.2 Implementation details

These five implementations were developed in C programming language using POSIX threads within the context of the paper [10]. The code is avialable at http://lpd.epfl.ch/site/ascylib.

2.7 Evaluation

2.7.1 System Configuration

The system we used to evaluate the implementations was a 60-core platform (Figure 2.14), NUMA architecture with the following characteristics.

- 4 sockets (Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz)
- 15 cores per socket (30 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 38MB L3 cache per socket
- 1TB RAM

2.7.2 Run Configurations

To evaluate the implementations of the described concurrent search trees, we perform random operations varying the number of threads, the range of the set of elements from which the keys are selected and the proportion of lookup, insertion and deletion operations. More specifically:

- Each software thread is manually pinned to a hardware thread in order to take advantage of the locality with the sockets. For instance, in case of a small number of threads we pinned them in the same socket, so as to share the same L3 cache. We also pin software threads to cores in such a way that all the available physical cores are being employed before utilizing hyperthreads. Otherwise, if we did not pin the software threads, the operating system may execute many software threads in the same core leaving another core idle.
- The duration of each execution is 5 sec, during which each thread performs randomly chosen operations, based on the percentage of operations we have selected.

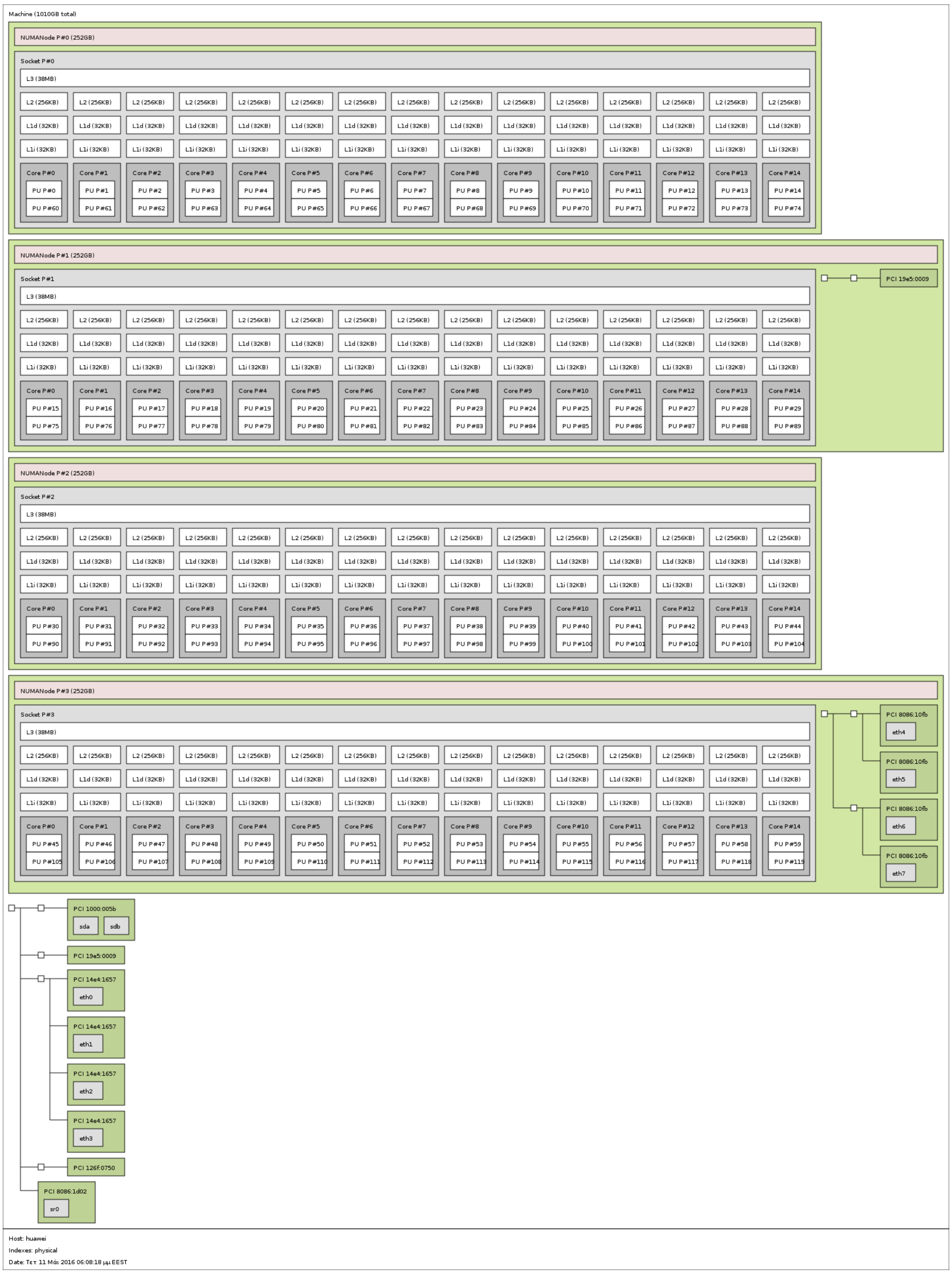

Machine (1010GB total)
NUMANode P#0 (252GB)
Socket P#0
L3 (38MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14
PU P#0 PU P#1 PU P#2 PU P#3 PU P#4 PU P#5 PU P#6 PU P#7 PU P#8 PU P#9 PU P#10 PU P#11 PU P#12 PU P#13 PU P#14
PU P#60 PU P#61 PU P#62 PU P#63 PU P#64 PU P#65 PU P#66 PU P#67 PU P#68 PU P#69 PU P#70 PU P#71 PU P#72 PU P#73 PU P#74
NUMANode P#1 (252GB)
Socket P#1
PCI 19e5:0009
L3 (38MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14
PU P#15 PU P#16 PU P#17 PU P#18 PU P#19 PU P#20 PU P#21 PU P#22 PU P#23 PU P#24 PU P#25 PU P#26 PU P#27 PU P#28 PU P#29
PU P#75 PU P#76 PU P#77 PU P#78 PU P#79 PU P#80 PU P#81 PU P#82 PU P#83 PU P#84 PU P#85 PU P#86 PU P#87 PU P#88 PU P#89
NUMANode P#2 (252GB)
Socket P#2
L3 (38MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14
PU P#30 PU P#31 PU P#32 PU P#33 PU P#34 PU P#35 PU P#36 PU P#37 PU P#38 PU P#39 PU P#40 PU P#41 PU P#42 PU P#43 PU P#44
PU P#90 PU P#91 PU P#92 PU P#93 PU P#94 PU P#95 PU P#96 PU P#97 PU P#98 PU P#99 PU P#100 PU P#101 PU P#102 PU P#103 PU P#104
NUMANode P#3 (252GB)
Socket P#3
L3 (38MB)
L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#4 Core P#5 Core P#6 Core P#7 Core P#8 Core P#9 Core P#10 Core P#11 Core P#12 Core P#13 Core P#14
PU P#45 PU P#46 PU P#47 PU P#48 PU P#49 PU P#50 PU P#51 PU P#52 PU P#53 PU P#54 PU P#55 PU P#56 PU P#57 PU P#58 PU P#59
PU P#105 PU P#106 PU P#107 PU P#108 PU P#109 PU P#110 PU P#111 PU P#112 PU P#113 PU P#114 PU P#115 PU P#116 PU P#117 PU P#118 PU P#119
PCI 8086:10fb
eth4
PCI 8086:10fb
eth5
PCI 8086:10fb
eth6
PCI 8086:10fb
eth7
PCI 1000:005b
sda
sdb
PCI 19e5:0009
PCI 14e4:1657
eth0
PCI 14e4:1657
eth1
PCI 14e4:1657
eth2
PCI 14e4:1657
eth3
PCI 126f:0750
PCI 8086:1d02
sr0
Host: huawei
Indexes: physical
Date: Τετ 11 Μάι 2016 06:08:18 μμ EEST

Figure 2.14: The platform used in evaluation of concurrent search trees.

- The key range effectively determines the size of the tree, we evaluate the concurrent search trees for ranges 2K, 32K and 2000000 keys. In the beginning of each execution the tree is initializes with half the possible keys of the selected range. This provides the guarantee that on average half of the operations are successful and that the average execution time of each operation remains the same all over the whole execution (the percentage of insertion operations is the same as deletion operations, so that the tree remains approximately the same).

- We use various proportion of operations, three different workloads 80-10-10 50-25-25 20-40-40, with 80%, 50% and 20% of operations respectively being lookups in the tree, i.e. read-only traversals, while the rest are equally divided between insertions and deletions. These workloads represent a read-dominated, read-write and write-dominated access pattern on the tree respectively.

2.7.3 Results

Naive concurrent search trees

Figures from 2.15 to 2.17 depict the throughput obtained from executions of the naive concurrent search trees on the described 60-core platform. Trees with key range 2K do not scale and in this case the performance is reduced as the number of threads increases. In small trees there are many conflicts on nodes among threads and as a result high contention. While the key range of the tree expands (32K and 2000000), the fine-grained implementations scales until a number of threads and after that point the performance collapses. More specifically, they scale up to 8 threads. After the point of 15 threads, the throughput is decreased and it appears the effect of NUMA architecture as threads employs more than one socket of the platform. In a cache miss during an operation the transfer of a cache line for a node is very expensive from one socket to another. On the other hand, coarse-grained locking versions do not provide parallelism. They are protected by a single global lock and as a result they serialize all accesses on the tree. They used as a baseline.

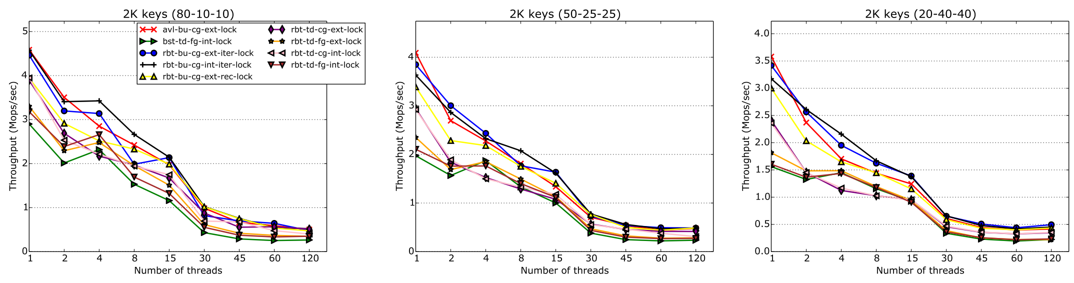


Figure 2.15: Throughput of concurrent naive implementations for 2K key range and the three workloads.

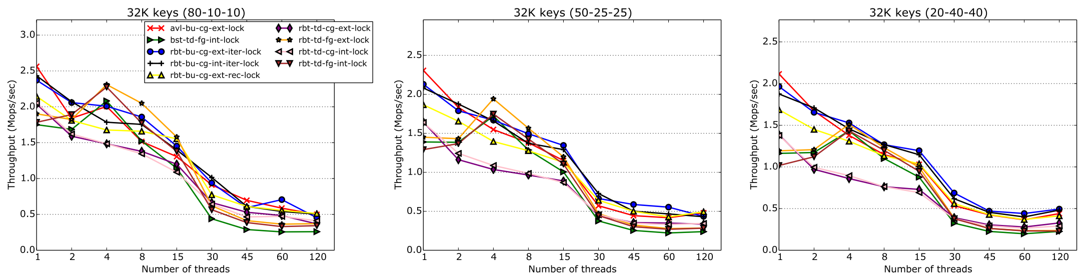


Figure 2.16: Throughput of concurrent naive implementations for 32K key range and the three workloads.

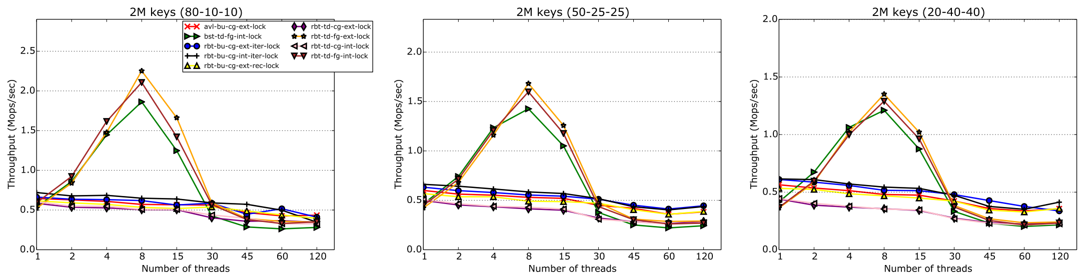


Figure 2.17: Throughput of concurrent naive implementations for 2000000 key range and the three workloads.

Among the fine grained-implementations, the RBT external tree version has the highest performance. This is because it is a height-balanced tree comparing to the BST fine-grained version and performs faster deletions comparing to the RBT internal version. As already mentioned, the deletion operation in an internal fine-grained locking tree requires exclusive access to every node between the node to be deleted and its successor. To achieve this in a fine-grained version the node to be deleted is kept locked until its successor is found. In this way, the whole subtree rooted at this node is locked and no other thread can proceed in it. All threads that attempt to proceed in this subtree block. This results to faster deletions in an external version which involves only leaf nodes. On the other hand, in coarse-grained implementations the internal version has a better throughput than the external version. In internal trees a lookup operation is faster as it can terminate in a small depth of the tree, while in external trees the operation terminates when a leaf is reached.

In top-down approach, while traversing the tree from the root to the appropriate leaf, modifications are proactively performed in order to guarantee that no bottom-up traversal of the tree is required. This pessimistic nature of top-down approach generally results to more tree modifications for each operation compared to bottom-up which performs only the necessary modifications for the operation. Consequently, top-down approaches have worse performance in serial executions (coarse-grained locking) and impose more overhead. In our results bottom-up coarse-grained locking trees have higher throughput than top-down, as the cost to traverse two times the path to the appropriate node executing only the necessary modifications is lower than traversing it one time, executing modifications in advance to avoid bottom-up traversal (typically more modifications are performed).

Comparing bottom-up iterative and recursive implementations, the iterative bottom-up concurrent search trees perform better than recursive versions. The iterative trees terminate the bottom-up traversal when no other rebalancing is needed (in a balanced node, not necessary the root node), while the recursive trees in bottom-up phase have to return up to the root of the tree. Thus, recursive trees traverse the path to the appropriate node exactly two times performing the necessary modifications. Furthermore, there is usually more overhead associated with the recursive calls due to the fact that the call stack is so heavily used during recursion.

Finally, we can note that in small trees the throughput in coarse-grained locking implementations is reduced significantly as the number of threads increases, in contrast to larger coarse-grained locking trees, where throughput is approximately the same for the different number of threads. This happens because the operations in small trees take a little time. As a result, as the number of threads increases, each thread blocks for a long time and performs a fast operation, while the same operation in a large tree lasts more time. Therefore, the operations performed in the same amount of time in small trees are much less in case of multiple threads than that in case of one thread.

More sophisticated concurrent search trees

The results (Figures 2.18, 2.19, 2.20) of our evaluation are close enough to those presented at [10]. Aside from BST Ticket search tree, Aravind concurrent search tree is generally the best concurrent implementation. This lock-free implementation uses two atomic operations on average per update operation, which is close to a concurrent search tree without any synchronization in terms of the number of stores and the number of the affected cache lines. More specifically, this tree has a small contention window because it operates at edge-level (marking edges as deleted instead of nodes). Secondly, it allocates fewer objects and executes fewer atomic instructions per update operation than the other algorithms. It does not use explicit objects for coordination between conflicting operations (like Info record in Ellen concurrent search tree). Helping strategy is performed only for deletion operations and instead of using explicit objects for coordination, the algorithm uses a small number of bits from child pointers stored in the node structure to enable coordination between operations. There is no helping strategy for insertions because it increases the overhead of an operation and may provoke duplication of work. And as already explained, in this algorithm, multiple leaf nodes may be deleted (unlinked from the tree) in a single step. To sum up, the algorithm reduces the contention between update operations and lowers the overhead of an update operation.

Comparing BST Ticket concurrent search tree with Aravind search tree, they have very similar behavior as they both are close to an asynchronous search tree. BST Ticket tree outperforms lightly the Aravind search tree (Figure 2.18 and 2.19 for smaller trees)

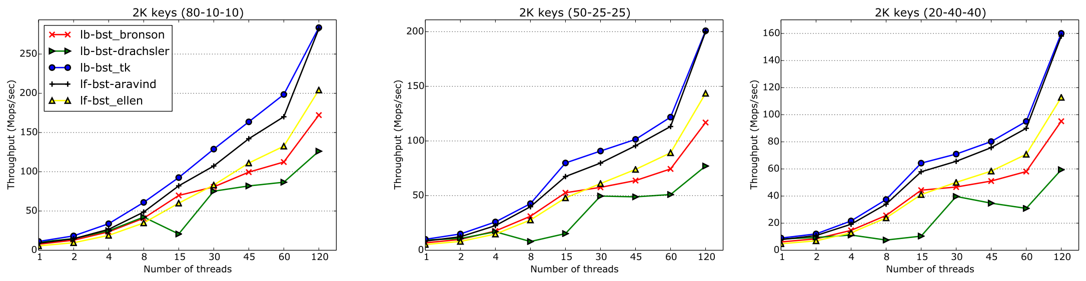


Figure 2.18: Throughput of concurrent sophisticated implementations for 2K key range and the three workloads.

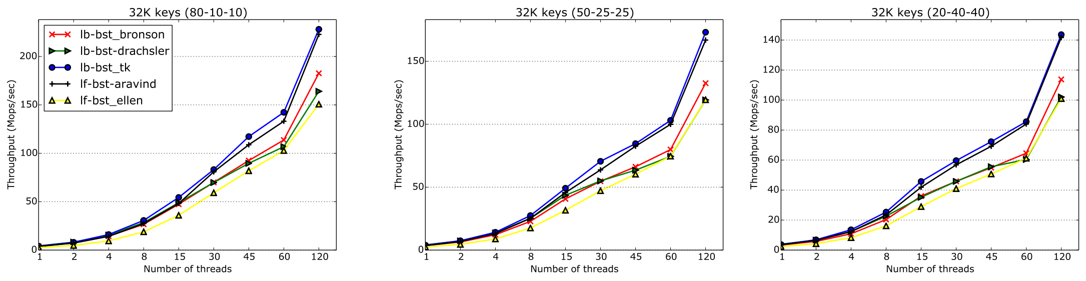


Figure 2.19: Throughput of concurrent sophisticated implementations for 32K key range and the three workloads.

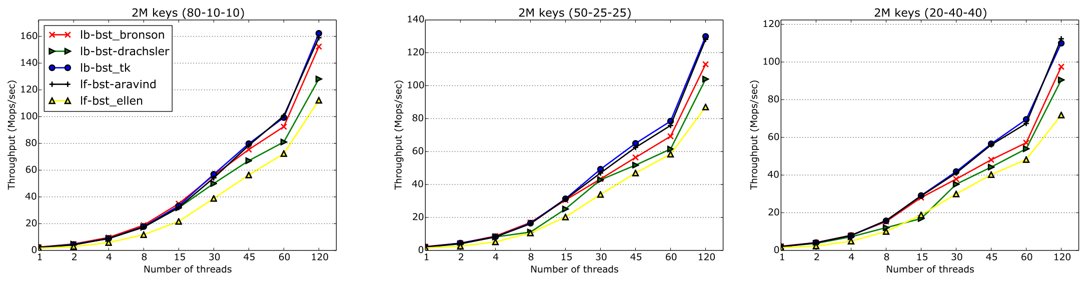


Figure 2.20: Throughput of concurrent sophisticated implementations for 2000000 key range and the three workloads.

as it executes less atomic operations per update. It executes two atomic operations per deletion while Aravind search tree executes three atomic operation per deletion (they both execute one atomic operation per insertion). However, BST Ticket has slightly increased parsing overhead compared to Aravind and in large trees where the contention between threads is not so high, they have approximately the same behavior (Figure 2.20).

