Data Structures Libraries

By: Leonor Frias Moya

Advisors: Jordi Petit Silvestre and Salvador Roura Ferret

PhD Thesis
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
June 2010

This PhD Dissertation was publically defended on June 8th, 2010, in Barcelona.
According to the judgement of the Jury, this PhD Dissertation obtained the qualification *excellent cum laude* and with the *European mention*.

In the public defense, it was my honor to have the following PhD Jury:

- Prof. Dr. Conrado Martínez, Universitat Politècnica de Catalunya, Spain
- Dr. Joaquim Gabarró, Universitat Politècnica de Catalunya, Spain
- Prof. Dr. Ulrich Meyer, Goethe Universität Frankfurt am Main, Germany
- Prof. Dr. Giuseppe Italiano, Università di Roma 'Tor Vergata', Italy
- Dr. Ricardo Baeza-Yates, Universitat Pompeu Fabra, Spain

To whom I am (and I will always be) very grateful.

Leonor Frias Moya

I dedicate this work to my parents,
Eusebio and Leonor

Dedico este trabajo a mis padres,
Eusebio y Leonor

Resum

Aquesta tesi doctoral estudia diversos problemes que sorgeixen quan es combina teoria i pràctica en algorísmia, en particular, quan es consideren els requirements de les biblioteques de programari. La *Standard Template Library (STL)* del llenguatge de programació C++ actua com a fil conductor en l'estudi de problemes fonamentals en algorísmia, com ara les seqüències, els diccionaris, l'ordenació i la selecció. Per fer-ho, seguim aproximacions diverses però interrelacionades, i que han estat parcialment influenciades pels canvis recents i radicals en l'arquitectura dels computadors.

Per una banda, presentem diversos algorismes i estructures de dades que es fonamenten en les capacitats dels computadors moderns. Concretament, de les `lists` de la STL presentem implementacions conscients de la memòria cau que, tot aprofitant les propietats d'aquestes jerarquies de memòria, aconsegueixen recorreguts molt eficients. Alhora, com també és el cas de les implementacions de la STL amb llistes doblement enllaçades, compleixen amb els requeriments de cost de l'estàndard per a les operacions, així com amb la validesa en tot moment dels iteradors.

També presentem diversos algorismes pràctics per a processadors *multi-core*. Concretament, descrivim algorismes paral·lels per a la construcció i la inserció múltiple en arbres vermell-negre. Aquests algorismes es basen en (a) l'accés en temps constant als elements usant el seu índex, com és el cas també dels algorismes en el model PRAM, (b) operacions d'unió i divisió d'arbres, així com (c) tècniques d'equilibri dinàmic de la càrrega paral·lela. La nostra implementació estén la del compilador GCC per als diccionaris de la STL. Els resultats experimentals en mostren la conveniència pràctica. A més a més, per preprocessar l'entrada de les anteriors operacions eficientment, hem definit un algorisme *online* general per particionar seqüències de mida desconeguda. Aquest mètode té un interès tant teòric com pràctic.

Per altra banda, descrivim millores en algorismes, ja existents, basats en reusar part dels càlculs previs. En primer lloc, presentem implementacions en paral·lel de la `partition` de la STL. Aquests són els primers algorismes pràctics que fan un nombre òptim de comparacions, és a dir, una per element. Més endavant, mostrem la utilitat d'aquesta propietat quan els elements són cadenes de caràcters (*string*s) i es fan particions repetitivament i jeràrquica, com ara en el *quicksort* i el *quickselect*. Certament, mostrem grans millores de rendiment en combinar-los amb tècniques ja existents per evitar comparacions de caràcters redundants. Els nostres algorismes poden usar-se per especialitzar el `sort` i el `nth_element` de la STL, respectivament. També, fem notar que algunes de les comparacions en aquests tipus d'algorismes i estructures de dades no accedeixen a les cadenes de caràcters pròpiament dites. Si, com és habitual, els *string*s s'implementen mitjançant punters, aquest fet és rellevant si es considera l'ús de la memòria cau. En aquest sentit, quantifiquem analíticament el nombre d'accessos a *string*s en arbres binaris de cerca, i també en *quicksort* i *quickselect* modificats d'aquesta manera. També analitzem els beneficis d'afegir redundància a aquests algorismes i estructures de dades.

Finalment, presentem i analitzem detalladament l'algorisme *multikey quickselect*, que és l'anàleg de l'algorisme d'ordenació *multikey quicksort* per al problema de la selecció per a *string*s. Aquest algorisme és eficient respecte el nombre de comparacions de caràcters, no necessita memòria auxiliar, i és fàcil d'implementar. A més a més, es pot usar per a especialitzar el `nth_element` de la STL. La nostra anàlisi considera tant el nombre de comparacions com el d'intercanvis sota la hipòtesi d'una distribució de claus aleatòria uniforme. Addicionalment, proposem diverses variants per millorar l'eficiència, que també es poden aplicar al *multikey quicksort*.

Abstract

This thesis studies several problems that arise from combining theory and practice in algorithmics, in particular, when considering requirements in software libraries. The well-known *Standard Template Library* (STL) of the C++ programming language is the common thread of this research. We have tackled fundamental problems in algorithmics, such as sorting and selection, sequences and dictionaries. This aim has been achieved using varied, intertwined and evolving approaches, in particular, partially influenced by fast and radical changes in technology.

Most of the contributions concern generic algorithms and data structures that take advantage of the processing capabilities of modern computers. First of all, we present cache-conscious implementations of STL `lists`. Taking advantage of the properties of memory hierarchies, very good performance is obtained for traversal-based operations. But like typical STL doubly linked list implementations, they keep up with the required cost for operations as well as with the validity of iterators at all times.

Besides, we present several practical algorithms for multi-core computers. Specifically, we describe parallel algorithms for the construction and insertion of many elements at a time into red-black trees. These algorithms use (a) constant-cost index computations to access the elements, as it is the case for some existing algorithms in the PRAM model, (b) split and union operations on trees, and (c) parallel load-balancing techniques. The implementation extends a red-black tree codification of STL dictionaries in the GCC compiler. The experiments show the practicability of this approach. In addition, motivated by efficiently preprocessing the input of the aforementioned bulk operations in dictionaries, we describe a general online algorithm for partitioning sequences whose size is unknown. Our algorithm is interesting from both a theoretical and practical perspective.

Furthermore, we have devised new variants of algorithms by exploiting the reusing of computations. In particular, we consider the parallel partitioning of an array with respect to a pivot element. We present practical algorithms that can be used to implement STL `partition`. These are the first such algorithms that perform an optimal number of comparisons, i.e., one per element. Later, we show that this property is particularly useful when the elements are strings and partitioning is repetitively used and in a hierarchical way, as in quicksort and quickselect. Specifically, we show big performance improvements combining parallel quicksort and quickselect with existing techniques to avoid redundant character comparisons. Furthermore, the resulting algorithms can be used to specialize STL `sort` and `nth_element`, respectively. Moreover, we highlight that some of the comparisons made in this kind of algorithms and data structures do not need to access the actual strings. Under the common assumption of a pointer string implementation, this is very relevant from a cache-conscious perspective. In this sense, we analytically quantify the number of string lookups in so modified binary search trees, quicksort and quickselect. Also, we analyze the benefits of adding some redundancy on the top of these algorithms and data structures.

Finally, we present and analyze *multikey quickselect*, the analog of multikey quicksort for the selection problem for strings. This algorithm is efficient with respect to the number of character comparisons, it is in-place and it is easy to implement. In particular, it can be used to specialize `nth_element` for strings. We present an analysis of its cost, measuring both the number of comparisons and the number of swaps, under a random uniform distribution. Last but not least, we propose several enhanced variants, that can be also applied to multikey quicksort.

Acknowledgements

In the development of my PhD thesis, I have accumulated knowledge about several topics, fluency in giving talks, but also, and above all, an international vision and experience about (research) work and life. But in order to make this moment a reality, I have had to overcome several challenges of various kinds. And I could not have made it without the support of the people around me. These acknowledgements are dedicated to them.

Firstly, I thank my advisors, Jordi Petit and Salvador Roura, for willing to advise my thesis. In particular, I am specially grateful for their useful advice and patience. Besides, I am also very grateful to Conrado Martínez for counting with me in his research projects and for being very accessible. In addition, I thank the rest of ALBCOM members, especially María J. Serna, for including me into account into their research activities. In particular, I enjoyed being part of several organizing committees, as well as attending several courses and conferences abroad. I had the chance to share some of them with Amalia Duch and María J. Blesa, to whom I am grateful for their constant moral support and trust. In particular, I thank María for giving me a copy of her dissertation with a moving dedication. Finally, I am also thankful to Kim Gabarró for his always very respectful and motivating views towards my work.

Secondly, I am also very grateful to Peter Sanders for hosting me at Universität Karlsruhe during four months in 2007. In particular, I thank him for providing an inspiring an relaxed research environment. In this context, I thank Johannes Singler, with whom I worked directly, for all the enriching discussions.

Thirdly, many thanks are due to the secretaries and to the people from the computing lab for their valuable work and for being always very nice and accessible. A special mention goes to Mercè Juan for her help with the bureaucracy in these very last steps to become a PhD.

Fourthly, I am specially grateful to friends and colleagues who volunteered in helping me in proof-reading and giving advice for some parts of this document. These include: Roberto Asín, Pere Comas, Cristina España, Edgar Gonzàlez, Meritxell Gonzàlez and Christoph Raupach.

Daily life in the Omega building also leaves some very special memories. In this context, I thank the previous and current inhabitants of the s109 office for all the good moments and, specially, for their patience with me in these last months. Besides, I will definitely miss the daily coffee with the inhabitants of the s107, my *second* office. I really want to thank them for all the laughs, volleyball matches, *freakexcursion*s, etc., but above all, for their welcoming character.

Besides, I feel lucky for the good friends I have, and I thank them for their moral support during these years. In particular, I am specially grateful to Anna Farró, Alina García, Edgar Gonzàlez, Meritxell Gonzàlez, and Montse Manubens for all the talks about everything and nothing.

But nothing would have been possible without the infinite love of my parents: Eusebio Frias and Leonor Moya. They are proud of me and supportive no matter they might not understand some of my decisions. I am more than grateful to them for that and I will never be able to correspond them enough. Giving up was never an option because they would never do it for me or for my brother.

Finally, I am very grateful to those people who expressed their availability for being a member of my jury, namely, Ricardo Baeza Yates, Gerth S. Brodal, Jordi Cortadella, Joaquim Gabarró, Giuseppe F. Italiano, Conrado Martínez, Ulrich Meyer and Peter Sanders.

Funding. The research conforming this PhD dissertation was partially supported by the Spanish technology project ALINEX (*Algorithms: Engineering and Experiments*) with contract number TIN2005-05446 and the ALBCOM recognized research group with contract numbers CIRIT 2005SGR 00516 and 2009 SGR 1137. Besides, I was granted by the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund. Specifically, I held the pre-doctoral grant number 2005FI 00856 from 2005 to 2008, and the grant number 2006BE-2 00016 for a 4-month research stay in Universität Karlsruhe in 2007. I thank all these institutions for their support, as well as all the people involved in these projects for making them a reality.

Contents

Chapter 1

Introduction

Most software systems are built on the top of predefined components. In this way, software architects can concentrate on the high-level design aspects and, as a result, save development costs and time. Data structures libraries are an important kind of software components that define interfaces and implement fundamental data structures and algorithms. Nowadays, data structures librares are very accessible because they are included as part of most programming languages.

