UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Towards Methods to Exploit Concurrent Data Structures on Heterogeneous CPU/iGPU Processors

Permalink
https://escholarship.org/uc/item/7r21z1s7

Author
Fuentes, Joel

Publication Date
2019

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Towards Methods to Exploit Concurrent Data Structures on Heterogeneous CPU/iGPU Processors

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Joel Fuentes

Dissertation Committee:
Professor Isaac D. Scherson, Chair
Professor Alex Nicolau
Professor Raymond Klefstad

2019

© 2019 Joel Fuentes

DEDICATION

To my family and parents.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

LIST OF ALGORITHMS **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **ix**

ABSTRACT OF THE DISSERTATION **xi**

1 Introduction **1**
1.1 Multi-thread and Data Parallelism 3
1.2 Challenges in Heterogeneous Concurrent Programming 3
1.3 Contributions of the Thesis 5

2 Preliminaries **7**
2.1 Common Terms 7
2.2 Formal definitions 9
2.3 Heterogeneous CPU/iGPU Architecture and Programming Framework . . . 10
2.3.1 Intel's Multi-Core CPU/iGPU Processor 10
2.3.2 C for Media 13

3 Related Work **18**
3.1 Synchronization Mechanisms for Multi-Core CPUs 18
3.2 Concurrent Data Structures for Multi-Core CPUs 19
3.2.1 Stacks 20
3.2.2 Queues 21
3.2.3 Lists 22
3.2.4 Maps 23
3.2.5 Trees 24
3.2.6 Skiplists 25
3.3 Concurrent Data Structures for GPUs 25

4 Exploiting Concurrent Data Structures **26**
4.1 Synchronization techniques . . . 28
4.2 Concurrent Data Structures with SIMD Processing . . . 31
4.2.1 SIMD-node transformations . . . 33
4.2.2 SIMD-node Traversals . . . 36
4.2.3 Restrictions and Considerations . . . 37

5 Contributed Multi-threaded Synchronization Mechanisms **38**
5.1 Spatial Locks: A Specilized Synchronization mechanism for Geometric Algorithms . . . 38
5.1.1 Paradigm and Interface . . . 40
5.1.2 Implementation . . . 42
5.1.3 Settings and restrictions . . . 46
5.1.4 Parallel Mesh Simplification using Spatial Locks . . . 47

6 Contributed Concurrent Data Structures with SIMD Processing **57**
6.1 k-ary Tree based on SIMD-node transformations . . . 57
6.2 Lock-free Skiplist based on SIMD-node transformations . . . 60
6.2.1 Structure details . . . 61
6.2.2 Implementation considerations . . . 63
6.2.3 Data structure operations . . . 64
6.2.4 Correctness . . . 73
6.2.5 Experimental Results . . . 78
6.3 SIMT-friendly Data Structures . . . 83
6.3.1 SIMT Tree Search . . . 85

7 Conclusions and Future Work **87**
7.1 Conclusions . . . 87
7.2 Future Work . . . 88

Bibliography **90**

LIST OF FIGURES

Page

2.1 Architecture components layout for an Intel processor with Gen9 Integrated Graphics Processor [40]. 11
2.2 Intel's Gen9 Execution Unit (EU) [40]. 12
2.3 Layout of the Intel's Gen9 iGPU[40]. 13

4.1 Classification of data structures 26
4.2 Multi-thread parallelism 27
4.3 Lock-based synchronization 28
4.4 Non-blocking synchronization 30
4.5 Multi-thread + SIMD parallelism 31
4.6 Θ-node transformation from a one-dimensional (a) and multi-dimensional (b and c) structures 34

5.1 Spatial hashing where objects are mapped into uniformly-sized cells 40
5.2 Surface mesh simplification algorithm applied to a mesh with different simplification levels [5]. 48
5.3 Contraction of the edge $(v,\ v_u)$ into a single vertex. The shaded triangles become degenerate and are removed during the contraction. 49
5.4 Total time spent by the parallel mesh simplification algorithm in simplifying surface meshes with up to 3,136,234 triangles. 52
5.5 Total time spent by the mesh simplification algorithm using internal locks and Spatial Locks. 55
5.6 Total time spent by both algorithms in simplifying a surface mesh with 2,436,234 triangles. 56

6.1 Skiplists before and after Θ-node transformation 61
6.2 Skiplist structure 61
6.3 Insertion of key 67 70
6.4 Insertion of key 21 71
6.5 Throughput of different Skiplists on GenX and CPU 79
6.6 Speedup of CMSL on GenX over Fraser's skiplist for CPU 80
6.7 Scalability of CMSL by the number of threads. Operations are 90% search, 5% insert and 5% delete. 83
6.8 SIMT processing 84

LIST OF TABLES

Page

4.1 Classification of concurrent data structures based on their implementation . 32

5.1 Performance of the PMS algorithm implemented with Spatial Locks 52
5.2 Performance of mesh simplification algorithms using internal locks and Spatial Locks 54

6.1 Performance comparison between CMSL and M&C skiplists. Relative efficiency of CMSL is calculated by $\frac{e_{CMSL}}{e_{M\&C}}$ from equation (1). Tests on 1M keys. 82

LIST OF ALGORITHMS

Page

1 Traversal of a Θ_X transformed tree 37
2 Lock object in Spatial Hash Table 44
3 Unlock object in Spatial Hash Table 45
4 Parallel Mesh Simplification using Spatial Lock 50
5 Find list 65
6 Search key 66
7 Insert key 67
8 Insert in Chunk 68
9 Delete key 72
10 General SIMT tree search 85

ACKNOWLEDGMENTS

I would like to thank many people without whom this thesis would not have been possible.

First, I thank my advisor Isaac D. Scherson for the guidance and inspiration he gave me during this journey towards a Ph.D. He not only taught me fundamental knowledge necessary for writing this thesis, but also gave me a lot of useful career advice. His broad knowledge and experience have given me a better perspective on the “big picture” of this thesis.

I thank the Department of Computer Science and Information Technologies (DCCTI) from the Universidad del Bío-Bío for providing me an opportunity to develop a research career. I am very grateful to all the DCCTI members for their motivation and encouragement to pursue a Ph.D. in the U.S.

Also, I would like to acknowledge the funding sources that made my dissertation research possible. Fellowship support was provided by the Universidad del Bío-Bío and Becas Chile CONICYT. Additional support was provided by the UCI AGS Division through travel grants.

I am also thankful to Intel Corporation and the CM Compiler team for providing internship appointments during my graduate studies. They also provided me the computing resources needed for the experimental part of this thesis. I am grateful to Wei-yu Chen and Guei-yuan Lueh for many interesting discussions on iGPUs and compilers, which inspired me to do research on concurrent data structure with SIMD/SIMT processing.

I am especially thankful to my wife, Lidia. She stood by my side during all of my ups and downs, and her constant support and encouragement enabled me to finish this journey. Finally, I thank my parents, Sebastian and Carmen, for always motivating me to pursue my dreams and being there whenever I needed them. Without them, I would not have come this far.

CURRICULUM VITAE

Joel Fuentes

EDUCATION

Doctor of Philosophy in Computer Science	**2019**
University of California, Irvine	*Irvine, CA, USA*
Master in Computer Science	**2016**
University of California, Irvine	*Irvine, CA, USA*
Bachelor in Science, Computer Engineering	**2011**
Universidad del Bío-Bío	*Chillán, Chile*

RESEARCH EXPERIENCE

Graduate Research Assistant	**2014–2019**
University of California, Irvine	*Irvine, CA, USA*
Research Intern	**2017–2018**
Intel Corporation	*Santa Clara, CA, USA*

TEACHING EXPERIENCE

Teaching Assistant	**2014–2019**
University of California, Irvine	*CA, USA*

REFEREED JOURNAL PUBLICATIONS

Synchronizing Parallel Geometric Algorithms on Multi-Core Machines **2018**
Joel Fuentes, Fei Luo, Isaac D. Scherson. International Journal of Networking and Computing 8 (2), 240-253

A Method to Find Functional Depen- dencies Through Refutations and Duality of Hypergraphs **2015**
Joel Fuentes, Pablo Sez, Gilberto Gutirrez, Isaac D. Scherson. The Computer Journal 58 (5), 1186-1198

REFEREED CONFERENCE PUBLICATIONS

A Lock-Free Skiplist for Integrated Graphics Processing Units **2019**
Joel Fuentes, Wei-yu Chen, Guei-Yuan Lueh, Isaac D. Scherson. International Parallel and Distributed Processing Symposium

Synchronizing Parallel Geometric Algorithms on Multi-Core Machines **2017**
Joel Fuentes, Fei Luo, Isaac D. Scherson. 2017 Fifth International Symposium on Computing and Networking (CANDAR), 401-407

Dynamic Creation of Virtual Machines in Cloud Computing Systems **2017**
Fei Luo, Isaac D. Scherson, Joel Fuentes. 25th International Conference on Systems Engineering (ICSEng), 316-323

A Novel Genetic Algorithm For Bin Packing Problem In jMetal **2017**
Fei Luo, Isaac D. Scherson, Joel Fuentes. IEEE International Conference on Cognitive Computing (ICCC), 17-23

Mining for Functional Dependencies Using Shared Radix Trees in Many-Core Multi-Threaded Systems **2017**
Joel Fuentes, Claudio Parra, David Carrillo and Isaac D. Scherson. Emergent Computation, 303-319

ABSTRACT OF THE DISSERTATION

Towards Methods to Exploit Concurrent Data Structures on Heterogeneous CPU/iGPU Processors

By

Joel Fuentes

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Isaac D. Scherson, Chair

Heterogeneous processors, consisting of CPU cores and an integrated GPU on the same die, are currently the standard in desktop and mobile platforms. The number of CPU cores is increased with every new generation, and integrated GPUs are constantly being improved in performance and energy efficiency. This raises the importance of developing programming methods and techniques to benefit general purpose applications on heterogeneous processors as more parallelism can be achieved. This dissertation addresses this challenge by studying new ways of exploiting parallelism on CPU cores as well as the integrated GPU, focusing on blocking and non-blocking data structures.

A new thread synchronization mechanism for parallel geometric algorithms dubbed Spatial Locks is first introduced. This synchronization mechanism ensures thread synchronization on geometric algorithms that perform concurrent operations on geometric surfaces in two- or three-dimensional spaces. A parallel algorithm for mesh simplification is chosen to illustrate the Spatial Locks usefulness when parallelizing geometric algorithms with ease on multi-core machines. Experimental results show the advantage of using this synchronization mechanism where significant computational improvement can be achieved compared to alternative approaches.

Non-blocking data structures are commonly used by many multi-threaded applications, and their implementation is based on the use of atomic operations. New computing architectures, such as Intel CPU/iGPU processors, have incorporated data-parallel processing through SIMD instructions, including in some cases support for atomic SIMD instructions and SIMT processing on the integrated GPU. A new framework called *SIMD-node Transformations* to implement non-blocking data structures using multi-threaded and SIMD processing is proposed. As a result, it is shown that one- and multi-dimensional data structures, such as skiplists, k-ary trees and multi-level lists, can embrace SIMD processing through these transformations. Finally, important performance gains obtained when applying these transformations to concrete data structures are reported.

Chapter 1

Introduction

With the advent of heterogeneous computing systems that use silicon devices with many CPU cores and an integrated GPU (iGPU) per chip, also known as Multi-Core CPU/iGPU processors, new challenges are posed to programming applications that attempt to use parallelism to achieve a significant computational improvement. Typical Multi-core CPU/iGPU computers use chips that contain two, four or more cores each, and an iGPU with a few dozens of GPU cores (Execution Units - EU). These heterogeneous processors also include an on-chip hierarchical shared cache system that provides local and shared caches for the CPU cores and iGPU's EUs. An interface to a large common main DRAM storage completes the solid state memory hierarchy. These systems are normally programmed using threads that are managed by the operating system to execute in the available cores attempting to use as much parallelism as possible. The main problem that arises is the management of shared data structures in the shared hierarchical memory to guarantee synchronized access to the data structures, avoiding deadlock and providing a correct access sequence as required by the program. Additionally, the inclusion of Data-Parallelism on these architecture through SIMD (single instruction multiple data) instructions has brought a new dimension of parallelism that has to be explored to exploit shared data structures.

SIMD processing is a term introduced by Michael J. Flynn in 1966 [13]. It refers to a processor architecture that executes a single instruction over multiple data items at the same time. However, it is not straightforward to take advantage of this architecture. The programmer requires to manually manage the data in arrays or vectors (SIMD-friendly format), so the SIMD execution unit can act over them. New CPU/iGPU computing architectures, such as Intel's heterogeneous processors, have brought new processing capabilities such as SIMD and SIMT processing on the iGPU. Along with the inclusion of GPU cores within the same CPU die, the performance and programmability of the Intel's integrated graphics processing unit (GenX iGPU) has been significantly improved over earlier generation of integrated graphics [40]. Besides providing specialized graphics hardware, GPU cores are also getting more programmable as new generations are released, making them amenable to general-purpose computing. In terms of use and presence, the GenX iGPU is massively used since it is present in almost every processor Intel releases for desktop and mobile.

Maurice Erlihy and Nir Shavit wrote a book [35] that discusses the methodologies used to properly program Multi-Core systems. Exploiting parallelism depends very much on the synchronization mechanisms available to avoid shared data conflicts. Their book has become a classic and has been adopted to teach Multi-core programming courses. Mutual exclusion concepts and blocking properties discussed in their book, such as deadlock-freedom and starvation-freedom, are fundamental when implementing new synchronization primitives. On the other hand, non-blocking techniques and properties such as lock-freedom, wait-freedom and obstruction-freedom, are essential for achieving correct non-blocking executions in concurrent programs. The authors also pioneered the linearization correctness condition for non-blocking data structures, which became widely used for different authors and researchers of new non-blocking designs.

1.1 Multi-thread and Data Parallelism

The spread of multi-core CPU architectures along with the deceleration of Moore's law have shown a change on how we develop software. Until recently, advances in technology meant advances in clock speed, so software would effectively "speed up" by itself over time. Now, however, this free ride is over. Advances in technology will mean increased parallelism and not increased clock speed, and exploiting such multi-thread parallelism is one of the outstanding challenges of modern Computer Science. Therefore, with the advent of computing systems with many CPUs per chip, well-known as multi-core and many-core machines, parallelizing algorithms is an objective in order to achieve significant computational improvement.

Data parallelism refers to scenarios in which the same operation is performed in parallel on elements in a source collection or vector. Accelerators targeting data-parallelism, such as GPU, have become very popular for graphics applications, media, video game, and machine learning. Notice that modern CPU architecture also support data-parallelism through vector operations; e.g. Advanced Vector Extensions (AVX, also known as Sandy Bridge New Extensions) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD.

1.2 Challenges in Heterogeneous Concurrent Programming

In recent years, GPUs have become essential for many computer tasks and they are used to accelerate applications in a wide variety of fields. They are well-known for having the potential to accelerate many kinds of computations, especially if these computations involve large number of data elements that can be executed in parallel. GPUs were initially created as a discrete component for computer machines, but today they are also widely available as System on a Chip (SoC) component in the same CPU die; as is the case of Intel processors.

Even when the use of these heterogeneous processors has become very common, especially on data parallel algorithms such as graphics/media applications, machine learning, etc., applications with concurrent data structures have not been explored yet. There exists a big group of data structures whose traversal algorithms can be improved by using data-parallel processing.

In a system where a discrete GPU is present, the work flow involves transferring data from main memory to GPU device memory, performing the computations on the GPU, and moving the results back to main memory. This data movement is done through the PCI express bus, which is orders of magnitude slower than making copies within the main memory. Heterogenous platforms with an iGPU, on the other hand, offer an environment where memory is shared between the CPU and the accelerator (in this case, iGPU). Thus, all the data movement can be avoided. The use of this shared memory presents demanding challenges in terms of thread synchronization when shared memory blocks from a concurrent data structures are accessed/updated from both devices; i.e. how mutual exclusion and/or linearizability are guaranteed.

In terms of programming languages, they need to provide an intuitive interface to express data-parallelism at a high level of abstraction. Compilers for these heterogeneous architectures need to efficiently exploit the SIMD capability of the iGPU. Optimizations and validations on data-parallel operations should be considered as well, especially when the used is in charge of defining vector operations and SIMD sizes.

Most research in frameworks aimed at scheduling tasks on heterogeneous architectures, composed of CPU's and GPUs, has focussed on optimizing execution time without considering energy consumption. However, a CPU core and an iGPU exhibit different performance/energy trade-offs, this is, a workload can run faster on one device but consume less energy on the other one. Thus, in order to benefit from the potential energy efficiency that the accelerators can provide in these heterogeneous chips, the runtime scheduler also needs to consider the performance/energy asymmetry when making a scheduling decision.

1.3 Contributions of the Thesis

The main contributions of this thesis are:

1. A thread synchronization mechanism dubbed Spatial Locks for parallel geometric algorithms is presented. We show that Spatial Locks ensure thread synchronization on geometric algorithms that perform concurrent operations on geometric surfaces in two-dimensional or three- dimensional space. The proposed technique respects the fact that these operations follow a certain order of processing, i.e. priorities. Parallelizing these kinds of geometric algorithms using Spatial Locks requires only a simple parameter initialization, rather than modifying the algorithms themselves together with their internal data structures. A parallel algorithm for mesh simplification is chosen to show the Spatial Locks usefulness when parallelizing geometric algorithms with ease on multi-core machines. Experimental results illustrate the advantage of using this synchronization mechanism where significant computational improvement can be achieved compared to alternative approaches.

2. We present a new framework called Θ*-node Transformations* to implement non-blocking data structures using multi-threaded and SIMD processing. As a result, it is shown that one- and multi-dimensional data structures, such as skiplists, k-ary trees and multi-level lists, can embrace SIMD processing through these transformations and remain linearizable. Some guidelines to implement new Θ-node-based data structures

from scratch are also provided.

3. A lock-free skiplist based on Θ-node transformations is presented. It capitalizes on the GenX iGPU computational model to accelerate its search and update operations. It uses non-blocking techniques and all its operations are lock-free. To the best of our knowledge, this is the first implementation of a lock- free data structure for iGPU. Experimental results show that our proposal is more compute-efficient than an existing discrete GPU implementation and outperforms state-of-the-art lock-free and lock-based skiplists for multi-core CPU, achieving up to 3.5x speedup. Additionally, energy savings of up to 300% are obtained when running different skiplist workloads on iGPU instead of CPU cores, hence further improving energy efficiency.

4. A SIMT-based tree search algorithm is presented. It capitalizes on the fact that binary search trees algorithms have low divergence and are simple. Experimental results show that SIMT-based search achieve higher throughput for some tree structures than multi-threaded algorithms for CPU.

Chapter 2

Preliminaries

2.1 Common Terms

- **Mutual exclusion**: It is the problem of making sure that only one thread at a time can execute a particular block of code, and is one of the classic coordination problems in multiprocessor programming.
- **Atomic operation**: Modern multiprocessor hardware provides special *read-modify-write* instructions that allow threads to read, modify, and write a value to memory in one atomic (i.e., indivisible) hardware step. In this thesis, we refer to the specific atomic *compare-and-exchange* operation (CMPXCHG) as *compare-and-swap* operation (CAS).
- **ISA**: Instruction set architecture.
- **Kernel**: A compute kernel is a routine compiled for high throughput accelerators (such as graphics processing units (GPUs), digital signal processors (DSPs) or field-programmable gate arrays (FPGAs)), separate from but used by a main program (typically running on a central processing unit). They are sometimes called compute

shaders, sharing execution units with vertex shaders and pixel shaders on GPUs, but are not limited to execution on one class of device, or graphics APIs.

- **GenX**: Intel architects colloquially refer to Intel processor graphics architecture as simply Gen, shorthand for Generation. A specific generation of the Intel processor graphics architecture may be referred to as Gen9 for generation 9 [40].
- **SISD**: Single instruction, single data
- **SIMD**: Single instruction, multiple data
- **SIMD Width**: Size of the vector operation to be processed by a SIMD instruction.
- **SIMT**: Single instruction, multiple threads.
- **Cache Coherency**: In most modern multiprocessors, each processor has an attached cache, a small, high-speed memory used to avoid communicating with large and slow main memory. Each cache entry holds a group of neighboring words called a line, and has some way of mapping addresses to lines. Each cache line has a tag, which encodes state information. The cache coherence protocol detects synchronization conflicts among individual loads and stores, and ensures that different processors agree on the state of the shared memory. When a processor loads or stores a memory address a, it broadcasts the request on the bus, and the other processors and memory listen in (sometimes called snooping).
- **Exponential Backoff**: It is an algorithm that uses feedback to multiplicatively decrease the rate of some process, in order to gradually find an acceptable rate.