The rest three trees (Bronson, Drachsler, Ellen) have worse throughput for all the three key ranges used in our evaluation. They have a more complicated synchronization mechanism. Ellen concurrent search tree employs helping strategy for both insertion and deletion operation and only on elements that the current operation wants to modify. Helping strategy is typically expensive, as it requires additional synchronization to be implemented and imposes additional overhead for each operation. Furthermore, Ellen concurrent search tree uses explicit objects (Info record) that augment the number of stores per operation and the number of cache line transfers. Secondly, Drachsler concurrent search tree is a lock-based tree that acquires a large number of locks for each successful update operation that limits the performance. It has a better throughput on workloads where the lookup percentage is high and the main advantage of this implementation is that during deletion operation it can find the successor of node to be deleted in a single step ($\mathcal{O}(1)$) through successor pointer stored in the node structure. Finally, Bronson concurrent search tree outperforms Ellen and Drachsler concurrent search trees in most of our experiments (Figures 2.19 and 2.20) because of its balance. It is a relaxed balance AVL tree. However, it is also a lock-based complex algorithm which uses hand-over-hand locking technique and threads can block for a long time waiting an update operation to complete. As a result, its performance is very low in small trees with high contention (Figure 2.18).

As explained in [10] cache coherence is the most significant limiting factor for scalability for concurrent search algorithms on multiprocessor systems, since the number of cache line transfers increases with the number of threads. Thus, most concurrent search tree algorithms attempt to minimize the amount of cache traffic (cache line transfers) it performs during each operation which is directly associated with the number of stores on shared data structure. Stores provoke invalidation in cache lines according the cache coherence protocol and result to cache misses of future accesses. Generally, the fewer cache misses an algorithm generates, the better it scales.

Finally, the results explain that lock-based and lock-free algorithms are close in terms of performance, but in case of high number of threads (high contention) lock-free implementations provides better scalability than lock-based. Lock-free implementations also offer robustness. Moreover, the number of stores in a successful operation should be close to an asynchronous, sequential algorithm. The closer to the sequential algorithm that provides no synchronization, an implementation is, the higher performance it has.

Chapter 3

Transactional Memory

3.1 Transactional Memory (TM)

As multiprocessor systems became the dominant computing systems, the discovery of a non-blocking scheme for synchronization that would provide better scalability and would simplify the parallel programming was necessary. This need led to Transactional Memory (TM). An important benefit of transactional memory is that there are not locks and deadlocks.

Transaction memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way. Complex operations can be performed concurrently, in isolation from each other, with those operations either completing or being undone, as transaction, a model that developers are already familiar with from database programming. Transaction is a unit of work that either completes in its entirety or has no effect at all (is executed atomically).

Transactions must be serializable (appear to execute sequentially). Serializability is a kind of coarse-grained version of linearizability. Linearizability is a guarantee about single operations on single objects. Each method call of a given object should appear to take effect instantaneously between its invocation and response. For instance, once a write completes, all later reads should return the value of that writer or the value of a later writer. Once a read returns a particular value, all later reads should return that value or the value of a later write. Serializability, on the other hand, defines atomicity for entire transactions, that is, instruction groups in the code that include calls to one or more objects. It ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call. Furthermore, it guarantees that the execution of a set of transactions over multiple objects is equivalent to some serial execution.

The idea of transactional memory is that during the execution of a transaction there is no need for synchronization. The underlying TM system can detect that a conflict has occurred because of a parallel execution of processes in multiple cores. A conflict occurs when two transactions perform conflicting operation to the same memory address. There are two types of conflicts:

1. A transaction writes to a memory address in which other processes perform a read or a write, too.
2. A transaction reads a memory address in which other processes perform a write.

If no conflicts detected during the execution of a transaction, then the underlying TM system attempts to persist the transaction's results and inform all other processors about the transaction's modifications (make all changes visible and permanent). This is a **transactional commit**. Otherwise, if conflicts are detected during the execution of a transaction, then the underlying TM system rolls back the current transaction, causes all the modifications made by the transaction to be discarded and revert the system to the previous stable state as if the transaction had never begun. This is a **transactional abort**. Therefore, we can conclude that the speed in which the conflicts are detected and the commits/aborts are executed is the most significant limiting factor for the performance of the underlying TM system.

TM system, depending on the implementation, can be divided into three categories, the Software Transactional Memory (STM), the Hardware Transactional Memory (HTM) and the Hybrid Transactional Memory.

3.1.1 Software Transactional Memory (STM)

Software transactional memory provides transactional memory semantics implemented exclusively in software, rather than as a hardware component. There is no use of hardware components in detecting conflicts or in performing transactional commits/aborts during transaction. It can be implemented as a lock-free algorithm or it can use locking and is a software runtime library. Some STM implementations released are: TiniSTM (C programming language), STMNet (C#), CL-STM (Common Lisp), STM Library (Haskell), Deuce, DSTM2 (Java), ScalaSTM (Scala).

The main advantage of software transactional memory is that it can be used in any platform/system, as there is no need for a particular hardware support. Furthermore, it is more flexible, as it permits implementation of a wider variety of more sophisticated algorithms and is easy to modify and evolve. However, transactional memory implemented entirely in software is slow and come with performance penalty, when compared to hardware solutions. Typically, detecting conflicts and performing the appropriate actions, after a transactional abort, to discard the modifications made by the transaction and revert the system to a previous stable state are very costly and time-consuming. As a result, software transactional memory can provide increased performance only in very particular cases.

3.1.2 Hardware Transactional Memory (HTM)

Hardware transactional memory was proposed as a performance improvement of software transactional memory. It consists of a full implementation of TM in hardware. Detecting conflicts and performing transactional commits or aborts are exclusively executed on hardware. The main purpose of HTM is to reduce the overhead of performing a transaction in a system. HTM can also have better power and energy profiles than STM and can provide strong isolation without requiring changes to non-transactional memory accesses.

As explained in [15], in attempt to implement HTM we have to add a transactional bit to each cache line's tag. Firstly, the transactional bit is unset, but when a value is placed in the cache line on behalf of a transaction, this bit is set and this entry is transactional. Modified transactional cache lines do not be written back to main memory before the transaction commits and invalidating a transactional cache line aborts the transaction. More specifically:

- If a transactional cache line is invalidated according to coherence protocol (e.g MESI), then the transaction is aborted. This invalidation indicated a synchronization conflict (another processor accessed on this cache line), either between two writes or a read and a write.
- If a modified transactional cache line is invalidated or evicted from the cache, the value is discarded (it must not be written to the main memory). While the transaction has not commit, we cannot evict tentative transactionally written values. In this case we must abort the transaction.
- If the cache evicts a transactional line, the cache coherence protocol cannot detect synchronization conflicts, since the cache line is no longer in cache. The transaction must be aborted, too.

Finally, when the transaction terminates and none of its transactional lines has been invalidated or evicted, the transaction commits, unsetting the transactional bits in its cache lines. If the transaction is aborted, its transactional cache lines are invalidated.

Although HTM is not so time-consuming as STM, it adds an important cost during transaction and especially in case of consecutive aborts. Nevertheless, the most limiting factor in HTM is the limited hardware resources. For instance, the size of the transaction is limited by the size of cache. Moreover, most operating systems flush the cache when a thread is descheduled, so the duration of the transaction may be limited by the time quantum of scheduling. As a result, HTM must be used for small transactions, whereas applications that need longer transactions should use STM or hybrid transactional memory. When a transaction aborts, the hardware should return a condition code indicating the reason of transactional abort. If the abort was due to a synchronization conflict (data conflict), the transaction should be retried. If the abort was due to a hardware resource exhaustion, there is no point in retrying the transaction. Another limitation in HTM is the usage of extensions in instructions of HTM. The code of the program must be rewritten for each processor that supports a different HTM implementation using different extensions each time. Finally, it is obvious that a program that employs an HTM implementation cannot be always executed in a platform which does not support HTM.

3.1.3 Hybrid Transactional Memory

This scheme is combination of both models, STM and HTM and provides benefits of both of them. For example, the papers [16], [17] propose an hybrid transactional memory, which implements TM in software so that it can use hardware TM (HTM) to boost performance but it does not depend on HTM and does not expose programmers to any of its limitations. The papers [18] and [19] propose an approach, in which hardware is used to accelerate a TM implementation controlled fundamentally by software (hardware accelerated STMs).

3.2 Basic TM Characteristics

Different TM implementations combine different options of the basic TM characteristics. The basic characteristics for TM implementations are:

- **Data versioning**

  Transactional memory systems require a mechanism to manage the tentative writes in concurrent transactions. This mechanism is known as data versioning. Transactional writes require two copies of the written location to be stored: the committed (old) version of data (to be used by other processes/threads while the transaction in the current thread is executed, and if the transaction aborts) and the uncommitted (new) version of data (to be used when the transaction commits). There are two approaches of data versioning depending on the location in which the commited and the uncommitted version of data are stored:

  – *Eager versioning (undo-log based)*

    This approach is also known as direct update because it means that the transaction directly updates the data in memory. The transaction maintains an "undo log" holding the committed values that it has overwritten. This undo log works like a stack. Every time a memory location is modified, the committed (old) values are copied in the stack. Thus, when a conflict is arised during the transaction, the transaction aborts and the system rolls back with each step of the undo log being executed in reverse order to restore the previous original state of memory. Figure 3.1[*] shows an example of eager versioning.

  – *Lazy versioning (write-buffer based)*

    This approach is also known as deferred update because the updated are delayed until a transaction commits. The transaction maintains its tentative writes in a write-buffer in cache instead of directly writing to memory. When a transaction commits, it updates the actual memory locations from the copies of write-buffer. Since transaction's updates are maintained in the write-buffer, a read inside the same transaction must consult the write-buffer so that earlier writes are seen. If a transaction fails to commit due to a transactional conflict, the write-buffer is discarded and the transactions do not modify memory at all. Figure 3.2[*] depicts an example of lazy versioning.

[*] Image taken from http://15418.courses.cs.cmu.edu/spring2013/article/40.

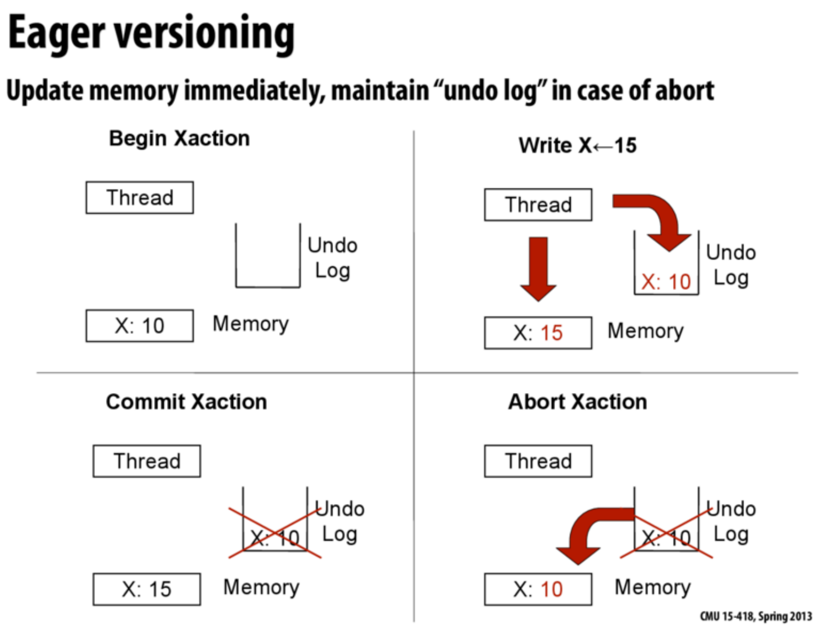


Figure 3.1: A simple example of eager versioning.

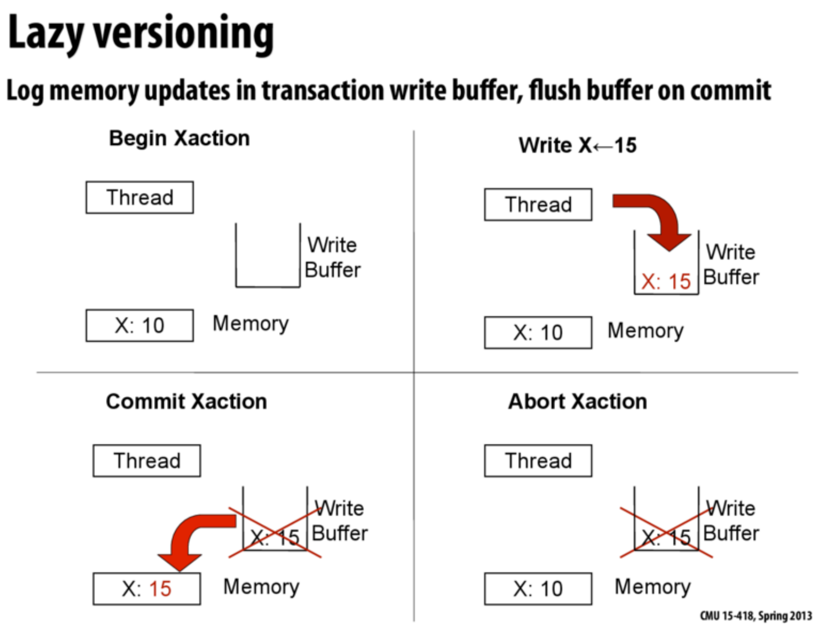


Figure 3.2: A simple example of lazy versioning.

Generally, commits are faster in eager versioning since new (modified) data are already stored in memory. On the other hand, rollback is faster (faster aborts) in lazy versioning as it just discards the write-buffer, while in eager versioning the committed (old) data has to be copied from the undo log to memory. Furthermore, in lazy versioning each store requires only one write to buffer, whereas in eager versioning it requires a write to memory as well as to the undo log. Finally, in case of a crash during transaction, memory will be in an inconsistent state in eager versioning, in contrast to lazy versioning that handles faults in a better way, since the memory is in a consistent state during transaction.

- **Conflict detection**

  Conflicts during transactions must be detected and handled properly to ensure correctness. There are two types of conflicts: read-write conflict and write-write conflict. A read-write conflict appears when a transaction reads an address, which was written to by another pending transaction. Similarly, a write-write conflict appears when two (or more) pending transactions write to the same address in memory. To achieve conflict detection the system keeps track of each transaction's read set and write set, which are the addresses read from or written to in each transaction. There are two policies for conflict detection:

  - *Pessimistic detection*

    This policy is also known as eager conflict detection. It attempts to detect conflicts early, as soon as a load or a store is requested. If a conflict is detected, then the contention manager (contention manager is responsible for making transactions look as if they are sequentially executed) either aborts the transaction, or stalls one of the transactions until the other completes. There are various priority policies to determine which transaction gets priority and handle common case fast. Figure 3.3* shows some pessimistic detection examples.

  - *Optimistic detection*

    This policy is also known as lazy conflict detection. It attempts to check conflicts only at commit time. Before committing, the write-set is communicated to all other pending transactions in order to check conflicts. Usually, on a conflict, the committing transaction has priority and other transactions may abort later on. Figure 3.4* depicts some optimistic detection examples.

  Finally, there are hybrid policies that use optimistic and pessimistic schemes together. For example, several STM systems use optimistic policy for reads and pessimistic for writes. Comparing the two policies, in pessimistic conflict detection, there is no forward progress guarantee and it may lead to a livelock (Figure 3.3 Case 4).

* Image taken from http://15418.courses.cs.cmu.edu/spring2013/article/40.

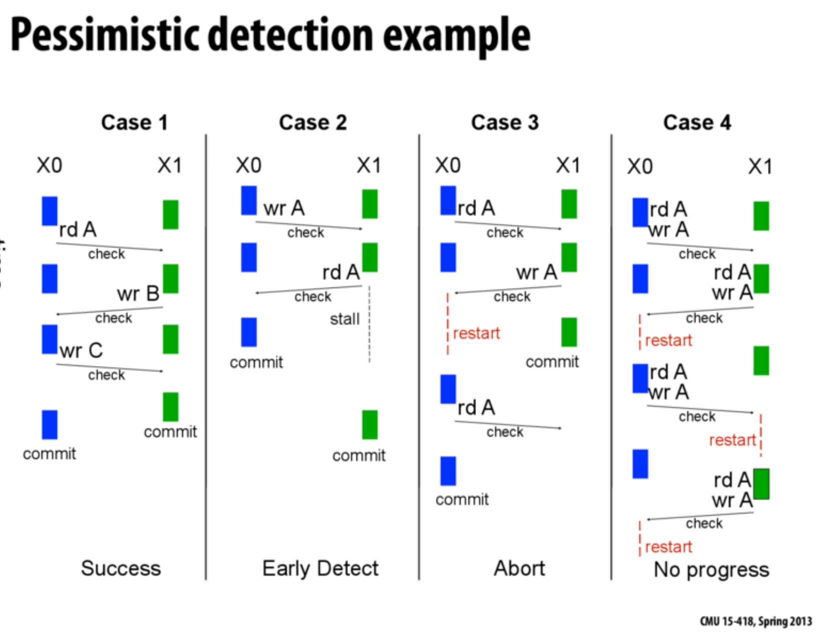


Figure 3.3: A discussion of pessimistic detection.

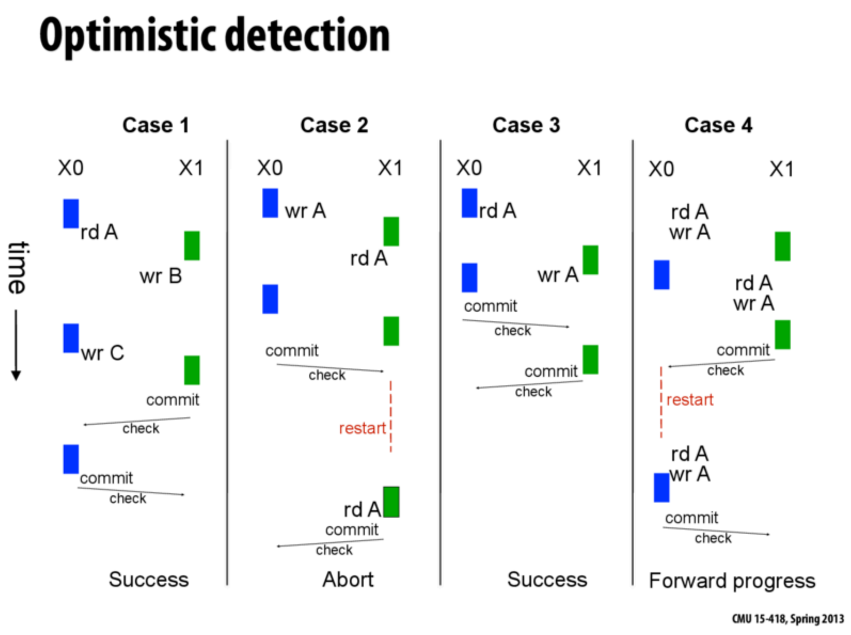


Figure 3.4: A discussion of optimistic detection.

- **Conflict resolution**

  The conflict is resolved when the underlying system take some action to avoid conflicts (e.g stall or abort one of the conflicting transactions). Eager conflict detection must resolve the conflict as soon as the transaction requests a load or a store that conflicts with one or more other pending transactions. The resolution policy can stall the transaction, abort the transaction, or abort others. Lazy conflict detection must resolve the conflict as soon as a transaction, that conflicts with one or more transactions, attempts to commit. The resolution policy can abort all others, stall or abort the committing transaction.

- **Isolation**

  As defined in [20] isolation requires that execution of a transaction does not affect the result of concurrently executing transactions. *Strong isolation* implies that transactional blocks are isolated from other transactional blocks and from concurrent non-transactional accesses. This means that a conflict is detected even if the conflicting access occurs in a non-transactional code. On the other hand, *weak isolation* implies that transactions are isolated only from other transactions. Therefore, in a system with weak isolation a non-transactional read may see the state of an incomplete transaction and a non-transactional write may appear to occur in the middle of a transaction.

- **Granularity**

  Transaction granularity is the unit of storage over which transactional memory system detects conflicts. There are three alternatives for transaction granularity, object granularity, word granularity and cache line granularity. *Object granularity* detects a conflicting access to an object even if the transactions referenced different fields. *Word granularity* is also known as *block granularity* and detects conflicting accesses to a memory word or adjacent (fixed-size group of words). And *cache line granularity* detects conflicting accesses to a cache line even if transactions modify disjoint parts of a cache line. Most HTM systems detect conflicts at cache line granularity, while most STM systems operate on an object granularity.

- **Best effort**

  This characteristic appears only in real HTM implementations. Using only transactional mode, no forward progress is guaranteed. A transaction may always fail to commit (Figure 3.3 Case 4) and therefore a non-transactional fallback path is necessary.