The *Standard Template Library (STL)* is the algorithmic core of the C++ standard library [50]. Its design principles include generic programming, abstractness without loss of efficiency, value semantics and the Random Access Machine (RAM) computation model. The key components of the STL are containers, iterators and algorithms. Containers consist of basic data structures such as lists, vectors, maps or sets. Iterators can be seen as high-level pointers that are used to access and traverse the elements in a container. Algorithms are basic operations such as sort, rotate or find. The specification of the STL components describes not only their external behavior but also their cost. From a theoretical point of view, the knowledge required to implement the STL is well laid down on basic textbooks on algorithms and data structures (see, e.g., [19, 69, 82, 104]). Indeed, most widely used STL implementations offered by compiler and library vendors are based on these.

Computing needs have varied over time together with the features of ever-evolving technology. But in the latest years, the changes in computer architecture have become more and more evident with respect to the RAM model of computation. This fact has motivated the definition of new, more realistic models of computation, as well as new algorithms and data structures for them.

For instance, as bigger, faster and cheaper secondary storage devices have become available, and as the gap between memory access time and arithmetic operation time has widen, the old RAM computation model has become more and more inaccurate. This fact has promoted the definition of IO and cache-conscious models, that is, algorithms and data structures that take into consideration the underlying memory hierarchy (see, e.g., [2, 21, 42, 58, 83]).

Besides, in the latest five years, the hardware industry has decidedly shifted to parallel chips, due to the current practical impossibility of providing more processing power by further increasing clock-rates (see, e.g., [76, Chapter 1]). In particular, most of today cheap laptop computers are *multi-core (multi)processors*, i.e., they have several independent pro-

cessors that share the memory system. As a result, designing general-purpose reusable parallel algorithms and data structures has become a must. A wide range of algorithms and data structures have been described under the Parallel Random Access Machine (PRAM) model (see, e.g., [52]). However, a PRAM bears low correspondence to a multi-core computer. Thus, adapting PRAM algorithms and data structures is not straightforward. On the other hand, some of the techniques that have been used in high-performance computing for big expensive parallel machines might be appropriate for multi-core computers.

Finally, it has become more and more important to efficiently manage textual data (i.e., strings) due to its growing applications in organizing the Web, in computational biology, in data mining, etc. Strings are represented as a sequence of symbols (digits, characters, ...) from a predetermined set or alphabet. Comparing strings lexicographically is a common operation, for instance when sorting and searching. Generic comparison-based data structures and algorithms can be used to solve these problems, but they make redundant symbol comparisons. In order to overcome this drawback, several algorithms and data structures have been defined *ad hoc* for strings. Specifically, tries [30] are the basis to implement efficient string dictionaries, and radixsort is their counterpart for string sorting. Moreover, comparison-based data structures and algorithms can be refined so that strings are efficiently compared (see, e.g., [45, 79]). Also, several cache-efficient algorithms and data structures for strings have been proposed (see, e.g., [7, 16, 48, 88]). Still, practical string algorithms and data structures for multi-core computers are scarce.

Anyhow, most common data structures libraries are still a step behind these novelties. Some exceptions are: Boost [13] (portable C++ libraries that extend the C++ Standard Library), the STL-XXL [22] (a STL implementation for big data sets), the MCSTL [86] (a parallel STL implementation for multi-core machines), or the STAPL [4] (a concurrent implementation of STL components).

Contributions of this thesis

This doctoral dissertation is aimed at building bridges between theory and practice in algorithmics. Overall, techniques from analysis and design of algorithms and data structures, high performance computing, experimental algorithmics, and algorithm engineering are used. The focus is on data structures libraries, because it is an effective way to transparently provide novelties. In particular, we consider the STL because it is standard and it is widely spread. This approach was also followed in my master thesis. See [31] for the complete version and [32] for a reduced version presented in ALENEX. That piece of work dealt with extending STL dictionaries with rank operations, and it can be considered as the starting point of my doctoral dissertation.

The contributions of this thesis concern three (non exclusive) areas: cache-conscious algorithms and data structures, parallel algorithms for multi-core computers and specialized algorithms and data structures for strings keys. Chapter 2 gives some essential background on these topics. It also gives an overview on the STL.

All the algorithms and data structures proposed in this doctoral dissertation have been implemented (in C++) and are available. Appendix A gives pointers to the actual implementations.

The following seven chapters are devoted to the contributions of this thesis, including some additional background for each of the problems.

Chapter 3 presents cache-conscious implementations of STL `lists`. The main issue is combining cache-efficiency with STL `lists` constraints, in particular, constant cost of operations and iterator validity under modifications. On the one hand, cache-efficient lists take advantage of memory hierarchies by packing elements together (typically, using arrays) so that the physical and logical order of elements is roughly the same. But this design enters in conflict with STL `lists` constraints. On the other hand, usual implementations of STL `lists` based on doubly linked lists can cope easily with STL `lists` constraints, but at the price of a poor usage of the cache memory. By contrast, we describe and analyze in detail three cache-conscious but standard-compliant implementations that combine features of both approaches. The experimental results show significant improvements in practice for common operations.

The results on this chapter appeared in the following publications:

- [36] L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. In C. Àlvarez and M. J. Serna, editors, *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*, pages 121–133, Berlin, Heidelberg, May 2006. Springer.

- [37] L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. *Journal of Experimental Algorithmics*, 14:3.5–3.27, 2009.

From that point on, we move on problems related with dictionaries, sorting and selection. We first present the analysis of several efficient string algorithms and data structures. Some of them are used later in this thesis.

Chapter 4 presents an analytical work on the number of string lookups in binary search trees (BSTs) and related algorithms with enhanced string comparisons. Enhancing string comparisons in comparison-based algorithms and data structure is achieved by taking advantage of the order and outcome of previous comparisons, as well as on the digital structure of strings. We highlight that, in some cases, this information suffices to decide the outcome of a comparison. Under the common assumption of a string implementation using pointers, this is relevant from a cache-conscious perspective. We analyze so-augmented BSTs using their relationship to ternary search trees (TSTs), which is a kind of trie representation. Moreover, we add some redundancy on the top of these BSTs to avoid the string lookups due to binary searching for a character position. We also analytically quantify the benefits of this approach. These techniques for strings, when applied on the top of a balanced BST, can be used to specialize STL dictionaries for strings. Furthermore, we extend our analysis for quicksort and quickselect, which in turn can be used to specialize STL `sort` and `nth_element`, respectively.

Most of the results on this chapter are included in the following research report:

- [33] L. Frias. On the number of string lookups in BSTs (and related algorithms) with digital access. Technical report LSI-09-14-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, 2009.

Chapter 5 presents *multikey quickselect*, which is the analog of multikey quicksort for the selection problem. First, we present our algorithm, which combines partitioning, as in quickselect, with digital access to strings. The result is an in-place, easy to implement algorithm that can be used to specialize `nth_element` for strings. We analyze the expected number of comparisons and swaps considering a random uniform distribution. The results lead us to introduce variants of multikey quickselect that reduce the expected number of comparisons and swaps. Some of the enhancements presented in this chapter can be applied to multikey quicksort as well.

The results on this chapter are included in the following research report:

- [38] L. Frias and S. Roura. Multikey Quickselect. Technical report LSI-09-27-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, 2009.

We also present several practical parallel algorithms for multi-core computers.

Chapters 6 and 7 present a collaborative work with the *ITI Sanders* research group at *Universität Karlsruhe*. This work was done in the context of the MCSTL, a parallel STL implementation for multi-core machines. Chapter 6 considers the parallelization of some bulk operations for red-black trees. Red-black trees are a kind of balanced binary search trees and are typically used to implement STL dictionaries. The considered operations are the construction from many elements, and the insertion of many elements at a time. We describe and analyze the algorithms. The construction from many elements shares some similarities with existing algorithms under the PRAM model. The insertion of many elements uses split and union operations on red-black trees, plus dynamic load-balancing techniques. We also present the results of a thorough experimental study, which indicates that great speedups can be obtained.

The results on this chapter appeared in the following publication:

- [39] L. Frias and J. Singler. Parallelization of Bulk Operations for STL Dictionaries. In L. Bougé, M. Forsell, J. L. Träff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, editors, *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007, Rennes, France, August 28-31, 2007, Revised Selected Papers*, volume 4854, pages 49–58, Berlin, Heidelberg, March 2008. Springer.

Chapter 7 presents a general technique to partition sequences of unknown size so that, later, they can be efficiently processed in parallel. In particular, this technique has been used to partition the input sequences of bulk operations in Chapter 6. We describe and analyze the properties of the proposed algorithm. It is online (i.e., it uses only one traversal) and it requires sublinear additional space. Experiments show the practicability of this approach in parallelization.

The results on this chapter appeared in the following publications:

- [40] L. Frias, J. Singler, and P. Sanders. Single-Pass List Partitioning. In *The Second International Conference on Complex, Intelligent and Software Intensive Systems*, pages 817–821. IEEE Computer Society Press, March 2008.

- [41] L. Frias, J. Singler, and P. Sanders. Single-pass list partitioning. *Scalable Computing: Practice and Experience*, 9(3):179–184, 2008.

Chapter 8 and 9 deal with the problem of partitioning an array in parallel. Partitioning an array is a basic building block of quicksort and quickselect. Besides, it is typically used in the implementation of STL `partition`, `nth_element` and `sort`. Chapter 8 considers the parallel partition of an array of generic keys. We review some existing practical parallel partitioning algorithms for multi-core architectures. They consist of a main parallel step and a cleanup step. We highlight that, in all cases, some elements might need to be compared again during cleanup. We describe a novel enhanced cleanup parallel algorithm to achieve one comparison per element, which is optimal. This cleanup is applied to the aforementioned partitioning algorithms, and then, it is experimentally evaluated. Unfortunately, the experiments show that the new cleanup algorithm is not especially worth in terms of time performance. Indeed, the improvement when measuring the total number of operations is small. But the properties of our cleanup algorithm regarding comparisons turn out to be crucial for the algorithms in Chapter 9.

The results on this chapter appeared in the following publication:

- [34] L. Frias and J. Petit. Parallel partition revisited. In C. C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 142–153, Berlin, Heidelberg, 2008. Springer.

Chapter 9 considers parallel partition for string keys. We describe how to enhance string comparisons in parallel partitioning-based algorithms, namely quicksort and quickselect, to avoid redundant character comparisons. This is achieved by means of the techniques analyzed in Chapter 4. To the best of our knowledge, parallel algorithms are a novel application of these techniques for strings. The main issue in the case of parallel quicksort and quickselect is that each partitioning step must compare each element at most once. This is achieved using the cleanup algorithms presented in Chapter 8. The resulting implementations can be used to specialize STL `nth_element` and `sort` for strings. Moreover, the experiments indicate its practicability.

The results on this chapter appeared in the following publication:

- [35] L. Frias and J. Petit. Combining digital access and parallel partition for quicksort and quickselect. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 33–40, Washington, DC, USA, 2009. IEEE Computer Society.

Finally, Chapter 10 sums up the contributions in this thesis and highlights some further future work.

Chapter 2

Preliminaries

The *Random Access Machine (RAM)* model [3] is a very common model for analyzing algorithms and data structures. This model essentially describes a 1-processor, 1-level memory computer. Nonetheless, most of modern computers have a memory hierarchy and several parallel processing units. Indeed, some problems can be solved much more efficiently in practice if these capabilities are taken into account. Not surprisingly, new computation models have been devised so that algorithms and data structures can be described and analyzed more accurately with respect to these features. Section 2.1 gives a background on algorithmic design conscious of memory hierarchies. The aim is to maximize the benefit from memory hierarchies, but cache memories always come into play. By contrast, taking advantage of several processing units to solve one task can only be done explicitly. Section 2.2 gives a background on parallel algorithmic design.

Furthermore, many comparison-based problems can be solved more efficiently taking into account peculiarities of the data type of the elements that are being compared. Section 2.3 gives an overview of existing efficient comparison-based algorithms and data structures for string elements.