2.2 Formal definitions

Definition 2.1. ***Mutual exclusion*** *is when critical sections of different threads do not overlap. For threads A and B, and integers j and k, either* $CS_A^k \rightarrow CS_B^j$ *or* $CS_B^j \rightarrow CS_A^k$.

Definition 2.2. ***Deadlock-freedom*** *guarantees that if some thread attempts to acquire the lock,then some thread will succeed in acquiring the lock. If thread A calls lock() but never acquires the lock, then other threads must be completing an infinite number of critical sections.*

Definition 2.3. ***Starvation-freedom*** *guarantees that every thread that attempts to acquire the lock eventually succeeds. Every call to lock() eventually returns. This property is sometimes called lockout freedom.*

Definition 2.4. *A method is* ***lock-free*** *if it guarantees that infinitely often some method call finishes in a finite number of steps.*

Definition 2.5. *A method is* ***wait-free*** *if it guarantees that every call finishes its execution in a finite number of steps.*

Definition 2.6. *A method is* ***obstruction-free*** *if, from any point after which it executes in isolation, it finishes in a finite number of steps.*

Definition 2.7. *An execution of a concurrent system is modeled by a* ***history*** H*, a finite sequence of method invocation and response events. A subhistory of a history* H *is a subsequence of the events of* H.

Definition 2.8. *A history* H *is* ***linearizable*** *if it has an extension* H' *and there is a legal sequential history* S *such that*

- *L1. complete*(H') *is equivalent to* S*, and*
- *L2. if a method call* m_0 *precedes method call* m_1 *in* H*, then the same is true in* S.

We refer to S as a linearization of H (H may have multiple linearizations).

Definition 2.9. *For implementations that do not use locks, the* ***linearization point*** *is typically a single step where the effects of the method call become visible to other method calls.*

Definition 2.10. *The* ***ABA problem*** *ocurrs especially in dynamic memory algorithms that use conditional synchronization operations such as CAS operations. Typically, a reference about to be modified by a CAS operation changes from a, to b, and back to a again. As a result, the CAS operation call succeeds even though its effect on the data structure has changed, and no longer has the desired effect.*

2.3 Heterogeneous CPU/iGPU Architecture and Programming Framework

2.3.1 Intel's Multi-Core CPU/iGPU Processor

Intel introduced a few years ago the first heterogeneous processor that integrates GPU cores along with the CPU cores on the same die. The integrated graphics processor that used to be part of MCH (Memory Controller Hub) or North-Bridge is now placed inside the processor. This move further improved the performance of Intel iGPU by a significant amount. Besides providing specialized graphics hardware, GPU cores are also getting more and more programmable making them amenable to general-purpose computing. GPU architecture, however, evolved out of its need to support real-time computer graphics and video/image processing with tremendous inherent parallelism. As a result their emphasis on throughput optimization and expectation of explicit parallelism is much more aggressive than the CPU cores. In other words, while the CPU cores are better suited for applications that are latency-sensitive and have implicit instruction-level parallelism, the GPU cores target throughput-oriented workloads with abundant parallelism. iGPU architectures are named GenX, where X is the number of a specific generation. In this thesis, all the computational experiments were carried out using a Gen9 iGPU.

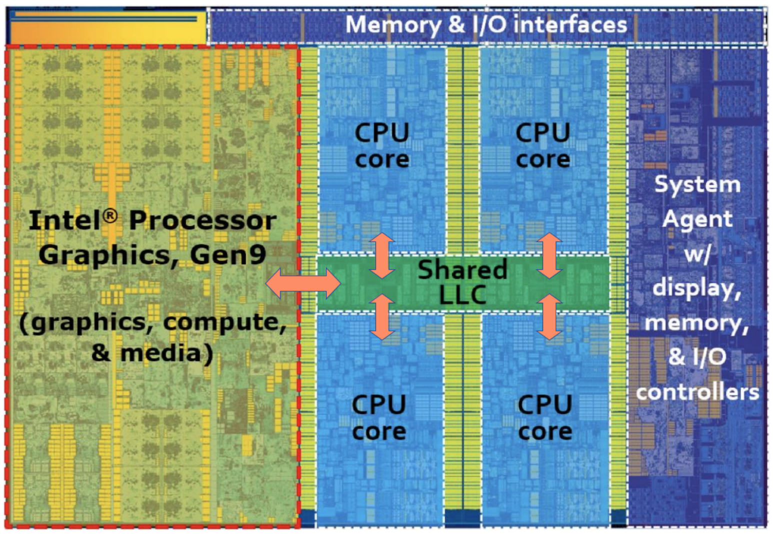

Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

Figure 2.1: Architecture components layout for an Intel processor with Gen9 Integrated Graphics Processor [40].

The Intel's iGPU delivers a full complement of high-throughput floating-point and integer compute capabilities, a layered high bandwidth memory hierarchy, and deep integration with on-die CPUs. The communication between CPU cores, caches and the iGPU is done through a bi-directional ring interconnect that has a 32-byte wide bus. Figure 2.1 illustrates the architecture components of an Intel processor with Gen9 iGPU. Some configurations include an embedded EDRAM exclusively for iGPU, along with coherence within its memory hierarchy and the CPU cores through a shared Last Level Cache (LLC) [40]. When we have both throughput-oriented and latency-oriented cores on the same die, it is but natural to think of exploiting such cores operating in tandem to address heterogeneous workloads. From a performance perspective, GPU cores placed inside the processor die get a boost from cutting-edge CPU manufacturing process and access to large on-chip cache. Furthermore, overhead of transferring data between CPU and GPU memory can virtually be eliminated as they share the same main memory.

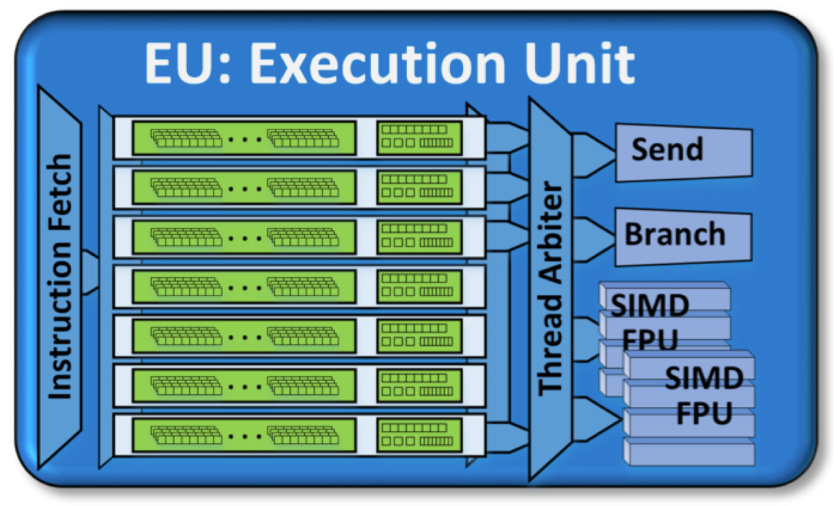


Figure 2.2: Intel's Gen9 Execution Unit (EU) [40].

The foundational building block of GenX iGPUs is the Execution Unit (EU). Figure 2.2 shows the diagram of a Gen9's EU. The architecture of an EU is a combination of simultaneous multi-threading (SMT) and fine-grained interleaved multi-threading (IMT). Each EU has 7 threads, a total of 128 32-byte general purpose registers (GRF), and a pair of SIMD floating point units (SIMD FPUs). Depending on the software workload, the hardware threads within an EU may all be executing the same compute kernel code, or each EU thread could be executing code from a completely different compute kernel. The execution state of each thread, including its own instruction pointers, are held in thread-specific ARF registers. On every cycle, an EU can co-issue up to four different instructions, which must be sourced from four different threads. The EUs thread arbiter dispatches these instructions to one of four functional units for execution.

The EU Instruction Set Architecture (ISA) and associated general purpose register file are all designed to support a flexible SIMD width. Thus for 32-bit data types, the Gen9 FPUs can be viewed as physically 4-wide. But the FPUs may be targeted with SIMD instructions and registers that are logically 1-wide, 2-wide, 4-wide, 8-wide, 16-wide, or 32-wide.

Subsequently, a subslice is an array of EUs that has its own local thread dispatcher and its own supporting instruction caches. Finally, subslices are clustered into slices, that manage thread dispatch routing, level-3 cache and atomic and barrier operations. Figure 2.3 illustrate the organization of slices, subslices and EUs in Gen9. Notice that, similar to CPU cores, the Gen9 iGPU includes a hierarchy of cache memory L3, L2 and L1.

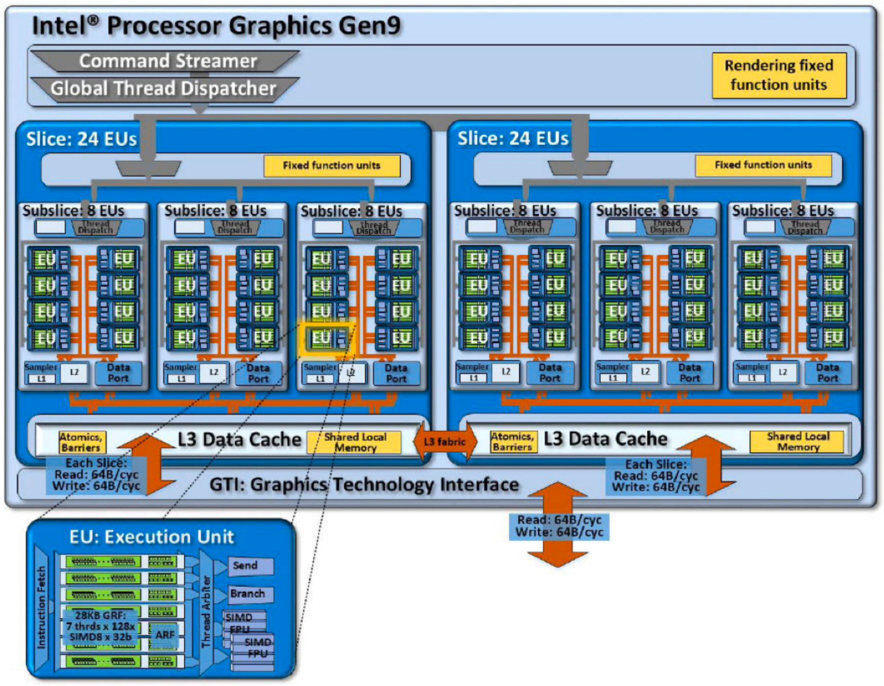


Figure 2.3: Layout of the Intel's Gen9 iGPU[40].

2.3.2 C for Media

C for Media (CM) is a high-level programming environment based on C/C++ that provides an efficient interface to utilize GenX iGPU. The CM toolset consists of two primary components: 1) the compiler for the CM language, which is a subset of C++, for the GPU-targeted code; and 2) the CM runtime, which provides an API to C++ programs, for setting up the GPU to transfer data and execute GPU-targeted code. The CM project was recently open-sourced [39].

CM supports two basic component data types, matrix and vector, and two reference component data types, *matrix_ref* and *vector_ref*, are defined as C++ template classes. Parameters are the element type and size of matrix/vector, which must be compile-time constants. The element type must be one of the basic types supported by CM. Following are examples of matrix/vector declaration:

```
matrix<int, 4, 8>  m; // a 4x8 integer matrix
vector<short, 8>  v; // a vector with 8 short integers
```

Matrix_ref and *vector_ref* define references to basic matrix/vector objects. No memory space is allocated to reference variables. For example,

```
vector_ref<int, 8> v_ref(m.row(2))
```

defines a vector referring to the second row of matrix m.

The primary goal of the CM platform is to allow programmers to keep their existing applications and delegate the data-parallel or massively-parallel segments of the application to the on-die iGPU cores. The language was designed to efficiently leverage SIMD capability of the iGPU, providing a syntax that facilitates implementing data-parallel applications. As an example, *16-int* vector operations can be easily expressed in CM as:

```
vector<int, 16> v1, v2, result;
...
result = v1 + v2;
```

Then, these *16-int* vector operations are converted into SIMD operations by the CM compiler. Thus, the CM compiler generates optimized GenX instructions extracting as much parallelism as possible from the underlying iGPU hardware.

C structures are also supported. They can contain all the supported scalar data types and vector/matrix object types, with the restriction that the structure data members must be properly aligned. The reference object cannot be declared as a structure field. Mask for SIMD comparison and merge operations has three kinds of representations in CM. It can be an integer, or a vector/matrix of short integers

The iGPU functions, also termed as kernels, are instantiated into user-specified number of threads. Each thread is then scheduled to run on an in-order EU. It is worth noting here that, unlike OpenCL or CUDA, a CM thread is equivalent to a CUDA warp and it executes SIMD instructions. SIMD computations over data blocks are expressed in CM via vector and matrix operations and efficiently translated to GenX-EU ISA (Instruction Set Architecture) by the CM compiler.

CM supports standard C++ operations for the allowed basic data types, with some deviation for float type. For component data types, CM supports some overloaded operators, including assignment, matrix/vector constructor, and component-wise arithmetic/shift/logic/comparison operators following C++ standard. In addition, some new operators are supported for matrix/vector manipulation.

- *Read and write*: reads data blocks from memory to registers and writes the computed output from registers to memory respectively. A vector of indices (offsets) can be used for scattered reads/writes. In order to reduce the load on the message gateway as well as maximize bandwidth utilization of read operation bandwidth we attempt to utilize the full capacity of the read operations (i.e. 32 dwords for linear block reads and 16 dwords for scattered reads) whenever possible.

- *Select*: CM provides a set of select functions for referencing a subset of the elements of vector/matrix objects. Some of these operations return a reference to the elements of matrix/vector objects, so they can be used as L-values in the statements.

- *Atomic read-modify-write*: GenX supports up to SIMD16 atomic read-modify-write operations on data block (contiguous data elements), i.e. it is guaranteed by GenX to perform the read-modify-write operation atomically on the data block. Similar to C++, it sup- ports the commonly used operations ADD, SUB, INC, COMPXCHG (compare-and-swap, CAS), and so on.

- *Format*: CM supports reinterpreting the element type of a matrix/vector and change the shape to another matrix/vector. As an example, format¡int, 4, 8¿(v) changes the type of elements in vector v to int and returns a 4x8 integer matrix.

- *Merge*: It is used to model masked copy operation. There are two forms of merge, v.merge(x, mask) and v.merge(x, y, mask). The elements in the file result are determined by the mask. The first one copies one element from x to v if the corresponding bit in mask is true. The second one copies an element in x if the corresponding mask bit is true; otherwise, it copies an element from y.

- *Boolean reduction*: it is applied to mask objects. Two operations are suppoted, any and or. v.any() returns 1 if any of the elements of v is 1; otherwise it returns 0. V.all() returns 1 if all of the elements of v is 1; otherwise it returns 0. rescheduling.

The CM toolset consist of a compiler, runtime and a few other components. The execution model as follows: first, the GenX kernels are compiled by the CM compiler to an intermediate language (called Common-ISA) file. Common-ISA is a high-level, generic assembly language that can be translated to run on any current or future GenX GPUs. At runtime, CM JIT compiler translates the Common-ISA into GenX executable code. Next, the application is compiled into an x86 binary with any C++ compiler of developers choice. At runtime, the application calls the CM-runtime APIs to setup and execute on the GPUs. The CM runtime provides the desired hardware abstraction layer to the application - it manages device-creation, setting-up input and output buffers, kernel-creation, setting-up thread arguments, and dispatching kernels to the GPU. During kernel-creation, CM runtime invokes the JIT compiler to generate GenX binary from Common-ISA. Subsequently, thread creation and scheduling is performed entirely in hardware by the GPUs thread dispatcher.

Chapter 3

Related Work

3.1 Synchronization Mechanisms for Multi-Core CPUs

Many synchronization mechanisms for shared-memory systems have been proposed over the years. Locks, Semaphores, Monitors and many other variants are used by most of the multi-threaded applications that require synchronization. Even though they were proposed several years ago, the design of efficient multi-core locks is still a hot research topic [22, 47, 9], considering also that the current trend is to include more CPU cores per chip.

Attempts to parallelize Geometric Algorithms have been focusing on partitioning the space into sub-regions, computing a sub-solution for each sub-region, and finally merging all of them into a final result. Some examples are the algorithms proposed in [7, 6] to solve the 3D Delaunay Triangulation. Another well-known approach is acquiring exclusive access to the containing sub-region and cells around it, as presented in [43] for the parallelization of randomized incremental construction algorithms. Batista et al. presented in [3] a few strategies to parallelize some geometric algorithms such as vertex-locking strategy, cell-locking strategy and other variants. These strategies use additional variables within vertices and through atomic operations they guarantee synchronization. Additionally, priority locks are used to avoid deadlock that occurs when a thread, that might already own other locks, waits for a lock owned by another thread. Our proposal differs from related works by offering a simpler approach that does not require modifications on geometric algorithms' data structures or surface information, in fact it only requires a simple initialization and the set of points –minimum and maximum points to be precise– representing the surface being updated in 2D or 3D space to guarantee mutual exclusion. Moreover, it provides better performance comparing with vertex-locking strategies, especially when the vertex degree of objects is high.

3.2 Concurrent Data Structures for Multi-Core CPUs

Most of the concurrent data structures for CPU available in the literature focus on multi-threaded programming only. The following subsections describe the main proposals with at least lock-free and wait-free properties.

In terms of data-parallel processing, there have been some proposals for using SIMD instructions to accelerate data structures and indices on CPU and GPU (discrete and integrated). The main focus is the use of SIMD instructions for traversals and search operations in different data structures and indices [58, 57, 42, 67]. However, non-blocking and linearizable data structures with SIMD processing have not been explored yet. The main restriction for implementing non-blocking data structures with SIMD processsing is the hardware support. Currently, most of the modern CPUs and GPUs support SIMD instructions, however not all of them support atomic SIMD instructions. There have been some proposals for adding multi-word CAS (CASN) support to CPUs [24, 63, 12], but their performance decreases considerably with high values of N.

3.2.1 Stacks

A concurrent stack provides concurrent push and pop operations with LIFO semantics. Several concurrent stacks have been proposed for shared memory systems. Lock-free implementations commonly support any number of concurrent operations, and some of them internally implement exponential backoff to help deal with high load and so improve scalability. This technique tackles the typical performance problem when implementing a concurrent stack, which is high contention on the head pointer.

R. Treiber proposed the first lock-free concurrent stack implementation in 1986 [64]. Treiber's stack consists of a singly-linked list with a top pointer that is modified atomically by CAS operations. Later proposals, such as [49], optimized Traiber's stack by using non-blocking techniques based on Herlihy's work [30].

Hendler et al. introduced a more scalable lock-free stack in [29]. It is based on the elimination technique proposed in [61]. This techniques allows pairs of operations with reverse semanticslike pushes and pops on a stackto complete without any central coordination, and therefore substantially aids scalability. The idea is that if a pop operation can find a concurrent push operation to "partner" with, then the pop operation can take the push operation's value, and both operations can return immediately. The net effect of each pair is the same as if the push operation was followed immediately by the pop operation, in other words, they eliminate each other's effect without the top of the stack involved. Elimination can be achieved by adding a collision array from which each operation chooses a location at random, and then attempts to coordinate with another operation that concurrently chose the same location. The number of eliminations grows with concurrency, resulting in a high degree of parallelism [51, 61].

Another technique to achieve scalable concurrent stack implementation is flat combining, introduced by Hendler et al. in [27]. It is a very general approach that has been successfully extended to complex lock-free data structures and allows to combine different algorithms; e.g. the elimination back-off. With this technique, if N threads access the stack at the same time, just one of them will acquire a lock, and the rest will wait for its release. But instead of passively waiting, the waiting threads could announce their operations. Thus, the winner-thread (lock owner) could perform tasks from the other threads, in addition to its own job.

3.2.2 Queues

A concurrent stack provides concurrent queue and dequeue operations with FIFO semantics. Several concurrent queues have been proposed for shared memory systems. Lock-free implementations commonly support any number of concurrent operations, and some of them internally implement exponential backoff, *helping techniques* and flat combining.