- **Conflicts**

  A transaction may fail to commit (abort) because of different reasons of conflicts. In STM systems a transaction aborts due to data conflicts, while in HTM systems a transaction may abort for several reasons like data conflicts, capacity aborts, explicit aborts and interrupts.

– *Data conflict*: This conflict appears when two or more threads perform conflicting operations to the same data. For example, when another process/thread writes to a memory location that has been added to the transaction's read or write set.

– *Capacity abort*: Transactional buffers of TM system have a fixed size for each process/thread. Thus, transaction's read and write set have limited capacity. When a transaction exceeds the maximum (write or read) buffering capacity imposed by the TM implementation, the transaction fails to commit due to a capacity abort.

– *Explicit abort*: This type of abort occurs when the programmer explicitly aborts the transaction. For instance, in real HTM implementations that use best effort, the code in the fallback path includes acquiring a global lock to protect the critical section. Thus, an explicit abort is performed at the beginning of a transaction if this lock is checked and found to be taken.

– *Other*: A transaction may abort due to several other reasons including interrupts, unsupported instructions, system calls e.t.c.

3.2.1 Real TM implementations

The following examples constitute real TM implementations.
HTM implementations

- Lazy + optimistic: Stanford TCC
- Lazy + pessimistic: MIT LTM, Intel VTM, Sun’s Rock
- Eager + pessimistic: Wisconsin LogTM

STM implementations

- Lazy + optimistic (rd/wr): Sun TL2
- Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
- Eager + optimistic (rd)/pessimistic (wr): Intel STM
- Eager + pessimistic (rd/wr): Intel STM

3.3 Intel's Haswell HTM

As already mentioned, there are different TM implementations which combines the above characteristics and manage differently the transactions. For example, Sun’s Rock combines lazy versioning with pessimistic conflict detection and Stanford TCC uses lazy versioning and optimistic conflict detection. In this section we will further analyze an HTM implementation that Intel provides in Haswell processors. Intel announced that its Haswell architecture would include hardware support for transactional memory in 2013 and Haswell became the first x86 processor to feature hardware transactional memory.

Haswell's HTM implementation uses lazy data versioning, pessimistic (eager) conflict detection, operates on cache line granularity (64 bytes), provides strong isolation and is a best-effort implementation.

The programmer have to mark the block of code which have to be executed as a transaction. Thus, the ISA (Instruction Set Architecture) of the processor has been extended with a set of instructions that allows programmer to use the HTM infrastructure. While this block of code is executed as a transaction, the system is in transactional mode. In transactional mode the memory addresses that the transactional code accesses, are transferred in cache memory and are separated in two sets, the read and the write set (consist of all memory addresses in which the transaction reads and writes, respectively). Each cache line in L1D and L2 cache contains bits that indicate whether the line belongs to the read set or the write set. Any transactional data must stay in L1D cache (or L2) and not be evicted to the L3 or memory, until the transaction commits.

While the transaction is executed, possible conflicts, that may arise from the parallel execution of different processes/threads, can be detected. This conflicts can be detected using cache coherence protocol (e.g MESIF). Briefly, when a process/thread that runs in a processor, reads or writes to a memory address, cache coherence protocol is responsible to communicate with other processors that have already stored in their L1D cache this memory address and inform them for the modification. Furthermore, since the cache coherence protocol informs immediately other processors for modifications, there is no need for additional communication among processes/threads that run concurrently, something that would limit the memory bus bandwidth.

If during transaction a conflict is detected, the transaction aborts and the HTM implementation have to roll back the current transaction, discard the modifications made by the transaction and revert the system to the previous initial state as if the transaction had never begun. This is simple for Haswell's HTM implementation, as all modifications made by the transaction are stored in L1D cache (and/or L2) and the main memory of the system has not been updated. As a result, the underlying TM system invalidates all the transactional cache lines of L1D (and/or L2) cache and main memory remains immutable. However, if a transaction attempts to commit and no conflict has been detected, the underlying system has to transfer the modified transactional cache lines from cache to main memory.

Finally, there is an important limitation in Haswell's HTM implementation. The cache detects concurrent accesses at a cache line granularity. For instance, the case of two threads that write to adjacent elements of a large array will often cause the transaction to fail (abort), because the hardware cannot distinguish between two accesses to the same address, and two accesses to different addresses in the same cache line.

3.3.1 Transactional Synchronizations Extensions (TSX)

In order to implement HTM in Haswell processors, its instruction set architecture (x86) has been extended with a set of instructions to ease the development and improve the performance of existing programming models. In this system it is more convenient the use of C or C++ programming language. Haswell's transactional support, which Intel is calling Transactional Synchronization Extensions (TSX) provides two software interfaces. The first, called Hardware Lock Elision (HLE) allows easy conversion of lock-based programs into transactional programs in a way that is compatible with current processors. The second, called Restricted Transactional Memory (RTM) is a more complete transactional memory implementation that allows programmers to define transactional regions in a more flexible manner than is possible with HLE. RTM also requires programmer to provide an alternate code path (fallback path) in case that transactional execution is not successful, since the hardware provides no guarantees as to whether an RTM region will ever successfully commit transactionally.

These extensions can help achieve the performance of fine-grained locking synchronization through a coarse-grained locking implementation in the code. Moreover, these extensions allow locks around critical sections and perform serialization of parallel executions of critical sections only when this is necessary. Multiple processes/threads that execute critical sections and do not perform any conflicting operations can proceed simultaneously without serialization. Although the software uses a global lock to protect critical sections, the hardware is allowed to recognize that processes/threads do not interfere with one another.

The main difference between HLE and RTM is that software written using HLE can run both on legacy hardware without TSX (in this case the critical section is executed directly in lock mode) and new hardware with TSX, while software written in RTM cannot run in a processor that does not support TSX. However, RTM offers more flexibility concerning the actions that can be done after a transactional abort. The programmer defines a memory address that points out the code that will be executed in case of abort (fallback handler). To employ TSX the programmer must have a complier gcc-4.8x (or a later version) and include in his program the library "immintrin.h". Otherwise, in case of an older version of gcc, the programmer must include the library "rtm.h".

Hardware Lock Elision (HLE)

Hardware Lock Elision is a simple way of deploying transactional memory in existing code. The idea of HLE is to remove locks and let CPU worry about consistency. Instead of assuming that a process/thread always protect the shared data from other threads, it can be assumed that the other processes/threads will not overwrite the variables that the current process/thread is working on (in the critical section). If another process/thread overwrites one of those shared variables, the whole process will be aborted by the CPU, and the transaction will be re-executed with a traditional lock.

HLE introduces two new instruction prefixes, named XACQUIRE and XRELEASE that are used to denote the bounds of critical section. XACQUIRE is a prefix for instructions that acquire a lock and it indicates the start of critical section (region for lock elision). When a process/thread acquires a lock with an XACQUIRE instruction, the lock is not actually acquired. The write operation is ignored, but the memory address of the lock instruction is added to the read set of the transaction, so the transaction will fail if something else writes to that address. The process/thread then enters transactional execution and continues on to the instructions inside the critical section, adding memory addresses to transaction's read and write set. The current process/thread will still think it has obtained the lock, but many processes/threads will be allowed to run simultaneously and make non-conflicting accesses to shared data.

Execution continues until the XRELEASE instruction. XRELEASE is a prefix that is used for the instruction that releases the lock address, and it marks the end of the critical section. When the processor reaches XRELEASE instruction, it attempts to commit the transaction. If it succeeds, the critical section was executed without acquiring or releasing the lock (none of the memory operations conflicted). If the transaction fails (when a conflict occurs), the processor will restore the architectural register state prior to XACQUIRE and discard any writes from the critical section. The process/thread will execute the critical section again, with the standard pessimistic locking behavior. In this case, it actually acquires the global lock. So the programmer can use coarse-grained locking as a "fall back" solution, and HLE can exhibit as much parallelism as is present in the access patterns, not in the locking designs.

As already mentioned, the software written using HLE can also run to hardware without TSX. The system is backwards compatible. The programmer can use the new TSX enabled library and gets the benefits of TSX if his program is executed on Haswell or a later Intel CPU. Every other processor will ignore the prefix and just operate on the lock, the traditional lock-based behavior. The prefixes of the instructions XACQUIRE and XRELEASE are treated as nops.

The listing 3.1 presents an example of elision of a TAS lock:

Listing 3.1: Example: elision of a TAS lock

```
/* Traditional lock implementation */
/* acquire lock */
while(__sync_lock_test_and_set(&lock_var) == 0)
 /* do nothing */;
... Critical section with lock acquired ...
/* release lock */
__sync_lock_release(&lock_var);


/* HLE implementation */
/* elide lock */
while(__hle_acquire_test_and_set(&lock_var) == 0)
/* do nothing */;
... Critical section with lock acquired ...
/* release lock */
__hle_release_clear(&lock_var);
```

Restricted Transactional Memory (RTM)

Restricted Transactional Memory (RTM) is an alternative implementation to HLE which gives the programmer the flexibility to specify a fallback code path that is executed when a transaction cannot be successfully executed. There are four new instructions, XBEGIN, XEND, XTEST and XABORT. The programmer marks the block of code that he wants to be executed atomically (critical section) using the instructions XBEGIN and XEND. XBEGIN and XEND mark the start and the end of the critical section, respectively. When the process/thread reaches the XEND instruction in the code, the transaction commits and the memory is updated according to the modifications made by the transaction. The instruction XTEST returns 1 if the process/thread is in transactional mode, otherwise it returns 0 and with XABORT(status) instruction the programmer can explicitly abort the transaction (as if a commit have been unsuccessful). The status is used to indicate the reason for transactional abort. An explicit abort instruction is useful when the programmer can determine that a transaction is going to fail, without any help from the hardware. Aborting the transaction early can also help reduce the power penalty.

If a conflict occurs during transaction, it may trigger an abort. After a transactional abort the fallback handler is responsible for the instruction that the process/thread will execute to resume the execution. The programmer defines the memory address of the code to be executed in case of transactional abort (fallback address). The fallback address is exactly the next instruction after XBEGIN. XBEGIN returns a value that indicates if the process/thread is in transactional mode or if the transaction has been aborted. As a result, the EAX register is updated according to the transaction's status (Figure 3.5). The programmer can check if the transaction has started or if it has been aborted performing a logical calculation and between the return value of XBEGIN and the following constants:

- _XBEGIN_STARTED: Transaction has successfully begun.
- _XABORT_CONFLICT: Transaction abort due to a memory conflict with another thread.
- _XABORT_CAPACITY: Transaction abort due to the transaction using too much memory.
- _XABORT_EXPLICIT: Transaction was explicitly aborted with _xabort. The parameter passed to _xabort is available with _XABORT_CODE(status).
- _XABORT_RETRY: Transaction retry is possible.
- _XABORT_DEBUG: Transaction abort due to a debug trap.
- _XABORT_NESTED: Transaction abort in an inner nested transaction.

Some causes of abort may always result to transactional abort. As a result, each time we execute the critical section in transactional mode using the HTM implementation, the transaction always fails and there is no progress in our program. For example, if transaction's read and write set exceed the size of cache, the transaction will always abort because of capacity aborts. In this case the programmer should use a back-off mechanism like a fallback path. The fallback path is an alternative implementation in the code of the program that does not employ RTM and is most likely a piece of code that does coarse-grained

EAX register bit position	Meaning
0	Set if abort caused by XABORT instruction.
1	If set, the transaction may succeed on a retry. This bit is always clear if bit 0 is set.
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted.
3	Set if an internal buffer overflowed.
4	Set if debug breakpoint was hit.
5	Set if an abort occurred during execution of a nested transaction.
23:6	Reserved.
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved).

Figure 3.5: Transaction's status is captured to EAX register's bits.

locking. After a specified number of aborts in a transaction the programmer can choose the execution of the fallback path instead of transaction. This implementation is necessary to guarantee progress in program's execution.

The fallback path is usually implemented as coarse-grained locking code that uses a global lock. In this case this global lock has to be added to transaction's read set. If the global lock does not be added to transaction's read set, it results to coherence problems. Figure 3.6 presents an example of two threads that attempt to execute the same code concurrently. The first thread enters the critical section acquiring the global lock and the second thread using RTM implementation. In this example the second thread will not detect any conflict and its transaction will commit updating the value of the "count" variable, while the first thread will not be informed about this modification. This constitutes a coherence problem. As a consequence, the programmer is responsible to add the global lock to transaction's read set. To achieve this the value of the global lock must be read without locking the global lock. If at the beginning of a transaction the global lock is used from another thread, the programmer must explicitly abort the transaction (explicit abort). Listing 3.2 is an example of adding a pthread_spinlock in transaction's read set, since the code reads its value, and in case that this is not free the transaction explicitly aborts via XABORT instruction.

Listing 3.2: Example: adding global lock to transaction's read set

```
if ((int)spin_lock != 1)
            _xabort();
if (pthread_mutex_t.__data.__lock != 0)
            _xabort ();
```

Finally, Listing 3.3 describes an RTM example in C programming language.

Listing 3.3: An RTM example

```
int aborts = MAX_TX_RETRIES;
lock_t = fallback_global_lock;
```

```
start_tx:
    int status = TX_BEGIN();
    if(status == TX_BEGIN_STARTED){
        if (fallback_global_lock is locked)
            TX_ABORT();
    ... Critical Section ...
    TX_END();
    }else{ /* status != TX_BEGIN_STARTED */
        if (--aborts > 0)
            /* retry transaction */
            goto start_tx;
        acquire_lock(fallback_global_lock);
        ... Critical Section ...
        release_lock(fallback_global_lock);
    }
```

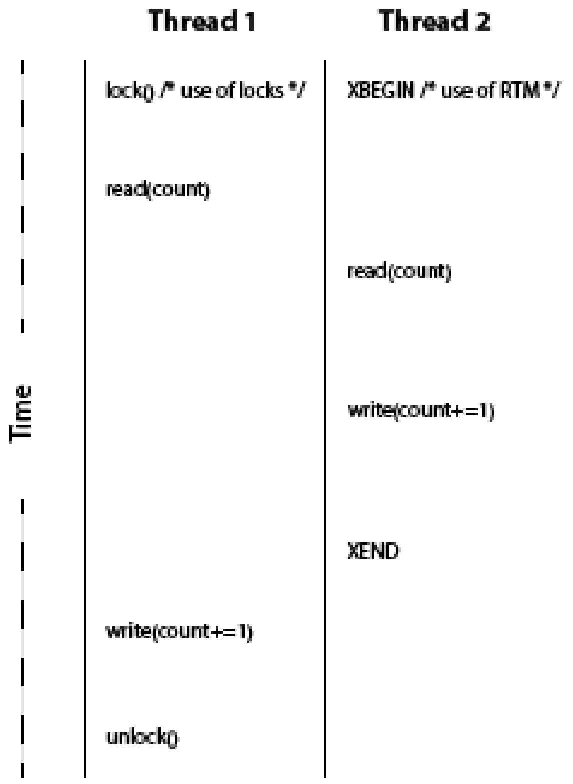


Figure 3.6: A parallel execution of two threads using RTM that results to coherence problems.

Chapter 4

A parallelization of Dijkstra's algorithm

4.1 Dijkstra's algorithm

4.1.1 Algorithm's history

Dijkstra's algorithm (also known as shortest path algorithm) is a fast algorithm for finding the shortest paths between nodes in a graph, which may represent a road network. It was conceived by a Dutch computer scientist from Netherlands Edsger Wybe Dijkstra (May 11, 1930 – August 6, 2002) in 1956 and published three years later [21]. Edsger Wybe Dijkstra is also known for his many essays on programming and received the A. M. Turing Award (widely considered the most prestigious award in computer science) in 1972.

As the history of shortest paths algorithms shown in figure 4.1 discloses, Dijkstra's algorithm is a simpler and faster version of Ford's algorithm. Wikipedia describes that Dijkstra thought about this algorithm when working at the Mathematical Center in Amsterdam in 1956 as a programmer to demonstrate capabilities of a new computer called ARMAC. His main purpose was to present both a problem as well as an answer, that would be produced by computer, that people could understand. He designed the shortest path algorithm and implemented it for ARMAC computer for a slightly simplified transportation map of 64 cities in Netherlands. A year later, he came across another problem from hardware engineers working on the institute's next computer: minimize the amount of wire needed to connect the pins on the back panel of the machine. As a solution, he re-discovered the algorithm known as Prim's minimal spanning tree algorithm [22] and published his algorithm in 1959, two years after Prim.

The algorithm exists of many variants. Dijkstra's original algorithm finds the shortest path between two nodes, but the most popular variant of this algorithm fixes a single node as the "source" node and finds the shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. Furthermore, in some fields (artificial intelligence)

Dijkstra's algorithm or a variant of it is known as uniform-cost search and formulated as an instance of the more general idea of best-first search.

Shimbel (1955)	Information networks.
Ford (1956).	RAND, economics of transportation.
Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, Seitz (1957).	Combat Development Dept. of the Army Electronic Proving Ground.
Dantzig (1958).	Simplex method for linear programming.
Bellman (1958).	Dynamic programming.
Moore (1959).	Routing long-distance telephone calls for Bell Labs.
Dijkstra (1959).	Simpler and faster version of Ford's algorithm.

Figure 4.1: Early history of shortest paths algorithms.

4.1.2 Description

Dijkstra's algorithm is a greedy algorithm that solves the single-source path problem when all edges have non-negative weights. This is asymptotically the fastest known single-source shortest-path algorithm for graphs with unbounded non-negative weights. The algorithm is based on the observation that any subpath of any shortest path is itself a shortest path (optimal substructure). Extending this idea we observe the existence of a shortest path tree in which the distance from source to vertex v is the length of shortest path from source to vertex in original tree. The length of a path $p = (v0, v1, ..., vk)$ is the sum of the weights of its constituent edges: $length = \sum_{i=1}^{k} w(v_{i-1}, v_i)$, where the function $w : E \to \mathbb{R}$ maps edges to the real-valued weights.

Intuitively, the algorithm reports the vertices in increasing order of their distance from the source vertex. Exploring a new vertex means exploring the vertex that has the smallest distance. This is why the algorithm uses the distance from the source to the vertex as the priority. Secondly, the algorithm constructs the shortest path tree edge by edge. At each step adding one new edge corresponds to the construction of shortest path to the current new vertex. The new edge is added to the shortest path to the current new vertex, if the new path from the source to the vertex is shorter than the previous distance from the source to vertex. The process by which an estimate of the distance from source to vertex is updated is called **relaxation**.

For a graph, $G = (V, E)$ where V is a set of vertices and E is a set of edges. Dijkstra's algorithm keeps two sets of vertices: S the set of vertices whose shortest paths from the source have already been determined and $V \backslash S$ the remaining vertices (unvisited set). The algorithm in steps is:

1. Set S to empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE (∞) and the distance value as 0 for the source vertex.

3. While there are still vertices in $V \backslash S$ (unvisited)
    (a) Choose an unvisited vertex u from $V \backslash S$ that has the minimum distance value from the source.
    (b) Include vertex u to the set S.
    (c) Relax all adjacent vertices of u that are still in $V \backslash S$. To achieve this, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge $u - v$, is less than the distance value of v, then update the distance value of v.

The listing 4.1 is a pseudocode for Dijkstra's algorithm using priority queue. The Q set is the set of unvisited vertices ($V \backslash S$) that is implemented as a priority queue and the arrays dist, prev have the distance from the source to a vertex v and the previous node of vertex v in the optimal path from source, respectively. The function add_with_priority() adds an element to the queue with an associated priority (minimum distance from source), the function extract_min() removes the element from the queue that has the highest priority (more specifically the vertex with the minimum distance from source), and return it, and the function decrease_priority() updates the priority of an element in the queue.

Listing 4.1: Dijkstra's algorithm

```
function dijkstra(graph, source):
        dist[source] ← 0 //Initialization

        create vertex set Q //Set of unvisited vertices

        for each vertex v in Graph:
                if v ≠ source
                dist[v] ← INFINITY //Unknown distance from source to v
        prev[v] ← UNDEFINED //Predecessor of v

                Q.add_with_priority(v, dist[v])


        while Q is not empty: //The main loop
                u ← Q.extract_min() //Remove and return best vertex
                for each neighbor v of u: //Only if v that is still in Q
                        sum = dist[u] + length(u, v)
                        if sum < dist[v] //A shorter path has been found
                                dist[v] ← sum
                                prev[v] ← u
                                Q.decrease_priority(v, sum)

        return dist[], prev[]
```

4.1.3 Complexity

The simplest implementation of the algorithm stores the vertex set as an ordinary linked list or array. Extract_min() takes $\mathcal{O}(V)$ time and there are |V| such operations. Therefore, a total time for extract_min() in while loop is $\mathcal{O}(V^2)$. Since the total number of edges in all the adjacency list is |E|, the for loop iterates |E| times with each iteration taking $\mathcal{O}(1)$ time. Hence, the complexity of the algorithm with an ordinary linked list or array implementation is $\mathcal{O}(V^2 + E) = \mathcal{O}(V^2)$.