In order to ease the delivery of these specialized algorithms and data structures, they can be included in data structure libraries. We consider the Standard Template Library (STL) of the C++ programming language. Section 2.4 gives an overview of the STL, including its implementations.

2.1 Cache-conscious algorithms and data structures

Memory hierarchies were introduced by computer architects to minimize the gap between memory access time and arithmetic operation time. See e.g., [76] for a detailed description on possible architectures. Each memory level may store copies of the actual data, being the access time and the size of the levels inversely proportional. Whenever the processor needs to access a memory location, it starts checking from the most to the less internal level (with respect to the processing unit).

Cache hierarchies conform the most internal levels of a memory hierarchy, namely those between the processing unit(s) and the main memory. Whenever the processor tries to access a memory location in a cache level, a *cache hit* occurs if the memory location exists; and a *cache miss* occurs, otherwise.

The benefits of memory hierarchies rely on the locality of data and programs. However, locality is not an universal property of data and programs. This has motivated looking for alternative data structures that organize data in such a way that logical access patterns are related to physically near memory locations. Several models have been devised to describe and analyze these data structures and algorithms.

The *external memory* or I/O model is a 2-level memory model introduced by Aggarwal and Vitter [2] to capture the existence of a fast internal memory and a slow external disk storage. Transfers between levels are done in blocks of consecutive elements. The internal level (main memory) has fixed size and is fully associative. Fully associative means that blocks can be mapped anywhere in memory. The external level (disk) is considered to be infinite and is explicitly managed. In this model, the cost measure of an algorithm is the number of disk accesses (the computation cost is neglected).

Similarly, *cache-conscious* models have been described to capture the existence of cache hierarchies that mitigate the access time to main memory. Cache hierarchies are mainly characterized by the number of levels and the size of each of them. Similarly to I/O, transfers between levels are done in blocks of consecutive elements. But by contrast, the cost of computation is not negligible and memory is not managed explicitly.

There are two main flavors of cache-conscious models, depending on whether they rely on specific cache hierarchy parameters or not. *Cache-aware* algorithms and data structures (see e.g., [58, 83]) are explicitly described using cache parameters and so, the resulting algorithms are not portable. Instead, *cache-oblivious* algorithms (see e.g., [21, 42]) use the parameters in the analysis but not in the actual algorithm description. They are very attractive from a theoretical point of view, but unfortunately they have some limitations: on the one hand, most cache-oblivious algorithms and data structures are not so efficient in practice as their cache-aware counterparts. In addition, for some problems it seems unlikely to find a reasonable cache-oblivious solution. Finally, there are problems for which an asymptotic gap has been proved between a cache-aware and a cache-oblivious solution (see e.g., [15]).

In the following, the cache-oblivious model is presented a bit more in detail, together with the basic algorithmic building blocks.

Cache-oblivious algorithms and data structures

The cache-oblivious model is a 2-level memory model introduced by Frigo et al. [42] in 1999. The internal level (cache memory) has fixed size M and is fully associative. The external level (main memory) is considered to be infinite and it is not managed explicitly. Besides, there is optimal cache replacement, i.e., whenever a cache block is evicted, it is chosen such that the total number of evicted blocks is minimal. Optionally, the *tall-cache assumption* might be added to the model. This assumption considers that the number of blocks fitting in the cache is larger than the block size B (which, to the best of our knowledge, is the case of every current cache configuration).

In this model, the cost measure of an algorithm is two-fold. Let n be the size of the input. Both the number of computational steps and the number of accesses to main memory are considered as a function of n. The main strength of the model is that provides reasoning in two levels, whilst the results are asymptotically valid for any number of cache levels and

more realistic cache configurations (namely, not fully associative).

For instance, consider the problem of computing an aggregate of a sequence. In the RAM model, this problem can be solved using $\Theta(n)$ computational steps, as long as finding the next element and the aggregate function have cost $\Theta(1)$. In the I/O model or cache-aware model, this can be done in $\lceil n/B \rceil$ memory accesses if the elements are stored in $\lceil n/B \rceil$ blocks of size B. In the cache-oblivious model, a similar bound can be obtained as follows: store the elements in any order in an array so that all elements are consecutive in memory; traverse the array in storage order. Doing so, at most $\lceil n/B \rceil + 1$ main memory accesses are done. Note that in contrast to the solution for the I/O model, the array data is not required to be aligned and no information about the cache parameters is used.

In order to solve more complex problems, more elaborated data structures are needed. Two popular building blocks for cache-oblivious data structures and algorithms are the van Emde Boas Layout (static structure) and the packed-memory structure (dynamic structure). Both were introduced by Bender et al. [6] to implement cache-oblivious search trees.

The *van Emde Boas Layout* is a cache-oblivious layout for trees that resembles the van Emde Boas data structure [101, 102]. It is is defined as follows. Split the tree at the middle level of edges, resulting in one *top recursive subtree* and roughly $\sqrt{n}$ *bottom recursive subtrees*, each of size roughly $\sqrt{n}$. Recursively, lay out the top recursive subtree, followed by each of the bottom recursive subtrees. Each recursive subtree is laid out in a single segment of memory, and these segments are stored together without gaps. Searching in a complete binary search tree with the van Emde Boas Layout, needs $O(\log_B n)$ memory accesses and $O(\log n)$ comparisons.

The *packed-memory structure* maintains an ordered sequence of elements in almost consecutive memory. It is an adaptation of the *ordered-file maintenance* problem solution in [51, 105]. The main idea is to distribute gaps in an array so that insertion and deletion takes $o(n)$ time but scanning needs only $\Theta(n/B)$ memory accesses. Specifically, the array is conceptually partitioned in blocks of size $\Theta(\log n)$. In each block, elements are kept contiguously, and a constant number of gaps is left between two blocks. Each of these blocks is the leaf of a conceptual tree. Each node of such a tree must keep with density constraints that are stronger as further up the tree. Inserting or deleting an element in a packed-memory structure takes $O(\log^2 n)$ amortized moves and $O(\log^2 n/B)$ amortized memory accesses

The packed-memory structure can also be used to implement a cache-oblivious list. Besides, splitting the array into smaller subarrays, the bounds for insertion and deletion can be improved. These bounds can be lowered until amortized constant if updates breaking the uniformity are allowed until the structure is reorganized when traversed.

Using a similar approach, we present standard-compliant cache-conscious STL `lists` in Chapter 3. In particular, our implementations achieve constant costs for insertion and deletion operations.

2.2 Parallel algorithms and data structures

Parallel computers provide several processing units that run in parallel. Additionally, communication and synchronization mechanisms are provided so that processing tasks can interact with each other. For an overview on parallel architectures, see e.g., [76].

Operating systems and libraries provide high-level abstractions to facilitate programming. For instance, POSIX threads [49] are a widely used abstraction of a processing task. They are commonly used as the base for higher level abstractions.

A parallel algorithm specifies how to solve a computational problem using several threads simultaneously. To do so, it is crucial to find a decomposition of the computation into *task*s. Tasks may interact with others, and may depend on finalization of other tasks. Tasks are mapped to threads and in turn, to processors. The aim is to maximize resource usage while minimizing (explicit and implicit) interaction between threads. In most cases, these are contradicting objectives, so a balance must be found. For an overview on parallel design techniques, see e.g [57].

Parallel data structures can be obtained in mainly two (not exclusive) ways. On the one hand, operations on data structures can be implemented in parallel. For instance, in Chapter 6 we consider the insertion of many elements at a time in STL dictionaries. On the other hand, data structures can provide mechanisms so that the data structure can be externally parallelized by executing concurrent, non-blocking operations. For instance, the STAPL library (see Section 2.4.3) offers such insertion operations for dictionaries.

The performance of a parallel program is measured by the parallel *speedup*. It is measured dividing the execution time of the best sequential implementation by the execution time of the parallel implementation. The maximum achievable speedup is limited by the portion of computation that is inherently sequential, i.e., tasks that cannot be divided further in concurrent tasks. *Amdahl's Law* states this relationship (see e.g., [76]). Indeed, the initial decomposition into parallel tasks (parallel setup) is in most cases inherently sequential. Specifically, it commonly deals with dividing the input into independent pieces to which essentially the same processing is applied (*data parallelism*). Doing so efficiently for no trivial inputs is tackled in Chapter 7.

Until recently, the practicability of parallel computation had been constrained to *high-performance computing*, that is, to solving highly computationally-intensive (typically numerical kind) problems. First, writing parallel programs was and is more involved and error prone than writing sequential problems due to concurrency. For an overview of concurrency and concurrency hazards, see e.g., [62]. Second, in most existing parallel machines the cost of synchronization and communication did not pay off for small or even medium-size computations. Indeed, finding the best parameters for a (parallel) program is a research issue in itself (see e.g., [81]). Third, fast improvements on clock-rates frequency during the last three decades made the parallelization efforts for general purpose software not attractive.

But in the last 4-5 years, hardware industry has had to shift to parallel chips to obtain more processing power significatively. Augmenting further the clock-rate was not a possibility anymore because of the resulting power consumption and the need for more and more expensive cooling systems. This phenomenon is known as the power wall (see e.g., [76, Chapter 1]).

These new parallel chips are mostly *multi-core (multi)processors*, which combine several independent processors (so-called *cores*) into a single socket. Each core alone may have several levels of cache memory. Additionally, all cores in a socket may share a higher level of cache memory. Besides, several sockets can be combined into a single chip so that all of them share the interconnection to main memory. Thus, a multi-core processor has a shared hierarchical memory structure.

The proliferation of multi-core computers has renewed the interest on parallel algorithms and data structures. Existing parallel algorithmic techniques also apply for multi-core computers. Nonetheless, the range of problems to be solved is much broader because parallel processing units have become cheap and available. Existing tools and abstractions to ease parallel programming have gone strong. For instance, Cilk [11] and OpenMP[72]. Besides, some new abstractions like *Intel Threading Building Blocks* [78] have been defined. All these tools completely abstract from thread management and their later mapping to processors. In particular, we have used OpenMP for the parallel implementations in this thesis. The *OpenMP API* consists of a set of compiler directives, library routines, and environment variables that support multi-platform shared-memory parallel programming in a portable, scalable and flexible way.

Besides, some efforts to automate parallel programming have been done, but in most cases they are limited to highly structured programs. A popular and simple way to benefit from multi-core processors is to provide parallel implementations of data structures libraries. In particular, several efforts have been devoted to parallelize the STL of the C++ programming language. Section 2.4.3 gives some examples of parallel STL-like libraries. In Chapters 8 and 9, we present parallel implementations for STL `partition`.

On the other hand, new parallel computational models have been defined. During the 80s and the 90s, the PRAM was the reference machine model for describing parallel algorithms and data structures. The PRAM model extends the RAM model with an arbitrarily large number of parallel processors, which share the memory and proceed synchronously. For an overview of algorithms under the PRAM model, see e.g., [52]. However, the PRAM bears low correspondence to a multi-core computer. By contrast, recent models like the *Multi-BSP* [99] by Valiant mirror multi-core computers. This is a hierarchical model (from the perspective of both processing and memory units). An arbitrary number of levels is considered; in each one, the number of processing components, the bandwidth, the synchronization cost and the memory size is defined. Also, the latest edition of an algorithmics book such as Cormen et al. [19] does not use the PRAM model anymore but a more realistic model with respect to current parallel computers.

2.3 Algorithms and data structures for strings

Strings, a common representation of textual data, are a sequence of symbols from a predetermined set or alphabet (digits, characters, ...). We consider the following common pointer representation: string data itself is stored in an array, and this array is accessed through a pointer. This is the case of C like strings, i.e., `char*` whose end is indicated with zero, and C++ `string`s, i.e., a pointer to an array plus the array size. `string` implementation adds namely a level of indirection with respect to rough `char*`, but in return, it can offer more advanced features.