Herlihy et al. introduced in [36] an array-based lock-free queue that considers unbounded array sizes. A linked-list-based proposal was introduced in [28]. Michael and Scott present in [49] a linearizable CAS-based lock-free queue with parallel access to both the head and tail. The algorithm uses helping technique to ensure that the tail pointer (enqueue operations) is always behind the end of the list by at most one element. In [44], the authors presented an optimistic approach to lock-free queues, the key idea behind their algorithm is a way of replacing the singly-linked list of Michael and Scott's queue, whose pointers are inserted using a costly compare-and-swap (CAS) operation, by an "optimistic" doubly-linked list whose pointers are updated using a simple store, yet can be "fixed" if a bad ordering of events causes them to be inconsistent.

Hoffman et al. [37] introduced a new approach called Basket, where instead of the traditional ordered list of nodes, the queue consists of an ordered list of groups of nodes (logical baskets). This produces a new form of parallelism among enqueue operations that creates baskets of mixed-order items instead of the standard totally ordered list. The operations in different baskets can be executed in parallel, and this implementation results in a linearizable FIFO queue.

An interesting new approach was introduced by Afek et al. in [1] for segment queues. The general idea is that the queue maintains a linked list of segments, each segment is an array of nodes in the size of the quasi factor, and each node has a deleted boolean marker, which states if it has been dequeued. Based on the fact that most of the time threads do not add or remove segments, most of the work is done in parallel on different cells in the segments. This ensures a controlled contention depending on the segment size, which is quasi factor. However, the segmented queue is an unfair queue since it violates the strong FIFO order but no more than quasi factor. It means that the consumer dequeues any item from the current first segment.

Similar to flat combining on lock-free stacks, the authors in [27] also presented a flat-combining-based lock-free queue.

3.2.3 Lists

Linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points supports insert, delete, and search operations. The most popular lock-based approach is hand-over-hand locking [45]. In this approach, each node has an associated lock. A thread traversing the linked list acquires and releases a nodes lock to obtain exclusive access to the current node.

In terms of lock-free proposals, Valois introduced a lock-free linked list in [65]. Valoiss algorithm is correct but this solution is not practical. Harris presented in [23] a lock-free list that uses a special “deleted” bit that is accessed atomically with node pointers in order to signify that a node has been deleted. Michael [48] overcomed this disadvantage by modifying Harriss algorithm to make it compatible with memory reclamation methods [51].

Heller et al. introduced in [26] a "lazy" list-based implementation of a concurrent set object. It is based on an optimistic locking scheme for inserts and removes, eliminating the need to use the equivalent of an atomically markable reference. It also has a novel wait-free membership test operation that does not need to perform cleanup operations and is more efficient than that of all previous algorithms.

3.2.4 Maps

A hash table is a resizable array of buckets, each holding an expected constant number of elements, and thus requiring on average a constant time for insert, delete and search operations [51].

Michael in [48] proposed a lock-free hash structure that performs well in multiprogrammed environments: a fixed-sized array of hash buckets, each implemented as a lock-free list. However, there it is not lock-free extensible on the array of lists. Greenwald showed how to implement an extensible hash table using his two-handed emulation technique in [19]. However, this technique employs a DCAS (double CAS) synchronization operation, which is not available on current architectures.

Shalev and Shavit [60] introduced a lock-free extensible hash table which works on current architectures. Their key idea is to keep the items in a single lock-free linked list instead of a list per bucket. To allow operations fast access to the appropriate part of the list, the Shalev- Shavit algorithm maintains a resizable array of pointers into the list.

Feldman et al. introduced in [11] a wait-free hash map dbased on perfect hashing. Their implementation makes use of a tree-like array-of-arrays structure, with data stored in single-element leaf arrays. Their design also includes dynamic hashing, the use of sub-arrays within the hash map data structure; which, in combination with perfect hashing, means that each element has a unique final, as well as current, position.

3.2.5 Trees

Some non-blocking implementations of balanced search trees have been achieved using Dynamic Software Transactional Memory mechanisms [14, 34]. These implementations use transactions translated from sequential code that performs rebalancing work as part of regular operations.

Ellen et al. proposed the first practical lock-free algorithm for a concurrent binary search tree in [10]. Their algorithm uses an external (or leaf-oriented) search tree in which only the leaf nodes store the actual keys. Howley and Jones proposed another lock-free algorithm for a concurrent binary search tree in [38]. Their algorithm uses an internal search tree in which both the leaf nodes as well as the internal nodes store the actual keys.

Natarajan and Mittal proposed a lock-free algorithm for an external binary search tree [53], which uses several ideas to reduce the contention among modify (insert and delete) operations. As a result, modify operations in their algorithm have a smaller contention window, allocate fewer objects and execute fewer atomic instructions than their counterparts in other lock-free algorithms.

Ramachandran et al. introduced in [56] a new lock-free binary search tree. Their design is based on the use internal representation of a search tree and is based on marking edges instead of nodes. It also combines ideas from two existing lock-free algorithms, namely those by Howley and Jones [38] and Natarajan and Mittal [53], and is especially optimized for the conflict-free scenario.

3.2.6 Skiplists

Skiplists, and concurrent data structures in general, have been mainly studied for CPU architectures. Pugh introduced the skiplist data structure in [54], and then in [55] he described highly concurrent implementation of skiplists using locks. Fraser in [14] discussed various practical non-blocking programming abstractions, including the skiplist. Herlihy et al. [32, 31] presented blocking skiplist implementations that are based on optimistic synchronization. Herlihy et al. [33] also provided a simple and effective approach to design non-blocking implementation of skiplists. Crain et al. proposed in [8] a non-blocking skiplist that alleviates contention by localizing synchronization at the least contended part of the structure. Guerraoui and Trigonakis introduced OPTIK in [21], a practical design pattern for designing and implementing fast and scalable concurrent data structures. They implemented a concurrent skiplist and modified Herlihy's using OPTIK patterns.

3.3 Concurrent Data Structures for GPUs

Even though concurrent data structures have been relatively new on discrete GPUs, there exist proposals of concurrent queues, linked lists, hash tables and trees [66, 59, 42, 41, 2] . With respect to skiplists, there exist two concurrent skiplist proposals for discrete GPU. Misra and Chaudhuri [50] implemented several well-known lock-free data structure algorithms in CUDA, including a lock-free skiplist. Their experimental results show important speedup when comparing to CPU lock-free implementations. Moscovici et al. proposed in [52] a fine-grained lock-based skiplist optimized for GPU that outperforms Misra and Chaudhuri's skiplist. Their implementation is based on chunked nodes and warp-cooperative functions that take advantage of the CUDA programming model.

Chapter 4

Exploiting Concurrent Data Structures

Data structures can be classified according to their organization in memory and how traversal operations can be performed. Figure 4.1 shows a classification of data structures according their orgnization in memory.

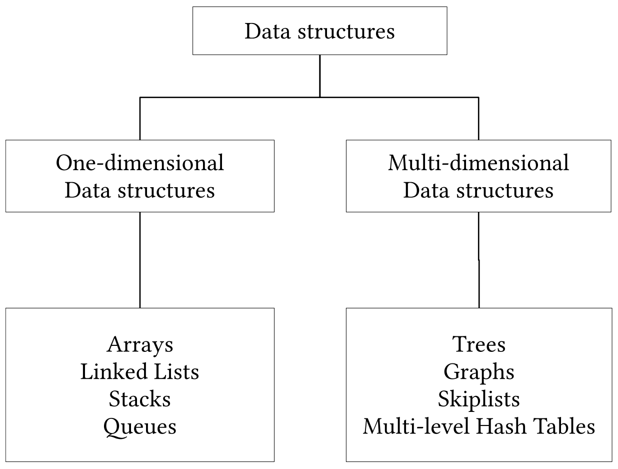


Figure 4.1: Classification of data structures

Elements of one-dimensional data structures are stored in a linear or sequential order and there is only two ways of traversal: forward and backward. On the other side, elements of multi-dimensional data structures are stored in a multi-dimensional organization, which allows traversals in any manner based on the links between elements.

A common way to exploit concurrent data structures is by having several threads accessing a shared data structure concurrently. Figure 4.2 illustrates how multi-thread parallelism works on shared concurrent data structures. Notice that different kinds of operations are performed on different memory addresses in parallel. The goal becomes challenging when having operations from different threads being performed on the same memory address, which can produce data race conditions and unexpected results.

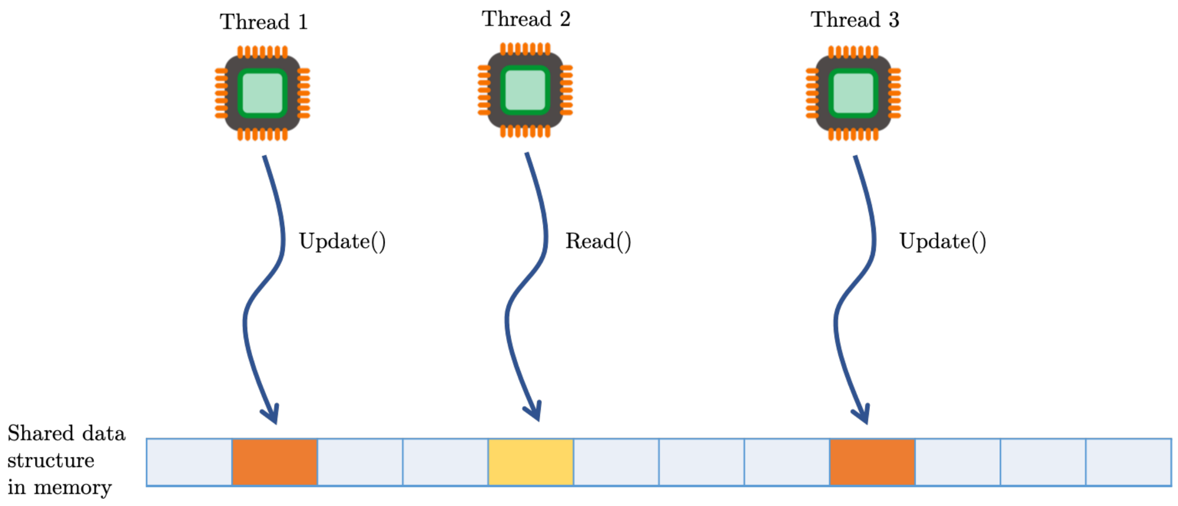


Figure 4.2: Multi-thread parallelism

There exist two ways of implementing data structures: static implementation using arrays and dynamic implementation using pointers. Parallelism on one-dimensional and multi-dimensional data structures can be achieved by using blocking and non-blocking synchronization techniques to make them thread-safe (without data race) in multi-threaded programs.

Even when non-blocking implementations can be designed for one-dimensional data structures, their performance suffers from high data contention when a large number of threads perform operations on similar memory locations. This is due to the fact that there is only one dimension of elements and the parallelism is restricted to the linear organization of elements.

4.1 Synchronization techniques

Parallelism can be exploited on both one-dimensional and multi-dimensional structures. There are five well-known techniques that allow multiple threads to access/update a single object at the same time [35].

- **Coarse-grain synchronization**: A spin lock is used to provide mutual exclusion in every access and update procedure. Each method call acquires and releases the lock. Figure 4.3 shows an example of a locked-based execution on a shared data structure over time. There are two threads performing concurrent operations on a shared queue. As both threads can eventually update the same memory address – inserting a new element at the end of the queue – there may be data race condition during their execution. For this reason, both threads require a lock first, and then once it is acquired their insertion can be safely performed. At the end, all the operations performed within critical sections are serialized.

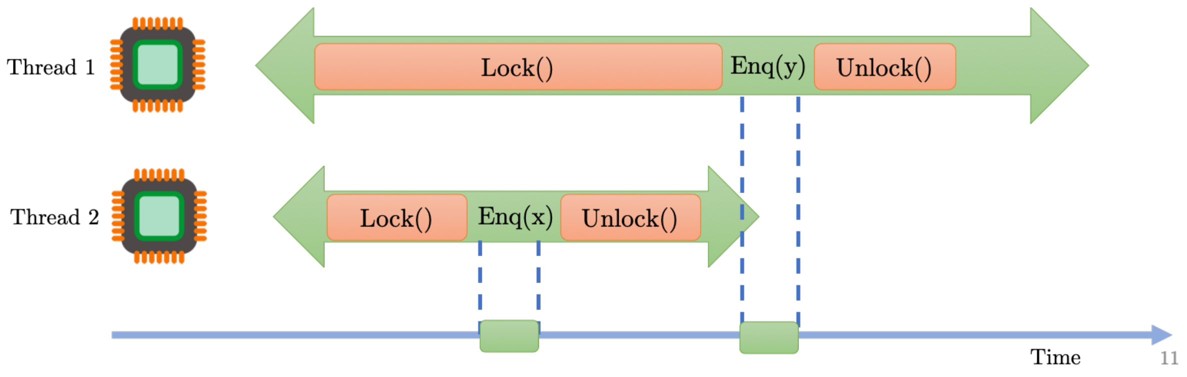


Figure 4.3: Lock-based synchronization

- **Fine-grained synchronization**: Instead of using single lock to synchronize every access to an object, the object is split into independently synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time.

- **Optimistic synchronization**: A search without acquiring any locks is performed first. Then, if the method finds the sought-after component, it locks it and checks that the component has not changed in the interval between when it was inspected and when it was locked. This technique is worthwhile only if it succeeds more often than not.

- **Lazy synchronization**: Consist of postponing the hard work. For example, a remove operation will first remove logically a component by setting a tag bit, and later, the component can be physically removed by unlinking it from the rest of the data structure.

- **Non-blocking synchronization**: Instead of using locks, built-in atomic operations such as *compareAndSwap()* are used directly in the update functions. By avoiding locks, non-blocking synchronization does not exhibit problems from lock-based synchronization such as deadlocks, blocking and priority inversion. Non-blocking shared data objects also have a higher degree of fault-tolerance than lock-based ones since they can tolerate any number of processes experiencing stop-failures [18].

  Figure 4.4 depicts a non-blocking execution of the same example from Figure 4.3. In this scenario, two threads perform the insertion of new elements in the queue without the use of locks. The insertions are perform in atomic steps during the method calls *enq(x)* and *enq(y)*.

  To verify the correctness of non-blocking executions, the linearizability property is used [35]. In a linearizable execution each method call should appear to "take effect" instantaneously at some moment between its invocation and response. So the approach

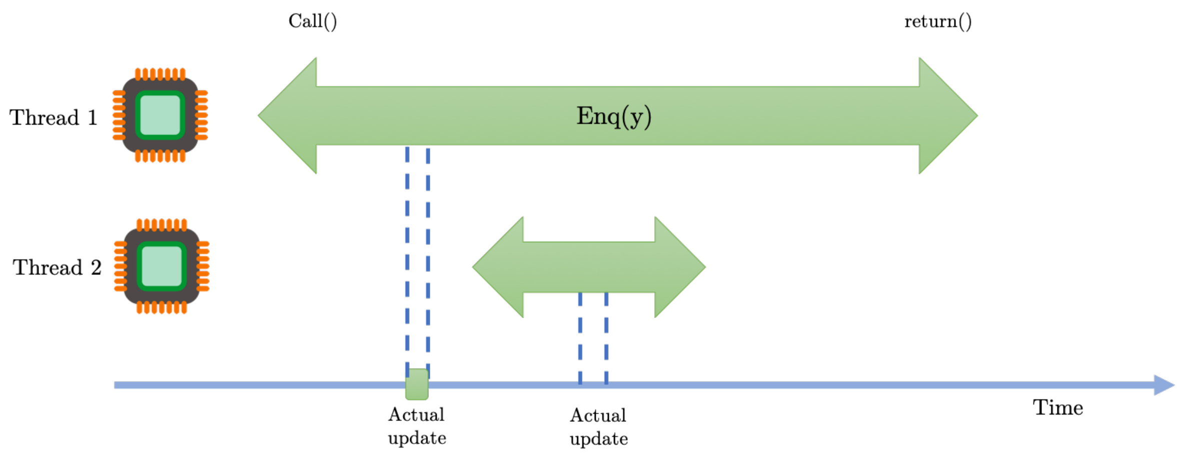


Figure 4.4: Non-blocking synchronization

to verify linearizable executions of concurrent threads on concurrent data structures is to identify the atomic step where an operation takes effect (linearization points) to show that an implementation is linearizable.

- **Lock-free**: A lock-free algorithm guarantees that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. However, as there is no fairness guarantee, some operation could be starved and take unbounded time to finish.
- **Wait-free**: A wait-free algorithm is both lock-free and fair, it guarantees that every operation finishes in a bounded number of its own steps, regardless of the actions of other operations. This is a very strong property, as it decouples the processes using the same shared data object from each other.

4.2 Concurrent Data Structures with SIMD Processing

To correctly implement concurrent data structures on SIMD architectures, support of atomic operations is required. The goal of implementing concurrent or non-blocking data structures is two-fold: to achieve higher thread concurrency by using non-blocking techniques and to achieve data parallelism at instruction level through SIMD processing.

Figure 4.5 shows the execution of concurrent operations from different threads on a shared data structures. Recall that in Figure 4.2 parallelism is achieved by having parallel threads performing operations on single data items of the shared data structures. For this new scenario, parallel threads are now able to perform concurrent operations on several data items at a time (in one instruction). Thus, a new dimension of parallelism is achieved: data-parallel processing.

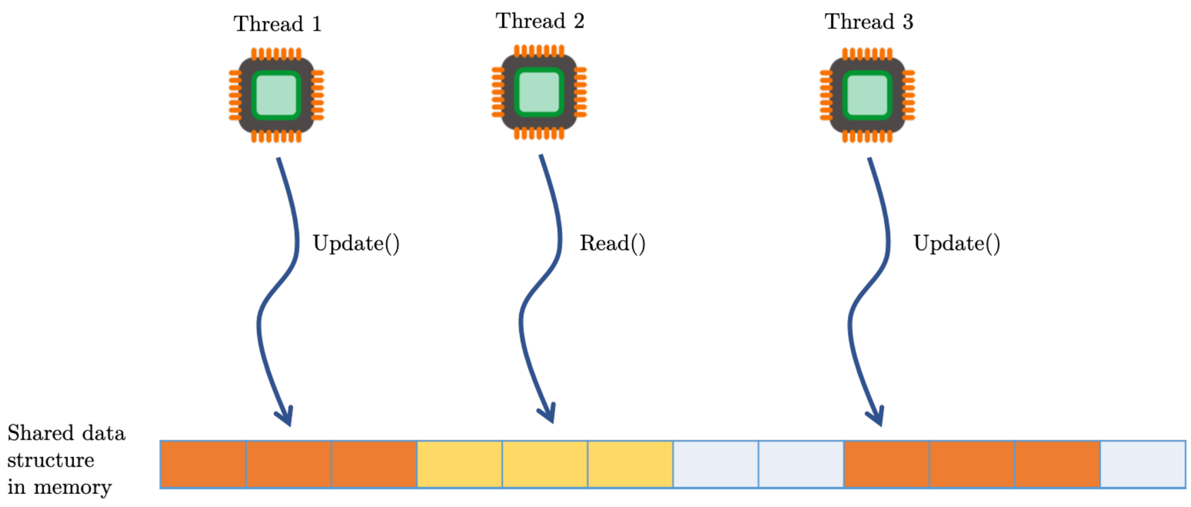


Figure 4.5: Multi-thread + SIMD parallelism

Similarly to CPU programming, concurrent data structures on SIMD architectures can be classified as blocking and non-blocking data structures. Additionaly, it is shown in this paper that non-blocking data structures such as lock-free and wait-free can be implemented

Table 4.1: Classification of concurrent data structures based on their implementation

Operation	Implemented with	Concurrent behavior per thread	Suitable concurrent data structures
Search	SISD	Nodes or data items are traversed and processed one a time.	Linked lists, queues, stacks
	SIMD	Several nodes or data items are traversed and processed within an instruction. Improvement in performance is potentially high.	k-ary trees, skiplists, multi-level hash tables, indices and succinct structures.
Update	SISD CAS	Nodes or data items are updated one at a time.	Linked lists, queues, stacks
	SIMD CAS	Several nodes or data items are updated atomically if they are contiguous memory addresses.	k-ary trees, skiplists, multi-level hash tables,

on SIMD architectures.