For sparse graphs (graphs with fewer edges), the algorithm can have a better complexity by storing the vertex set in the form of adjacency lists and using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue. This results to a more efficient implementation of extract_min() function. Extract_min() takes $\mathcal{O}(\log V)$ time and there are |V| such operations. The function decrease_priority() (or decrease_key()) takes $\mathcal{O}(\log V)$ in case of a self-balancing binary search tree or a binary heap and $\mathcal{O}(1)$ in case of a fibonacci heap for each of the |E| edges. Thus, the running time of the algorithm with self-balancing binary search tree or binary heap provided is $\mathcal{O}((E + V)\log V)$ and with fibonacci heap provided is $\mathcal{O}(E + V\log V)$.

4.1.4 Proof of correctness

Lemma: When a vertex u is added to S set (visited nodes), then $dist[u] = [s, u]$, where $[s, u]$ the length of the shortest path from the source to the vertex u.

Proof: Suppose that the algorithm first attempts to add a vertex u to the set S for which $dist[u] \neq [s, u]$. Then, $dist[u] > [s, u]$.

Consider the shortest path (Figure 4.2) from source s to vertex u ($s\epsilon S$ and $u\epsilon V\backslash S$). Let (x,y) be the edge taken by the path, where $x\epsilon S$ and $y\epsilon V\backslash S$ (it may be that $x = s$ and/or $y = u$).

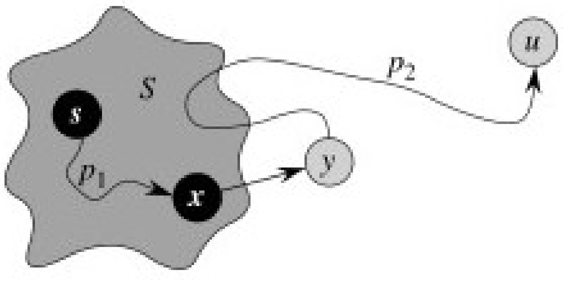


Figure 4.2: Proof of corectness of Dijkstra's algorithm. When a vertex u is added to S set (visited nodes), then $dist[u] = [s, u]$.

Having done the relaxation in vertex x we can conclude that

$$dist[y] \leq dist[x] + w[x, y], \tag{4.1}$$

where the function $w : E \to \mathbb{R}$ maps edges to the real-valued weights.

By hypothesis x is in the set S, so:

$$dist[x] = [s, x]. \tag{4.2}$$

Since <s,...,x,y> is a subpath of a shortest path, by 4.2

$$(s,y) = (s,x) + w(x,y) = dist[x] + w(x,y). \tag{4.3}$$

By 4.1, 4.3

$$dist[y] <= (s,y).$$

Therefore,

$$dist[y] = (s,y).$$

So $y \neq u$, as we suppose that $dist[u] > (s,u)$.

As a result,An example of a graph.

$$dist[y] = (s,y) < (s,u) <= dist[u].$$

Thus, y would have been added to S set before u, since it has a smaller estimate of the distance from the source. This contradicts with the assumption that u is the next vertex to be added to the set S.

By the lemma, $dist[u] = (s,u)$ when is added to the set S and at the end of the algorithm, all vertices are in the set S and all distance estimates are optimal.

4.1.5 Applications

As described above, the algorithm finds the shortest path between a node and every other. It can be also used for finding the shortest path from a single node to a single destination by stopping the algorithm when the shortest path to the destination node has been determined. For instance, in a road network, supposing that the cities are represented as the nodes of the graph and driving distances between pairs of cities connected by a direct road are represents as edges paths, Dijkstra's algorithm can find the shortest route between one city and all other cities. Thus, the shortest path algorithm is a widely useful problem-solving model used in network routing protocols, VLSI design, social networks and TeX typsetting. Figure 4.3 presents some applications of Dijkstra's algorithm.

4.2 The concept of Helper Threads

Helper threads is an optimization technique used in non traditional parallelism to accelerate a program and provide performance speedups. Helper threads are "assist" threads that perform certain critical computations on behalf of a main thread in order to help the main ("master") thread and reduce its tasks. Typically, this optimization has been exploited either to prefetch future data accesses or to precompute the outcome of blocks of code that would otherwise be executed by the main thread.

To improve the performance of an application program using the concept of helper threads, there are several key issues that need to be taken into consideration. First, in

Maps
Robot navigation
Texture mapping
Typesetting in TeX (e.g LaTeX)
Urban traffic planning
Optimal pipelining of VLSI chip
Subroutine in advanced algorithms
Telemarketer operator scheduling
Routing of telecommunications messages
Approximating piecewise linear functions
Network routing protocols (OSPF, BGP, RIP)
Exploiting arbitrage opportunities in currency exchange
Optimal truck routing through given traffic congestion pattern

Figure 4.3: Applications of Dijkstra's algorithm.

hyper-threaded processors some structures are shared or partitioned in between logical processors in multi-threading mode, and thus, resource contention can be an issue. As a consequence, helper threads can be invoked judiciously to avoid potential performance degradation due to the increased resource contention. Second, the program behavior changes dynamically, and hence helper thread invocation should be adaptable. For instance, a particular load might experience a significant number of cache misses over the total program execution, but the temporal distribution of those misses might not be uniform. As a result, a helper thread should be able to detect the dynamic program behavior at runtime. Finally, to adapt to the dynamic behavior, helper threads need to be activated and synchronized frequently. Thus, a low overhead thread synchronization mechanism is needed. Compared to traditional multi-threading technique where each tread should be executed in a pre-defined order to guarantee the correctness of the program, helper threads only affect the performance speedup of the program. Accordingly, a helper threads can be deactivated whenever it does not improve the performance of the main thread. Finally, dynamic program behaviors can be effectively captured at runtime and various dynamic optimizations can be applied.

4.3 Parallelizing Dijkstra's algorithm

4.3.1 Introduction

This section describes a parallelization of Dijkstra's algorithm presented in the papers [23], [24]. As explained in these papers, dijkstra's algorithm is based on the iterative extraction of nodes (vertices) from a priority queue. This property limits the explicit parallelism of the algorithm and any attempt to utilize the remaining parallelism results to performance degradation due to synchronization overheads. Thus, the two major issues inherent to the algorithm is the limited explicit parallelism and excessive synchronization. To deal with this problems the authors of the papers employed the concept of Helper Threads (HT) to extract more parallelism and Transactional Memory (TM) as a means of concurrent accesses to shared data structures.

The authors of [23], [24] chose the idea of Helper Threads to coarsen the granularity of parallelism. The key idea is that helper threads will offload operations from the main thread. More specifically, the main thread performs many relaxations of the nodes of the priority queue. Therefore, parallel helper threads can simultaneously relax the distances of several nodes. While the main thread extracts and updates the neighbors of the head of the priority queue, k helper threads can update the neighbors of the next k nodes in the priority queue in order to offload operations of the main thread in its next iteration.

Finally, TM system is a promising approach for dynamic data structures and applications with independent threads providing performance gains. The programmer is able to envelop blocks of code within a transaction, indicating that within this segment of the code exist accesses to memory addresses that may be performed by other threads as well. The TM system monitors the concurrent transactions of the threads and if two or more perform conflicting accesses, it resolves the conflict. In the case of non-conflicting accesses, TM systems perform the appropriate accesses with no overhead.

4.3.2 The algorithm

The algorithm exploits the basic property of Dijkstra's algorithm: the relaxations result to monotonically decreasing values for the distances of unvisited nodes until each distance reaches its final optimal (minimum) value. When a node is inserted in the queued set (its distance from the source is no longer infinite) its neighbors could also be relaxed to newer updated values. The original algorithm does not take into consideration this property and avoids computing intermediate distances that will be overwritten by updating only the neighbors of the extracted node. The idea is that Helper Threads can relax neighbors belonging to the queued set. Some of these relaxations will be offloaded by the main thread.

Helper threads perform relaxations to the top k positions in the queue and the corresponding nodes might have already obtained their optimal distance from source with some probability. Therefore, when helper threads read their distances and relax their outgoing edges, the corresponding neighbors related with these outgoing edges may obtain their optimal distance from source, as well. As a result, when in the next iteration the main thread checks these nodes (vertices), it will not perform any relaxations. On the other hand, a helper thread may perform a relaxation to a node that has not obtained its optimal (minimum) distance yet. In this case the node will eventually be set to its optimal minimum distance, when it will be examined by the main thread later on.

The main thread operates like in the sequential version. In each iteration it extracts the minimum vertex from the priority queue and performs its relaxations. At the same time, the k-th helper thread reads the tentative distance of the k-th vertex in the queue and attempts to relax its outgoing edges according to this value. When the main thread finishes all its relaxations, it notifies helper threads to stop their relaxations, and they all proceed to the next iteration. This scheme is demonstrated in Figure 4.4 [*].

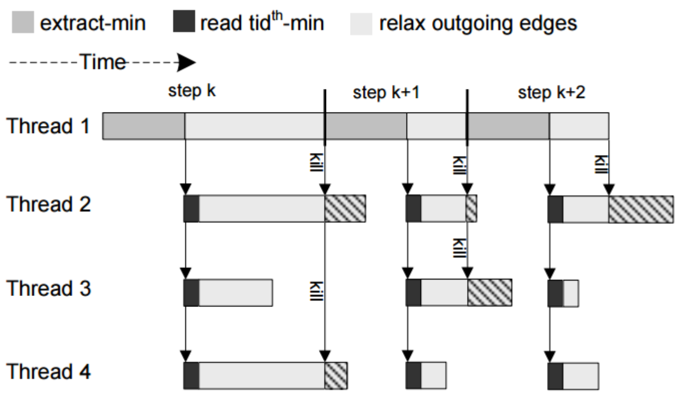


Figure 4.4: Execution pattern of the HT scheme.

In case that helper threads are forced by the main thread to stop their computations and proceed with it to the next iteration, it is possible that at this time a helper thread might have updated only some of the neighbors of its vertex, leaving the rest neighbors with their old distances. Nevertheless, this is not a problem since all neighbors of this vertex will eventually obtain their optimal distances when the vertex reaches at the top of the priority queue.

The code executed by the main and helper threads is shown in listings 4.2 and 4.3, respectively. In each iteration, the main thread extracts the vertex with the high priority (minimum distance from source) from the priority queue. At the same time, helper threads wait (spinning in a while loop) until the main thread finishes its extraction. Subsequently, each helper thread reads (without extracting) one of the top k vertices in the queue. This is done by ReadMin() function. In the next step, all threads, both the main and helper threads, perform the appropriate relaxations related with the outgoing edges of the vertices they have taken over. As explained above, helper threads offload relaxations of the main thread and thus, it will evaluate the expression of line 7 in Listing 4.2 as true fewer times and will not need to perform the operations of lines 8-10.

The proposed scheme should provides atomicity because a conflict can arise when two or more threads update concurrently the same neighbor, or update different neighbors but change the same part of the queue. To achieve atomicity updates to the queue via the DecreaseKey() function, as well as updates to the shared distance and predecessor arrays (d[], p[] respectively) are enclosed within a single transaction for both main and helper

[*] Image taken from [24].

threads. In this way, when a conflict arises, only one thread will be allowed to proceed, commit the transaction and perform the update to the queue, while the rest will have to repeat their work.

As already mentioned, when the main thread finishes its relaxations, it notifies helper threads to stop and proceed all to the next iteration. To implement this, the algorithm employs transactional memory (TM). More specifically, when the main thread completes the iteration of the inner loop for relaxations (line 4), it sets the notification variable "done" to 1. This means that the main thread will proceed to the next iteration for the next vertex and it also forces all helper threads to stop and follow, terminating their computations that they were performing on the queue. Since helper threads are in transactional mode and "done" variable is in their read sets, they will abort and they will retry the transaction. However, when helper threads will attempt to perform a new transaction for their work, they will find, with some strong probability, the "done" variable set to 1 and then they will stop their relaxations for the remaining neighbors in the inner loop and will proceed to the next iteration of the outer loop. If the main thread performs the ExtractMin() function too quickly and "done" variable will set back to 0, helper threads will miss the last notification, continuing from the point where they have stopped. This does not affect the guarantee of correctness. Although helper threads may update the distances of the neighbors with a suboptimal value, these will be overwritten with the optimal value when the vertices examined by helper threads reach at the top of the priority queue.

Finally, the main purpose of the algorithm is to employ helper threads only to offload work of the main thread and not to interfere in main thread's progress. Furthermore, this scheme attempts to minimize the time spent on synchronization events and transactional aborts. Helper threads perform operations on the queue, intruding at the same time as less as possible on main thread's work, even if they do not perform useful work. By using the underlying TM system there should exist a conflict resolution policy that favors the main thread and minimizes its transaction abort overheads.

Listing 4.2: Main thread's code.

```
 1  while Q not empty do
 2          u ← ExtractMin(Q);
 3          done ← 0;
 4          foreach v adjacent to u do
 5                  sum ← d[u] + w(u, v);
 6                  Begin−Transaction
 7                  if d[v] > sum then
 8                          DecreaseKey(Q, v, sum);
 9                          d[v] ← sum;
10                          p[v] ← u;
11                  End−Transaction
12          end
13          Begin−Transaction
14          done ← 1;
```

```
15        End−Transaction
16  end
```

Listing 4.3: Helper threads' code.

```
while Q not empty do
    while done = 1 do ;
    x ← ReadMin(Q, tid);
    stop ← 0;
    foreach y adjacent to x and while stop = 0 do
        Begin−Transaction
        if done = 0 then
            sum ← d[x] + w(x, y);
            if d[y] > sum then
                DecreaseKey(Q, y, sum);
                d[y] ← sum;
                p[y] ← x;
        else
            stop ← 1;
        End−Transaction
    end
end
```

4.3.3 Optimizations

The authors of the papers [23] and [24] evaluated this algorithm in a full-system simulation. On the contrary, we evaluated the proposed algorithm in a real HTM system (Intel's Haswell HTM). Therefore, in order to achieve performance speedup as the number of cores increases we have applied some optimizations on the algorithm. More specifically:

- Since the basic characteristic of real HTM systems is strong isolation, there is no need to set "done" variable within a separate transaction (lines 13-15, listing 4.2). In strong isolation a conflict can be detected even if the conflicting access occurs in non-transactional code. Thus, removning this transaction (line 42 listing 4.4) we reduce the number of transactions and avoid additional cost of unnecessary transactions.

- Instead of "stop" variable used in helper threads' code to exit the inner (for) loop, we explicit abort the transaction in helper threads when "done" variable is set to 1. In this case, helper threads do not perform any other relaxations as they are explicitly aborted with an abort code that indicates this reason.

- Our implementation employs a binary heap (array representation of heap) for the priority queue. A binary heap is a complete binary search tree. All levels of the tree except possibly the last one are full filled, and, if the last level of the tree is not complete the nodes of that level are filled from left to right. It also satisfies the min-heap ordering property, which states that the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root. Since the heap is a complete binary search tree, a heap with n nodes has $\mathcal{O}(\log n)$ height. As a result, the ReadMin function takes a constant time ($\mathcal{O}(1)$) and the ExtractMin() and DecreaseKey() functions take $\mathcal{O}(\log n)$ time.

- In an attempt to increase performance speedup we implemented a more coarse-grained transaction for the main thread. In the original algorithm the main thread performs one transaction for each edge examined (possible relaxation). This results to many small transactions, especially in dense graphs and as a consequence to additional overhead associated with the beginning and ending of many consecutive transactions. As a result, we examine more than one edge (perform possible relaxations a certain number of neighbors) within a separate transaction (line 21 listing 4.4). We expect that this coarse-grained approach is able to provide better performance speedup in small graphs and may result to more capacity aborts in large graphs. Therefore, we have to find a solution that trades off between the overhead of performing many transactions and the cost of many transactional (capacity) aborts.

- As explained in previous chapter false sharing is a liming factor for scalability. Different threads may modify independent parts of the structure that share the same cache line. Since real HTM systems detect conflicts at cache line granularity, the case of different threads that perform update operations (in transactional mode) in independent data that share the same cache line will cause data conflicts and one or more transactions will fail (abort). To avoid such conflicts and transactional aborts we applied structure padding to all shared structures like the priority queue, the distance array and the predecessor array, such as different elements of these structures to reside on different cache lines.

- Finally, the real HTM system used in evaluation part is a best effort implementation. Consequently, a fallback path is necessary. We employ a global lock, shared among threads, to protect the critical section. However, when a thread acquires the global lock, the rest will be aborted, as they have the global lock in their read set, and they will continue to fail until the lock is released. The purpose of the algorithm is to take advantage of the concept of helper threads such that to reduce main thread's relaxations and not to delay its progress. So, if a helper thread acquires the global lock, the main thread will always fail (transactional abort) until the release of the global lock and will not progress. The main thread should be allowed to run almost at the speed of the serial execution. To implement a policy that favors the main thread the global lock can only be acquired by the main thread. Helper threads always attempt to perform updates (relaxations) in the shared data through transactions and may always fail to commit, as no forward progress is guaranteed.

The listings 4.4 and 4.5 present our implementations in C programming language for the main and helper threads, respectively.

Listing 4.4: Main thread's code for a real HTM.

```
while(heap->curr_size > 0){

        my_min = bh_extract_min(heap);
        done = 0;

        /* Find the id of the vertex. */
        my_min_id = my_min->vertex_id;

        /* Read the key (weight) of my vertex.*/
        my_min_key = dist[my_min_id].value;

        if(my_min_key < INFINITY){

                /* adjacency list for neighbors */
                v = g->adj[my_min_id];

                if(v != NULL){
                        while(1){

                                /* Check neighbors for relaxation. */
                                begin_transaction(num_retries, &fallback_lock, tid);
                                for(i=0; i< num_neighbr; i++){

                                        distv = dist[v->id].value;
                                        sum = my_min_key + v->weight;

                                        /* Relax */
                                        if(distv > sum){
                                                decrease_key_mt(heap, v->id, sum);
                                                pred[v->id].value = my_min_id;
                                                dist[v->id].value = sum;
                                        }
                                        v = v->next;
                                        if(v == NULL)
                                                break;
                                }
                                end_transaction(&fallback_lock, counter);
                                if(v == NULL)
                                        break;
                        }
                }
                done=1;
        }
}
```

Listing 4.5: Helper threads' code for a real HTM.

```
while(heap->curr_size > 0){

        while(done == 1);

        /* ReadMin */
        my_min_id = heap->node_array[tid].vertex_id;
        my_min_key = dist[my_min_id].value;

        if(my_min_key < INFINITY){

            /* Check neighbors for relaxation. */
            for(v=g->adj[my_min_id]; v!=NULL && !done; v=v->next){


                    if(begin_transaction(num_retries, &fallback_lock, tid) != -1){
                            if(done == 0){
                                    distv = dist[v->id].value;
                                    sum = my_min_key + v->weight;

                                    if(distv > sum){
                                            decrease_key_mt(heap, v->id, sum);
                                            pred[v->id].value = my_min_id;
                                            dist[v->id].value = sum;
                                    }
                            }else
                                    _xabort(0xaa);
                    }else
                            break;
                    end_transaction(&fallback_lock, counter);

            }
    }
}
```

4.4 System Configuration

The system we used to evaluate the proposed parallelization of Dijkstra's algorithm was a 28-core platform (Figure 4.5), NUMA architecture with the following characteristics.