Comparing strings lexicographically is a common operation, for instance, when sorting and searching. Generic comparison-based data structures and algorithms can be used to solve these problems. They consider keys as atomic to make comparisons. This is simple, and in many cases leads to a good performance. However, many redundant symbol comparisons might be performed, in particular if the strings share long common prefixes. This source of inefficiency can be avoided taking into consideration the digital structure of

strings.

To compare strings efficiently, *ad hoc* string algorithms and data structures have been defined. Many of them are based on the same principle as a thumb index on an encyclopedia: from the first letter of a given word, the pages that contain all the words beginning with that letter can be immediately located. If this idea is pursued, a searching scheme based on repeated subscripting is achieved, which is optimal in the number of symbol comparisons. Section 2.3.1 gives an overview of string data structures and algorithms following this principle. We focus on string dictionaries, and the sorting and selection problems. The description of the data structures and algorithms is first done in the RAM model of computation. Then, Section 2.3.2 discusses how to build cache-conscious string data structures and algorithms from them.

The behavior of *ad hoc* string algorithms and data structures greatly depends on the characteristics of the dataset (string length, how skewed are the strings,...). In many common cases, that leads to a better (or at least good) performance, but there is no guarantee. Besides, many of these algorithms and data structures need to know the specific value of the alphabet cardinality. A more robust approach consists in specializing comparison-based data structures and algorithms so that no redundant digit comparisons are made, whilst keeping the rest of combinatorial properties (see Roura [79] and Grossi et al. [45, 28]). Comparisons in such specialized algorithms and data structures take advantage of the digital structure of strings, and the order and the outcome of other comparisons. To efficiently reuse this information, some extra data must be kept per element (i.e., the data structures and algorithms are augmented). Figure 2.1(d) shows an example for a binary search tree (BST): each node additionally keeps the maximum common prefix of its string with its ancestors in the access path, which is shown in parenthesis. The resulting implementations are amenable, and the data structures and algorithms are competitive in practice. In particular, this approach is convenient for long and skewed strings. See e.g., [20] for an experimental study.

In this thesis, we explore further advantages of specializing comparison-based data structures and algorithms for strings. Chapter 4 deals with the analysis of some properties related to cache-efficiency. Chapter 9 combines these specializations with parallelization. Besides, our proposals are aimed to be generic with respect to the string type. Indeed, most of existing C-C++ implementations of efficient string algorithms and data structures are keyed for C like strings (actually, in most cases they are pure C implementations).

2.3.1 *Ad hoc* string algorithms and data structures

A *trie* [30] (name that comes from re*trie*val) is an efficient dictionary data structure for strings. A trie is a σ-ary tree, where σ denotes the cardinality of the alphabet (tries are aware of the value of σ). Each trie node at depth ℓ represents the set of strings that begin with a certain prefix of length ℓ. A search branches in the tree until the searched string is completely matched or a leaf is reached. Thus, successfully searching a string of length d requires $\Theta(d)$ symbol comparisons, which is optimal. Besides, for a given set of strings, there is only one associated trie. Figure 2.1 (a) shows an example of trie. Trie nodes are typically implemented with an array of pointers with null pointers indicating not existing branches. This is particularly efficient for small alphabets, e.g., the English alphabet.

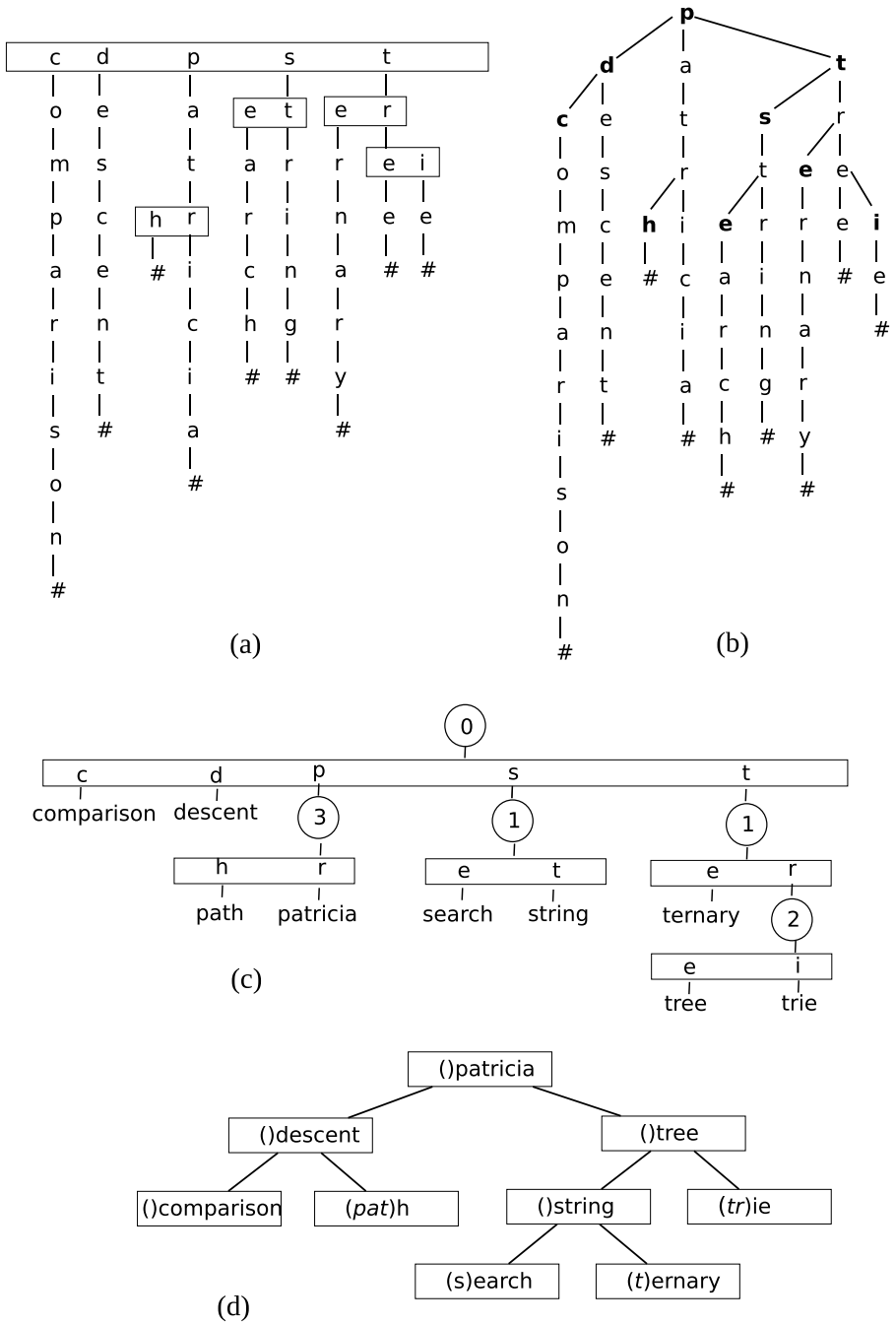


Figure 2.1: Examples of string data structures for dictionaries. (a) Trie. (b) TST. (c) Patricia trie. (d) Specialized BST.

If the alphabet cardinality is big (e.g., the Unicode alphabet), implementing trie nodes with arrays is not efficient. An elegant alternative are *ternary search trees* [9] (TSTs). TSTs are tries in which BSTs are used to guide descent for each character. Figure 2.1 (b) shows an example of a TST. An internal TST node contains two possibly null *comparison pointers* (shown with a diagonal line) and a not null *descent pointer* (shown with a vertical line). A leaf contains a special character (shown with #). For each character position in the searched string, several comparison pointers might be followed (binary searching in their values) until the matching node or a null pointer is found. In case of success in the current character, the descent pointer is followed. All the nodes for a given character position and prefix can be placed in several ways. The actual TST shape depends on insertion order and whether balancing techniques are applied. In a not balanced TST that stores n string keys, a search needs $\Theta(n)$ symbol comparisons in the worst-case. But if the TST is kept balanced, only $\Theta(d + \log n)$ symbol comparisons are needed. The main advantage of TSTs is their generality. In particular, no information on the alphabet is needed. However, as in plain tries, much more nodes than keys are needed. This causes a poor usage of memory.

Patricia tries [71](name that comes from *P*ractical *a*lgorithm *t*o *r*etrieve *i*nformation *c*oded *i*n *a*lphanumeric) reduce the total number of nodes to $\Theta(n)$. Patricia tries are tries in which all nodes that only have one possible follow-up (i.e., one not null pointer) are compacted. Thus, they are especially suitable for extremely long, variable-length keys with common prefixes. The compaction is done including the number of symbols to skip over before making the next comparison, and placing the keys in the leaves. An example of a Patricia trie is shown in Figure 2.1 (c). A search in a Patricia trie examines several nodes but only one string (at the leaf). Thus, $\Theta(d)$ symbol comparisons are needed.

For a detailed analysis of trie-like data structures, see e.g., [17, 24].

Furthermore, there are several sorting algorithms that follow the same main ideas as tries. Specifically, Most Significant Digit (MSD) radixsort is a sorting algorithm isomorph to constructing a trie. *Radixsort* (see e.g., [56]) is based on recursively distributing the elements into groups w.r.t the current character position. MSD radixsort needs $O(dn)$ symbol comparisons, where d is the maximum string length. The main issue in implementing radixsort is dealing efficiently with groups. There are many variants. One main distinction is whether one or two passes on the data are done. One-pass algorithms typically use an array of groups, in which each group is implemented with a dynamic data structure. Some of the groups might be empty. If there are many of them, it turns out in a considerable slowdown (apart from the waste of space). Radixsort can be implemented in-place by using an extra pass to previously determine the size of the groups before partitioning the array. There are several proposals to mitigate the drawbacks of both approaches, as well as hybrids. For further reading on the tuning of radixsort, see e.g., [5, 54, 67]. For further details on the analysis of radixsort, see e.g., [63].

Multikey quicksort [9] is the sorting algorithm isomorph to constructing a TST. According to [5], multikey quicksort can be seen as a variation of MSD radixsort with bucketing replaced by (three-way) quicksort. Multikey quicksort performs $\Theta((d + \log n)n)$ symbol comparisons using a randomly chosen pivot. It is simple to implement, in-place, needs no knowledge on the alphabet cardinality, and it has been shown to be relatively fast in practice (see e.g., [5]).

However, not much attention has been paid to adapting radixsort or alike to selection.

Selection consists in finding an element in a unsorted sequence of a given rank using a certain order function. As far as we know, only [63] analyzes a very generic radixselect algorithm. In this thesis, we contribute multikey quickselect algorithms. Section 5 describes and analyzes them.

2.3.2 Cache-efficiency considerations

The algorithms and data structures described in the previous subsection are not cache-conscious. Cache-conscious algorithms and data structures for strings can be obtained as follows. On the one hand, trie or alike building blocks can be combined in a cache-efficient way. The resulting data structures (externally) are not tries. On the other hand, trie alike data structures can be build using cache-efficient blocks. In the following, we describe some examples.

Several cache-conscious data structures and algorithms for strings use Patricia tries as a building block. Patricia tries are used because they perform only one string lookup per search. Patricia tries are used in IO efficient string dictionaries, namely *String B-tree*s [27], and their corresponding sorting algorithm (see e.g., [26]). Also, this approach has been used to build cache-oblivious string dictionaries in [7] and [16].

By contrast, *burst tries* [48] are cache-aware tries whose nodes are buckets. Buckets store a collection of strings that share a common prefix. Cache-efficiency is attained inside the bucket. Whenever a bucket does become too large, it *bursts*, i.e., it expands so that there is a new child node for each symbol. Buckets can be implemented in several ways. Burstsort [88, 89, 90, 87] is a sorting algorithm based on burst tries, where the buckets are kept unsorted until the strings are outputted. One of the variants uses multikey quicksort for sorting the buckets. Burstsort has been shown to be very fast in practice for common data sets.