To efficiently exploit the SIMD capabilities within concurrent operations the organization of internal data items and nodes is the most important design aspect. Achieving correctness, non-blocking properties, and data parallelism requires a new abstraction when implementing SIMD-friendly concurrent data structures from scratch or transforming existing ones.

Table 4.1 shows a classification of concurrent data structures based on their operation implementations using SISD (Single Instruction Single Data – classical CPU instructions) and SIMD. Atomic operations and specifically the compare-and-swap operation (CAS) are fundamental for implementing update operations in non-blocking data structures. SISD CAS is supported in most multi-core CPU and GPU architectures. Differently, SIMD CAS is not commonly supported on CPU architectures, nonetheless, modern iGPU on heterogeneous CPU/iGPU architectures do support it [40].

A search operation, that traverse data structures, is typically the most common operation in applications such as databases, search engines, file systems, etc. So it is important to implement them efficiently. SIMD traversals are search operations where comparisons are performed on several data items within a single instruction [42]. For example, chunk-based nodes of a k-ary tree can contain several keys that serve as decision to continue the search to their children nodes and whose search is performed using data-parallel comparisons with SIMD instructions. Besides chunk-based trees, multi-dimensional data structures such as skiplists, multi-level hash tables, and index structures are good candidates to be implemented in such way and thus, take advantage of the SIMD processing. On the other hand, one-dimensional data structures such as linked lists, queues, and stacks are not well-suited for SIMD processing due to their nature of operations, i.e. in some cases data items are obtained directly without search, or traversals are sequential.

4.2.1 SIMD-node transformations

In order to implement non-blocking data structures based on SIMD processing, a new abstraction is defined. This abstraction will serve as guideline for designing new SIMD-friendly non-blocking data structures or transforming existing SISD non-blocking data structures to SIMD-friendly ones.

Definition 4.1. *Let Θ_X denote a super node encapsulating single nodes or data items of a data structure that can be processed by a SIMD instruction of size X (SIMD width). Θ_X can be updated atomically by SIMD CAS operations in concurrent executions and it is restricted to have the X data items contiguously in memory.*

Definition 4.2. *Let S be a one- or multi-dimensional data structure, then a Θ-node transformation S^Θ can be attained such that*

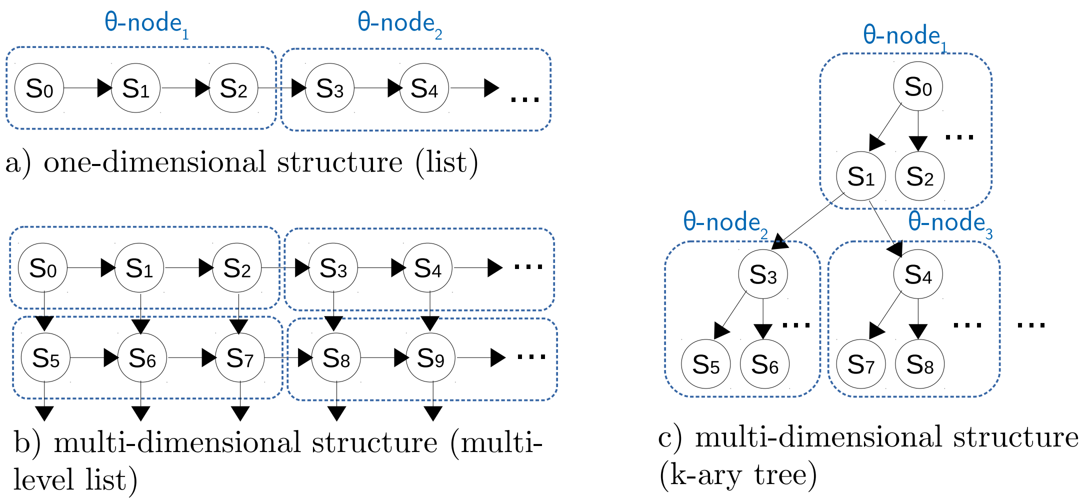


Figure 4.6: Θ-node transformation from a one-dimensional (a) and multi-dimensional (b and c) structures

- *Every node in S is in some Θ_X of S^Θ.*

$$\forall r \in S, r \in S^\Theta \wedge |S| = \sum_{i=0}^{n} |\Theta_X^i|$$

  where $|S|$ is the number of single nodes or data items in S.

- *S^Θ preserves the same S's traversal structure and thus the complexity of all its operations.*

- *S^Θ can be traversed using the orginal traversal algorithms of S (single node-based traversal), and new traversal algorithms using SIMD instructions (Θ_X-based traversal).*

Figure 4.6 depicts Θ-node transformations from three different data structures. Sub-figure 4.6a shows the layout of a simple linked list (one-dimensional structure) whose nodes have been encapsulated by two Θ_X. Sub-figures 4.6b and 4.6c illustrate the Θ-node transformations of two multi-dimensional structures, a multi-level list and a k-ary tree. Notice that these examples illustrate pointer-based data structures, but Θ-node transformations can also be applied on to other kinds of data structure, e.g. array structures, where several data items can be contained in X.

The manner that Θ-node transformation is applied depends mainly on the organization of the data structure. In hierarchical data structures such as k-ary trees, Θ-node transformations are to be performed in *parent-children* form, i.e. Θ_X encapsulates a parent and all its children. Therefore, the maximum size of X, which is defined by hardware, is the most important factor when performing Θ-node transformations; it defines whether the transformation in a hierarchical data structure is possible or not. In one-dimensional structures such as lists, queues, etc., the only restriction is to encapsulate more than one node or data item in Θ_X.

Theorem 4.1. *A Θ-node transformation, applied on a one- or multi-dimensional data structure S, produces S^Θ with the same traversal structure and complexity.*

Proof. Assume that the produced S^Θ has different traversal structure and complexity than the original S. However, for this to happen, new pointers must have been added/removed between nodes or data items during the Θ-node transformation. Then, by Definition 4.2, a Θ-node transformation on one- or multi-dimensional structures only encapsulates contiguous nodes or data items and no new pointers are added and no existing pointers are modified outside each Θ_X. □

The concept *linearizability*, introduced in [36], is used to prove the correctness of a Θ-node transformation from a concurrent data structure. Linearizability is a global property that ensures that when two processes each execute a series of method calls on a shared object there is a sequential ordering of these method calls that do not necessarily preserve the program order (the order in which the programmer wrote them down), but each method call does seem to happen instantly (i.e., invocation and response follow each other directly), whilst maintaining the result of each method call individually and consequently the object its state [35]. This definition of linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and their response.
- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

Conjecture 4.1. *A Θ-node data structure S^{Θ} can be linearizable.*

Observation. As any Θ_X can be updated atomically by Definition 4.1, S^{Θ} can be proved linearizable when all its functions hold the linearization properties in any Θ_X of S^{Θ}.

If we consider a data structure traversal as a graph traversal, after applying Θ-node transformations we obtain a transformed graph with the same traversal by definition. In this sense, the data structure traversal has not changed. However, the transformations refer to the change of internal processing within nodes. So this change is reflected in the algorithm to traverse the data structure by Θ_X nodes instead of single nodes.

4.2.2 SIMD-node Traversals

Traversing a Θ-node-based concurrent data structure involves performing comparison operations on Θ_X with single SIMD operations. Algorithm 1 illustrate a simple traversal for a general Θ-node transformed tree where there are only two ways to finish: the key is found or not (we reached the end of the structure).

Notice that all the operations on Θ_X are performed using SIMD instructions. So not only assignments and comparisons are SIMD (lines 2, 4, 7 and 10), but also the read operation (line 11) is performed by reading the entire SIMD block from memory. This tree traversal and its operations can be easily generalized to any Θ-node transformed structure.

Algorithm 1 Traversal of a Θ_X transformed tree

```
1: procedure treeTraversal(root, key)
2:     $\Theta_X \leftarrow root$
3:     while true do
4:         if $\Theta_X == null$ then
5:             break                                        ▷ Key not found
6:         end if
7:         if $\Theta_X == key$ then
8:             break                                        ▷ Key found
9:         end if
10:        $next \leftarrow \Theta_X.children\ within\ key\ range$
11:        $\Theta_X \leftarrow read(next)$                  ▷ Visit next Θ-node
12:    end while
13: end procedure
```

4.2.3 Restrictions and Considerations

The main restriction for implementing non-blocking data structures with SIMD processsing is the hardware support. Currently most of the modern CPUs and GPUs support SIMD instructions, however not all of them support atomic SIMD instructions. There have been some proposals for adding multi-word CAS (CASN) support to CPUs [24, 63, 12], but their performance decreases considerably with high values of N.

The proposed framework also considers the availability of a memory allocator that is capable of allocating contiguous memory addresses for each SIMD node. Recall that SIMD-node transformations are based on data item encapsulation within contiguous memory addresses.

In terms of performance, defining a proper width of each SIMD node could impact the performance significantly. SIMD instructions might have many performance implications due to their width and interpretations from the compilers: under-utilization of the SIMD block, incorrect SIMD width defined by user, unmatched cache line size, and so on.

Chapter 5

Contributed Multi-threaded Synchronization Mechanisms

5.1 Spatial Locks: A Specilized Synchronization mechanism for Geometric Algorithms

The parallelization of algorithms that update models in 2D or 3D space has been commonly solved by dividing the model into sub-models, processing each sub-model with different threads in parallel to finally merge the individual results, taking special care of the stitches. Even though these algorithms are known to be embarrassingly parallel and show good performance, they bogdown when the update operations must be performed in certain order or with priorities. For example, in the simplification algorithm for triangulated meshes, parallelizing the simplification by decomposing the mesh into submeshes and run the algorithm sequentially on each part can lead to bad quality results [20].

Multi-threaded algorithms that perform a mixed set of operations in 3D space are also appealing. A well-known example is the 3D Delaunay triangulation, which is typically implemented using shared containers for vertices and cells. Building a Delaunay triangulation requires threads to perform efficient alternating addition and removal of new and old cells, and addition of new vertices, updating the shared containers. These operations come out as the algorithm runs based on the current properties of the mesh.

We introduce a new synchronization mechanism called Spatial Locks that allows thread synchronization on shared-memory and multi-core architectures [16]. This synchronization becomes very useful when the algorithm to be parallelized performs updates over objects in 2D or 3D space by following a certain order of processing. Even though the example problem in this paper is the synchronization of concurrent updates done by geometric algorithms on shared objects in 2D and 3D spaces, it can also be applied to other kinds of algorithms that might need this type of synchronization.

The basic idea behind the proposed Spatial Locks mechanism is to protect 2D or 3D regions by dividing them into space cells and using lock-protection for each acquired cell.

The novelty in Spatial Locks stems from the merging of two well-known concepts, namely Spatial Hashing and Axis-aligned Bounding Box (AABB). Spatial hashing is the process by which a 3D or 2D domain space is projected into a 1D hash table [25]. The hash function takes any given 2D or 3D positional data and returns a unique grid cell that corresponds to a 1D bucket in the hash table. The hash function for hashing 2D to 1D can be as simple as $hash = x * conversion\_factor + y * conversion\_factor * width$, where $conversion\_factor$ is computed by $1/cell\_size$ and $width$ is the number of uniformly-sized cells per axis. The cell size is defined by the user and will depend on the algorithm domain. It is called spatial hash because the cell index of a data element can be obtained in constant time by its coordinates (e.g., x, y and z) with the hash function. Evidently, a 2D or 3D matrix can be used instead of the hash table and using a simplified hash function will be enough to get the corresponding cell: $grid[x * conversion\_factor,\ y * conversion\_factor]$. The programmer has the option to manage this addresses himself or leave this work to the compiler by using matrices. Figure 1 shows the elementary use of spatial hashing, where geometric points are hashed into cells placed in a 3D grid.

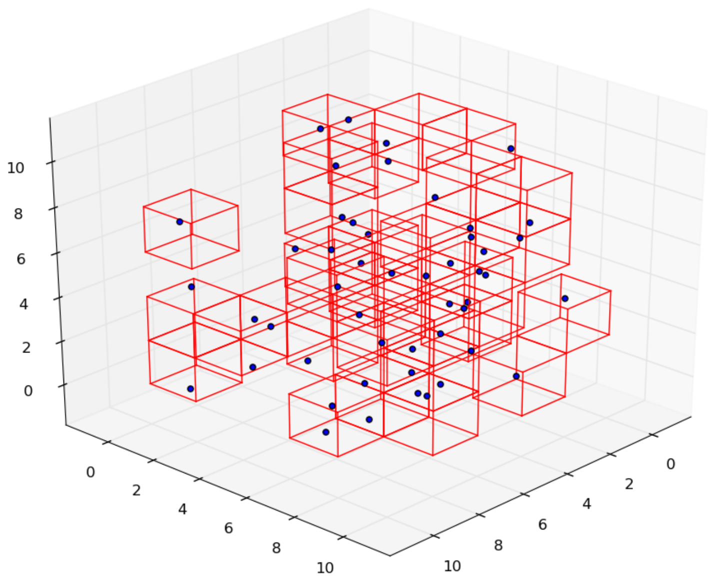


Figure 5.1: Spatial hashing where objects are mapped into uniformly-sized cells

An AABB is a rectanguloid whose faces are aligned with the coordinate axes of its parent coordinate system. An AABB can be represented or constructed with just minimum and maximum extents along each axis. For our purpose, AABBs will represent the building boxes of geometric shapes, so Spatial Locks can be seen as AABBs placed in a 2D or 3D grid indicating that threads are performing concurrent updates in them. Thus, the logic behind Spatial Locks lies on protecting threads' updates on sections of shapes or objects by placing AABBs into the spatial hash table.

5.1.1 Paradigm and Interface

Many applications exist with geometric algorithms that update objects in 2D or 3D space. The objective of using Spatial Locks is to allow safe parallelization of these algorithms by guaranteeing mutual exclusion. This synchronization mechanism works by protecting objects being updated by one thread from other thread's updates over the same object. It maintains an internal spatial hash table that reflects the status of concurrent updates by threads, identifying in which cells threads are performing concurrent updates. Synchronization can be achieved from the user perspective by means of the Spatial Locks' functions $lock(object)$ and $unlock(object)$, where $object$ corresponds to either a set of points defining a section of a shape or an AABB of that section. The $lock(object)$ and $unlock(object)$ functions work similarly to regular mutexes, i.e. the first function is blocking while the latter is not.

Synchronizing thread's updates does not only mean locking subsections of the grid for exclusive access, yet providing enough functionality to give different possibilities for thread synchronization and achieve the best performance and parallelism. Hence, this primitive also provides $check(object)$ and $tryToLock(object)$ functions which are not blocking. As its name describes, $check(object)$ returns a boolean value indicating whether the cell to which $object$ belongs is occupied or not. On the other hand, $tryToLock(object)$ tries to acquire the corresponding cell for $object$ but only considering one attempt. A boolean value is returned indicating whether the cell was acquired or not.

Imagine that an algorithm needs to perform updates to several parts of a 3D shape, and these updates can be performed by any thread in any part of the shape, so dividing the shape and assigning each part to each thread is not the best option –updates can be heavily located on a certain section of the shape, overloading a single thread and as a result serializing the entire execution–. Threads can perform their updates safely by using $lock(object)$ and $unlock(object)$ as regular locks. However, better performance can be achieved by implementing a less blocking technique if the updates do not require a specific order of processing. For example, $tryToLock(object)$ can be used instead of $lock(object)$, and if the object's cell is already taken by another thread this update is enqueued to be tried later and a different object update can be performed instead.

5.1.2 Implementation

The implementation of Spatial Locks in this paper assumes hardware support for atomic read-modify-write operations on a single memory location. The fundamental operation is *compare and swap (CAS)*, specified as *int CAS(addr, old, new)*, which checks in one atomic operation the equality between the value on *addr* and the value on *old*. If equal, it changes the value of *addr* to *new* value and returns a flag when the change is successful.

The $lock(object)$ operation is shown in Algorithm 1. It can be seen that the spatial hash table is updated by using the *compare and swap (CAS)* atomic operation, which guarantees thread safety and allows thread synchronization. The minimum and maximum indices obtained from the AABB's minimum and maximum points serve as guidelines for obtaining the cell indices to be locked by the current thread. These cell indices are calculated by the function $getCell(point)$ that performs the hashing operations explained at the beginning of section 2. If the minimum and maximum cell indices are equal, only one cell has to be taken; else 2 or 4 can be taken in 2D space and 2, 4 or 8 in 3D space. For the latter scenario, thread synchronization and safety is achieved by using an internal mutex, otherwise deadlock-freedom cannot be guaranteed even with CAS operations.

The situation when the minimum and maximum cell indices are not equal is handled as follows: based on AABB's minimum and maximum points, new 2D/3D points are created to potentially lock additional cells. For instance, for an object in 2D space up to 4 cells can be locked when the coordinates (x, y) of its minimum and maximum AABB points are different, and their two additional cell indices are obtained from the points $(max.x,\ min.y)$ and $(min.x,\ max.y)$. A similar procedure is performed for objects in 3D space, where the additional cell indices are obtained from the points $(min.x,\ min.y,\ max.z)$, $(min.x,\ max.y,\ min.z)$, $(max.x,\ min.y,\ min.z)$, $(min.x,\ max.y,\ max.z)$, $(max.x,\ min.y,\ max.z)$, and $(max.x,\ max.y,\ min.z)$. Observe that some of these new points can fall into the same cell index, and the average number of cells to be locked depends directly on the spatial hash table's cell size.

An alternative approach to handle lock conflict when a thread attempts to take multiple cells is using priorities. So instead of using a mutex as in Algorithm 1, lock conflicts can be handled by using priority locks where each thread is given a unique priority. If the acquiring thread has a higher priority it simply waits for the lock to be released. Otherwise, it retreats, releasing all previous cells and restarting the operation. This approach also avoids deadlocks and guarantees progress.

Note that before performing the CAS operation in Algorithm 1, we first check if the cell is available (if value is 0). It might seem redundant since the following CAS operation also checks if the cell is 0, nonetheless this presents better performance than using only CAS operations due to cache coherency and the costs of the atomic operations on modern architectures [35].

The *unlock*(*object*) operation, shown in Algorithm 2, is very simple and also similar to *lock*(*object*), however the internal mutex is not needed. Once the cell indices are obtained from the AABB, these are immediately released by using *CAS* operations. It is guaranteed that the thread that acquires a specific cell, is the same who releases it. It is also guaranteed that the status of every cell that will be released by a thread is *locked* (value 1 of *table* in Algorithm 1 and 2); so no additional verification is needed by the time of releasing cells within this operation.