- 2 sockets (Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz)
- 14 cores per socket (28 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 35MB L3 cache per socket
- 128GB RAM
- Hardware Transactional Memory:
    - lazy data versioning
    - eager conflict resolution
    - best effort HTM
    - strong isolation
    - cache line granularity
    - 4MB read set
    - 22KB write set

In the evaluation part, each software thread is manually pinned to a hardware thread (to a core) in order to take advantage of the locality with the sockets. We first pin software

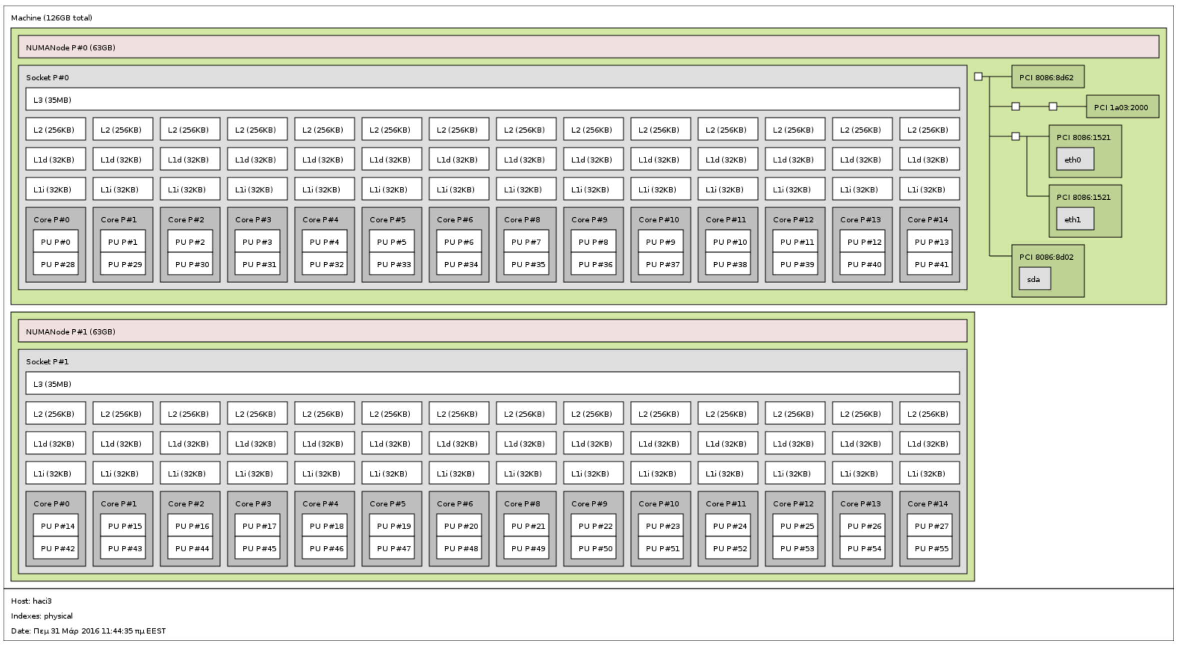


Figure 4.5: The platform used in evaluation of Dijkstra's algorithm.

threads such that to fill the first socket (first 14 threads) and share the same L3 cache. And then we pin threads in the second socket. Our evaluation reveals that the NUMA effect negatively influences the scalability. In case of a cache miss the transfer of the memory address from one socket to another is costly.

4.5 Experimentation

4.5.1 Experimentation in the serial algorithm

We first evaluated the serial Dijkstra's algorithm. More specifically, we executed on the above platform the main thread's code (simple dijkstra's algorithm) for graphs of different sizes, both dense and sparse graphs. In this experimentation there is no need to perform relaxations within a transaction, as this is a serial execution. Executing relaxations in transactional mode causes an additional overhead associated with the transaction and it is more time consuming. However, the difference in the runtime between using and not using transaction is not important and does not affect our analysis. The main purpose of this experimentation is to evaluate the parallelization of the inherently serial Dijkstra's algorithm.

We separated the algorithm in four phases and measured the runtime of each phase. The first phase is the ExtractMin() operation that takes time proportional to $\mathcal{O}(\log n)$, the second phase is the compute operation for the distance (line 9 in listing 4.6) from source, the third is the DecreaseKey() operation and the fourth is the update operation in the distance and predecessor arrays (lines 18-19 in listing 4.6). Figure 4.6 presents our results for two dense graphs (a rmat graph with 1M nodes and 100M edges, and a random graph with 10M nodes and 500M edges) and a sparse graph (a rmat graph with 100M nodes and 100M edges).

Listing 4.6: The four phases of the algorithm.

```
 1  while Q not empty do
 2          start_timer(extract_min)
 3          u ← ExtractMin(Q);
 4          stop_timer(extract_min)
 5
 6          done ← 0;
 7          foreach v adjacent to u do
 8                  start_timer(compute_time)
 9                  sum ← d[u] + w(u, v);
10                  stop_timer(compute_time)
11
12                  Begin−Transaction
13                  if d[v] > sum then
14                          start_timer(decrease_key)
15                          DecreaseKey(Q, v, sum);
16                          stop_timer(decrease_key)
17                          start_timer(update_time)
18                          d[v] ← sum;
19                          p[v] ← u;
20                          stop_timer(update_time)
21                  End−Transaction
22          end
23
24          Begin−Transaction
25          done ← 1;
26          End−Transaction
27  end
```

Taking into consideration the concept of this parallel algorithm, only the decrease_key and update part of the algorithm can be offloaded from the main thread. Helper threads perform some operations such that the if branch of the main thread (line 13 in listing 4.6) can be evaluated as true fewer times. According to figure 4.6 this branch constitutes almost the 12-25% of the total main thread's runtime. Thus, the main thread can gain a very small percentage of the total runtime and this parallelization can offer small performance speedups theoretically. The extract_min and compute phases of the algorithm have to be executed from the main thread too, for all the nodes and the edges of the graph. As a result, we conclude that dijkstra's algorithm is a hard algorithm to parallelize as the most part of

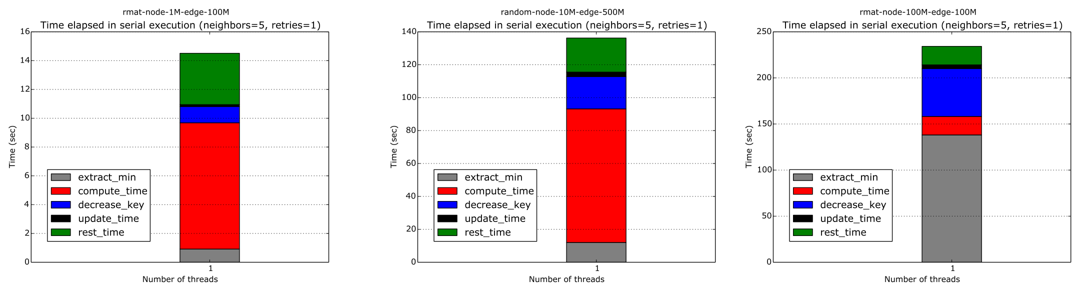


Figure 4.6: Evaluation of the four phases of the algorithm in different graphs.

the algorithm is serial.

In the rmat-node-100M-edge-100M graph the extract_min phase constitutes a great portion of the total runtime, comparing with the other two graphs. ExtractMin() in a binary heap removes the root of the heap and replace it with the node with the next highest priority (minimum distance from the source). To implement this a traversal from root to leaves is needed and it takes time proportional to $\mathcal{O}(\log n)$. Thus, the more nodes a graph has, the longer the ExtractMin() function lasts. This conclusion is depicted in rmat-node-100M-edge-100M graph that is a very large graph. We can also conclude the same for the DecreaseKey() function, as it also takes time proportional to $\mathcal{O}(\log n)$.

On the other hand, in the other two graphs (rmat-node-10M-edge-500M and random-node-1M-edge-100M) the compute phase is the most time-consuming phase of the algorithm. Compute phase calculates a new distance associated with an edge. It repeats this calculation for all edges in the graph. As a result, the more dense a graph is, the more time the compute phase lasts. The rmat-node-100M-edge-100M is a sparse graph, as the ratio of the number of nodes to the number of edges is 1. In this graph, every repetition of the while loop (line 1 in listing 4.6) demands only one repetition on average of the compute phase, while in more dense graphs the compute phase is repeated many times. Therefore, this phase is the most time-consuming phase in dense graphs and does not constitute a large portion of the total time in sparse graphs.

In the serial execution there is no data conflict aborts, since there is only one thread. We used structure padding technique in order to avoid false sharing that would lead to data conflicts aborts in case of multiple threads in the execution. Thus, the structures of the algorithm like the distance and predecessor arrays have not to be padded in the serial execution. We evaluated the serial algorithm for different graphs without using structure padding technique in the shared structures and our results are shown in figure 4.7.

Our experiments demonstrate that although we do not use padding in the structures, the proportion of the 4 phases of the algorithm (extract_min, compute, decrease_key and update) to the total runtime remains the same. The pattern of the different phases is similar in all executions. However, we have to notice that the total runtime is reduced significantly when we remove the padding from the distance array in executions where the compute phase is the most time-consuming phase (dense graphs). Without padding in the distance array, consecutive elements are stored in the same cache line. Therefore, the main thread

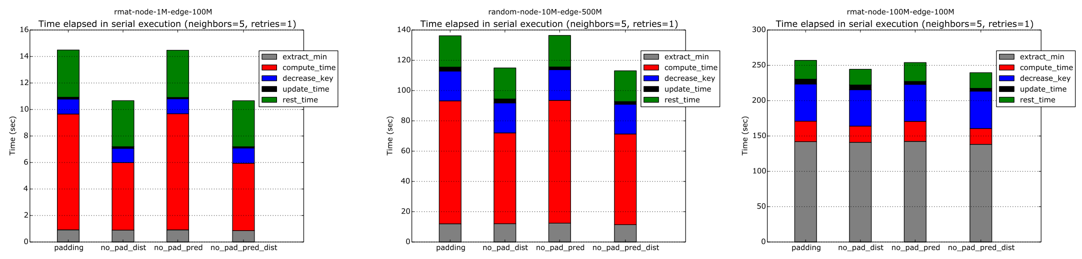


Figure 4.7: Experimentation in padding technique on the serial algorithm.

can find some elements in its cache memory and avoid transferring cache lines from the main memory in each read/write operation. Furthermore, without structure padding the capacity misses are reduced. One cache line can store multiple elements of the array and cache memory can store more elements than before (in case of padding).

As a consequence, the main gain of removing padding is in compute phase, where we avoid transferring the distance value of the current node d[u] (line 9 in listing 4.6) in each iteration of the for loop (line 7 in listing 4.6). This element remains in the cache memory and the main thread can read its value quickly. In sparse graphs like rmat-node-100M-edge-100M graph (figure 4.7) there is not considerable gain, as the compute phase is not so time-consuming. Finally, removing padding from the predecessor array (p[]) does not have any important contribution in the total runtime. In each iteration of the for loop the adjacent nodes (v nodes) of the current u node can be in the same cache line of the predecessor array with a very small probability. It depends on the shape of the graph. In the most common case, edges connect nodes which are quite remote one from each other (not consecutive nodes). These nodes are not in the same cache line and as a result the main thread has to transfer the element $p[v]$ from the main memory every time that if branch is evaluated as true despite not using padding. It cannot exploit spatial locality.

4.5.2 Experimentation in the parallel algorithm

Experimentation in the number of retries per transaction before acquiring the lock

As explained in previous chapter, when a transaction fails to commit, it retries. Nevertheless, there are transactions that they always fail to commit. For example, when transaction's read/write set exceeds HTM system's read/write set, the transaction will be always aborted due to capacity abort. In this case the fallback path must be executed. In our scheme the fallback path is implemented as coarse-grained locking code that uses a global lock shared among all threads that protects the critical section. If a thread acquires this global lock, any other thread cannot proceed in the critical section and it will always fail until the lock is released.

We evaluated the performance of the algorithm for different numbers of retries for a transaction before acquiring the global lock. First, the main thread performs a certain number of retries for each transaction and if it exceeds this number due to consecutive transactional aborts, it acquires the lock, aborts all helper threads and executes the critical section in a coarse-grained locking mode. Helper threads can never acquire the lock and they always attempt to execute the critical section in transactional mode (unlimited number of retries for a transaction). In the opposite case, where helper threads could acquire the global lock, they would delay the main thread and would interfere its progress decreasing the performance of the algorithm. Figures 4.8 and 4.9 present a performance evaluation for different numbers of retries for a main thread's transaction (a random-node-1M-edge-10M and a rmat-node-10M-edge-500M respectively) investigating 5 neighbors for relaxation per transaction.

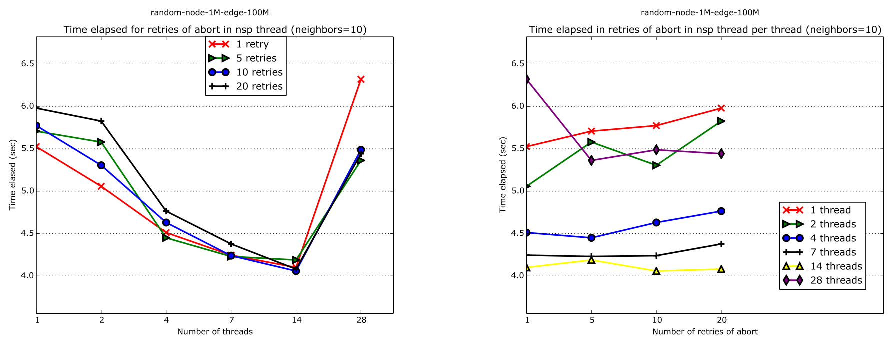


Figure 4.8: Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of the main thread.

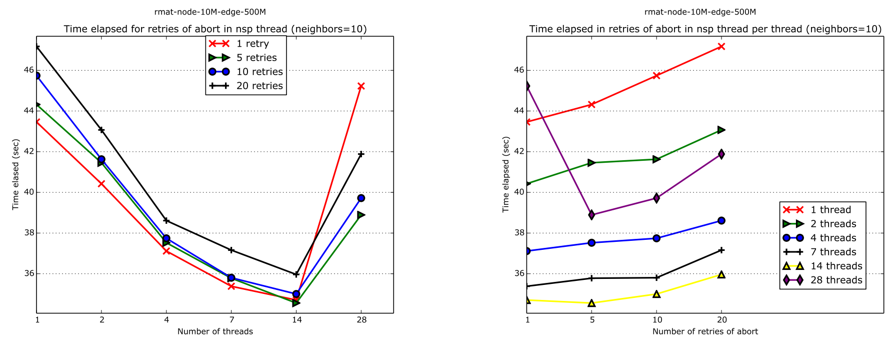


Figure 4.9: Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of the main thread.

According to these figures, the more times a transaction can retry, the more time the execution of the algorithm lasts. In case of one possible retry per transaction, when the main thread fails to commit, it immediately acquires the lock and executes the critical section successfully while helper threads are stalled. Thus, this execution of the main thread is very close to the serial execution. Since the main thread can retry its transaction more times (the number of retries per transaction increases), each transaction lasts more when consecutive transactional aborts appear. This is why the runtime is better in the execution with one possible retry per transaction comparing with the rest executions with more possible retries per transaction, where the main thread's runtime diverge more and more from the serial execution's runtime.

Secondly, we can notice that in executions with more than one possible retry there is a better scalability. For example, the runtime of 14 threads in executions of 5 and 10 retries per transaction becomes approximately the same with that of one possible retry per transaction (figure 4.9), while in case of one thread they differ significantly. This is because in executions of one possible retry per transaction helper threads are stalled immediately without performing many relaxations. In executions with more possible retries per transaction helper threads have more time to perform relaxations and commit them, so as the main thread can gain more work.

Finally, in executions with 28 threads it appears the effect of NUMA architecture. The transfer of a cache line from the memory of one socket to the another is costly and it negatively influences the scalability. The runtime increases because of expensive transfers of cache lines from the memory of a different socket. We can also observe that the time elapsed of the execution with one possible retry per transaction is worse than in executions with more possible retries per transaction. This is due to the shared global lock. The main thread writes (locks) the global lock and each time that a helper thread starts a transaction and attempts to read the global lock, it has to transfer it (because of the coherence protocol) possibly from a remote memory. This is quite costly in executions where the main thread writes the global lock frequently like in execution with one possible retry per transaction. In this execution when a helper thread, which resides in different socket from the main thread, reads the global lock, it has to transfer it from a remote memory because it has been written from the main thread with some strong probability. On the other hand, in executions with more possible retries per transaction the main thread does not write the global lock so frequently and thus, helper threads do not perform costly transfers of the global lock in each transaction (frequently). However, each transaction lasts more when consecutive transactional aborts show up. To sum up, in case of 28 threads where 2 sockets are used, we have to find a solution about the number of possible retries per transaction that trades off the costly frequent transfers of the global lock and the more time-consuming transactions in case of consecutive transactional aborts in the main thread.

Subsequently, we examined the number of possible retries per transaction in helper threads. In previous executions helper threads could retry their transaction until they commit or are explicitly aborted by the main thread when "done" variable is set to 1. We attempted to limit the number of possible retries per transaction in helper threads, such that to reduce conflict aborts of the main thread. Helper threads can now perform a specific number of retries per transaction and if they exceed this number they will wait in a while loop until they are forced by the main thread to proceed in the next iteration (until "done" is set to 1). We suppose that retrying a transaction to commit after a certain number of retries can only cause conflict aborts and can never lead to a transactional commit. Figures 4.10 and 4.11 depict our results for different numbers of possible retries per transaction in helper threads.

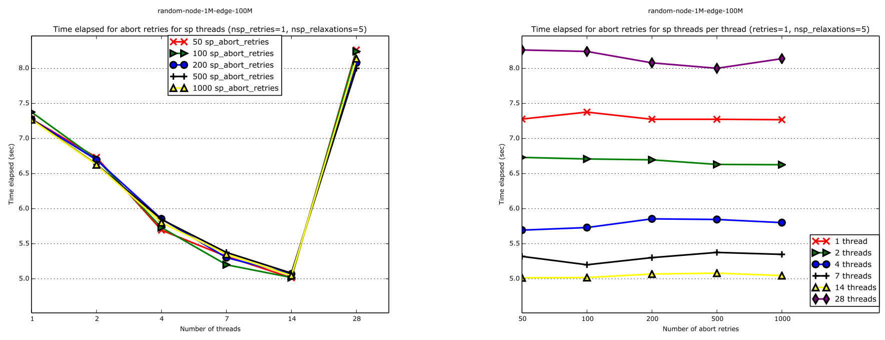


Figure 4.10: Time elapsed in random node-1M-edge-10M graph for different number of possible retries per transaction of helper threads.

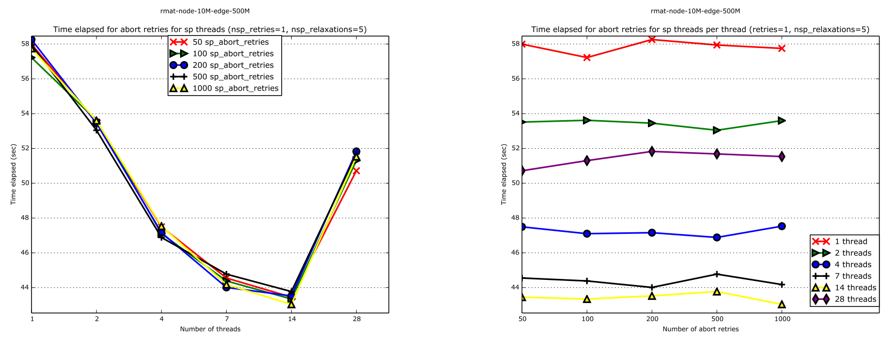


Figure 4.11: Time elapsed in rmat node-10M-edge-500M graph for different number of possible retries per transaction of helper threads.

Our results demonstrate that although we limited the number of possible retries per transaction, the total runtime of the algorithm was not improved. Helper threads can commit their transactions retrying them less times than the given number (limit) of retries. Even though we reduced the number of possible retries per transaction to a very small number, 50 retries, this number is like an infinite value for retries. Thus, we conclude that there is no gain by limiting the number of retries per transaction in helper threads.

Experimentation in the number of neighbors examined for relaxation per transaction

To avoid the overhead of beginning and ending many consecutive small transactions, we implemented a more coarse-grained scheme in transactions for checking the neighbors of the current node to perform relaxations on them. If we check more than one neighbor to relax in a single transaction, the overhead of performing transactions reduces, as we have bigger and fewer transactions. However, in large graphs performing a more coarse-grained transaction can lead to capacity transactional aborts, since this scheme accesses a large part of the memory that can exceed HTM system's read or write set. Therefore, we examined the number of neighbors checked for relaxation within a single transaction, such that to trade off the overhead of many consecutive small transactions and the capacity transactional aborts that may appear in a more coarse-grained scheme.