In addition, cache-efficiency might be improved by means of redundancy or a *superalphabet*. Redundancy consists in storing some extra symbol(s) together with the string so that these are looked up in the first instance. The gain is obtained by avoiding looking up the string in some cases. A *superalphabet* consists in changing the base alphabet by considering groups of more than one symbol instead of one alone. These techniques can be used on their own. For instance, they can be used in combination with radixsort [54]. Besides, they can be combined with a cache-conscious design, as it is the case of some variants of burst tries.

2.4 The Standard Template Library

The C++ language specification [50] includes the definition of a standard library. This specification dates back to 1998. A new Standard (unofficially known as *C++0x*) has been in preparation for long but it is not finished yet. More details on the development of the new Standard can be found in [18].

The C++ standard library defines components to deal with input/output, numbers and strings; abstract data types and algorithms; support for the language itself (dynamic memory management, exceptions, ...); internationalization and other utilities (allocators, date and time, ...).

The *Standard Template Library (STL)* is a widely used subset of the C++ standard library comprising basic algorithms and data structures. Besides, it has had and has a great impact in the rest of the library (see e.g., [95]).

Section 2.4.1 describes the main STL components. Section 2.4.2 outlines the main issues on STL compliance. Finally, Section 2.4.3 gives an overview of existing implementations. This includes existing efforts to provide advanced algorithmic features in STL and alike C++ libraries.

2.4.1 STL components

The STL is based on the interaction of three key components: *containers*, *iterators* and *algorithms*. We describe them in the following. For examples on how to use STL components, see e.g., [53].

Containers

Containers store and manage collections of elements. Elements stored in a container are copies of the provided elements and their life depend on the life of its container. The following two types of containers are devised, depending on how the relative order between the elements is determined:

- *Sequences:* Sequences are linear collections of elements. The relative order of their elements is determined according to the time and place of insertion, but not on the element characteristics. The following sequences are provided: `vector`, `deque` and `list`. All of them provide sequential access to elements. In addition, `vector` and `deque` support efficient random access. Besides, given an iterator, elements can be inserted and deleted at any point. The cost depends on the modification point and the container. In particular, `lists` provide efficient insertion and deletion at any point, `deque` provide these operations efficiently only at the beginning and end of the sequence, and `vector` only at the end of the sequence. Common implementations for `deque` and `vector` use arrays. By contrast, `lists` are typically implemented using doubly linked lists.

  Further, the STL specifies three container adaptors that take sequences as default: `queue`, `stack` and `priority_queue`. Essentially, a container adaptor takes a container type and transforms it into another with restricted functionality. In particular, none of the previous adaptors allows iteration through its elements.

- *Associative containers:* Associative containers correspond with the notion of dictionary. The relative order of their elements is determined by its key and a comparison function. Associative containers support efficient search, insertion and deletion of elements by key. Besides, given an iterator to the structure, associative containers also offer efficient traversal by sorted key order.

  The following ordered associative containers are defined: `set`, `map`, `multiset` and `multimap`. They are differentiated according to the following two criteria. On the one hand, elements can be made up only by a key, which is the case of `set` types, or be a pair {key, value}, which is the case of `map` types. On the other hand, associative

containers can allow repeated keys (`multi` types) or not. Associative containers are usually implemented with balanced binary search trees.

Additionally, many library vendors offer associative containers which do not keep an order among their elements, and whose aim is optimizing query operations by key. They are typically implemented with a hash table. This kind of associative containers will be probably be included in the forthcoming standard.

Iterators

Iterators offer a container-independent interface to traverse the elements of a collection. In particular, they offer a dereference operation to access the element 'pointed to' by the iterator. If the iterator is mutable, the element can also be modified.

The main iterator types are: forward, bidirectional and random access. Forward iterators provide one-by-one forward access in a sequence of values. Additionally, bidirectional iterators provide one-by-one backward access. `list`, `(multi)map` and `(multi)set` offer bidirectional iterators. Furthermore, random access iterators provide constant time access to any element of the collection. This is done using iterator arithmetic (in analogy to pointer arithmetic). Specifically, a displacement can be added and subtracted, the difference between the position of two iterators can be calculated, or iterators can be compared using < and > relational operators. `vector` and `deque` offer random access iterators.

Algorithms

STL algorithms use iterators to process elements of a collection in a generic way. A collection is described by means of a begin iterator and an end iterator.

Algorithms in the STL can be classified in the following categories: non-modifying sequence operations (`for_each`, `find`, `count`...), modifying sequence operations (`copy`, `swap`, `partition`...), sorting-like (`sort`, `nth_element`..), binary search (`lower_bound`, `binary_search`,...), merge (`merge`, `set_union`...), heap (`push_heap`, `make_heap`...), and min/max (`min_element`, `next_permutation`...).

2.4.2 STL specification

The specification of STL components comprises pre and post conditions and complexity requirements. Besides, some general exception handling requirements are stated.

Time and space complexity requirements are commonly described using asymptotic notation. But in some cases, exactly a certain number of operations is required. The analysis is done under the RAM model of computation and considering elements as black boxes (i.e., the size of the input considers only the number of elements, not their length).

Error checking in the STL is minimal because efficiency has a higher priority. In general, if preconditions are violated, the behavior is unspecified (no exception is thrown). Nonetheless, objects that are managed by the STL can throw exceptions that if not caught, can reach the user program. Unfortunately, recovering from an exception is not always possible or, if it is, compromises efficiency. In this sense, the Standard establishes that the STL will not lose resources or violate the containers invariants when exceptions are thrown. Besides, in some cases, the Standard requires that if an exception occurs, the operation

has no effect. That is, the Standard requires these operations to be atomic with respect to exceptions. For further reading on exception handling, see e.g., [1, 94].

Finally, the library is not concerned about concurrency or parallelism. Specifically, it does not provide tools for building multi-threaded applications. But also the expected behavior of concurrently accessing a component is unspecified, i.e., it is implementation specific. By contrast, the new Standard constraints the expected behavior. Up to now, most of existing implementations follow the *thread-safety* criteria by the SGI STL [84]: query concurrent operations and concurrent operations performed over different instances are thread-safe. Thus, query operations must be reentrant and containers must not share data among instances.

2.4.3 STL implementations

The first public STL implementation was released by HP [92] in 1994, i.e., even before the Standard was published. The *SGI STL* was one of its successors. Some features were corrected, and new features were added, such as thread safety. Besides, it became and still is specially popular for its comprehensive web-based documentation.

Nowadays, most known C++ compilers implement their own STL. For instance, the *GNU Compiler Collection (GCC)* [43] offers an STL implementation together with the C++ compiler. GCC is is a free software project that also includes front ends for other programming languages like Java or Fortran. GCC's implementation of the STL is based on SGI's. In all the experiments in this thesis, we have used the GCC compiler.

Additionally, there are some STL implementations distributed unto themselves. In particular, Dinkum [60] and STLPort [93] put special emphasis on the conformance with the Standard. The *Dinkum STL* is a proprietary implementation by DinkumWare Ltd. Customized versions of Dinkum are distributed with Microsoft and IBM compilers. By contrast, the *STLPort* is freely available through SourceForge [91] and offers debugging support.

But the STL is not the only C++ data structure library. Boost [13] and the CPH-STL [25] libraries are strongly related to the STL. *Boost* provides STL-like extensions for graph, math and string algorithms, among others. In particular, the string library includes specific string searching algorithms, but does not include sorting or string containers. Boost libraries are free, peer-reviewed and portable. Besides, they are intended to be widely useful, to establish existing practice, and to provide reference implementations so that they are suitable for eventual standardization. The *CPH-STL* is a research project that includes alternative implementations and extensions to the STL. For instance, they include cache-conscious STL dictionaries based on B-trees (though without supporting iterators functionality). Lately, they have rather focused on alternative design approaches of the library and on defining stronger guarantees for some of the components.

Furthermore, *LEDA* [68] is a C++ library for algorithms and data structures that is completely independent from the STL. It has been specially popular in the research community, in which the project started. Among its features there are a comprehensive library for graphs, as well as mechanisms to choose among different implementations of a component.

Finally, some STL implementations are designed to take advantage of particularities of modern computer architectures. For instance, the *STL-XXL* [22] is an implementation of the STL for efficient external (out-of-core) memory computations.

Several STL or alike C++ libraries are aimed to ease parallel computation. The *Standard Template Adaptive Parallel Library (STAPL)* [4] is a parallel and concurrent C++ library. It provides a collection of parallel algorithms (*pAlgorithms*), parallel containers (*pContainers*), and 'iterators' for parallel containers (*views*). Besides, it provides a run-time system. It is designed for both shared and distributed-memory parallel computers. The focus is on providing a concurrent framework more than on the parallelizations themselves (i.e., their parallel algorithms are rather simple). For instance, *pContainers* provide mechanisms for efficient concurrent modifying operations. Unfortunately, there is no publicly release available.

By contrast, the *Multi-Core Standard Template Library (MCSTL)* [86] is a parallel STL implementation that focus on the parallelization of algorithms. Specifically, the MCSTL consider parallel algorithms for shared-memory multiprocessors, namely multi-core multiprocessors. Under this assumption, some techniques like fine-grained load-balancing can be efficiently implemented. Fine-grained load-balancing techniques are specially useful to adapt to changes in resources and tasks, as well as to accommodate to other forms of parallelism (multi-core computers are not typically used as dedicated machines, thus, several processes might run concurrently). Also, they scale down for small inputs, i.e., as the inputs become smaller, the number of used threads decreases. Scaling down makes sense under the assumption of a multi-core computer, but not in many other parallel platforms. The MCSTL is implemented using OpenMP 2.5 [73]. Furthermore, I collaborated with the MCSTL on the implementation of parallel bulk insertion operations for STL dictionaries (see Chapter 6) and a helper algorithm in parallelization for partitioning sequences whose size is unknown (see Chapter 7). Both pieces of code are included in the 0.8.0-beta version. Besides, the later piece of code is also included on the successor of the MCSTL, the *libstdc++ parallel mode*, which is included with the GCC compiler version 4.3 on. Nowadays, development is focused on the *libstdc++ parallel mode*. See [85] for further considerations on its design.

The *OpenMP Multi-Threaded Template Library* [74] has a similar aim and approach to those of the MCSTL, but the algorithms they use are less competitive.

Chapter 3

Lists revisited: cache-conscious STL `lists`

In this chapter, we present three standard-compliant cache-conscious implementations of STL `lists`. `list` is one of the most simple but essential objects in the STL. Doubly linked lists are typically used to implement them because they easily cope with STL `lists` requirements. However, they are not cache-efficient. On the other hand, the design of existing cache-conscious lists addresses rather different requirements than those in STL `lists`.

This chapter merges both approaches, paying special attention to iterators constraints. The remainder of this chapter is organized as follows. In Section 3.1, we present some basic facts on STL `lists`. Then, in Section 3.2, we consider the behavior of classic doubly linked list implementations with regard to the cache memory. In Section 3.3, we give a background on previous work and present the main design issues on our data structure. Then, in Section 3.4 we present an specific reorganization algorithm and in Section 3.5, further analyze it. Finally, in Section 3.6 we give the results of a thorough experimental evaluation. In particular, great speedups are shown with respect to doubly linked lists implementations. Section 3.7 sums up some conclusions.

3.1 STL `lists`

STL `lists` are sequences, which are a kind of container. See Section 2.4.1 for an overview of STL containers. As sequences, they provide insertion and deletion of elements at any point. In particular, `lists` provide all the variants of insertion and deletion efficiently, namely using linear time in the number of elements being inserted or deleted. Additionally, they provide efficient operations to sort, reverse or splice a list, also to merge two lists or make one list unique, among other minor utilities. Specifically, the standard requires that on a list with n elements, `sort()` takes $O(n \log n)$ and is stable, `splice()` takes only $O(1)$ time, and `reverse()` and `unique()` take $O(n)$ time. Besides, `size()` is allowed to take $O(n)$ time.