Lemma 5.1. *Spatial Locks satisfy mutual exclusion when objects fall into the same unique cell.*

Proof. By contradiction. Suppose that 2 threads, A and B, are going to update the objects X and Y respectively, which fall into the same cell S in 3D space. It means that both X's and Y's minIndex and maxIndex are equal. These 2 threads first call *lock*(*object*), so the

Algorithm 2 Lock object in Spatial Hash Table

```
 1: procedure LOCK(aabb)                                  ▷ object represented as an AABB
 2:     minIndex ← getCell(aabb.minPoint)
 3:     maxIndex ← getCell(aabb.maxPoint)
 4:     if minIndex == maxIndex then
 5:         while true do
 6:             if spatialHashTable[minIndex] == 0 then
 7:                 if CAS(spatialHashTable[minIndex], 0, 1) then
 8:                     break
 9:                 end if
10:             end if
11:         end while
12:     end if
13:     mutex.lock()
14:     while true do
15:         if table[minIndex] == 0 then
16:             if CAS(spatialHashTable[minIndex], 0, 1) then
17:                 break
18:             end if
19:         end if
20:     end while
21:     while true do
22:         if table[maxIndex] == 0 then
23:             if CAS(spatialHashTable[maxIndex], 0, 1) then
24:                 break
25:             end if
26:         end if
27:     end while
            ▷ Continue with other indices. AABB can be part of 2, 4 cells in 2D and 2, 4 or 8
        in 3D
28:     mutex.unlock()
29: end procedure
```

order of events from Algorithm 1 is:

$read_A(cell_S = false) \rightarrow write_A(cell_S = true) \rightarrow UpdateObject_A(X)$

$read_B(cell_S = false) \rightarrow write_B(cell_S = true) \rightarrow UpdateObject_B(Y)$

Without loss of generality, assume that A was the last thread to update the object X. Since thread A still entered its critical section, it must be true that thread A reads $cell_S == false$. Thus it follows that $read_B(cell_S = false) \rightarrow write_B(cell_S = true) \rightarrow UpdateObject_B(Y)\ read_A(cell_S = false) \rightarrow write_A(cell_S = true) \rightarrow UpdateObject_A(X)$

Algorithm 3 Unlock object in Spatial Hash Table

```
1: procedure UNLOCK(aabb)                    ▷ object represented as an AABB
2:     minIndex ← getCell(aabb.minPoint)
3:     maxIndex ← getCell(aabb.maxPoint)
4:     if minIndex == maxIndex then
5:         spatialHashTable[minIndex].store(0);
6:     else
7:         spatialHashTable[minIndex].store(0);
8:         spatialHashTable[maxIndex].store(0);
           ▷ Continue with other indices. AABB can be part of 2, 4 cells in 2D and 2, 4 or 8 in 3D
9:     end if
10: end procedure
```

This is impossible because there is no other write $false$ to $cell_S$ between $write_B(cell_S = true)$ and $read_A(cell_S = false)$. Contradiction. □

Lemma 5.2. *Spatial Locks satisfy mutual exclusion when objects fall into multiple cells, having or not common cells between them.*

Proof. Suppose that 2 threads, A and B, are going to update the objects X and Y respectively in 3D space. Object X falls into the cells S_1 and S_2, and object Y falls into S_2 and S_3. These 2 threads first call $lock(object)$, so the order of events from Algorithm 1 is:

$mutex.lock() \rightarrow read_A(cell_{S_1} = false) \rightarrow write_A(cell_{S_1} = true) \rightarrow read_A(cell_{S_2} = false) \rightarrow write_A(cell_{S_2} = true) \rightarrow mutex.unlock() \rightarrow UpdateObject_A(X)$

$mutex.lock() \rightarrow read_B(cell_{S_2} = false) \rightarrow write_B(cell_{S_2} = true) \rightarrow read_B(cell_{S_3} = false) \rightarrow write_B(cell_{S_3} = true) \rightarrow mutex.unlock() \rightarrow UpdateObject_B(Y)$

By simply holding the mutual exclusion property of the shared $mutex$, both sequences of events A and B are sequentialized. □

Theorem 5.1. *Spatial Locks (Algorithm 1 and 2) satisfy mutual exclusion.*

Proof. Follows from Lemma 1 and 2. □

Theorem 5.2. *Spatial Locks (Algorithm 1 and 2) satisfy deadlock-freedom*

Proof. By contradiction. Without loss of generality, suppose thread A waits forever in the $lock(object)$ function. Particularly, it runs $while()$ forever waiting $cell_S$ becomes $false$. If thread B also got stuck in its $while()$ loop, then it must have read $true$ from $cell_S$. But since $cell_S$ cannot be $true$ from the beginning, the hypothesis that thread B also got stuck in its $lockObject()$ method is impossible. Thus, thread B must be able to enter its critical section. Then after thread B finishes its critical section and calls $unlock()$ method, $cell_S$ becomes false, and this triggers thread A to enter its own critical section. Hence, thread A must not wait forever at its $while()$ loop. □

The $check(object)$ operation can be implemented by using *load* operations on the atomic cells from the spatial hash table. If all the cells are available, the object is reported as available to be updated. On the other hand, the $tryToLock(object)$ operation works by doing something very similar to Algorithm 1, but instead of waiting until the cell is available, it returns immediately. In the scenario of having multiples cells to be acquired, when there is one that is not available along the way it rolls back by releasing the ones already acquired.

5.1.3 Settings and restrictions

The use of Spatial Locks requires initial settings according to the set of objects to be updated and the algorithm itself. Let S be the set of objects or sub-shapes that will be updated by the algorithm. In geometric terms every object $s \in S$ corresponds to a set of 2D or 3D points. The following invariants must be held when using Spatial Locks to synchronize concurrent updates over S; where the Γ operator returns the size of an object's AABB:

- The dimension of the spatial hash table T covers all the objects in S:

$$\Gamma(T) \geq \Gamma(\bigcup_{i=1}^{n} S_i)$$

- Let c be the cell size of the spatial hash table. Then,

$$\Gamma(c) > \Gamma(s), \forall s \in S$$

Since the performance of a concurrent algorithm using Spatial Locks depends directly on the chosen value c, the spatial hash table can be re-built with a new c' any time during the execution of the algorithm. However, these invariants must preserve with the remaining objects in S and the new T'.

5.1.4 Parallel Mesh Simplification using Spatial Locks

Surface mesh simplification is the process of reducing the number of faces used in a surface mesh while keeping the overall shape, volume and boundaries preserved as much as possible [5]. It can be considered as the opposite of the subdivision or mesh refinement. Models are usually represented as triangulated meshes and their simplification algorithms can be used to automatically generate them in different resolutions, so that designers only need to model the finest level. The size of models are reduced dramatically, so they are also used in streaming and network applications. Figure 2 shows different levels of simplifications of a hand obtained with this algorithm.

Due to the considerable processing time, simplification algorithms are normally used as a preprocessing step. Considering that the average performance of computing edge-collapse simplification with quadric error metrics, one the most common methods for mesh simplification, is about 50,000 operations per second [20]. Although the parallelization of this algorithm seems to be simple, most of the edge-collapsing simplification algorithms work sequentially due mainly to the large neighborhood information required for the computation of optimally ordering of operations, where triangles are simplified following priorities and costs.

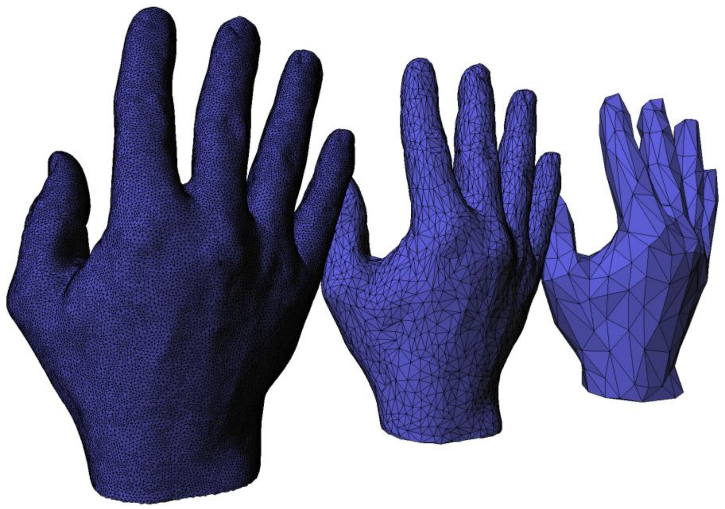


Figure 5.2: Surface mesh simplification algorithm applied to a mesh with different simplification levels [5].

For our study we chose the mesh simplification algorithm presented in [46, 17, 5]. The algorithm proceeds in two stages. In the first stage, called collection stage, an initial collapse cost is assigned to each and every edge in the surface mesh and maintained in a priority queue. In the second stage, called collapsing stage, edges are processed in order of increasing cost. Some processed edges are collapsed while some are just discarded. Collapsed edges are replaced by a vertex and the collapse cost of all the edges now incident on the replacement vertex is recalculated, affecting the order of the remaining unprocessed edges. The process ends when a desired number of triangles is reached or the collapse cost is over a specific threshold. Notice that processing edges by their collapse costs has been proven to provide better quality results than any other ordering.

Figure 3 illustrates the edge-collapse operation over the edge (v, v_u). This operation takes the edge (v, v_u) and substitutes its two vertices v and v_u with a new vertex $\underline{v}$. In this process, the triangles t_l and t_r are collapsed to edges, and are discarded. The remaining edges and triangles incident upon v and v_u, e.g., t_{n0}, t_{n1}, t_{n2}, t_{n3}, t_{n4}, t_{n5} and t_{n6} respectively, are modified such that all occurrences of v and v_u are substituted with $\underline{v}$. The computation involved on this process can be summarized as: computing the cost of collapsing the edge

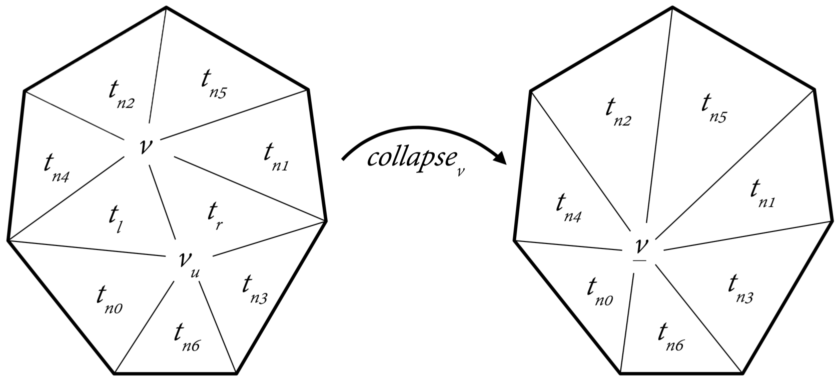


Figure 5.3: Contraction of the edge (v, v_u) into a single vertex. The shaded triangles become degenerate and are removed during the contraction.

e, choosing the position of the vertex $\underline{v}$ that replaces the edge, and updating the adjacent triangles.

Parallelizing the Mesh Simplification Algorithm implies synchronizing all the threads' updates on the shared mesh. The specific region to synchronize on every update is given by the edge to be collapsed and its adjacent triangles, as it is shown in Figure 3. Even though the external edges of adjacent triangles (t_{n0}, t_{n1}, and so on) are not modified during the edge-collapse operation, their vertices are part of several edges being updated, so they cannot be collapsed by another thread at the same time. However, these external edges can be borderline of two parallel edge-collapse operations.

The parallelization of the aforementioned algorithm is explained as follows:

- A concurrent priority queue P is used to store all the edges with their costs as priorities. If priorities are relaxed a concurrent vector can be used instead.
- Every thread takes an edge e from P and analyzes its quadratic errors.
- If an edge is set to be collapsed, the corresponding AABB given by the edge's adjacent triangles is locked in the spatial hash table.

- Once the edge is collapsed and the shaded triangles removed, the corresponding AABB is removed safely from the spatial hash table.

A simplified version of the parallel mesh simplification algorithm (*PMS*) is shown in Algorithm 3. Note that the use of *parallel for* does not mean the subdivision of range for every thread, but the parallelization for obtaining an edge from the priority queue at every iteration. If two threads are attempting to update adjacent triangles, only one will succeed locking the triangle given by its AABB and the other must wait.

The complete mesh simplification process can be done on several iterations depending on the level of simplification desired by the user. After every iteration, triangles have become bigger, so the *rebuil()* function re-builds the spatial hash table with a greater cell size, as explained in section 2.3. The new cell size is calculated similarly to its initial value at the beginning of the process: it must be greater than every remaining triangle's AABB in the mesh.

Algorithm 4 Parallel Mesh Simplification using Spatial Lock

```
 1: procedure SIMPLIFY(edges)                          ▷ Input: Set of edges
 2:     for every iteration do
 3:         parallel for every edge e do
 4:             err ← calculateError(e)
 5:             if err > threshold then
 6:                 continue
 7:             else
 8:                 aabb ← createAABB(getTriangle(e))
 9:                 spatialLocks.lock(aabb)
10:                 collapseEdge(e)
11:                 spatialLocks.unlock(aabb)
12:             end if
13:         end parallel for
14:         spatialLocks.rebuild();
15:     end for
16:     return edges                                   ▷ Reduced set of edges
17: end procedure
```

Recall that processing edge-collapse operations based on their costs as priorities is very crucial for the quality of the resulting mesh. Therefore, the property is maintained even when there could be threads that might reflect their results later than others due to the operating system scheduler. Under normal conditions, once a thread acquires a Spatial Lock for an edge-collapse operation, it is guaranteed to perform this operation in a finite number of steps.

Two other variants of the parallel mesh simplification algorithm were implemented. In the first variant, called *PMS-RP*, the cost priorities are relaxed and threads attempt to acquire Spatial Locks by using the non-blocking function $tryToLock(object)$. If the corresponding cell is taken by another thread, the next edge is taken from the queue and the same process is tried again. The second variant, called *PMS-MBB*, allows having bigger cells and multiples AABBs per cell. For this purpose an AABB tree [4] (which provides add, remove and intersection operations) with a regular mutex are used in every cell, so the protection of an edge-collapse operation is given by the presence of its AABB in the tree rather than a specific flag within the cell.

Experimental Results

A set of experiments were carried out in order to evaluate the performance of the *PMS* algorithm using Spatial Locks and to compare it to its sequential version as well as the parallel variants *PMS-RS*, *PMS-MBB* and one using internal locks in vertices. Algorithms were implemented in the C++11 programming language. The experiments were carried out on a Dual 14 Core Intel(R) Xeon(R) CPU E5-2695 v3, with a total of 28 physical cores running at 2.30GHz. Hyperthreading was disabled. The computer runs Linux 3.19.0-26-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 32KB and 256KB, respectively and a per-processor shared L3 cache of 35MB, with a 32GB DDR RAM memory and 1TB SSD. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in *time.h*.

Table 5.1: Performance of the PMS algorithm implemented with Spatial Locks

Model	# triangles	# removed triangles	1 thr (sec.)	2 thr (sec.)	4 thr (sec.)	6 thr (sec.)	8 thr (sec.)
Bunny	69,664	34,832	0.3	0.26	0.23	0.20	0.20
Head	281,724	140,862	1.4	1.3	1.1	1.0	0.9
Wall	651,923	325,961	3.3	2.8	2.5	2.2	2.0
Einstein	674,038	337,018	4.1	3.7	3.5	3.0	2.7
Motors	1,516,759	758,360	13.5	9.1	5.5	4.4	3.8
Facew	2,402,732	1,001,364	22.1	18.4	14.2	7.2	5.9
Castle	3,136,234	1,218,116	32.1	25.8	17.2	9.1	7.4

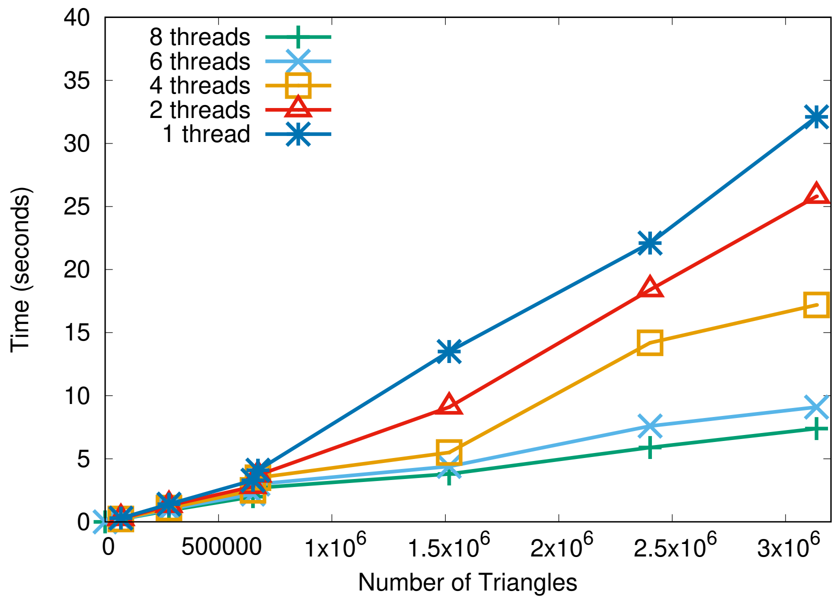


Figure 5.4: Total time spent by the parallel mesh simplification algorithm in simplifying surface meshes with up to 3,136,234 triangles.

Table 1 shows detailed information of mesh simplifications using the original algorithm (sequential with one single thread) and PMS using Spatial Locks with up to 8 threads. Meshes with more than 1 million of triangles are particularly of our interest, since their simplifications require a big amount of edge-collapse operations. These meshes, which have from 60K to 3.2M triangles, were simplified using a 0.5 simplification ratio with both algorithms. All the surface meshes were obtained from [68].

It can be seen that as the number of triangles in the surface mesh increases, the runtime for all variants also increases. However, the increment of PMS' runtime with several threads is slower than its sequential counterpart, getting much better performance with bigger meshes. It is due to the fact that with bigger meshes threads have less probabilities to interfere each other within the same spatial hash table's cell, getting parallelism benefits. Figure 4 shows the runtime trend of all the variants when the number of triangles increases. It turned out that for some experiments, specifically those with small meshes (under 100K triangles), the sequential algorithm showed better or very similar performance than our parallel implementation. This observation can be used as a key factor when deciding whether to implement a parallel geometric algorithms with Spatial Locks or avoid its use.

We also counted the number of cells that $lock(object)$ operations occupied with different configurations for the spatial hash table. As it can be anticipated, with greater cell size the number of occupied cells by one *lock* operation is smaller. i.e. with a cell size of 4 times the average of edge size, the *lock* operations that occupied only 1 cell was over 89%, and the percentage of *lock* operations using 8 cells was only 2%. The trade-off is given by adjusting the cell size to have less *lock* operations falling on 8 cells (worst case), but as the cell size is increased there is less parallelism due to the fact of having cells covering more space and threads spending more time in the locking phase. For this specific algorithm, we obtained the best performance with a cell size of 3 times the average of the edge size, where the *lock* operations occupying 1 cell were around 86% of the total, and those falling on 8 cells represented only 2% of the total.

To make the comparison fair, we implemented a parallel mesh simplification algorithm with internal locks using a similar approach to the one proposed in [3] (called fine-grained locks). To guarantee thread safety when collapsing edges, the internal locks are managed in vertices. A lock conflict occurs when a thread attempts to acquire a lock already owned by another thread. Systematically waiting for the lock to be released is not an option since a thread may already own other locks, potentially leading to a deadlock. Therefore, lock conflicts are handled by priority locks where each thread is given a unique priority (totally ordered). If the acquiring thread has a higher priority it simply waits for the lock to be released. Otherwise, it retreats, releasing all its locks and restarting an insertion operation.

Table 5.2: Performance of mesh simplification algorithms using internal locks and Spatial Locks

Model	# triangles	# removed triangles	PMS Internal Locks (sec.)	PMS Spatial Locks (sec.)
Bunny	69,664	34,832	0.3	0.2
Head	281,724	140,862	1.4	0.9
Wall	651,923	325,961	3.3	2.0
Einstein	674,038	337,018	4.1	2.7
Motors Car	1,516,759	758,360	11.5	3.8
Facew	2,402,732	1,001,364	26.7	5.9
Castle	3,136,234	1,218,116	20.1	7.4

Similarly to previous experiments, we carried out several simplifications on different models. Table 2 and Figure 5 summarizes and plots respectively the running times of the mesh simplification algorithms using internal locks and Spatial Locks. Similar to previous setup, both algorithms were tested with 8 threads. As it can be seen, the alternative with Spatial Locks performs all the simplifications in less time, being approximately 3 times faster than the alternative with internal locks. Additionally, it is not only a better alternative in terms of performance, but its use is also much less invasive since there is no need to modify internal data structures as we did with internal locks approach.

There are several other synchronization alternatives that can be created from the concept of Spatial Hashing. We also measured a couple of them explained in the previous section. Figure 5 illustrates the throughput (number of removed triangles per second) by PMS and its two variants: *PMS-RP* and *PMS-MBB*. In this scenario PMS-RP obtains the highest throughput since waits for available cells are avoided by relaxing priorities when choosing edges to collapse. Recall that this variant might lead to worse quality results for their

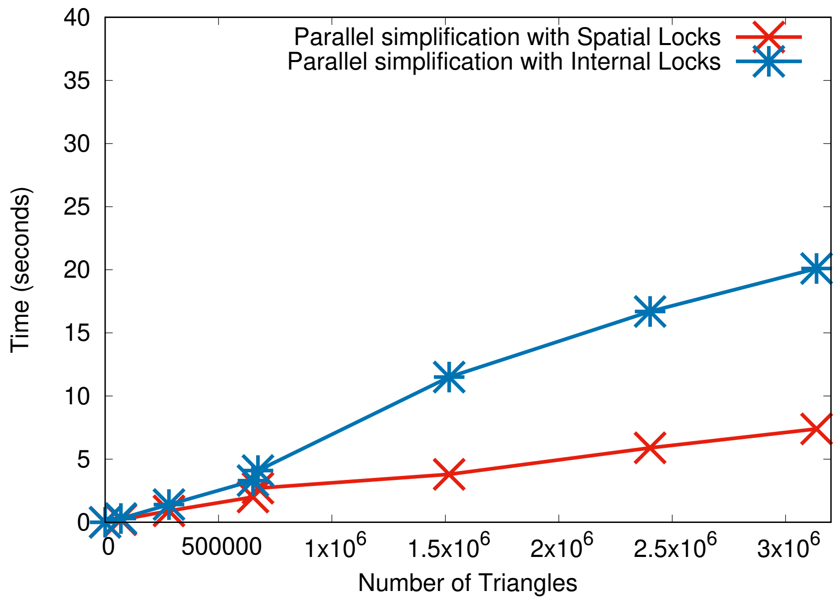


Figure 5.5: Total time spent by the mesh simplification algorithm using internal locks and Spatial Locks.

resulting meshes. On the other hand PMS-MBB, that allows more than one AABB per cell, presents lower throughput than PMS. It can be explained by the use of local mutexes as well as the complex operations that imply using the AABB-tree with insertions and removals.