We evaluated the algorithm as described in listings 4.4 and 4.5 for the main and helper threads, respectively. More specifically, we performed a coarse-grained transaction for different number of neighbors to examine in the main thread, while helper threads check only one neighbor for relaxation in each transaction. We executed the algorithm in different sizes of graphs for 1, 2, 5, 10, 20 and 50 neghbors examined for relaxation in a single transaction. Figures 4.12 and 4.13 demonstrate our results for a random node-1M-edge-100M graph and a rmat node-10M-edge-500M graph.

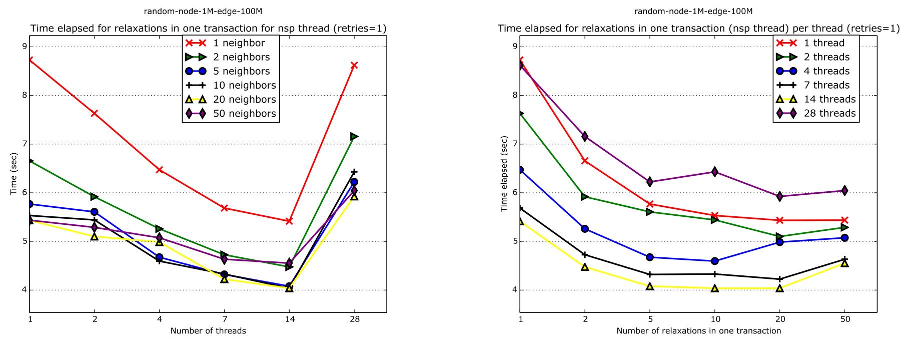


Figure 4.12: Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of the main thread.

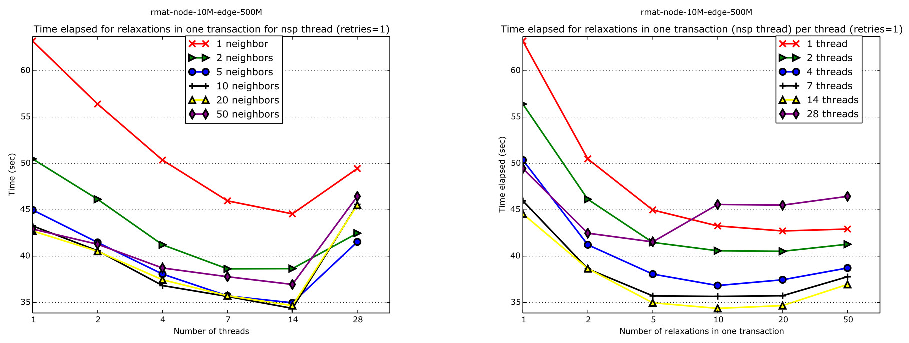


Figure 4.13: Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of the main thread.

Our results confirm our hypothesis about a more coarse-grained transaction. We can observe that the case of checking one neighbor per transaction has the worse total time

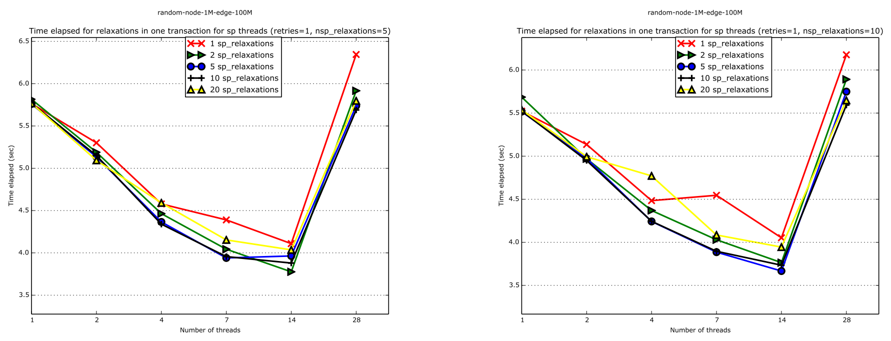


Figure 4.14: Time elapsed in random node-1M-edge-100M graph for different number of neighbors examined for relaxation per transaction of helper threads.

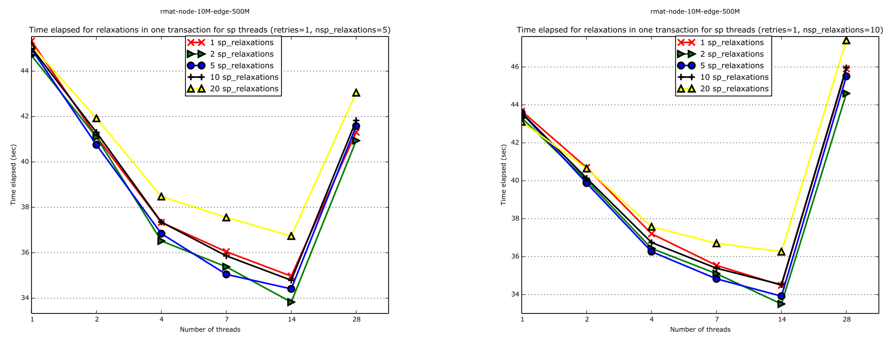


Figure 4.15: Time elapsed in rmat node-10M-edge-500M graph for different number of neighbors examined for relaxation per transaction of helper threads.

elapsed, since there are many small consecutive transactions that clear and fill frequently transactional cache lines in memory and this is quite time-consuming (high overhead). As the numbers of neighbors checked per transaction increases the total time elapsed is improved until a certain number of neighbors. In the smaller graph (random-node-1M-edge-100M figure 4.12) the best time elapsed can be shown for 20 neighbors checked within a single transaction, while in the larger graph (rmat-node-10M-edge-500M figure 4.13) the best time elapsed is appeared in case of 10 neighbors. Consequently, we can conclude that the larger a graph is, the less coarse-grained the transaction should be, since more coarse-grained transactions in large graphs access more memory (the priority queue is larger) and they may lead to capacity transactional aborts.

Secondly, the case of checking one neighbor per transaction scales better as the number of threads increases. In this execution, the time elapsed for one thread only (the main thread) is the worst and thus, the more threads we add, the more gain we have. The executions with more than one neighbor checked for relaxation per transaction are not so time-consuming and adding more threads do not have so much gain (smaller scalability) than in case of examining one neighbor per transaction. Finally, executions with 28 threads are time-consuming because of the NUMA architecture effect of our system which causes costly cache line transfers from one socket to another.

In the next step, we performed the same evaluation for helper threads, too. We fixed the coarse-grained transaction of the main thread to 5 and then to 10 neighbors per transaction and implemented a coarse-grained transaction for helper threads (1, 2, 5, 10, 20 neighbors examined for relaxation per transaction). Our purpose is to analyze which execution has the best scalability, since all executions start from the same point, the one thread execution (main thread) with fixed (5 or 10) neighbors checked per transaction. Our results are depicted in figures 4.14 and 4.15.

In random-node-1M-edge-100M graph the executions of 5 and 10 neghbors examined per transaction have the best scalability. Similarly to the above conclusions, as the transaction becomes more coarse-grained (bigger transactions), we can avoid costly consecutive small transactions, but we do not gain in performance if we exceed a certain number of neighbors. In rmat-node-10M-edge-500M graph the executions of 2 and 5 neighbors checked per transaction give the best scalability. Since this graph is larger, the transaction has to be less coarse-grained than in the smaller graph. It accesses a larger binary heap and can exceed HTM system's read/write set by checking for relaxations a smaller number of neighbors within a single transaction.

4.6 Results

4.6.1 Performance results

In our performance evaluation we tested graphs of different density and structure. We used graphs with 10K, 1M, 10M, 100M vertices from the Random and R-MAT families. We also evaluated the algorithm on a real road network, a full USA road network (USA-road-d.USA). Figure 4.16 presents the speedups achieved by the implementation of Dijkstra's algorithm described in listings 4.4 and 4.5. The speedup obtained for n threads is the ratio of the execution time of the serial algorithm to the execution time with n threads, $n-1$ of them being helper threads. This scheme is able to achieve significant speedups in most cases. The maximum speedup achieved is 1.39 for the random-node-1M-edge-100M graph (14 threads).

As explained in [24] in the serial execution, time can be estimated as:

$$T_{serial} = n * \mathcal{O}(\log n) + d * n * \mathcal{O}(\log n), \tag{4.4}$$

where n represents the number of vertices in the graph and d the average out-degree of the vertices. The ExtractMin() operation spends time $n * \mathcal{O}(\log n)$ and DecreaseKey spends $d * n * \mathcal{O}(\log n)$ time, approximately.

The execution time of the described parallel scheme can be estimated as:

$$Tparallel = n * \mathcal{O}(\log n) + a * d * n * \mathcal{O}(\log n), a < 1 \tag{4.5}$$

where a is the ratio of the main thread's DecreaseKey() operations to those executed in the serial case. This is a simple theoretical approach. It does not take into account the time

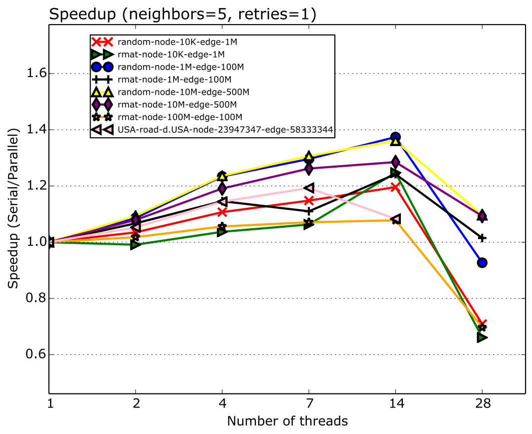


Figure 4.16: Multithreaded speedups for graphs of different density.

spent for thread coordination or delays due to transactional aborts. Thus, a theoretical speedup can be calculated as:

$$s = \frac{1+d}{1+a*d} \tag{4.6}$$

We have to notice that the performance is strongly related to the density of the graph. This conclusion is implied to speedup's definition 4.6, too. According to the results of figure 4.16, for more dense graphs, the speedup is greater, as more parallelism can be exposed in the inner loop of the algorithm. Conversely, sparse graphs like rmat-node-100M-edge-100M leave limited space for parallelism leading to low performance. Furthermore, the figure also reveals that the speedup increases as more threads are utilized. The performance is improved up to a maximum point, after which utilizing more threads leads to performance degradation. The number of threads to achieve this maximum is again related to the graph's density. For example, in rmat-node-100M-edge-100M (sparse) graph increasing the number of threads from 7 to 14 slightly reduces the performance and in USA-road-d.USA-node-23947347-edge-58333344 graph the performance remains nearly the same. Finally, using 28 threads degrades the performance in all evaluated graphs because of the NUMA effect. Our system is a NUMA architecture with 14 threads per socket. Therefore, pinning 28 threads in two sockets leads to expensive cache line transfers from one socket to another and negatively influences scalability.

4.6.2 A closer look at the results

In this subsection, we attempted to have a closer look into the behavior of the described scheme. We examined main thread's gain in the number of relaxations, the abort ratio and the time spent in each main thread's operation and we tested them to all previous graphs.

We only present some representative graphs with different density, as the other graphs exhibit similar behavior.

Figure 4.17 shows the distribution of performing relaxations between the main and helper threads (lines 28-32 and 20-24 in listings 4.4 and 4.5 respectively). As more threads are used, main thread's relaxations are reduced and helper threads' relaxations are increased, justifying the performance improvement. Similar reductions in the main thread's operations are also achieved for the sparse graph (rmat-node-100M-edge-100M). In this graph, helper threads' relaxations can never exceed main thread's relaxations, as this is a very sparse graph and helper threads cannot offload many relaxations of the main thread. Employing 28 threads degrades scalability of main thread's relaxations because of the NUMA effect, that was depicted in the previous figure 4.16 for speedup, too.

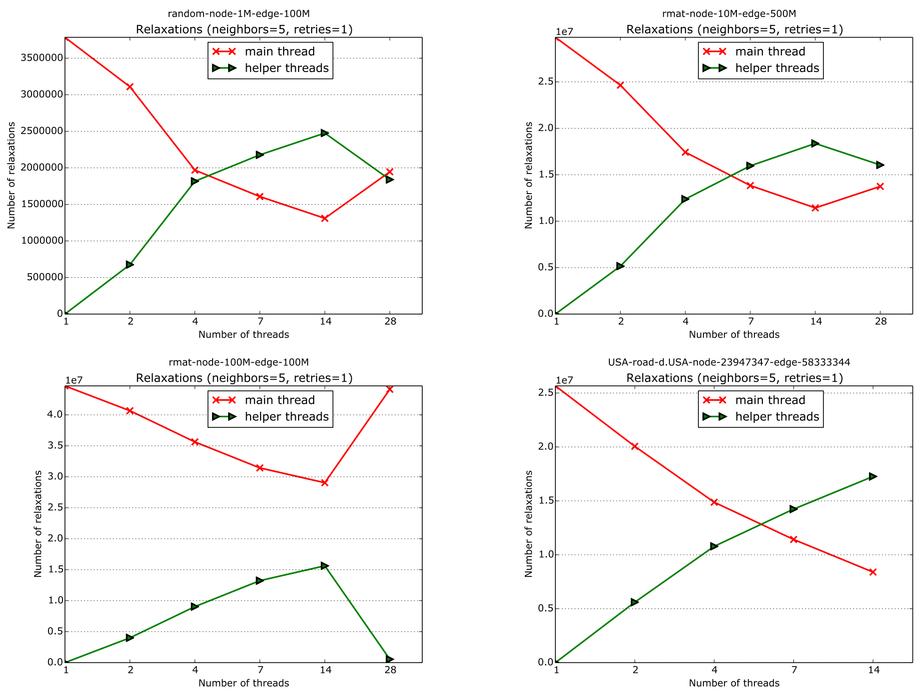


Figure 4.17: Distribution of relaxations between the main and helper threads.

Figure 4.18 depicts the number of commits and aborts of the main thread for the evaluated executions. The main thread suffers a really low number of aborts, especially in dense graphs. This means that even when helper threads are not contributing any useful work, they still do not obstruct main thread's progress. The main thread is allowed to run almost at the speed of the serial execution. An important observation though, is that the number of transactional aborts in the main thread depends on the size of the transaction's write set. The larger the write set is, the higher probability of a conflict. Furthermore, in the more coarse-grained transaction that we have implemented for the main thread, there is a higher probability of capacity aborts in large graphs like the full USA road network, where in main thread's execution there are many transactional capacity aborts. Finally, the addition of more threads does not lead to an increase of the number of aborts. Thus, we can suppose that if the NUMA effect did not exist, the algorithm would lead to a better performance speedup for more than 14 threads.

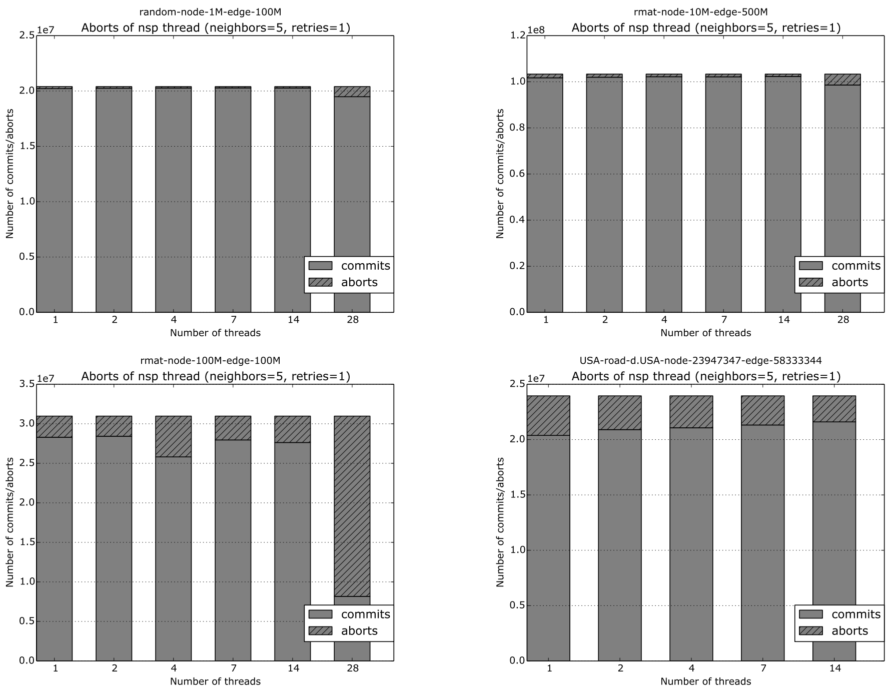


Figure 4.18: The number of main thread's commits/aborts.

In case of 28 threads transactional aborts are increasing. A single transaction lasts more time due to the NUMA effect. The cache line transfers are expensive (take a long time), since cache lines may be transferred from a remote memory. As the transaction is more time-consuming, it can be aborted with a stronger probability. Firstly, data conflicts are more possible to be detected in longer transactions. And secondly, if a transaction last more than the time quantum, the scheduler of the operating system will schedule out the process and the transaction will be aborted because of a timer interrupt. In rmat-node-100M-edge-100M graph, the transactional aborts in 28 threads due to the NUMA effect are significantly increased, since this is the largest graph and a single transaction takes up a lot of time.

To gain a better understanding of the wasted work due to transactional aborts, figure 4.19 presents the percentage of the total transactional aborts for all threads in total number of transactions. Again, for graphs of high density the percentage of transactional aborts is relatively small (about 3%), justifying the observed speedups, while in sparse graphs this percentage is higher. The small percentage of transactional aborts shows that most of the concurrent accesses to the shared data structures are non-conflicting. Moreover, we can notice that in dense graphs, as the number of threads increases, the percentage of transactional aborts also increases, as the probability of performing conflicting accesses

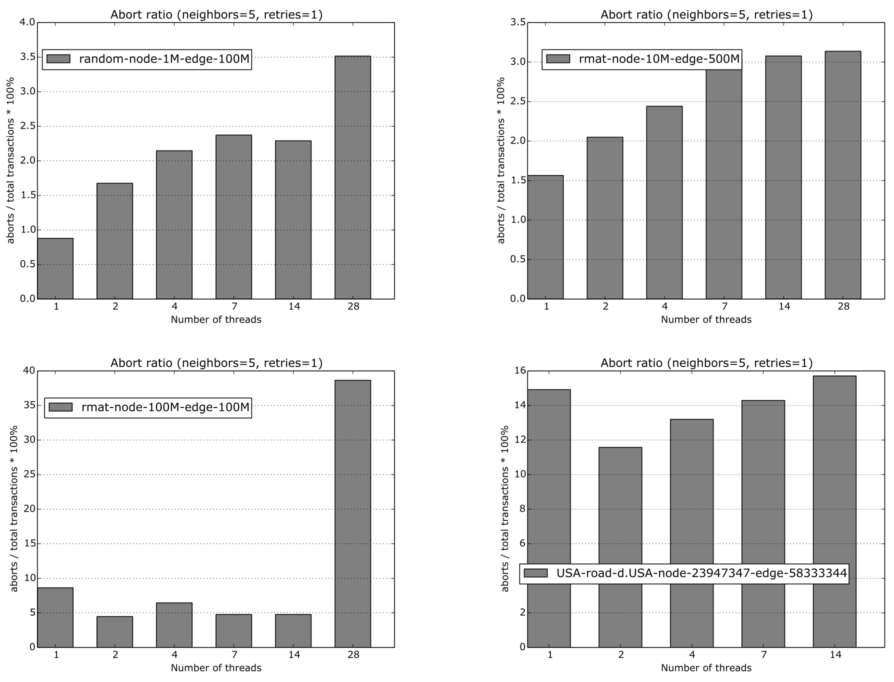


Figure 4.19: The percentage of total transactional aborts for all threads in total number of transactions.

becomes stronger. On the contrary, in sparse graphs like the rmat-node-100M-edge-100M graph the number of transactional aborts does not augment when adding more threads because threads perform conflicting accesses with a very small probability in sparse graphs. Similarly to figure 4.18, the percentage of transactional aborts is increased in 28 threads, since the transaction lasts longer due to the NUMA effect and becomes more vulnerable to a transactional abort.