`list` provides bidirectional iterators, i.e., `lists` can be backward and forward traversed. An iterator is valid if the element that points to is a valid element of a collection. List iterators can only be invalidated, when the element they point to is deleted. Any other

operation does not affect their validity. Besides, the number of iterators per list and per element is unbounded.

Figure 3.1 illustrates how to use an STL `list` to solve the *Josephus problem*: Given a group of n men arranged in a circle under the edict that every k-th man will be executed going around the circle until only one remains, find the position in which one should stand in order to be the last survivor.

```
int survivor (int n, int k) {
    if (k==1) return n;
    list<int> L;
    for (int i = 1; i ≤ n; ++i) L.push_back(i);
    list<int>::iterator it = L.begin();
    for (int i = 1; i < n; ++i) {
        int c = (k − 1) % n;
        for (int j = 0; j < c; ++j) {
            ++it;
            if (it == L.end()) it = L.begin();
        }
        it = L.erase(it);
        if (it == L.end()) it = L.begin();
    }
    return *L.begin();
}
```

Figure 3.1: Using STL `lists` to solve the Josephus problem

3.2 Cache behavior of doubly linked lists

In order to implement a list fulfilling all the STL requirements, a classical doubly linked list suffices. Indeed, this is what all known STL implementations do. In this case, nodes store an element together with a pointer to the previous and the next node in the list. The size of the list is not stored in order to guarantee constant time `splice()` operations. Besides, `sort()` is typically implemented by bottom-up mergesort to guarantee stability and quasi-linear performance. Moreover, list iterators are just pointers to these nodes. Because of the existence of `begin()` and `end()` iterators, implementations include a fake node after the end of the list. Figure 3.2 depicts this classical solution and Figure 3.4 gives its simplified definition in C++.

Pointer-based data structures use *memory allocators* to allocate and deallocate their nodes. Once a node is allocated, its physical position remains unaffected until it is freed. The logical position of a node in the list, instead, can be changed by simply modifying the nodes to which it is linked. This is a key property of any pointer-based data structure. It also makes implementing iterators trivial: iterators are not affected by these logical movements and they always remain valid.

Allocators typically answer consecutive memory requests with consecutive addresses

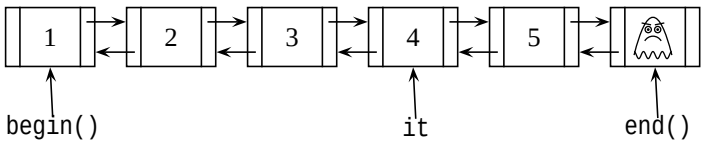


Figure 3.2: Classical doubly linked implementation.

of memory (whenever possible). This is particularly important in the case of `list`, because if elements are added at one end (and no other allocations are performed at the same time), there is a good chance that logically consecutive elements are also physically consecutive, which leads to a good cache performance when elements are traversed. However, if elements are inserted at random points or if the list is shuffled or if different lists are constructed, logically consecutive elements will rarely be at nearby physical locations. Therefore, a traversal may incur in a cache miss per access, thus dramatically increasing the execution time.

In order to give evidence of the above statement, we have performed the following experiment with the GCC `list` implementation: Given an empty list, n random integers are pushed back one by one. Then, we measure the time to fully traverse it. Afterwards, we sort the list (using the sort method of `list`) to randomly shuffle the links between nodes. Finally, we measure again the time to fully traverse it. Traversal times before and after sorting the list are shown in Figure 3.3. Except for very small lists, it can be seen that the traversal of the shuffled list is about ten times slower than the traversal of the original list; and the only difference can be in the memory layout (and so, in the number of cache misses).

Taking into account that `list` is used when elements are often reorganized (e.g., sorted) or inserted and deleted at arbitrary positions (e.g., the Josephus problem), the previous experiment shows that it is worth to try to improve the performance of `list` using a cache-conscious approach. Note that if a user only wished to perform operations at the ends, (s)he would better have selected `vector`, `stack`, `queue` or `deque` rather than `list`.

3.3 Design of cache-conscious STL `lists`

A good survey of previous work on cache-conscious lists can be found in [21]. That survey considers traversal (as a whole), insertion and deletion operations. Let be n the list size and B be the block size. The cache-aware solution consists of a sequence of $\Theta(n/B)$ pieces, each having between $B/2$ an B consecutive elements. In this way, constant cost in the number of memory accesses is achieved for updates and $O(n/B)$ for traversals. The cache-oblivious solutions are based on the packed memory structure (see Section 2.1). Specifically, updates require $O(\log^2 n/B)$ memory accesses using its basic form. By allowing more flexibility on the data structure, and then, reorganizing it when traversing, updates only require an amortized constant number of memory accesses. However, this approach is not convenient in the case of STL `lists` because they are not traversed as a whole but step by step via iterators.

Therefore, theory shows that cache-conscious lists fasten scan based operations and hopefully, do not significantly rise update costs compared to traditional doubly linked lists.

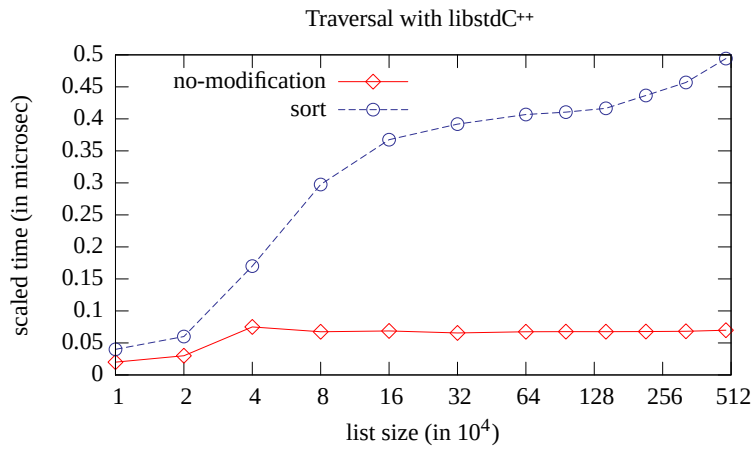


Figure 3.3: Time measurements for list traversal before and after shuffling it. The vertical axis is scaled to the list size.

However, none of the previous designs take into account common requirements of software libraries. In particular, combining iterator requirements and cache consciousness rules out some of the most attractive choices.

In this section, we describe precisely the challenges that poses keeping with list iterator requirements. Besides, we present some hypotheses that have guided some of our choices. Finally, we present the key ideas of our approach.

3.3.1 Iterator concerns and hypotheses on common `list` usages

In cache-conscious data structures, logical and physical locations of elements are heavily related to take advantage of cache hierarchies. As a consequence, it is difficult to implement iterators trivially with pointers, because these should be kept coherent whenever a modification in the list affects the position of the pointed element. In fact, the main issue here is that an unbounded number of iterators can point to the same element. Besides, any arbitrary restriction on its number would make the solution non standard compliant.

In order to overcome this difficulty, our design has been guided by some hypotheses on common `list` usages. In particular, from our experience as STL users, we can state that many uses of `list` are in keeping with the following:

1. The list is often modified at any position: insertions, deletions, splices.
2. A particular list instance has typically only a few iterators on it.
3. Many traversals are expected.
4. The stored elements are not very big (e.g., integers, doubles, ...).

Let us briefly justify these points: Point 1 applies because if there were no updates at arbitrary points, the programmer would have selected another container rather than `list`

(`stack`, `deque`, `vector`, ...). Point 2 raises from the observation that iterators are mainly declared as local variables to perform traversals or update the list. Truly, one can conceive creating arrays of (not initialized) iterators, but this is very rare. Point 3 is a direct consequence of Point 2 and the fact that list elements can only be accessed through iterators, which provide sequential access. Finally, Point 4 is a common assumption in the design of cache-conscious structures that also appears implicitly or explicitly in general cache-conscious data structures literature. Since this last condition can be checked at compilation time, a traditional implementation could be used instead and this could neatly be achieved with template specialization.

3.3.2 Our data structure

In the following, we present the main characteristics of our data structure. This data structure will prove to be specially convenient when the hypotheses on common `list` usages hold, but, in any case, it shall always be compliant with the costs required by the STL even when the above conditions do not apply.

The core of the data structure is inspired by the cache-aware solution previously mentioned in Section 3.3. Specifically, it consists in a doubly linked list of buckets. A bucket contains an small array of K elements, pointers to the next and previous buckets, and extra fields to manage the data in the array. This simple data structure ensures locality inside the bucket, but logically consecutive buckets are let to be physically far.

From now on, K will denote the *capacity* of the buckets, that is, the number of elements a bucket can hold. Moreover, we will say that the *occupancy* of a bucket is its number of elements divided by its capacity.

In addition, we must deal with the following issues:

1. *Capacity of a bucket.* The capacity of a bucket has been fixed according to the outcome of the experiments (see Section 3.6.1).

2. *Elements arrangement inside a bucket.* We devise three possible ways to arrange the elements inside a bucket:

   (a) *Contiguous*: The elements are stored contiguously from left to right. Consequently, insertions and deletions must shift some elements towards their left or their right.

   (b) *With gaps*: Elements are stored from left to right, but gaps between elements are allowed. Gaps though do not improve the worst-case with respect to the number of shifts. Besides, an extra bit per element is needed to distinguish real elements from gaps.

   (c) *Linked*: The order of the elements inside the bucket is set by internal links rather than the implicit left to right order. This requires extra space for the links, but avoids shifting inside the bucket. Thus, this solution is more scalable.

   These three possibilities are depicted on Figure 3.7 and Figure 3.5 gives their simplified definition in C++.

```
struct node {
    node* next;
    node* prev;
    elem x;
};

struct iterator {
    node* p;
};
```

Figure 3.4: C++ definition for a doubly linked list implementation.

```
struct bucket {
    bucket* next;
    bucket* prev;
    elem[K] a;
    int beg;
    int end;
};
```

(a) Contiguous

```
struct bucket {
    bucket* next;
    bucket* prev;
    elem[K] a;
    bool[K] occupied;
};
```

(b) With gaps

```
struct bucket {
    bucket* next;
    bucket* prev;
    elem[K] a;
    int[K] next;
    int[K] prev;
};
```

(c) Linked

Figure 3.5: C++ definition for bucket arrangements.

```
struct bucket {
    bucket* next;
    bucket* prev;
    pair<elem,relay*>[K] a;
};

struct relay {
    bucket* b;
    int index;
    int count;
};

struct iterator {
    relay* r;
};
```

(a) Bucket of pairs

```
struct bucket {
    bucket* next;
    bucket* prev;
    elem[K] a;
};

struct relay {
    relay* next;
    relay* prev;
    bucket* b;
    int index;
    int count;
};

struct iterator {
    relay* r;
};
```

(b) 2-level list

Figure 3.6: C++ definition for iterator implementations (obviating arrangement details).

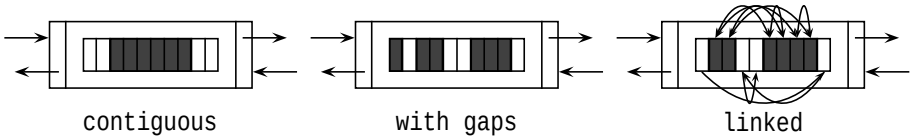


Figure 3.7: Bucket arrangement.

3. *Bucket reorganization when inserting or deleting elements.* The algorithms involved in the reorganization of buckets preserve the data structure invariant after an insertion or deletion. Our actual invariant will be given latter, in the next section. The main issue here is keeping a good balance between high occupancy, few bucket accesses per operation, and few of elements movements.