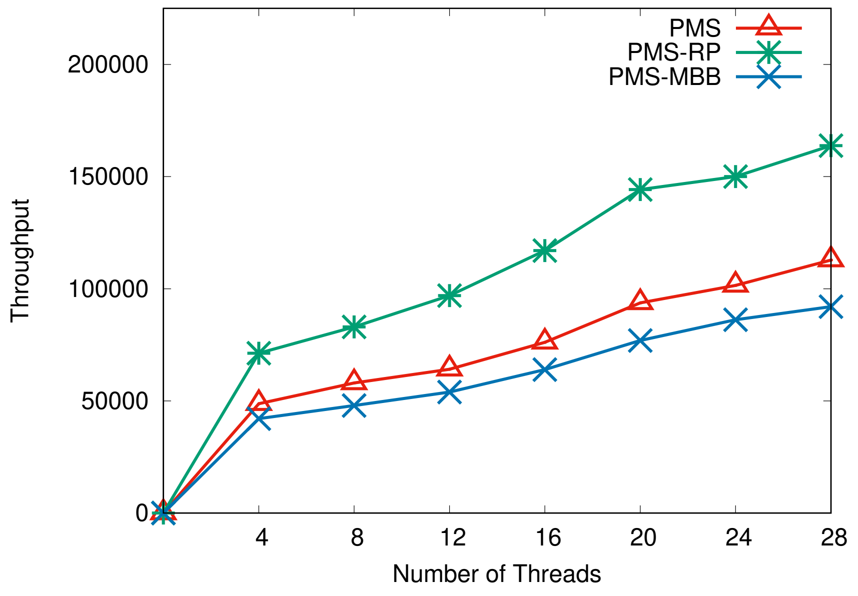


Figure 5.6: Total time spent by both algorithms in simplifying a surface mesh with 2,436,234 triangles.

Chapter 6

Contributed Concurrent Data Structures with SIMD Processing

This chapter presents two new concurrent data structures based on SIMD-node transformations: a k-ary tree and a lock-free skiplist. Additionally, a SIMT-based algorithm for data-parallel traversals on tree structures is introduced.

6.1 k-ary Tree based on SIMD-node transformations

The k-ary tree is a rooted tree in which each node has no more than k children. A binary tree is the special case where $k = 2$, and a ternary tree is another case with $k = 3$ that limits its children to three. Traversing a k-ary tree is very similar to binary tree traversal. The pre-order traversal goes to parent, left subtree and the right subtree, and for traversing post-order it goes by left subtree, right subtree, and parent node. For traversing in-order, since there are more than two children per node for $k > 2$, one must define the notion of left and right subtrees.

Consider a SISD non-blocking k-ary tree S that maintains non-duplicated keys and has $O(log_k(n))$ complexity for its operations. A Θ-based non-blocking k-ary tree S^Θ results from the Θ-node transformation where the k children and parent nodes of every subsequent level are encapsulated in a Θ_X (similar to Figure 4.6.c). S and S^Θ have the same properties and data items, and they represent the set abstraction. This can be generalized to any similar non-blocking hierarchical data structure, and more simple multi- or one-dimensional structures. Linearizability of S and S^Θ is proved as follows:

Lemma 6.1. *The successful insertion of a children node in some Θ_X of S^Θ with less than k children is linearizable with any other search/update operation.*

Proof. The precondition is that there is no node with the key to be inserted in S^Θ. The post-condition is a valid S^Θ with the new key in some children node in Θ_X. Notice that there is available space in Θ_X to insert the new key, so the linearization point is the successful SIMD CAS operation inserting the key in Θ_X. □

Lemma 6.2. *The successful insertion of a new node in some Θ_X of S^Θ with k children is linearizable with any other search/update operation.*

Proof. Similar to the previous case, the precondition is that there is no node with the key to be inserted in S^Θ. The post-condition is a valid S^Θ with the new key in some Θ_X. Observe that Θ_X has k children, so it is full. A new Θ_X is created with the new key as parent node. Here the linearization point is the successful SIMD CAS updating one of the children node's next pointer of the current Θ_X to the new Θ_X with the new key. □

Lemma 6.3. *The successful remove operation of a node in some Θ_X of S^Θ with more than one child node is linearizable with any other search/update operation.*

Proof. This operation has a precondition that there exists a Θ_X in the S^Θ with a key to be removed. The post-condition is having the same Θ_X in the tree without the key. The linearization point is the successful SIMD CAS removing the key from Θ_X. The successful SIMD CAS ensures the post-condition of Θ_X without the key along with Definition 4.6 that the tree is still valid after the removal. Additionally, whether the key to be removed is the parent or one of the children nodes, the succesful SIMD CAS updates the nodes internally when necessary. □

Lemma 6.4. *The successful remove operation of a parent node in any Θ_X of S^Θ with no children is linearizable with any other search/update operation.*

Proof. Similar to previous case, this operation has a precondition that there exists a Θ_X in S^Θ with a key to be removed. Furthermore, the key is in a parent node in Θ_X with no children, so it corresponds to a leaf of the tree. The post-condition is having S^Θ without the key and its Θ_X since it only had the parent key at the time of linearization point. The linearization point is the successful SIMD CAS removing the entire Θ_X by updating its parent's next pointer. The successful SIMD CAS ensures the post-condition of S^Θ without the key along with Definition 4.6 that S^Θ is still valid after the removal. □

Theorem 6.1. *S^Θ maintains the linearizability of S.*

Proof. This follows from lemmata 6.1, 6.2, 6.3, and 6.4. □

Notice that other transformations from similar non-blocking data structures can be directly derived from the lemmata described above. For example, non-blocking linked list, queues, skiplists, and multi-level hash tables.

There are two main benefits from adding SIMD instructions to concurrent data structures like the k-ary tree. The obvious benefit of SIMD instructions is the data parallelism available in each SIMD instruction, e.g. traversing several data items within one instruction. Second, in multi-dimensional and hierarchical structures, the use of SIMD instructions reduces the number of conditional branches needed to select the next data item from next levels (children nodes), which are difficult to foresee in the branch predictor of the processors for random data inputs; therefore, the overhead of branch misprediction can be reduced.

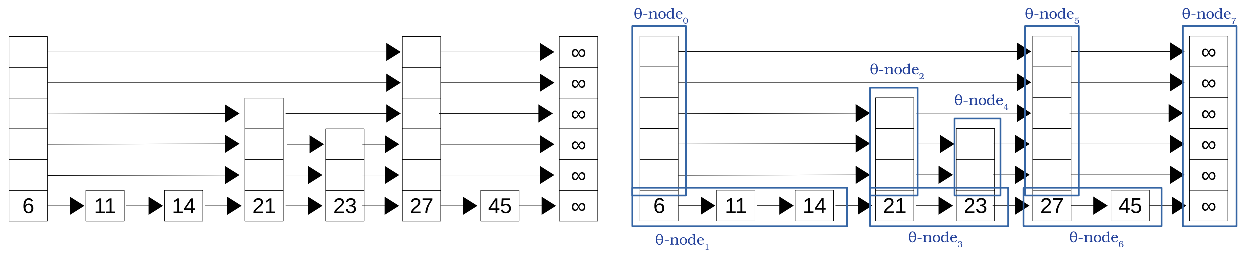
Regarding data structures operations. Not only search operations can be significantly speed up. Update operations, e.g. insertions or removals, can also benefit from SIMD instructions. Consider the case of an insertion operation that involves several updates of the neighbor nodes. If a Θ-node encapsulates correctly the data items to be updated, all the update operations can be done in a single atomic SIMD operation.

6.2 Lock-free Skiplist based on SIMD-node transformations

Skiplists are popular in concurrent algorithms, as they offer a probabilistic alternative to balanced search trees without costly balancing operations and still providing $O(logn)$ for search and update operations. We capitalize on the fundamental SIMD instructions of the Intel's GenX architecture to design a skiplist, named CMSL, based on chunked lists [15]. So instead of having nodes and individual updates when inserting or deleting keys, entire chunks are updated using SIMD atomic operations. Thus, the parallelism of this data structure is achieved at thread level, within multiple EUs, and instruction level, through SIMD within each FPU. A detailed proof of correctness using linearization points establishes that CMSL guarantees the lock-freedom property for the insert and delete operations, and wait-free property for its search operation.

6.2.1 Structure details

As mentioned above, the proposed skiplist takes advantage of the GenX architecture and its instruction set by keeping its levels and sorted keys in chunked lists instead of individual nodes. This allows to perform SIMD operations on entire chunks rather than single nodes every time there are search or update operations involved. Furthermore, the support of atomic SIMD16 operations by CM and GenX allows atomic updates of an entire chunk with one single instruction.



(a) A 6-level skiplist (b) A Θ-node 6-level skiplist

Figure 6.1: Skiplists before and after Θ-node transformation

Figure 6.1a and 6.1b illustrates the layout of the original skiplist and the Θ-node transformed skiplist.

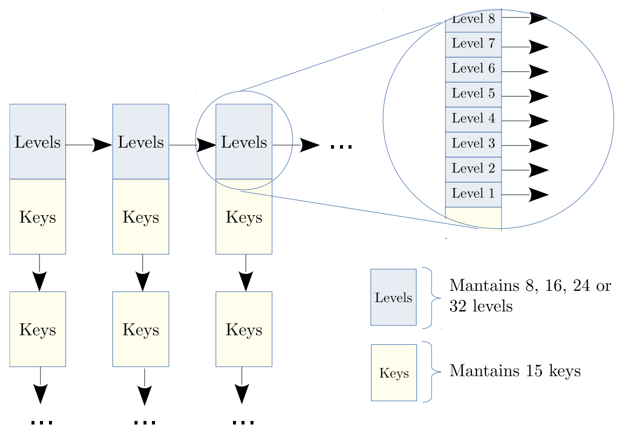


Figure 6.2: Skiplist structure

Figure 6.2 details the sizes of every Θ_X and their internal organization. the Every chunk of levels can be a 8-, 16-, 24- or 32-dword vector which stores the offset to pointed lists. On the other side, every chunk of keys is a *16-dword* vector which maintains up to 15 sorted keys, and has one *dword* reserved for a pointer to next chunk of keys. Recall that the skiplist's maximum level is directly related to the total number of keys that it can store while maintaining its logarithmic complexity. As the CMSL can keep up to 32 levels in its main list, only one chunk is enough to maintain up to 2^{32} sorted keys with logarithmic running times for search operations and $p = 1/2$.

A major consideration to improve the performance of CMSL on GenX is memory access. The number of global memory reads should be minimized and every read should use full bandwidth by reading 32 elements (*dwords*) at a time. This is the reason why every main vertical list is read as *32-dword* vector, with half of it being for levels and the other half sorted keys. If more levels or keys are required during the search, additional read operations might be necessary. An additional optimization following this idea is allocating two *16-dword* chunks of keys together. So that when searching for a key within a list, every single read operation will return two attached chunks.

One could think of a drawback of CMSL as its space complexity or unused space in chunks. While CMSL does save space by having different SIMD width for the chunks of levels, it only allocates *16-dword* chunks of keys. This comes from the logic of the key distribution set by p, where there are twice as many nodes (uniformly balanced) at level l than at its upper level $l + 1$. Even when the chunks of keys store keys for all the levels, its worst case comes from level 0, storing 50% of the total number of keys. Additionally, a *16-dword* chunk matches with the hardware support for atomic SIMD16 CAS operations. So that a single SIMD operation is preferred rather than multiple reads and comparison operations when having smaller chunks.

6.2.2 Implementation considerations

CM supports standard C++ operations and specialized operations for graphics applications aimed to exploit data parallelism. CM can be considered as a vector/matrix language oriented, so that it allows working with vector operations with ease. Operations such as reading data elements from memory into vectors, selecting a subset of elements from a vector, arithmetic and boolean operations between vectors, boolean reductions, etc. are optimized and translated into SIMD instructions by the CM compiler. For further details of the CM language, refer to its documentation at [39].

The main CM operations used by CMSL are:

a. *Read and write:* reads data blocks from memory to registers and writes the computed output from registers to memory respectively. A vector of indices (offsets) can be used for scattered reads/writes. In order to reduce the load on the message gateway as well as maximize bandwidth utilization of read operation bandwidth we attempt to utilize the full capacity of the read operations (i.e. 32 dwords for linear block reads and 16 dwords for scattered reads) whenever possible.

b. *Atomic read-modify-write:* GenX supports up to SIMD16 atomic read-modify-write operations on data block (contiguous data elements), i.e. it is guaranteed by GenX to perform the read-modify-write operation atomically on the data block. Similar to C++, it supports the commonly used operations ADD, SUB, INC, COMPXCHG (compare-and-swap, CAS), and so on.

c. *Select:* references a subset of the elements of vector/matrix objects. Some of these operations return a reference to the elements of matrix/vector objects, so they can be used as L-values in the statements.

d. *Boolean reduction:* it is applied to mask objects. Two operations are supported, *any* and *all*. *v.any()* returns 1 if any of the elements of v is 1; otherwise it returns 0. *V.all()* returns 1 if all the elements of v are 1; otherwise it returns 0.

Even when GenX supports Shared Virtual Memory and atomic operations, the current implementation of CMSL is based on pre-allocated buffers. These buffers are managed by the CM runtime internally.

6.2.3 Data structure operations

CMSL provides three main operations: *searchKey*, *insertKey* and *deleteKey*. The source code (adapted CM pseudocode) of the skiplist implementation is described in the following subsections.

To express instruction level parallelism through SIMD operations, lines are highlighted in Algorithms 5, 6, 7, 8 and 9. For example, the highlighted line in Algorithm 5 $mask \longleftarrow (k \geq keys)$ reflects that the comparison between the vector $keys$ and the value k has been performed in one SIMD instruction, storing the result in $mask$.

The compare-and-swap operation (CAS) in Algorithms 8 and 9 follows the standard format *bool CAS(addr, old, new)* but considering it as SIMD CAS. It checks in one atomic operation the equality between the values on *addr* and the values on *old*. If equal, it changes the values of *addr* to *new* values and returns *true* when the change is successful.

Search

The search operation is typically the most common operation in skiplist workloads, so it is important to traverse the skiplist as fast as possible.

Following the original skiplist algorithm, the *searchKey* operation searches for the key's range first and then performs a linear search within the list found. Considering the skiplist structure from Figure 6.2 as reference, Algorithm 5 and 6 illustrate the search operation in two dimensions: *findList* performs the search within the chunk of levels horizontally, while *searchKey* (starting from line 4) performs the search within the chunks of keys vertically.

Algorithm 5 Find list

```
 1: procedure FINDLIST($skiplist, k$)
 2:     $offset \leftarrow 0$                                   ▷ Root of skiplist
 3:     next_list:
 4:     $offsets \leftarrow read(skiplist, offset)$
 5:     $keys \leftarrow read(skiplist, offsets)$             ▷ Scattered read of minimum keys
 6:     $mask \leftarrow (k \geq keys)$
 7:     if $mask.any()$ then
 8:         $i \leftarrow getIndex(mask)$                      ▷ Index of the furthest list
 9:         $offset \leftarrow list[i]$
10:         goto $next\_list$
11:     else
12:         return $list$
13:     end if
14: end procedure
```

The *findList* procedure begins by selecting the offsets of the pointed lists from the current list. Then, on line 5 it performs a scattered read of the minimum keys from the pointed lists whose offsets were just selected. Notice that the minimum key of each pointed list corresponds to the first key within the list (offset to the pointed list + index of first key), so if the pointed list exists, it will have a minimum key that indicates the range's beginning for that list. As we are interested in finding the list that contains k, the horizontal traversal continues while $k \geq keys$, so that it gets to the list where no minimum keys from its pointed lists smaller than k exist. Finally, it returns the last list found in the traversal and whose minimum key is the closest to k.

The *searchKey* procedure in Algorithm 6 begins by calling *findList*. Once it gets the list that might contain k, it starts the SIMD linear search by comparing each attached chunk of keys with k. When the boolean reduction on line 6 is true, it returns the key from its chunk

Algorithm 6 Search key

```
 1: procedure SEARCHKEY($skiplist, k$)
 2:     $list \leftarrow findList(skiplist, k)$
 3:     $keys \leftarrow list.keys$
 4:     next_chunk:
 5:     $mask \leftarrow (keys == k)$
 6:     if $mask.any()$ then
 7:         $i \leftarrow getIndex(mask)$                ▷ Index of the key
 8:         return $keys[i]$
 9:     else
10:         if $keys.next == null \mid\mid keys[0] > key$ then
11:             return $null$
12:         end if
13:         $keys \leftarrow read(skiplist, keys.next)$
14:         goto $next\_chunk$
15:     end if
16: end procedure
```

or object associated to it; otherwise, it checks if there is another attached chunk and if it is still in the k's range to continue the search.

While traversing lists and chunks in the *searchKey* procedure, there might exist the case where the algorithm encounters lists or chunks marked as removed. These marked lists and chunks are logically removed by the *deleteKey* procedure and correspond to lists or chunks that became empty at some point. The algorithm simple ignores them by continuing the lateral search following the pointed lists in *findList*, or continuing the down search following the pointer to the next chunk of keys.

Insert

The insert operation receives a key k to be inserted in the skiplist. The insertion is successful only if the key does not exist in the data structure. The procedure follows the same steps from *searchKey* in order to find the right chunk and position for the new key. Once they are found, atomic operations are performed to guarantee synchronization and the correct state of the structure at every phase of the algorithm.

Algorithm 7 Insert key

```
1: procedure InsertKey($skiplist, k$)
2:     $level \leftarrow randomLevel()$
3:     restart:
4:     $list \leftarrow findList(skiplist, k)$
5:     $chunk \leftarrow list.keys$
6:     next_chunk:
7:     $mask \leftarrow (chunk \geq k) \mid (chunk == 0)$
8:     if $mask.any()$ then
9:         $i \leftarrow getIndex(mask)$                ▷ Index of current match
10:        if $chunk[i] == key$ then
11:            return $false$                             ▷ Key already exists
12:        end if
13:        if $!insertInChunk(chunk, level, k, i)$ then
14:            goto $restart$
15:        end if
16:    else
17:        if $chunk.next == null$ then
18:            if $!insertInChunk(chunk, level, k, -1)$ then
19:                goto $restart$
20:            end if
21:        else
22:            $chunk \leftarrow read(skiplist, chunk.next)$
23:            goto $next\_chunk$
24:        end if
25:    end if
26:    return $true$
27: end procedure
```

Algorithm 7 describes the *insertKey* procedure. It begins by obtaining a random value to define the level for the current insertion, which is within the range $0 \leq level \leq 31$. On line 4 it calls *findList* and obtains the list whose range corresponds to the new key. Then, on line 7 it searches for the right position for k, which can be at the position where a key greater or equal than k exists or a vector cell is available (zero in the cell). When a position is found, a final check needs to be done: whether it corresponds to the same key or not. If it is not the same key, it finally calls the subroutine *insertInChunk* to perform the insertion. In the case that the current chunk has no keys greater than k or cells available, the algorithm continues by checking if there exists an attached chunk. When there is no attached chunk, it means that the new key corresponds to the greatest key in the current list and must be placed in a new chunk since there is no space available in the last one. Line 18 reflects this situation by calling *insertInChunk* with no position for the new key, which will be interpreted as a insertion in a new chunk.

As the thread synchronization in CMSL is managed by CAS atomic operations directly within its operations, whenever a thread fails to insert a new key because of the failing CAS operation, it will try the insertion again until it gets successfully completed. This is valid for the two cases explained above, whether the insertion fails on lines 13 or 18 the process is restarted.