Figure 4.20 shows the time spent by the main thread in ExtractMin() operation, in transactional mode part (lines 21-37 in listing 4.4) and in the rest operations of the main thread. The ExtractMin() operation remains stable for each graph, as it is not affected in the described scheme. The time spent in ExtractMin() is only increased in case of 28 threads because of the costly cache line transfers in our NUMA architecture. The NUMA effect is notably depicted in the large graph (rmat-node-100M-edge-100M graph). Secondly, the addition of helper threads reduces the time spent in transactions, the parallel part of the scheme. The main thread performs less relaxations (lines 28-32 in listing 4.4). It executes fewer times the costly DecreaseKey() operation, that takes time proportional to $\mathcal{O}(\log n)$ (where n is the number of vertices) and as a result the runtime in transactional mode reduces. Finally, as explained in the evaluation part of the serial execution, the ExtractMin() operation lasts more time and constitutes a large percentage of the total runtime in larger graphs like rmat-node-100M-edge-100M graph and USA-road-d.USA graph, since it also takes time proportional to graph's size ($\mathcal{O}(\log n)$).

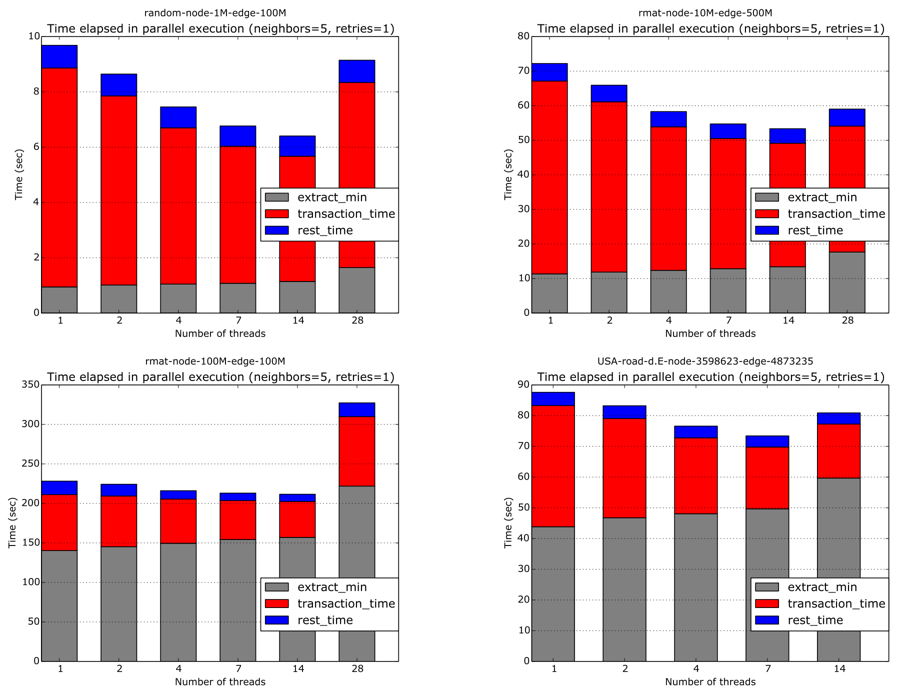
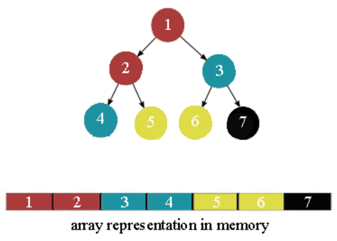


Figure 4.20: Distribution of time spent in different phases of main thread's execution.

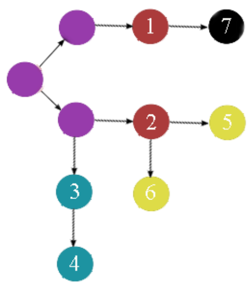
4.6.3 Experimentation in padding technique

All previous evaluations were implemented using padding technique in shared structures. We padded the distance and predecessor arrays as well as the node_array and the where_in_heap array, which are used for the graph representation and are shared among threads, too. We supposed that if we were not using padding technique, the number of transactional aborts would be higher and thus, the execution in transactional mode would take a longer time. Different threads that perform operations (in transactional mode) in independent data that reside in the same cache line would be aborted, as our real HTM system detects conflicts at cache line granularity. However, without using padding memory can store more elements of the structures. As a consequence, a thread can find an element in its cache memory with a stronger probability, avoid an expensive transfer and can exploit temporal and spatial locality. Temporal locality defines that a data which is referenced at one point in time will be referenced again sometime in the near future and spatial locality defines that likelihood of referencing a data is higher if a data near it was just referenced.

In order to verify our hypothesis for longer time spent in transactional mode in case of not using padding technique, we executed the algorithm removing padding of the shared structures. Our results can be shown in figure 4.22. They prove that our hypothesis was incorrect, as the transactional runtime was reduced. Different threads access independent data that reside in the same cache line with a very small probability. It depends on the shape of the graph. In the most common case, edges connect vertices that are quite remote one from each other (not consecutive nodes) and as a result they do not share the same cache line. Thus, we can conclude that not using padding does not affect the abort ratio. Figure 4.21a presents a representation of a binary heap in memory (vertices that share the same cache line are depicted with the same color) and figure 4.21b the real network in which edges connect nodes that reside in different cache lines. In case of 3 threads, the first thread will relax node 7, the second nodes 5, 6 and the third node 4. These simultaneous relaxations are performed in nodes that reside in different cache lines despite not using padding. As a result, it will not appear any transactional abort.



(a) Binary heap representation in memory.



(b) The real network.

Figure 4.21: Despite not using padding, simultaneous relaxations are performed in nodes that reside in different cache lines.

On the other hand, removing padding improved the total runtime of the algorithm. More specifically, removing padding reduces the time spent in transactional mode, especially in case of removing padding from the distance array, since the distance array is accessed mostly inside transaction (line 24 in listing 4.4). The if branch (lines 28-32) that accesses the rest structures (predecessor array, node_array and where_in_heap array) is evaluated as true fewer times. Therefore, the main gain of temporal and spatial locality for a thread is in the distance array. Threads can find an element in their cache memory with a stronger probability when no padding is used and avoid transferring cache lines from the main memory. Furthermore, by avoiding frequent transfers we can also avoid delays related to the common memory bus (memory bus congestion).

Finally, we can notice that despite removing padding the execution pattern remains the same. As the number of threads increases, the scalability of the execution without using padding in structures is the same with that of using padding. The performance speedups of the algorithm have the same values for all evaluated graphs. The only difference is that the total runtime of the algorithm is reduced when no padding is used in structures. Consequently, our analysis for the scalability and the performance of the algorithm is not affected.

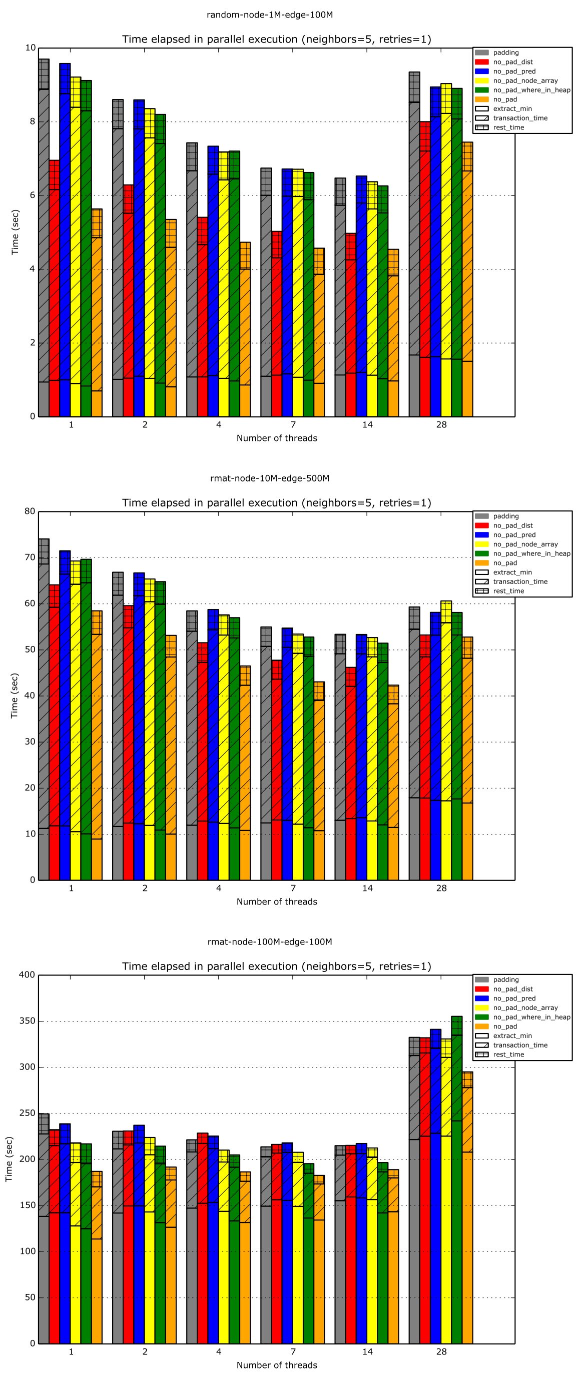


Figure 4.22: Experimentation in padding technique on the parallel algorithm.

4.7 Employing skip list

4.7.1 The skip list structure

Our implementation described employs a binary heap for the priority queue. In this section we attempted to evaluate the algorithm employing a skip list instead of binary heap. DecreaseKey() operation takes time proportional to $\mathcal{O}(\log n)$ for both skip list and binary heap and ReadMin() operation has the same complexity, too. However, ExtractMin() operation takes time proportional to $\mathcal{O}(1)$ when using a skip list, in contrast with the binary heap where it takes $\mathcal{O}(\log n)$.

Skip list is a data structure that allows fast lookup within an ordered sequence of elements (key-value pairs). Fast search is made possible by maintaining a linked hierarchy of subsequences, each skipping over fewer elements. This structure is built in layers. The bottom layer is an ordinary ordered linked list. A key in layer i appears in layer i+1 with some fixed probability p. Thus, each element of the structure has a random height that represents the layers in which the associated key appears. Skip list has a maximum height and every time that an element is inserted, it takes a random height between 1 and skip list's maximum height. Figure 4.23 depicts an example of a skip list.

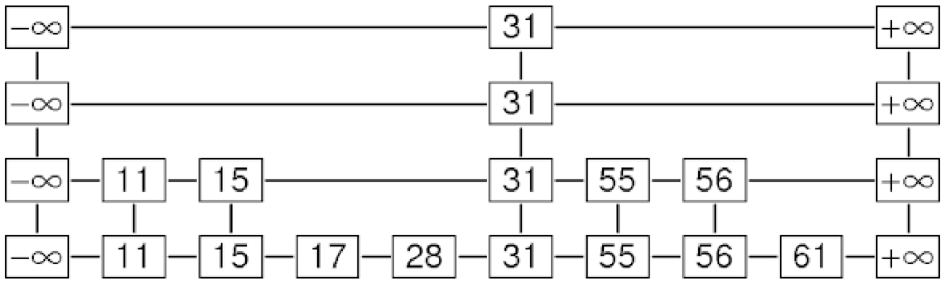


Figure 4.23: An example of a skip list.

A lookup for a target begins at the head element in the top layer and proceeds horizontally until an element greater or equal to the target is reached. If this element is equal to the target, it has been found and the operation returns. Otherwise, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower layer. The total expected complexity of the lookup operation is $\mathcal{O}(\log n)$.

In skip list, the height of each element (number of element's layers) is a random number. Thus, it is possible (though with a very low probability) that it will be produced a badly balanced structure. However, skip lists work well in practice, and the randomized balancing scheme has been argued to be easier to implement than the deterministic balancing schemes used in other structures like balanced binary search trees. Finally, skip lists are useful in parallel computing, where insertions can be done in different parts of the skip list concurrently without any global rebalancing of the data structure.

4.7.2 Comparison with the binary heap

In this section we will compare the implementation that employs a skip list for the priority queue with the previous implementation (binary heap for the priority queue). In our first approach we constructed a skip list that had an element for each vertex of the graph. Thus, in DecreaseKey() operation the element that represents the vertex to be relaxed, is removed from the skip list and is placed at a closest position to the top of the list. As in case of using a binary heap, this operation takes time proportional to $\mathcal{O}(\log n)$. However, the execution time of the algorithm was extremely large and the scalability was not remarkable. The reason was the DecreaseKey() operation, which was more time-consuming than that in case of using a binary heap.

In our attempt to improve the implementation with the skip list, we noticed that the DecreaseKey() operation was performing many steps to place the extracted element to its new position. To place an element in skip list, there is a traversal from the head of the list until an element with an equal or a greater key is reached. This traversal was taking too many steps, while placing the element to its new position in the binary heap requires a small number of swaps in the elements of the structure. We also noticed that our skip list had too many elements with the same key during the execution of the algorithm. While the algorithm were executed, the distances of the vertices from the source were being updated with the same value with some strong probability. As a consequence, our skip list had many elements (different vertices of the graph) with the same key, was needing a lot of memory (more cache line transfers) and the traversal from the head of the list to the desired node, was much time-consuming (due to the large number of elements which had to be overtaken).

Therefore, we implemented an optimized skip list that contains only discrete keys. Each element of the list has a unique key and a simple internal nested list that stores the ids of the vertices which have the same distance-key from the source. In this way, our skip list demands less memory, since it contains fewer items, and the DecreaseKey() operation takes fewer steps, since it traverses a smaller number of elements.

In our evaluation we firstly compare the serial execution of the algorithm for the three different structures, the binary heap, the simple skip list (there is an element for each vertex of the graph) and the optimized skip list (it contains only discrete keys). Figure 4.24 shows our results for two different graphs, the rmat-node-10M-edge-500M and rmat-node-100M-edge-100M. The ExtractMin() operation is much less time consuming in implementations that employ a skip list, since it takes time proportional to $\mathcal{O}(1)$, while in case of binary heap it takes $\mathcal{O}(\log n)$. In large graphs like the rmat-node-100M-edge-100M graph, where the ExtractMin() operation constitutes a great portion of the total runtime, there is a significant gain. Furthermore, we have to remark that the DecreaseKey() operation takes more time in the simple skip list for both two graphs. As explained above, this is because it takes too many steps to place the extracted element to its new position, since it traverses many elements of the same key. In our optimized skip list we avoid such a long traversal and as a result the DecreaseKey() operation can place an element to its new position with approximately the same number of steps with the binary heap. The elements that have to be overtaken in the optimized skip list to place the extracted element to its new position is comparable with the number of elements' swaps that are performed in DecreaseKey() operation in case of binary heap.

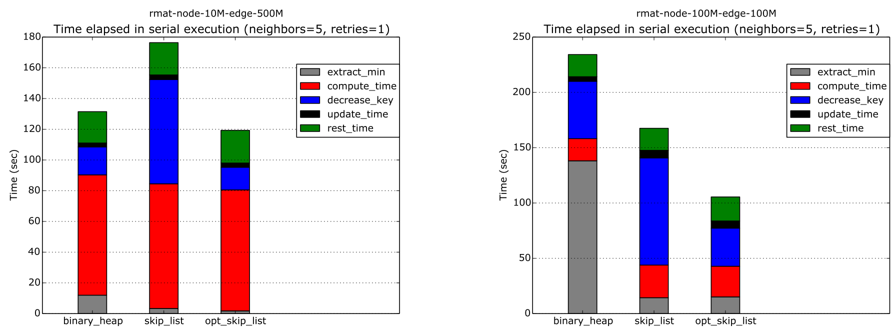


Figure 4.24: Time elapsed in serial execution for the three different structures used for the priority queue.

Subsequently, we evaluated the parallel execution of the algorithm for different number of threads. Figure 4.25 depicts the total runtime elapsed for the random-node-10M-edge-500M and the rmat-node-100M-edge-100M graph. As in the serial execution, the DecreaseKey() operation of the random-node-10M-edge-500M graph in case of using the simple skip list is much time-consuming and as a result the total runtime is the worst among the three executions. This is because the random-node-10M-edge-500M graph is very dense. The more dense a graph is, the more relaxations (DecreaseKey() operations) are performed due to the large number of edges. Thus, the DecreaseKey() operation is executed many times and the total execution time is increased significantly. On the contrary, the rmat-node-100M-edge-100M graph is a sparse graph and the DecreaseKey() is not executed so many times, since there are not many edges to cause relaxations, and the total runtime is not influenced. As we noticed in the serial execution, in this graph the ExtractMin() operation constitutes a great portion of the algorithm when using a binary heap for the priority queue. In this execution, the ExtractMin() operation depends on the number of vertices of the graph ($\mathcal{O}(\log n)$), while the skip list performs the ExtractMin() operation in a constant time ($\mathcal{O}(1)$). As a consequence, in the rmat-node-100M-edge-100M graph the binary heap execution has the worst total runtime.

Concerning the scalability of the parallel execution we observe that the execution with the optimized skip list achieves the highest scalability for both the rmat-node-10M-edge-500M graph and the rmat-node-100M-edge-100M graph. However, the rmat-node-100M-edge-100M graph does not appear good scalability, since this is a sparse graph and helper threads cannot offload many operations from the main thread. The skip list scales better because helper threads perform more relaxations and the main thread can offload more work. As we can see in figures 4.26a and 4.26b, the number of main thread's relaxations decreases much more in case of using a skip list. On the other hand, the execution with the binary heap cannot offload so much work. We suppose that the aborts performed when using the binary heap are useful aborts that would perform useful relaxations. Moreover, the optimized skip list case scales better than the simple skip list case, as it needs a smaller read/write set and in this way, it avoids costly traversals and capacity transactional aborts. Lastly, in case of 28 threads, we can conclude that the NUMA effect degrades the scalability of all our executions due to expensive cache line transfers from one socket to another. The more cache lines are transferred from one socket to another, the worse performance the execution has.

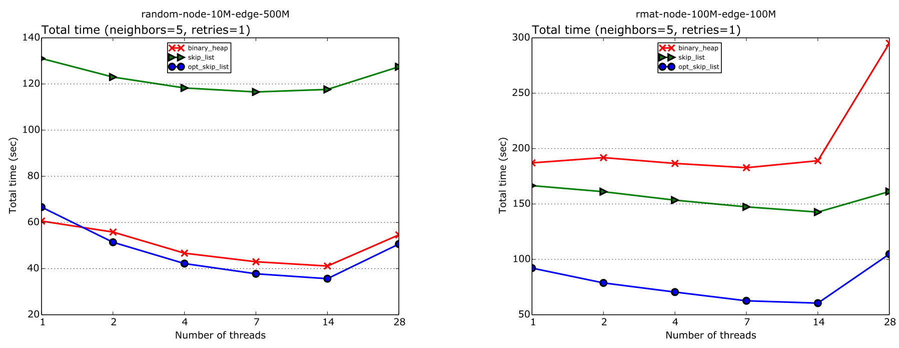
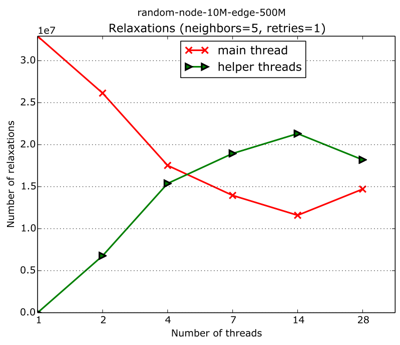
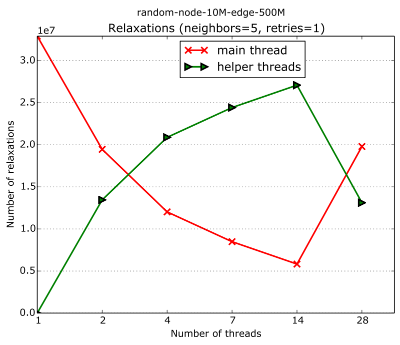


Figure 4.25: Time elapsed in parallel execution for the three different structures used for the priority queue.



(a) Using the binary heap structure.



(b) Using the optimized skip list structure.

Figure 4.26: Distribution of relaxations between the main and helper threads for the random-node-10M-edge-500M graph using two different structures for the priority queue.

Finally, we present the number of transactional commits/aborts of the main and helper threads in the three previous executions. Figure 4.27 depicts the number of commits/aborts of the main thread. The number of transactional aborts in the simple skip list execution is extremely high in comparison with the other two executions, especially in the larger graph. This is due to the large memory that the traversal of the DecreaseKey() operation demands. The traversal in the simple skip list demands a very large read set and this may lead to transactional capacity aborts. Furthermore, there is also a higher probability of data transactional abort. The larger transactional sets that threads need, the higher probability of performing conflicting accesses among them it exists. Secondly, we can notice that the binary heap execution and the optimized skip list execution have comparable number of transactional aborts. In both of these executions the main thread suffers a really low number of transactional aborts and this means that helper threads do not obstruct main thread's progress. Furthermore, the number of transactional aborts is not importantly affected by the number of threads. It remains approximately the same as the number of threads increases, and we conclude that if the NUMA effect did not exist, we could have a better scalability.