4. *Iterators management.* Our key idea is identifying all the iterators referred to an element with a dynamic node called *relay* that points to it. Thus, there are two levels of indirection: iterators pointing to a relay, and relays pointing to elements.

   The pointer from a relay to an element is actually a pointer to its bucket and its index in that bucket. In this way, the rest of bucket data can also be accessed. Besides, it keeps count of the number of iterators that are referring to that relay so that it can be destroyed when there are none. A problem remains: when the physical location of an element changes, its relay (if it exists) must be reached and changed accordingly in constant time. We devise two approaches for that:

   (a) *Bucket of pairs*: This solution is depicted in Figure 3.8 and Figure 3.6(a) gives its simplified definition in C++ (arrangement details are obviated here).

   In this obvious solution, for each element, a pointer to its relay is kept. This technique is easy to implement and still uses less space than a traditional doubly linked list because it needs one pointer per element rather than two.

   (b) *2-level*: This solution is depicted in Figure 3.9 and Figure 3.6(b) gives its simplified definition in C++ (arrangement details are obviated here).

   The key point is taking advantage of the fact that STL `list` elements can only be accessed by library users through iterators. Let *it* denote an iterator given as operation parameter, *r* the relay to which *it* points and *elem* the element to which *r* points; our reorganization algorithm guarantees that if the location of another element *other* changes due to the operation on *elem*, *other* is at constant (logical) distance of *elem*. Then, in order to find its relay also in constant time, we keep a doubly linked list of relays in the same relative order that their elements and perform a sequential search starting at *r*. Note that if the relay of *other* does not exist, the search finishes when the first relay pointing to another bucket is found. This solution is more involved to implement but uses less space compared to the previous one, provided there are not many relays.

   It is clear that, in both solutions, the locality of iterator accesses decreases with the number of elements with iterators, because the locations of the relays are not related

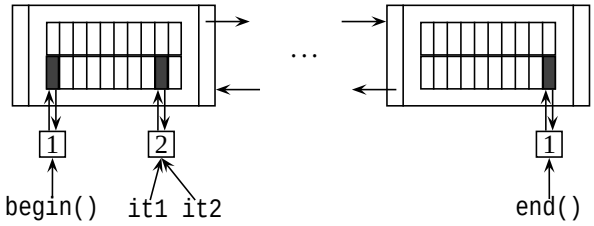


Figure 3.8: Bucket of pairs.

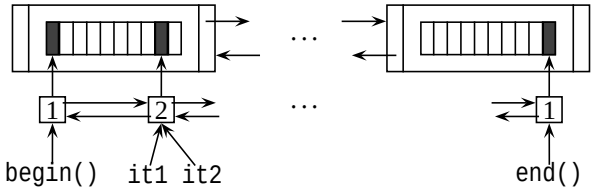


Figure 3.9: 2-level list.

to their elements. However, assuming that there will not be many iterators in a single application, we expect to find the relays in the cache. In any case, our two approaches conform to the standard for an arbitrarily large number of iterators.

5. *Relay allocation:* Given that iterators are frequently incremented or decremented and, typically, moving from/to places where there are no other iterators, our implementations avoid (whenever possible) to destroy the relay (to leave an element) and recreate it (to arrive to the next element). Not only that, our implementations also make use of a pool of relay nodes to delay as much as possible the use of the allocator to get or release memory for them.

3.4 Reorganization of the buckets

In this section, we present with some detail how we keep the buckets reorganized when insertion and deletion operations are applied to the list. For the sake of simplicity, we assume that the elements in the buckets are arranged using the linked strategy so that we may ignore intra-bucket movements.

This section is organized as follows. First, we present the notation that we will use. Then, we state the representation invariant of our data structure. Finally, we give the necessary reorganization algorithms to keep up this invariant when insertion and deletions are applied to the list. To do so, we first present some useful operations, then present the algorithm for the general case and, finally, present the algorithms for various particular cases.

3.4.1 Notation

In the following, we denote by $b(i)$ the i-th bucket of the list, by $\ell(i)$ the bucket at its left, and by $r(i)$ the bucket at its right, provided that they exist. We also denote by $\ell^j(i)$ the bucket j positions to the left of $b(i)$ and by $r^j(i)$ the bucket j positions to the right of $b(i)$. On the other hand, we denote by $|b(i)|$ the occupancy of bucket $b(i)$, that is, its number of elements divided by its capacity (K). Moreover, we define the occupancy of buckets

from i to j as $|b(i \dots j)| = |b(i)| + \cdots + |b(j)|$ and $s_i = |\ell(i)| + |b(i)| + |r(i)|$. From now on, x always denotes the new element to be inserted or deleted and m denotes the number of buckets in the list (there is always at least one bucket).

3.4.2 Representation invariant

Roughly, our goal is to try to achieve an average occupancy of at least $\frac{2}{3}$ for each bucket. This is a compromise between guaranteeing a very high occupancy for any pattern, which is costly, and very loose restrictions that could turn the data structure useless. Besides, endpoint buckets constraints are relaxed to achieve almost maximum occupancy when lists are built adding elements at the back or the front, which is a common building pattern.

Specifically, the representation invariant of our data structure is given by the following constraints, which are depicted in Figure 3.10.

- *General constraint:* For all $b(i)$, s.t. $2 < i < m-1$, we have $s_i \geq 2$.

- *Endpoints constraints:* When $m \geq 3$, we have

$$\begin{cases} |b(1)| = 0 \implies |b(2)| > \frac{5}{12}, \\ |b(1)| \neq 0 \implies |b(2)| = 1, \end{cases} \quad \text{and} \quad \begin{cases} |b(m)| = 0 \implies |b(m-1)| > \frac{5}{12}, \\ |b(m)| \neq 0 \implies |b(m-1)| = 1. \end{cases}$$

- *Small lists constraints:* We have

$$\begin{cases} m = 4 \implies |b(1 \dots 4)| > \frac{3}{2}. \\ m = 3 \implies |b(1 \dots 3)| > 1. \\ m = 2 \implies |b(1 \dots 2)| > \frac{1}{2}. \end{cases}$$

Note that lists with one single bucket do not have occupancy constraints.

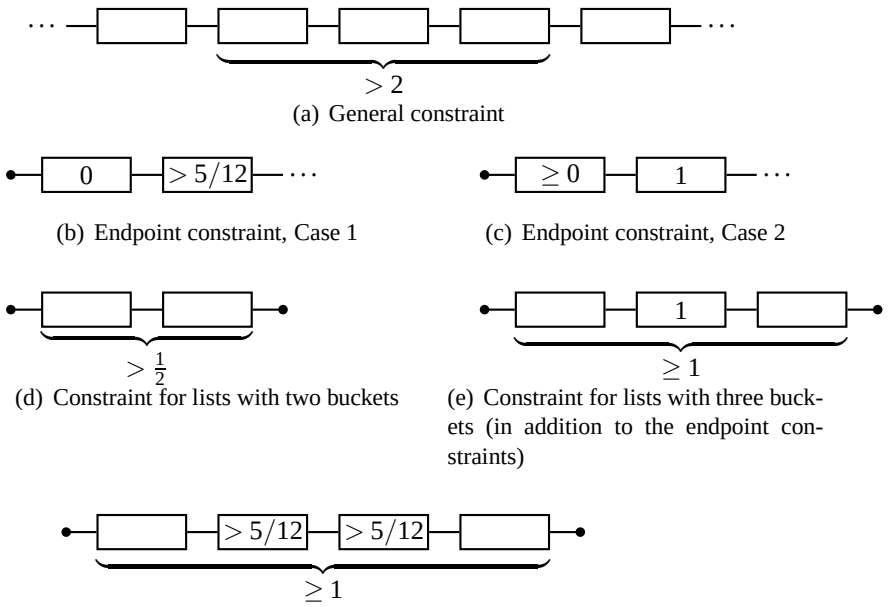
3.4.3 Useful reorganization operations

In order to present our reorganization algorithm, let us define the following basic reorganization operations:

- *Transfer from $b(i)$ to $b(j)$:* One or several elements are moved from bucket $b(i)$ to bucket $b(j)$, which typically corresponds to either $\ell(i)$ or $r(i)$.

- *Split of $b(i)$:* A new bucket is allocated to the left or right of $b(i)$, and the elements are redistributed between $\ell(i)$, $b(i)$, $r(i)$ and the new bucket.

- *Merge of $b(i)$:* The elements from $b(i)$ are redistributed among $\ell(i)$ and $r(i)$, then, $b(i)$ is deallocated.



(a) General constraint

(b) Endpoint constraint, Case 1

(c) Endpoint constraint, Case 2

(d) Constraint for lists with two buckets

(e) Constraint for lists with three buckets (in addition to the endpoint constraints)

(f) Constraint for lists with four buckets (in addition to endpoint constraints).

Figure 3.10: Occupancy constraints.

3.4.4 Reorganization algorithm for the general case

We first consider the reorganization algorithm on large lists for buckets far from the endpoints. Assume that a deletion or an insertion takes place in bucket $b(i)$, with $2 < i < m-1$. There are three possible cases where reorganization is needed:

1. *Deletion from $b(i)$, which makes $s_k = 2$ for some $k \in \{\ell(i), b(i), r(i)\}$:*

   In this case, first x is removed and then $b(k)$ is merged.

2. *Insertion in $b(i)$ such that $|b(i)| = 1$ and $s_i < 3$:*

   In this case, an element must be transfered from $b(i)$ to either $\ell(i)$ or $r(i)$ to make room for x. It seems reasonable to choose the neighbor with lowest occupancy, but this is not necessary for the forthcoming analysis.

3. *Insertion in $b(i)$ such that $|b(i)| = 1$ and $s_i = 3$ and $3 < i < m-2$:*

   In this case, either a split or a transfer may be performed. The algorithm is the following:

   (a) *General case:* First, $b(i)$ is split so that $\ell(i)$, $b(i)$, $r(i)$ and the new bucket have capacity of $\frac{3}{4}$ at least. Afterwards, x is put in its place.

   (b) *When $|\ell^2(i)| < \frac{1}{2}$:* The elements in $\ell^2(i)$ and $\ell(i)$ are equally distributed among these two buckets. In this way, $s_{\ell^3(i)}$ increases, $s_{\ell^2(i)}$ and $s_{\ell(i)}$ remain the same, and s_i decreases (never under $\frac{5}{2}$). Finally, one element from i is moved to $\ell(i)$, thus making room for x, which is put in its place.

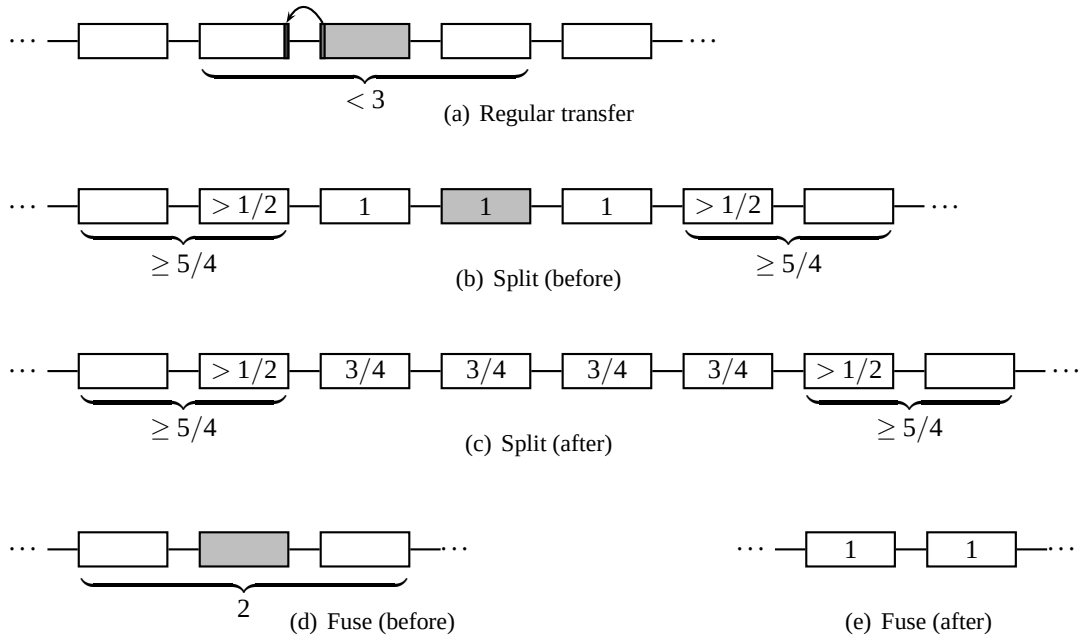


Figure 3.11: General reorganization operations.