Algorithm 8 Insert in Chunk

```
procedure INSERTINCHUNK($chunk, level, k, i$)
    if $level == 0$ then
        $newChunk \leftarrow insert(chunk, k, i)$
        return $CAS(offset, chunk, newChunk)$
    else
        $newList \leftarrow allocate()$
        $newList.keys[0] \leftarrow key$
        $newList.keys \leftarrow stealKeys(chunk, i)$
        $newChunk \leftarrow removeKeys(chunk, i)$
        $newChunk.next \leftarrow newList.chunk.offset$
        if $!CAS(offset, chunk, newChunk)$ then
            return $false$
        end if
        $updateLinks()$
    end if
    return $true$
end procedure
```

The subroutine *insertInChunk*, in Algorithm 8, is responsible for doing all the atomic updates needed to insert new keys in existing chunks, new chunks or new lists. It receives the chunk where the insertion will be done, the level for this insertion, the key k, and the position i. There are basically two cases depending on *level* when performing the final step of insertion:

a. *level* is equal to zero, k is inserted in current chunk: a new vector representing the chunk is created, and k is inserted at position i. When the position i is occupied by another key, it and greater keys are shifted to the right. Finally, the SIMD CAS operation is performed as shown on line 4.

b. *level* is greater than zero, k is inserted in a new list with the given *level*: a new list is allocated. k is inserted at the first position of the new list's chunk of keys. Keys greater than k from the current chunk are copied to the new list's chunk of keys starting at position 1 (line 8); in case that an attached chunk exists, its offset is also copied. Then, a new vector representing the chunk without the stolen keys is prepared: all the stolen keys are removed (line 9) and the pointer to attached chunk is updated to a pointer to the new list's chunk of keys (line 10). This last step ensures that the state of the structure remains correct even if there are threads traversing the current chunk when the insertion has been partially completed (without updating the pointers in the chunk of levels yet). Finally, on line 11 the current chunk is updated through a SIMD CAS operation. If the CAS operation fails, the subroutine returns *false* and the insertion is restarted; otherwise, the algorithms proceeds by updating the offsets to the new list in the chunk of levels from the current list and previous ones until the desired *level* is reached (line 14).

   Observe that when updating offsets in chunks of levels there might exist the case of moving to previous lists in order to achieve the desired level. For this case, a traversal path obtained from the *findList* procedure is used.

Example 6.1. *Inserting a key with level 0 in a chunk that is full. Consider that insertKey(67) is called and the procedure already found the list and the chunk where the key should be inserted. Figure 6.3 shows the current state of the chunk of keys where 67 should be inserted.*

As the current chunk of keys is full, the algorithm proceeds by creating a new chunk with the *new key* 67 *and some keys stolen from the current chunk. It also includes the pointer to the next attached chunk. Finally, it performs the SIMD CAS operation on the current chunk to remove the already copied keys and update the next pointer to the new chunk.*

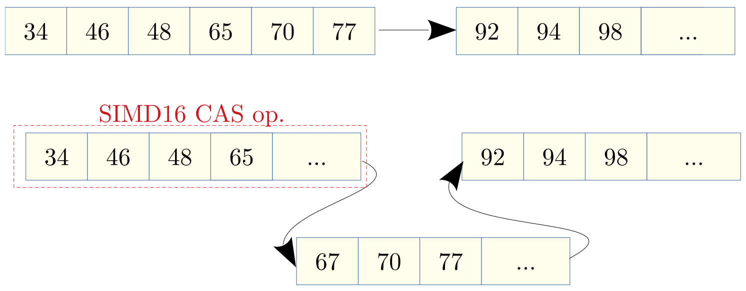
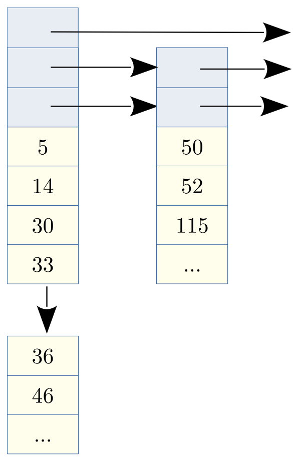


Figure 6.3: Insertion of key 67

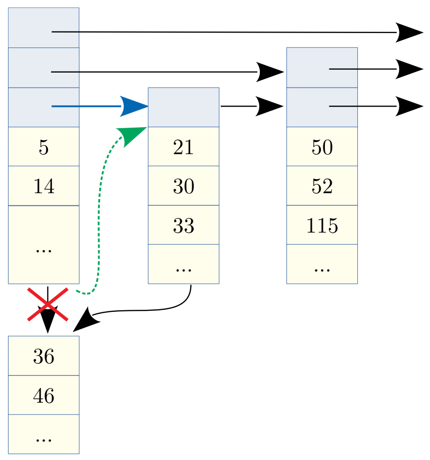
Example 6.2. *Inserting a key with level 1. Consider that insertKey(21) is called and the procedure already found the list and its chunk where the new key should be inserted (current list and current chunk in this example). Figure 6.4 shows the state of the skiplist before and after inserting 21. As it can be seen in Figure 6.4a, the current list with minimum key 5 is the one used for this insertion.*

Notice that the insertion procedure has level 1, so all the keys greater than 21 must be moved to the new list with level 1. The algorithm begins by allocating a new list (shown in the middle of Figure 6.4b), and copying the following elements to it: offset of the pointed list from the current list, key 21 at the first position of the new list's chunk keys, keys greater than 21, and the attached chunk of keys with keys that are also greater than 21. Finally, a new chunk that reflects the changes in the current chunk of keys is prepared: copied keys greater than 21 are removed and the pointer to next chunk is updated to a pointer to the new list's chunk of keys.

By this time everything is prepared to perform the SIMD CAS operation safely on the current chunk of keys. This update will guarantee that if any thread traverses the current chunk looking for a key greater than 21, it will follow to the new list's chunk of keys just created. Hence a SIMD CAS operation is performed on the current chunk of keys to remove copied



(a) Original skiplist



(b) Updated skiplist

Figure 6.4: Insertion of key 21

keys and update the next pointer to the new list's chunk of keys as shown in Figure 6.4b. Notice that the pointer (green arrow) to the new list's chunk of keys is temporal and allows to maintain the correct status of the structure while the insert operation has not been finished.

Finally, the algorithm performs another CAS operation on the current list to update the pointer to the new list (blue arrow in Figure 6.4b). At this moment, the new list can be reached by the search operation through horizontal traversals.

Observe that if the level of key 21 were greater than the level of the current list (level 3), the algorithm proceeds by updating levels from previous lists in a bottom-up fashion.

Delete

The delete operation is similar to the insert operation. Algorithm 9 describes the *deleteKey* procedure. It begins by searching the key to be deleted, k. If k exists in the skiplist, line 7 of Algorithm 9, it prepares a new chunk by removing k and shifting greater keys one position to the left.

Notice that at this point the current chunk can become empty. If that is the case, it is marked as removed (line 11). Additionally, the current list can be also marked as removed if the chunk being marked is the last chunk of the list. This first step can be considered as a logical removal. Marked lists and chunks are omitted by threads traversing horizontally through lists and vertically through chunks respectively. Insert and delete operations on the previous list and chunk will eventually perform a physical removal of marked lists and chunks by updating their next pointers.

Algorithm 9 Delete key

```
1:  procedure DELETEKEY($skiplist, k$)
2:      restart:
3:      $list \leftarrow findList(skiplist, k)$
4:      $chunk \leftarrow list.keys$
5:      next_chunk:
6:      $mask \leftarrow (chunk == k)$
7:      if $mask.any()$ then
8:          $i \leftarrow getIndex(mask)$                          ▷ Index of the match
9:          $updatedChunk \leftarrow removeKey(chunk, i)$
10:         if $updatedChunk.isEmpty()$ then
11:             $maskAsRemoved(updatedChunk)$
12:         end if
13:         if $!CAS(offset, chunk, updatedChunk)$ then
14:             goto $restart$
15:         end if
16:     else
17:         if $chunk.next == null \; || \; chunk[0] > k$ then
18:             return $false$                                  ▷ Key not found
19:         end if
20:         $chunk \leftarrow read(skiplist, chunk.next)$
21:         goto $next\_chunk$
22:     end if
23:     return $true$
24: end procedure
```

Ultimately, the algorithm performs the SIMD CAS operation on the current chunk (line 13); if it fails the process is restarted. The rest of the cases are analogous to the *insertKey* procedure.

6.2.4 Correctness

The proposed skiplist represents a set data structure. A key k is in the set if there is a path from the first list to a chunk of keys that contains k, otherwise it is not in the set.

A valid skiplist has the following properties:

- The keys in the skiplist are sorted by their ascending order
- There are no duplicated keys
- Every key has a path to it from the first list

Support for the atomic SIMD16 CAS operation by the GenX hardware is assumed. We use the concept linearizability [36, 35] to prove the correctness of CMSL. Linearizability is a global property that ensures that when two processes each execute a series of method calls on a shared object there is a sequential ordering of these method calls that does not necessarily preserve program order (the order in which the programmer wrote them down), but each method call does seem to happen instantly (i.e., invocation and response follow each other directly), whilst maintaining the result of each method call individually and consequently the object its state. This definition of linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and their response.
- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

The following fundamental lemma is derived from the properties of a valid skiplist:

Lemma 6.5. *For any list l, key $k \in l$ and $0 < i \leq l.level$*

$$l[i].minKey \neq null \Rightarrow k < l[i].minKey$$

Where $l[i].minKey$ represents the minimum key of the pointed list from l at level i.

Proof. Line 7 in *insertKey* ensures that new keys are inserted at the corresponding position (ascending order) in the chunk of keys. On the other hand, lines 6 and 7 in *findList* ensure that always the right list l is found such that $l.keys[0] \leq k < l[0].minKey$. □

Lemma 6.6. *The successful searchKey and failed insertKey of an existing key operations on a valid skiplist are linearizable.*

Proof. First note that neither of these operations make modifications to the skiplist so they result in a valid skiplist. For these operations we must have a chunk with key k at the point of linearization. The linearization point is when the chunk is read on line 13 in *searchKey* and line 22 in *insertKey*. The check in the following step ensures that this is the right chunk and it contains k. Finally, lemma 6.5 ensures that the skiplist is valid and it is the only list with a chunk containing key k. □

Lemma 6.7. *The failed searchKey and failed delete operations on a valid list are linearizable.*

Proof. Firstly, note that neither of these operations make modifications to the skiplist so they result in a valid skiplist. We must have no key k in the skiplist at the linearization point. After locating the list where the key could have been located (whose first key is smaller or equal than k), the linearization point is where the last chunk of keys is read, line 13 in *searchKey* and 20 in *deleteKey*. Later, on line 10 in *searchKey* and 6 in *deleteKey* the entire chunk is compared with k which must not have a key equal to k. Observe that even though chunk of levels maintains pointers to every next list, the *findList* operation always goes to the actual list that contains the chunk with the key. So in the case of a lists marked as deleted, they are discarded. □

Lemma 6.8. *The successful insert operations on a valid list are linearizable.*

Proof. The precondition is that there is no key equal to k in the skiplist. The post-condition is a valid skiplist with the key k in some list's chunk. There are two cases of linearization point. The first linearization point is the successful CAS operation inserting k in a chunk of keys on line 4 in *insertInChunk*, in this case there is available space to insert the new key in the chunk. The second case is when a chunk is full of keys or a new list is created due to a level greater than 0, here the linearization point is the successful CAS changing chunk's next pointer on line 4 for the first scenario and on line 11 for the latter. Notice that for the case of creating a new list with level greater than 0, there is at least one more CAS operation on the previous list that updates the pointer to the new list (line 14). Even when the key was already inserted and the skiplist is still valid, the post-processing step on line 14 to update upper levels is necessary to achieve the probabilistic assignment of levels on new lists. □

Lemma 6.9. *The successful delete operations on a valid list are linearizable.*

Proof. This operation has a precondition that there exists a chunk of keys in the skiplist with key k. The post-condition is having the same chunk in the skiplist without the key k or without the entire chunk if it only had the key k at the time of linearization point. The linearization point is the successful CAS removing the key from the chunk on line 13 and removing the entire chunk on line 11 and 13. The existence of k for the precondition is ensured by the CAS on line 13 along with line 7. The CAS ensures the post-condition of chunk without k along with lemma 6.6 and that the skiplist is still valid. □

Theorem 6.2. *CMSL satisfies linearizability for the set abstraction.*

Proof. This follows from lemmata 6.6, 6.7, 6.8 and 6.9. □

Notice that other combinations of operations and their linearization points can be directly derived from the lemmata described above.

A similar proof can be extended for Θ-node transformations on CMSL.

Lemma 6.10. *The successful searchKey and failed insertKey of an existing key operations on a valid skiplist are linearizable.*

Proof. First note that neither of these operations make modifications to the skiplist so they result in a valid skiplist. For these operations we must have a Θ_X with key k at the point of linearization. The linearization point is when Θ_x is read in *searchKey* and in *insertKey*. The check in the following step ensures that this is the right Θ_X and it contains k. Finally, lemma 6.5 ensures that the skiplist is valid and it is the only list with a Θ_X containing key k. □

Lemma 6.11. *The failed searchKey and failed delete operations on a valid list are linearizable.*

Proof. Firstly, note that neither of these operations make modifications to the skiplist so they result in a valid skiplist. We must have no key k in the skiplist at the linearization point. After locating the list where the key could have been located (whose first key is smaller or equal than k), the linearization point is where the last Θ_X of keys is read in *searchKey* and *deleteKey*. Later, in both procedures, the entire Θ_X is compared with k which must not have a key equal to k. Observe that even though Θ_X of levels maintains pointers to every next list, the *findList* operation always goes to the actual list that contains the chunk with the key. So in the case of a lists marked as deleted, they are discarded. □

Lemma 6.12. *The successful insert operations on a valid list are linearizable.*

Proof. The precondition is that there is no key equal to k in the skiplist. The post-condition is a valid skiplist with the key k in some list's Θ_X. There are two cases of linearization point. The first linearization point is the successful SIMD CAS operation inserting k in a Θ of keys in *insertInChunk*, in this case there is available space to insert the new key in the chunk. The second case is when a Θ_X is full of keys or a new list is created due to a level greater than 0, here the linearization point is the successful SIMD CAS changing $Theta_X$'s next pointer. Notice that for the case of creating a new list with level greater than 0, there is at least one more SIMD CAS operation on the previous list that updates the pointer to the new list. Even when the key was already inserted and the skiplist is still valid, the post-processing step on line 14 to update upper levels is necessary to achieve the probabilistic assignment of levels on new lists. □

Lemma 6.13. *The successful delete operations on a valid list are linearizable.*

Proof. This operation has a precondition that there exists a Θ_x of keys in the skiplist with key k. The post-condition is having the same Θ_X in the skiplist without the key k or without the entire Θ_X if it only had the key k at the time of linearization point. The linearization point is the successful SIMD CAS removing the key from Θ_X or removing the entire Θ_X. The existence of k for the precondition is ensured by the SIMD CAS itself (it would fail otherwise). The CAS ensures the post-condition of Θ_X without k along with lemma 6.10 and that the skiplist is still valid. □

Theorem 6.3. *Θ-node based CMSL satisfies linearizability for the set abstraction.*

Proof. This follows from lemmata 6.10, 6.11, 6.12 and 6.13. □

CMSL is lock-free. It only uses CAS operations (atomic SIMD1, SIMD8 and SIMD16) as a synchronization technique for update operations and no synchronization during search operations. The only way for an operation to not exit the main loop (line 3 in Algorithm 7 and line 2 in Algorithm 9) is to have its CAS interfere with another CAS executed by another thread on the same memory location. Note that this guarantees that each time a CAS fails, another thread succeeds and thus the whole system always makes progress, satisfying the *lock-freedom* property.

6.2.5 Experimental Results

A set of experiments were carried out in order to evaluate the performance of the proposed skiplist and to compare it to CPU and discrete GPU state-of-the-art implementations.

Algorithms were implemented in CM for GenX and in C++11 for CPU. The experiments were carried out on a Intel(R) Core i7-8670HQ CPU, with a total of 4 physical cores and 8 logical processors running at 2.70GHz. This machine has per-core L1 and L2 caches of sizes 32KB and 256KB, respectively and a shared L3 cache of 8MB, with a 16GB DDR RAM memory. In terms of graphics, it has an Intel(R) Iris Pro Graphics 580, dubbed GT4e version and Gen9 microarchitecture, with 128MB of dedicated eDRAM memory and features 72 Execution Units (EUs) running at 350 - 1100 (Boost) MHz. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in *time.h*.

Power consumption is calculated by using energy counters between two time samples and converting to power for that time sample.

The initial size of CMSL is 1 million keys for all the experiments in Sections 6.2.5, 6.2.5 and 6.2.5. The same initial size is used for the rest of the skiplist implementations used for comparison.

Comparison with CPU

Use cases to test concurrent data structures are typically lists of combined concurrent operations distrbuted among all the threads. We defined a set of combined operations by ratios, i.e. from the total amount of operations performed by a thread, percentages of them that correspond to search, insert and delete. These use cases were measured using CMSL and state-of-the-art concurrent Skiplist for CPU [21, 62, 14, 54].

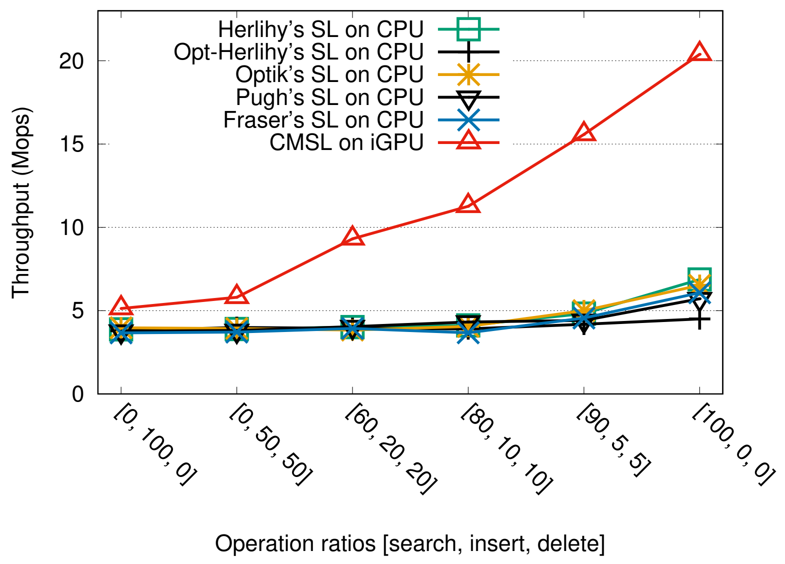


Figure 6.5: Throughput of different Skiplists on GenX and CPU

Figure 6.5 shows the number of operations per second (in millions) that are achieved by different skiplists on CPU and our proposal on iGPU. The $x - axis$ depicts the ratio of [search, insert, delete] operations, i.e. the leftmost data item corresponds to insert-only operations, while the rightmost data item corresponds to search-only operations.

The skiplists tested on CPU are: *a) Herlihy's*, update operations are optimistic, so it finds the node to update and then acquires the locks at all levels, validates the nodes, and performs the update. Searches simply traverse the multiple levels of lists; *b) Herlihy-Optik's*, optimized version of Herlihy's skiplist using Optik patterns; *c) Optik's*, skiplist implementation using Optik pattens; *d) Pugh's*, optimized implementation of the first skiplist proposal by Pugh, it maintains several levels of lists where locks are used to perform update operations; *e) Fraser's*, optimistically searches/parses the list and then does CAS at each level (for updates). A search/parse restarts if a marked element is met when switching levels. The same applies if a CAS fails.

Notice that the concurrent skiplists tested on CPU are mainly designed for many-core CPUs. They scale well for multiple threads (e.g. over 16). However, when testing up to 8-thread on a 4-core CPU their performance is not scalable. In contrast, CMSL scales well on the iGPU of the same 4-core processor.