As mentioned in previous section the NUMA effect influences the number of transactional aborts. Since the cache line transfers are expensive, a single transaction lasts a lot of time. As a consequence, data conflicts are more possible to be detected in longer transactions and there is a stronger probability of transactional abort due to time interrupt. When a transaction lasts more time than the time quantum, the scheduler of the operating system schedules out the process and the transaction is aborted. In the large rmat-node-100M-edge-100M graph, where many transactional cache lines have to be transferred in DecreaseKey() operation, the transaction is much time-consuming and this results to a large number of transactional aborts.

Similarly to the main thread, the number of transactional aborts of helper threads is increased in the simple skip list execution, as shown in figure 4.28. The traversal in the DecreaseKey() operation is very costly and causes many conflicts for helper threads, too. In the other two executions (binary heap and optimized skip list), the number of transactional aborts is relatively small and shows that the most of the concurrent accesses to the shared data structures are non-conflicting. Moreover, the number of transactional commits increases when adding more threads, since more threads perform relaxations. In this figure, the number of transactional commits in case of binary heap is higher than in the other two executions that use a skip list. Although the number of commits in case of binary heap is larger, these transactional commits do not result to useful relaxations. As we see in figures 4.26a and 4.26b, the optimized skip list structure results to more useful relaxations. We suppose that in case of using the binary heap structure, helper threads have more time to run until the main thread stops them. Thus, they may execute more than once the outer while loop (line 1 in listing 4.3) during the execution of one iteration (outer while loop) of the main thread. However, since helper threads do not extract elements from the priority queue, they will always read the same element in the ReadMin() function (the n-th helper thread reads always the n-th first element in the priority queue). Only when the main thread proceeds to its next iteration, helper threads will read another element to examine. Therefore, in case that helper threads have much time to run and they repeat more than once the outer while loop, they will perform the same process (same relaxations) more than once, executing relaxations for the one and only element that they can read in each main thread's iteration. They will perform many transactional commits (for the same element) without performing new useful relaxations.

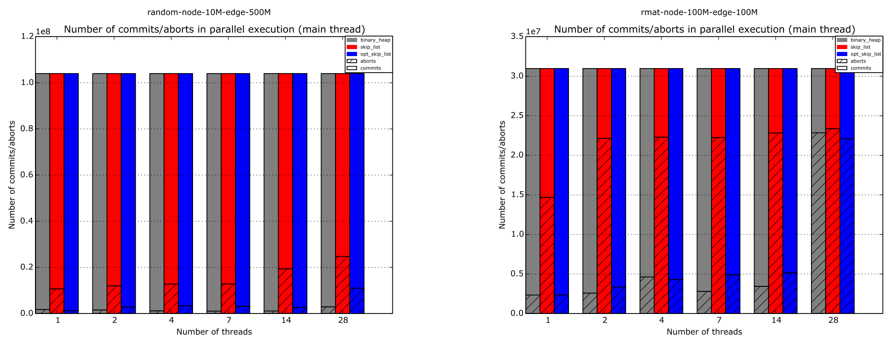


Figure 4.27: The number of commits/aborts of the main thread in parallel execution for the three different structures used for the priority queue.

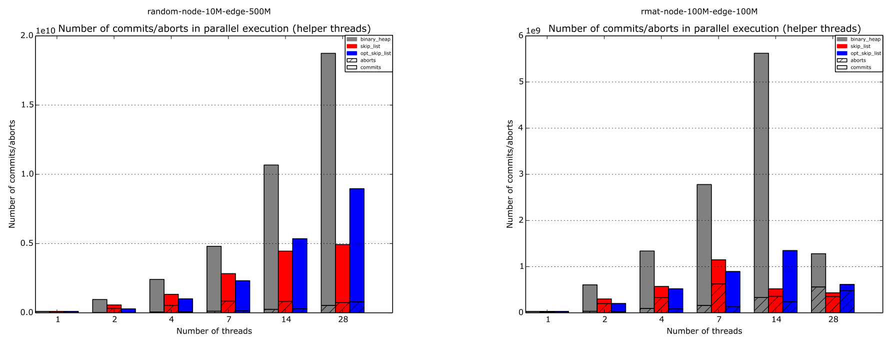


Figure 4.28: The number of commits/aborts of helper threads in parallel execution for the three different structures used for the priority queue.

4.7.3 Results

In the last part of our analysis we evaluated the performance of the parallel execution that employs our optimized skip list. We used the same graphs with the previous performance evaluation of the binary heap version. Figure 4.29 presents the speedups achieved using the optimized skip list in Dijkstra's algorithm. The proposed optimized scheme achieves significant speedups for all evaluated graphs. The maximum speedup achieved is 1.94 for the dense random-node-10M-edge-500M graph (14 threads).

As mentioned in the previous evaluation, the speedup is related to the density of the graph. For more dense graphs, the speedup is greater, as more parallelism can be exposed. Helper threads can offload more work from the main thread in dense graphs, where the number of edges is greater. Conversely, sparse graphs leave limited space for parallelism. In case of 28 threads the performance is degraded due to the NUMA effect.

Our implementation of the optimized skip list achieves higher performance speedup and scales better than the implementation with the binary heap. The main reason is the relaxations performed by the helper threads. As already mentioned, using the optimized

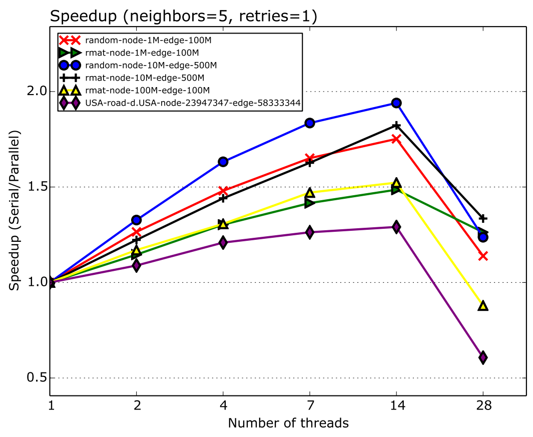


Figure 4.29: Multithreaded speedups for graphs of different density when using our optimized skip list.

skip list helper threads execute more useful relaxations. The number of relaxations performed using the optimized skip list is much larger than using the binary heap structure as figures 4.26a and 4.26b show, while these two implementations have comparable transactional aborts (figures 4.27 and 4.28). In some executions, the relaxations performed using our optimized skip list are double that performed when using the binary heap structure. This may be due to transactional aborts. We suppose that in case of binary heap the transactional aborts performed would result to useful relaxations. Since we cannot determine a policy when a conflict is detected, we are not sure for which transaction will be aborted. It is possible that a large number of the aborted transactions in case of using a binary heap would result to useful relaxations, while this does not happen when using our optimized skip list.

Secondly, the binary heap and the skip list are two completely different data structures. The skip list is a totally ordered data structure. Thus, at a given time, helper threads read the vertices with the first minimum distances (key) from the source (in ReadMin() function) to perform their relaxations. On the contrary, the bineary heap structure is not a totally ordered data structure. At a given time, helper threads read the elements that are located higher (in a small depth) in the binary heap, but these elements are not necessarily the vertices with the first minimum distances from the source. For example, in the binary heap of figure 4.30 if we had 5 helper threads, they would read the elements with the keys 9, 4, 26, 20 and 18. In case case of using a skip list, the 5 helper threads would examine the elements with the keys 4, 9, 11, 18 and 19, since skip list is totally ordered. Thus, we conclude that in each step of the algorithm helper threads examine different elements to perform their relaxations depending on the data structure used. We suppose that the total order that the skip list structure has, may lead helper threads to examine vertices that have obtained their optimal distance from the source with some stronger probability than that in case of using the binary heap structure and perform more useful relaxations than that

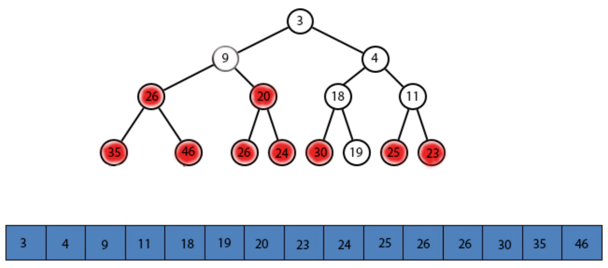


Figure 4.30: An example of binary heap. The vertices with the red color have not obtained their optimal distance from the source.

performed when using the binary heap structure.

Finally, to obtain an estimate of speedup we used the formula 4.6. It gives a speedup approximation based on main thread's relaxations. It also implies that the speedup should increase with the average out-degree. The more dense a graph is, the more parallelism can be exposed. However, this theoretical formula is a simple estimate and constitutes a theoretical upper bound for any performance improvement. It does not take into account the time spent in thread orchestration or delays due to consecutive transactional aborts. Figures 4.31 and 4.32 present a theoretical speedup and the speedup achieved for all evaluated graphs in case of 14 threads using the binary heap structure and the optimized skip list structure, respectively.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4	1.45
rmat-node-1M-edge-100M	3.58	1.27
random-node-10M-edge-500M	3.74	1.46
rmat-node-10M-edge-500M	3.09	1.35
rmat-node-100M-edge-100M	1.22	1.1
USA-road-node-23M-edge-58M	1.91	1.08

Figure 4.31: A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the binary heap structure.

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4.57	1.75
rmat-node-1M-edge-100M	5.11	1.49
random-node-10M-edge-500M	5.14	1.94
rmat-node-10M-edge-500M	4.74	1.82
rmat-node-100M-edge-100M	1.79	1.52
USA-road-node-23M-edge-58M	1.48	1.29

Figure 4.32: A speedup approximation based on main thread's relaxations in case of 14 threads and the speedup achieved using the optimized skip list structure.

Chapter 5

Conclusion and Future Work

In the first part of this thesis we studied concurrent data structures, in particular binary search trees. Search trees are one of the most frequently used in a wider range of applications and parallelizing them introduces many challenges.

The first implementations presented constituted a naive approach for search tree data structure. We implemented binary search trees, AVL trees and Red-Black trees using coarse-grained and fine-grained locking technique for synchronization between threads. Our results demonstrate that coarse-grained implementations do not scale as they do not provide parallelism (serial execution) and fine-grained implementations scale in large trees until a small number of threads. Red-Black tree implementation has the highest performance, since this is a height-balanced tree.

On the other hand, more complex implementations scale better. They have a better synchronization mechanism, smaller contention windows and some of them perform a helping strategy. Secondly, according to the results presented there is no inherent difference between lock-free and lock-based algorithms. The main goal of more complex implementations is to reduce the number and the granularity of locks such that the execution to be close to the asynchronous algorithm. The closer to the sequential algorithm an implementation is, the better it scales. Finally, the scalability of synchronization is mostly a property of hardware. Synchronization primitives are non-scalable on NUMA architectures due to expensive cache line transfers. Therefore, the amount of synchronization on concurrent data structures must be reduced in order to achieve scalability on NUMA architectures.

Apart from the implementations examined on this thesis, there are still many interesting implementations of concurrent search trees to be studied, each of them has its own set of characteristics and innovations. For example, papers [25], [26], [27], [28] and [29] also present complex concurrent search trees to be studied. Furthermore, we could implement concurrent search trees using transactional memory as synchronization mechanism. Even a simple coarse-grained locking HTM implementation of a balanced tree like Red-Black tree could outperform lock-based and wait-free alternatives. However, to enable scalability to high numbers of threads, the programmer needs to be aware of the underlying HTM system's limitations and optimize the code appropriately.

Since the purpose of this analysis is to present concurrent data structures that scale efficiently and could be used in parallel software to improve its performance, besides search trees, there are many other data structures to be studied. Linked lists like FIFO queues, hash tables and priority queues like binary heaps and skip lists are also frequently used data structures and a performance and scalability analysis of such concurrent data structures would be quite challenging, too.

In the second part of this thesis, we applied some parallelization techniques to Dijkstra's algorithm, which is known to be hard to parallelize. We experimented on the algorithm proposed in [23], [24] in order to achieve high scalability in a real HTM system.

The aim of our experimentation is to find a policy that favors the main thread. Helper threads can perform operations simultaneously without interfering in main thread's progress. Thus, to avoid consecutive transactional aborts that would delay main thread's execution, we have forced main thread to acquire the global lock immediately. It can retry the execution of its work in transactional mode only once. Furthermore, helper threads can never acquire the global lock so as not to postpone main thread's execution and they always attempt to execute the critical section in transactional mode.

Secondly, we implemented a more coarse-grained transaction in main thread's code such that to reduce transactions' overhead and we tested the same for helper threads, too. However, performing a coarse-grained transaction can lead to capacity transactional aborts, especially in large graphs, due to the large part of the memory that this scheme accesses. We also experimented in padding technique used on the shared data structures. We concluded that padding does not affect the scalability of the algorithm, but it provides better runtime, as threads can exploit temporal and spatial locality.

Subsequently, we presented graphs for main thread's and helper threads' relaxations as well as the abort ratio of our executions. We can notice that the gain in main thread's relaxations is considerable, but there is no corresponding gain in speedup due to synchronization costs. Moreover, helper threads may perform relaxations that are not useful as the algorithms proceeds. Finally, our results also depict the effect of NUMA architecture. The performance collapses when threads are pinned in both two sockets of our system. As future work, it could be designed a variation of the presented algorithm in which the amount of synchronization would be reduced, and it would scale even in the presence of non-uniformity.

In the last part of this thesis we evaluated Dijkstra's algorithm employing a skip list instead of a binary heap for the priority queue. The results achieved using a simple skip list show that it has much worse performance than the binary heap version of the algorithm. The simple skip list requires a lot of memory and the traversal to its elements is very time-consuming. Thus, we implemented an optimized skip list that contains only discrete keys. Vertices that have the same key are stored to an internal nested list in skip list's element. In this way, our optimized skip list needs less memory, as there are many vertices with the same key during the execution of the algorithm and the traversal to its elements is much faster. This implementation achieves better performance than the binary heap and has higher scalability. Helper threads modify locally the optimized skip list and perform more useful relaxations than in case of using a binary heap.

Our optimized skip list does not contain one element for each vertex of the graph. The number of elements depends on the number of vertices that obtain the same distance (key in the skip list) from the source during the execution of the algorithm. It depends on the weights of graph's edges. As a consequence, we cannot conclude a fixed maximum height for the optimized skip list. We have to examine the average number of elements that our skip list has during the execution and set the maximum height as the logarithm of this average number. Generally, the parameter of the maximum height of our optimized skip list must be examined separately for each graph.

At a given step of the algorithm, the first element of our optimized skip list has the minimum distance-key from the source to be examined. This element may contain many ids of vertices of the graph which have the same minimum distance from the source at this step. Thus, a proposition would be that the main thread could extract the element from the skip list and delegate the different vertices (ids) to helper threads to perform their relaxations. In this way, there would be more than one settled node in a single step and this would result to greater gains. However, in case that a vertex is extracted from the priority queue, all of its edges have to be examined and all of its possible relaxations have to be executed. Otherwise, the algorithm will not be correct. Therefore, the algorithm have to be redesigned, such that each step (iteration of the outer loop) takes as much the most time consuming vertex examination among all threads (both the main and helper threads). In such an execution, the main thread should not stop helper threads when they still have edges to examine.

Our evaluation was performed in Intel's Haswell HTM. As continuation of the present work, the algorithm could be evaluated in other systems that support Hardware Transactional Memory. It could be explored the impact of various TM characteristics on the behavior of the presented schemes, such as resolution policy, version management and conflict detection. For example, an Hybrid conflict resolution policy which tends to favor older transactions against younger ones could be more efficient or a system with larger read/write transaction set would be more suitable for larger graphs. The programmer has to be informed of the underlying HTM system and its characteristics and change the code appropriately in order to achieve high scalability.

Eventually, in paper [24], results demonstrate interesting variations in the available parallelism between different execution phases. As the algorithm proceeds the available parallelism is reduced and the gains from the use of more helper threads are negligible. Thus, as future work it can be examined the different phases of the algorithm and explored more adaptive schemes in terms of the number of helper threads. Helper threads have to be dynamically adjusted as well as the tasks assigned to them.

Bibliography

[1] Amdahl, G., *The validity of the single processor approach to achieving large scale computing capabilities*, In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, April 1967

[2] Flynn, M., *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, 1972.

[3] Georgy Adelson-Velsky, G.; Evgenii Landis (1962)., *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences (in Russian) 146: 263–266. English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

[4] Leonidas J. Guibas and Robert Sedgewick (1978)., *A Dichromatic Framework for Balanced Trees*, Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein., *Introduction to Algorithms*, The MIT Press, 3rd ed., 2009.

[6] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Performance analysis of concurrent red-black trees on htm platforms*, TRANSACT, 2015.

[7] Siakavaras D., Nikas K., Goumas G., and Koziris N., *Massively Concurrent Red-Black Trees with Hardware Transactional Memory*, 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.

[8] R. A. Tarjan., *Efficient top-down updating of red-black trees*, Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.

[9] R. Bayer and M. Schkolnick., *Readings in database systems*, ch. Concurrency of Operations on B-trees, pp. 129–139, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[10] Tudor, D., Guerraoui, R., Trigonakis, V., *Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures*, In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Pages 631-644. ASPLOS 2015.

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun., *A Practical Concurrent Binary Search Tree*, PPoPP 2010.

[12] Drachsler D., Vechev, M.T., Yahav, E., *Practical concurrent binary search trees via logical ordering*, In: PPoPP, pp. 343-356. ACM(2014).

[13] Aravind Natarajan and Neeraj Mittal., *Fast Concurrent Lock-free*, Binary Search Trees. PPoPP 2014.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel., *Non-blocking Binary Search Trees*, PODC 2010.

[15] Maurice Herlihy., Nir Shavit., *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©2008.

[16] Damron P., Fedorova A., Lev Y., Luchangco V., Moiv M., Nussbaum D., *Hybrid transactional memory*, In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, Pages 336-346. ASPLOS 2006.

[17] M. Moir., *Hybrid transactional memory*, July 2005.

[18] Casper J., Oguntebi T., Hong S., Bronson N., Kozyrakis C., Olukotun k., *Hardware acceleration of transactional memory on commodity systems*, In: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Pages 27-38. ASPLOS 2011.

[19] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott., *An integrated hardware-software approach to flexible transactional memory*, SIGARCH Computer Architecture News, 35, June 2007.

[20] J. R. Larus and R. Rajwar., *Transactional Memory. Synthesis Lectures on Computer Architecture.*, Morgan & Claypool, 2007.

[21] Dijkstra, E. W., *A note on two problems in connection with graphs.*, Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390, 1959.

[22] R. C. Prim, *Shortest connection networks and some generalizations.*, In: Bell System Technical Journal, 36 (1957), pp. 1389–1401.

[23] N. Anastopoulos, K. Nikas, G. Goumas, and N. Koziris, *Early experiences on accelerating dijkstra's algorithm using transactional memory.*, in Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), 2009.

[24] K. Nikas, N. Anastopoulos, G. Goumas, N. Koziris, *Employing transactional memory and helper threads to speedup Dijkstra's algorithm.*, Parallel Processing, 2009. ICPP'09. International Conference on, 388-395.

[25] Crain, T., Gramoli, V., Raynal, M., *A speculation-friendly binary search tree*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 161-170, ACM New York, 2012.

[26] Chatterjee, B., Nguyen, N., Tsigas, P., *Efficient lock-free binary search trees*, In: PODC '14, Proceedings of the 2014 ACM symposium on Principles of distributed computing Pages 322-331, ACM New York, 2014.

[27] Prokopec, A., Bronson, N., Bagwell, P., Odersky, M., *Concurrent tries with efficient non-blocking snapshots*, In: PPoPP '12, Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming Pages 151-160 , ACM New York, 2012.

[28] Crain, T., Gramoli, V., Raynal, M., *A contention-friendly binary search tree*, In: Euro-Par'13, Proceedings of the 19th international conference on Parallel Processing Pages 229-240, Springer-Verlag Berlin, Heidelberg, 2013.

[29] Natarajan, A., Savoie, L., Mittal, N., *Concurrent Wait-Free Red Black Trees*, In: SSS 2013 Proceeding of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255 Pages 45-60, Springer-Verlag New York, 2013