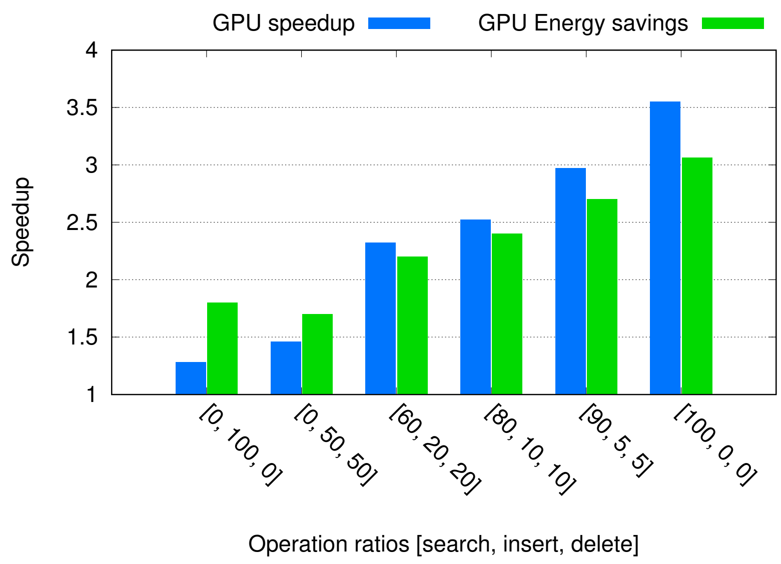


Figure 6.6: Speedup of CMSL on GenX over Fraser's skiplist for CPU

CMSL achieves higher number of operations for all the scenarios, and its best performance is when all the operations are search with up to 3.1x speedup. When all the operations are updates, it still presents 1.2-1.5x speedup. We chose the Fraser's skiplist to perform an individual comparison in terms of GPU speedup and energy savings. Figure 6.6 illustrates the speedup of CMSL over Fraser's skiplist. CMSL achieves up to 3.5x speedup when all the operations are search and 1.3x speedup when all the operations are insert; the rest of mixture operations present speedups between these numbers. Similarly, it also shows the energy savings for all the experiments, achieving of up to 300% when all the operations were search.

It is noteworthy where the performance gains of CMSL come from. All the CPU implementations are based on single-node design; e.g, traversals on levels are done through node pointers one at a time, comparison operations within the bottom level are performed one by one, and so on. In contrast, CMSL performs traversals on levels through 8, 16, 24 or 32 levels at a time, and comparisons at the bottom levels are performed every 16 elements with one single SIMD operation.

Comparison with discrete GPU

Even though CMSL is the first proposal for iGPU, there exists a skiplist implementation for discrete GPU using CUDA. Misra and Chaudhuri (M&C) in [50] ported several concurrent data structures to GPU, including a lock-free skiplist. The main difference between CMSL and M&C is that the latter is pointer-based, so physical updates within the lists are performed by single-word CAS operations.

It is evident that a direct comparison cannot be done since they are executed on different graphics processors, which vary in architecture, area of architecture components, power, etc. However, we measured efficiency e in terms of measured performance ($1/time$) per compute-power (peak GFLOPS – pG), which is calculated as

$$e = \frac{1}{time \, * \, pG} \tag{6.1}$$

Thus we can characterize and compare algorithms in terms of efficiency running on different kind of graphics processors.

A Nvidia GTX 970 GPU was used to run different workloads of the M&C skiplist. This graphics card has 13 active streaming multiprocessors, a total of 1,664 cores, and 4GB of GDDR5 memory capacity. In terms of GFlops, the Nvidia GTX 970 has a peak GFlops of 3,920, while the Intel's GT4 Gen9 has a peak GFlops of 1,152.

Table 6.1 shows the running times of each skiplist when executing 1 million operations. The operation ratios are the same used in the comparison with CPU algorithms. The relative efficiency in the rightmost column of the table provides the efficiency of CMSL divided by

Operations [search, insert, delete]	CMSL on iGPU (in ms)	M&C on dGPU (in ms)	Relative efficiency of CMSL
[0, 100, 0]	243	110	1.5
[0, 50, 50]	211	109	1.6
[60, 20, 20]	141	69	1.7
[80, 10, 10]	93	48	1.8
[90, 5, 5]	71	42	2.0
[100, 0, 0]	61	44	2.5

Table 6.1: Performance comparison between CMSL and M&C skiplists. Relative efficiency of CMSL is calculated by $\frac{e_{CMSL}}{e_{M\&C}}$ from equation (1). Tests on 1M keys.

the efficiency of M&C – in a sense, this measures how much more performance CMSL is able to extract from the iGPU with the same compute-power.

As the column on relative efficiency depicts, in all the cases CMSL is capable of achieving more performance on a CM-GenX than the M&C skiplist on CUDA-GPU with the same compute-power. The performance of CMSL is significantly more efficient in search-dominant scenarios due to effective utilization of varying SIMD width within a kernel and usage of CM intrinsic functions to make use of the GenX hardware features. When having mostly updates operations, efficiency of CMSL is slightly more than M&C with a performance of 50% higher.

Scalability

In this experiment, a predefined number of threads is set before dispatching the kernel on the iGPU. Then, each thread performs a randomly-generated list of operations in the skiplist. As the skiplist data structure is mainly used for queries, the ratio of operations chosen is defined as: 90% search, 5% insert, and 5% delete.

Figure 6.7 shows the performance in milliseconds of CMSL considering different amount of operations and varying the number of threads. It can be seen that the performance improves

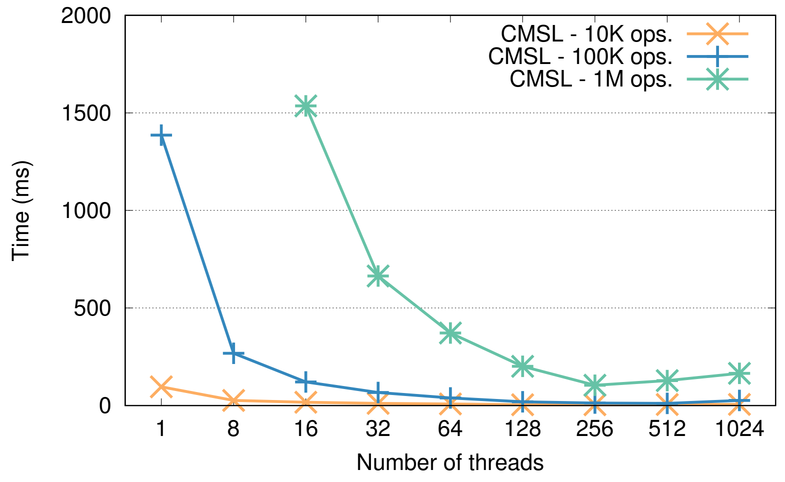


Figure 6.7: Scalability of CMSL by the number of threads. Operations are 90% search, 5% insert and 5% delete.

considerably as the number of threads increases for the three scenarios. The best performance is achieved between 256 and 512 threads, which is consistent with the GT4 Gen9 hardware capabilities where the experiments run on. The Gen9 iGPU features 72 EUs with a total of 504 hardware threads. Note that increasing the number of threads to 1024 produces degradation of the performance due to higher data contention and bandwidth limits.

6.3 SIMT-friendly Data Structures

In SIMT architectures, threads are created, managed, scheduled, and executed in groups of parallel threads called warps (CUDA) or thread group (CM). Comparing with SIMD architecture, SIMT and SIMD are both architectures are similar, the difference relies on how the instruction is applied. While SIMD applies one instruction to multiple data, SIMT applies one instruction to multiple independent threads in parallel. In contrast to SIMD architectures, SIMT enables programmers to write thread level parallel code for independent threads and they can essentially ignore the attributes such as warps. On the other hand, SIMD architectures require to manually manage the data in vectors (SIMD-friendly format).

Figure 6.8 illustrates the same example presented in previous sections but with SIMT processing. A shared data structure is available to be accessed from multiples threads. Each computational unit performs SIMD instruction, however every data item belongs to a different thread. Every thread can be seen as a light thread, and every thread group or wrap can share common instructions that are performed in a data-parallel fashion.

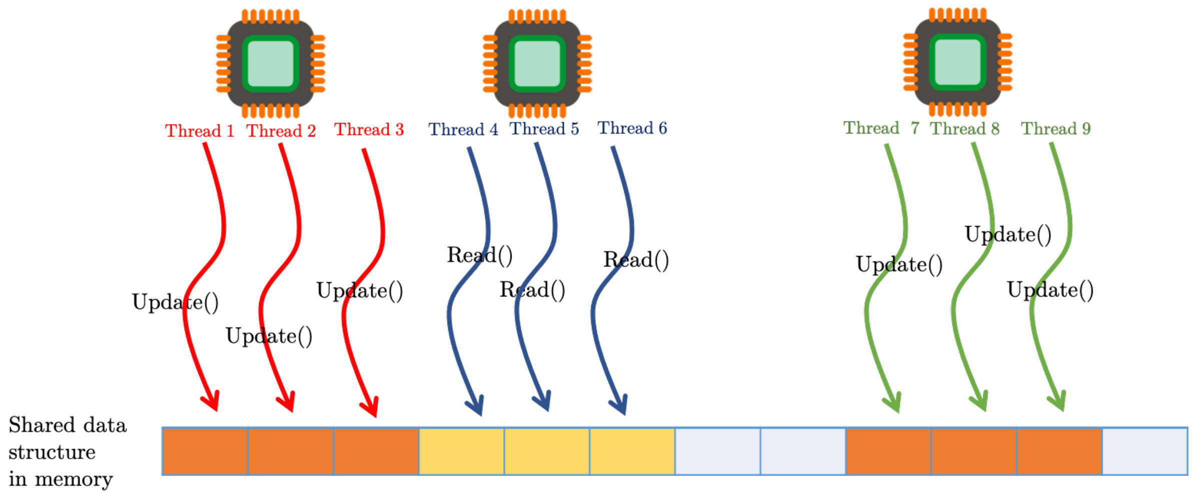


Figure 6.8: SIMT processing

Data structures operations that have low divergence are good candidates to be implemented with SIMT processing. Update operations often have a high number of branches and non-blocking techniques, such as trying to update until succeeding, might produce a high number of cache misses. Search operations, on the other hand, are candidates for SIMT processing as long as they have a reduced number of branches.

The general idea of implementing search operation using SIMT processing is that multiple threads perform traversals in groups or warps. Each thread group executes the same instruction but on different data, i.e. different thread of the same group might be at different nodes in a search tree a at certain moment.

6.3.1 SIMT Tree Search

Algorithm 10 shows the general SIMT algorithm for searching keys in tree structures. It is noteworthy to point out that this implementation represents a node-based tree structure with 2 children (left and right), but it can be generalized to any tree structure with an arbitrary number of children. Line 12 in Algorithm 10 defines the decision to continue the traversal to a next node, so this line can be changed in order to define the number of paths (children) to new nodes.

Algorithm 10 General SIMT tree search

```
 1: procedure treeSearch($root$, $keys$)
 2:     $nodes \leftarrow root$
 3:     while $true$ do
 4:         if $nodes == null$ then
 5:             $break$                                   ▷ Key not found
 6:         end if
 7:         $vals \leftarrow scatter\_read(nodes)$
 8:         if $vals == keys$ then
 9:             $break$                                   ▷ Key found
10:         end if
11:         $offsets \leftarrow 0$                        ▷ Stores next node's offset
12:         if $keys\ is\ within\ a\ range$ then          ▷ Condition for traversing the structure
13:             $offsets \leftarrow node_1\_offset$
14:         else
15:             $offsets \leftarrow node_2\_offset$
16:         end if
17:         $nodes \leftarrow scatter\_read(offset)$      ▷ Visit next node
18:     end while
19: end procedure
```

The search starts by defining the starting point for the thread group (line 2 in Algorithm 10) which is the root of the structure. Then each thread enters the while loop that will end whenever the key is found or not. Notice that the live range of every thread is independent, i.e. a thread that finds its key earlier will break the while loop and finish its operation. All the results will be synchronized and returned once all threads finish their search.

It is expected that more parallelism can be achieved in SIMT processing versus CPU multi-threaded processing if algorithms have low divergence. In terms of data structures, tree structures are good candidates to be implemented with SIMT processing as long as they present low divergence and keys are stored in external nodes, e.g. external binary search trees.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Modern heterogeneous CPU/iGPU architectures present new programming challenges to programmers. First, the inclusion of more CPU cores per chip requires the efficient implementation of new synchronization primitives that allows thread-safe execution of concurrent programs with shared data structures. On the other hand, the new capabilities on Data-Parallelism on these architectures open new a dimension for exploiting parallelism on existing multi-threaded data structures.

Spatial Locks constitute a useful synchronization mechanism that allows to make parallel geometric algorithms thread-safe. Based on Spatial Hashing and Axis-aligned Bounding Boxes (AABB), they provide constant-time lock/unlock operations when updating an object in 2D or 3D space. It has been proven that this synchronization mechanism satisfies mutual exclusion and deadlock-freedom, fundamental properties for any thread synchronization mechanism. Many geometric algorithms can benefit from Spatial Locks given that its use does not require significant changes on the implementation of the geometric algorithms themselves. Our experiments show that highly parallel executions with important speed-up can be obtained when using this synchronization mechanism for mesh simplification processes and big meshes.

A new framework called SIMD-node Transformations to implement non-blocking and linearizable data structures using multi-threaded and SIMD processing was proposed. One- and multi-dimensional data structures, such as skiplists, k-ary trees and multi-level lists, can benefit from new SIMD computing capabilities available in accelerators such as iGPUs by applying these transformations. Ex- perimental results show important performance gains obtained when applying these transformations to a lock-free skiplist. In the future, we plan to implement more non-blocking data structures using SIMD-node transformations.

A lock-free skiplist for iGPU was introduced to show the usefulness of Θ-node transformation on an existing non-blocking data structure. CMSL design is based on chunked lists which helps the effective utilization of SIMD width on GenX and its implementation was proven linearizable. Experimental results show important speedups and energy savings over CPU proposals as well as a more compute-efficient implementation when compared to a discrete GPU proposal. We believe that other array-based concurrent data structures can be implemented using similar design considerations for iGPU.

7.2 Future Work

There is important work to be explored on new applications that benefit from heterogeneous CPU/iGPU processors. This thesis work opens new research in the field of concurrent data structures with SIMD processing as well as using heterogeneous CPU/iGPU processors for general purpose processing. Potential future works are:

- Study how compilers can optimize data structures with SIMD processing. Compilers play an important role when optimizing implementations of data structures with SIMD processing. As in the proposed framework Θ-node transformations, it is the user who defines the sizes of Θ_X, validations should be done in order to avoid impact in performance.

- Extend Θ-node transformations for specific and stronger non-blocking properties, such as lock-free, wait-free, etc., will be useful for application-specific scenarios. Additionally, identifying and implementing more data structures with embedded data-parallelism is an important future step.

- In terms of the Intel CPU/iGPU architecture, it is interesting to study the performance impact of allocating data structures in the Last Level of Cache (LLC) memory, that is shared between the CPU cores and iGPU. An important challenge is to study how mutual exclusion and/or synchronization when updating concurrent data structures from CPU and iGPU is guaranteed. The implementation concurrent data structures allocated in LLC is possible since current the architecture Gen9 supports atomic operations from the executions units on this memory as well as from the CPU cores.

- Extend this work to discrete accelerators such as discrete GPUs. This would allow to implement scalable algorithms for data centers and super computers.

Bibliography

[1] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

[2] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Building an efficient hash table on the gpu. In *Gpu Computing Gems Jade Edition*, pages 39–53. Elsevier, 2011.

[3] V. H. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8):663–677, 2010.

[4] G. v. d. Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[5] F. Cacciola. Triangulated surface mesh simplification. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017.

[6] P. Cignoni, D. Laforenza, R. Perego, R. Scopigno, and C. Montani. Evaluation of parallelization strategies for an incremental delaunay triangulator in e3. *Concurrency and Computation: Practice and Experience*, 7(1):61–80, 1995.

[7] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3d delaunay triangulation. In *Computer Graphics Forum*, volume 12, pages 129–142. Wiley Online Library, 1993.

[8] T. Crain, V. Gramoli, and M. Raynal. Brief announcement: a contention-friendly, non-blocking skip list. In *International Symposium on Distributed Computing*, pages 423–424. Springer, 2012.

[9] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[10] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *PODC*, volume 10, pages 131–140. Citeseer, 2010.

[11] S. Feldman, P. LaBorde, and D. Dechev. Concurrent multi-level arrays: Wait-free extensible hash maps. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 155–163. IEEE, 2013.

[12] S. Feldman, P. Laborde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

[13] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[14] K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[15] J. Fuentes, W. Chen, G. Lueh, and I. D. Scherson. A lock-free skiplist for integrated graphics processing units. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 36–46. IEEE, 2019.

[16] J. Fuentes and F. Luo. Synchronizing parallel geometric algorithms on multi-core machines. *International Journal of Networking and Computing*, 8(2):240–253, 2018.

[17] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997.

[18] A. Gidenstam. Non-blocking algorithms and data structures library reference manual.

[19] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 260–269. ACM, 2002.

[20] N. Grund, E. Derzapf, and M. Guthe. Instant level-of-detail. In *VMV*, pages 293–299, 2011.

[21] R. Guerraoui and V. Trigonakis. Optimistic concurrency with optik. In *ACM SIGPLAN Notices*, volume 51, page 18. ACM, 2016.

[22] H. Guiroux, R. Lachaize, and V. Quema. Multicore locks: The case is not closed yet. *USENIX Annual Technical Conference*, pages 649–662, 2016.

[23] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[24] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

[25] E. J. Hastings, J. Mesit, and R. K. Guha. Optimization of large-scale, real-time simulations by spatial hashing. In *Proc. 2005 Summer Computer Simulation Conference*, volume 37, pages 9–17, 2005.

[26] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*, pages 3–16. Springer, 2005.

[27] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[28] D. Hendler and N. Shavit. Work dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 164–172. ACM, 2002.

[29] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[30] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[31] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[32] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[33] M. Herlihy, Y. Lev, and N. Shavit. A lock-free concurrent skiplist with wait-free search. *Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts*, 2007.

[34] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[35] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2011.

[36] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[37] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *International Conference On Principles Of Distributed Systems*, pages 401–414. Springer, 2007.

[38] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 161–171. ACM, 2012.

[39] Intel Corporation. C-for-media compiler. `https://github.com/intel/cm-compiler`, 2018.

[40] S. Junkins. The compute architecture of Intel® Processor Graphics Gen9. *Intel whitepaper v1*, 2015.

[41] F. Khorasani, M. E. Belviranli, R. Gupta, and L. N. Bhuyan. Stadium hashing: Scalable and flexible hashing on gpus. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 63–74. IEEE, 2015.

[42] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[43] J. Kohout, I. Kolingerová, and J. Žára. Parallel delaunay triangulation in e 2 and e 3 for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[44] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *International Symposium on Distributed Computing*, pages 117–131. Springer, 2004.

[45] D. Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[46] P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 279–286, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[47] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and portable locking for multicore architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(4):13, 2016.

[48] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[49] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[50] P. Misra and M. Chaudhuri. Performance evaluation of concurrent lock-free data structures on gpus. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 53–60. IEEE, 2012.

[51] M. Moir and N. Shavit. Concurrent data structures., 2004.

[52] N. Moscovici, N. Cohen, and E. Petrank. A gpu-friendly skiplist algorithm. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 246–259. Ieee, 2017.

[53] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014.

[54] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[55] W. Pugh. Concurrent maintenance of skip lists. Technical report, 1998.

[56] A. Ramachandran and N. Mittal. A fast lock-free internal binary search tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, page 37. ACM, 2015.

[57] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. Simd parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.

[58] B. Schlegel, R. Gemulla, and W. Lehner. K-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 52–60. ACM, 2009.

[59] T. R. Scogland and W.-c. Feng. Design and evaluation of scalable concurrent queues for many-core architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 63–74. ACM, 2015.

[60] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[61] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 54–63, 1995.

[62] N. N. Shavit, Y. Lev, and M. P. Herlihy. Concurrent lock-free skiplist with wait-free contains operator, May 3 2011. US Patent 7,937,378.

[63] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *International Conference On Principles Of Distributed Systems*, pages 240–255. Springer, 2004.

[64] R. K. Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research ..., 1986.

[65] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC*, volume 95, pages 214–222. Citeseer, 1995.

[66] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.

[67] S. Zeuch, F. Huber, and J.-c. Freytag. Adapting tree structures for processing with simd instructions. 2014.

[68] Q. Zhou and A. Jacobson. Thingi10k: A dataset of 10, 000 3d-printing models. *CoRR*, abs/1605.04797, 2016.