(c) *When* $|\ell^3(i)| + |\ell^2(i)| < \frac{5}{4}$*:* The elements in $\ell^3(i)$, $\ell^2(i)$ and $\ell(i)$ are equally distributed among these three buckets. In this way, $s_{\ell^4(i)}$ and $s_{\ell^3(i)}$ increase, $s_{\ell^2(i)}$ remains the same, $s_{\ell(i)}$ decreases (never under $\frac{7}{3}$) and s_i decreases (never under $\frac{8}{3}$). Finally, we move one element from i to $\ell(i)$, thus making room for x, which is put in its place.

The three most general cases (Cases 1, 2 and 3a) are shown in Figure 3.11. The two special transfer cases (Cases 3b and 3c) are shown in Figure 3.12.

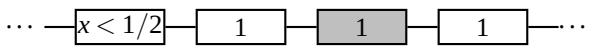
3.4.5 Reorganization algorithm at the endpoints

We consider now the reorganization algorithm at the endpoints of large enough lists. Here, the rules change to achieve almost perfect occupancy for typical list building patterns at the endpoints. We wish to note that if there were no special constraints for the endpoints, and the list was built adding elements at the ends, the general reorganization algorithm would lead to an average occupancy of about $\frac{3}{4}$, which is already greater than the lower bound of $\frac{2}{3}$.

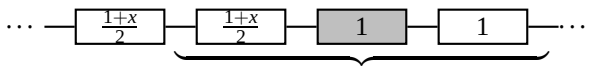
The rules are simple and apply symmetrically at the right endpoint. First, we present the transfer rules and second, rules that allocate or deallocate buckets.

Transfer rules

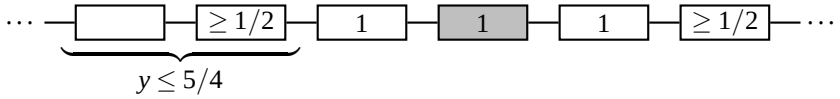
1. *Deletion in* $b(2)$ *such that* $|b(1)| > 0$: First, the element x is removed. Then, the last element from $b(1)$ is transfered to $b(2)$.



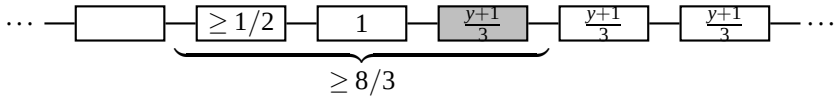
(a) Special transfer. Case 1 (before)



(b) Special transfer. Case 1 (after)



(c) Special transfer. Case 2 (before)



(d) Special transfer. Case 2 (after)

Figure 3.12: Special transfer operations.

2. *Insertion in $b(2)$ such that $|b(2)| = 1$ and $|b(1)| < 1$*: The first element from $b(2)$ is transfered to $b(1)$. Then, x is put in its place.

Allocation and deallocation rules

1. *Deletion in $b(2)$ such that $|b(2)| \leq 5/12$*: First, the element of $b(2)$ is deleted and $b(1)$ is deallocated. Then, $b(2)$ elements are transfered to $b(3)$ until this is full.

2. *Insertion in $b(1)$ such that $|b(1)| = 1$*: First of all, an empty bucket b is allocated and placed as the new $b(1)$. Then, if x was inserted at the front of the list, we just put x in b. Otherwise, the first element of $b(2)$ is shifted to b to make room for x.

3. *Insertion in $b(2)$ such that $s_2 = 3$*: We perform a split operation with final occupancies (in this order) of $1/2$, 1, $3/4$ and $3/4$. Then, x is put in its place.

4. *Insertion in $b(3)$ such that $s_3 = 3$*: First, we perform a split operation as in the general case. Second, elements from the $b(1)$ are transfered to $b(2)$ until $b(2)$ is full or $b(1)$ is empty. (Note: if the list has only five buckets, this step must be repeated for the right endpoint, because the third bucket from the left is also the third bucket from the right.) Then, x is put in its place.

3.4.6 Reorganization algorithm for small lists

We consider now lists with four or less buckets ($m \leq 4$). The rules are adapted as follows:

- *Transfer rules in Section 3.4.5*: All apply, except for lists s.t. $m = 2$, in which elements are inserted with no transfers when possible.

- *Allocation of new buckets in lists s.t.* $m \leq 3$: A new bucket is allocated only when there is no room at all. Further, the elements are reorganized as follows: the endpoint buckets are at $\frac{1}{2}$ or more of its capacity; the rest become full.
- *Allocation of new buckets in lists s.t.* $m = 4$: Here, we assume that the left endpoint is the nearest to the insertion point.
  - *Insertion in* $b(1)$: The same algorithm as in Section 3.4.5.
  - *Insertion in* $b(2)$: If $b(4)$ is not full, one element is transfered from $b(3)$ to $b(4)$, and then, one from $b(2)$ to $b(3)$. If $b(4)$ is full (i.e., all the buckets are full), a split operation must be performed with final list occupancies of $\frac{5}{8}$, 1, $\frac{3}{4}$, 1 and $\frac{5}{8}$.
- *Deallocation of buckets*: A bucket is deallocated when the corresponding minimum average occupancy is reached.
  - $m = 3$ *or* $m = 4$: The endpoint bucket b which is empty is deallocated. If $m = 4$, the elements of b neighbor are transfered to the following neighbor until this is full.
  - $m = 2$: The elements from the bucket with lowest occupancy are transfered to the other one. Then, it is deallocated.

3.5 An upper bound for the number of created and destroyed buckets

In this section, we present and prove two asymptotically optimal properties of our reorganization algorithms related to the number of created/destroyed buckets. In practice, this means that our bucket management strategy is robust whatever the applied sequence of list operations. This goes towards a good (cache) performance.

3.5.1 Interleaved operations at the same point

Theorem 3.5.1. Let a list conform to the representation invariants given in Section 3.4. Consider an arbitrary long alternating sequence of insertions and deletions at the same point. Then, the total number of allocated and deallocated buckets is at most two.

Proof. We have the following case analysis:

1. *The first operation performs no allocation neither deallocation.* There are two cases:
   (a) *No element is transfered.* In this subcase, the next operation returns the bucket to its initial state, and so happens again and again, and so, no allocation or deallocation are performed (in fact, neither transfers).
   (b) *An element is transfered from bucket* b_i *to bucket* b_j. If b_i or b_j are endpoint buckets, then this operation is reversed at each step, and so, no allocations or deallocations are performed. Otherwise, this operation must be an insertion (because deletions do not transfer elements except when at the endpoint buckets),

and so, the next is a deletion. In this case, a deallocation cannot occur either because the value of s_i is the initial one.

2. *The first operation is an insertion and allocates a bucket*. Then, the next deletion cannot deallocate a bucket. For small lists, this is due to the fact that allocation and deallocation thresholds do not coincide. For big lists, when the allocated bucket becomes an endpoint, its next bucket is full and therefore, cannot be deallocated afterwards. Finally, in the most general case, a split guarantees that the two central buckets i, where the element is actually inserted, have $s_i > \frac{9}{4}$, and so, a right afterwards deallocation cannot either occur.

   Therefore, we turn to Case 1, for which has been shown that no further allocations or deallocations are performed.

3. *The first operation is a deletion and deallocates a bucket*. There are two subcases:

   (a) *The next operation (insertion) allocates a bucket*. Then, we are in Case 2.

   (b) *The next operation (insertion) does not allocate a bucket*. Then, we are in Case 1.

   In any case, we turn to a case for which has been shown that no further allocations or deallocations are performed. Further, for Case 2, the total number of allocated and deallocated buckets is exactly two.

□

3.5.2 Arbitrary sequences of insertions and deletions

Theorem 3.5.2. Let L be an empty list. Consider a sequence of r insertions and/or deletions at arbitrary positions conform to the rules given in Section 3.4 applied to L. Then, the number of allocated and deallocated buckets is at most $6r/K$.

Proof. The amortized cost is shown using the potential method.

Let L_0 be the initial list. For each $i = 1,\ldots,r$, let c_i be the actual cost of the i-th operation, and let L_i be the list resulting after applying the i-th operation to L_{i-1}. Since we are interested in the number of created/destroyed buckets, we have

$$c_i = \begin{cases} 1, & \text{if a split/merging is performed during the } i\text{-th operation;} \\ 0, & \text{otherwise.} \end{cases}$$

Let Φ be a potential function (defined later) that assigns a real number to every list. We define:

- The increment of potential of the i-th operation: $\Delta_i = \Phi(L_i) - \Phi(L_{i-1})$.
- The amortized cost a_i of the i-th operation: $a_i = c_i + \Delta_i$.
- The total actual cost: $C = \sum_{i=1}^{r} c_i$.
- The total amortized cost: $A = \sum_{i=1}^{r} a_i$.

An easy calculation shows that $C = A + \Phi(L_0) - \Phi(L_m)$.

Every list update can be separated conceptually into two steps:

1. one normal split, or one normal merging, or the equal redistribution of the elements of some non-endpoint buckets, or one specialized operation at a bucket near an endpoint, or just nothing;
2. one element insertion or one element deletion.

Note that the actual cost c_i comes from the first step. Let us denote by $\Delta_i^{(1)}$ and by $\Delta_i^{(2)}$ the increment of potential due to the first step and to the second step, respectively; that is, $\Delta_i = \Delta_i^{(1)} + \Delta_i^{(2)}$. The rest of this section is devoted to define an adequate potential function, and prove with it

1. $\Delta_i^{(1)} \leq 0$ when $c_i = 0$,
2. $\Delta_i^{(1)} \leq -1$ when $c_i = 1$,
3. and $\Delta_i^{(2)} \leq 6/K$.

Altogether, this will imply $a_i \leq 6/K$ for every i, and thus will prove an upper bound of $6r/K$ for C.

The potential function

To define the potential function, it is convenient to use the following definitions:

$$|L| = \sum_{b \in L} |b|.$$

$$f(x) = \begin{cases} 6x-5, \text{ if } x \geq \frac{5}{6}; \\ 5-6x, \text{ if } x \leq \frac{5}{6}. \end{cases}$$

$$f_1(x) = \begin{cases} 4x-2, \text{ if } x \geq \frac{1}{2}; \\ 2-4x, \text{ if } x \leq \frac{1}{2}, \end{cases}$$

$$f_2(x) = \begin{cases} 4x-4, \text{ if } x \geq 1; \\ 4-4x, \text{ if } x \leq 1, \end{cases}$$

$$f_3(x) = \begin{cases} 4x-6, \text{ if } x \geq \frac{3}{2}; \\ 6-4x, \text{ if } x \leq \frac{3}{2}. \end{cases}$$

Then, we define the potential of a bucket b as follows:

$$\Phi(b) = \begin{cases} 2|b|, \text{ if } b \text{ is an endpoint;} \\ f(|b|), \text{ if } b \text{ is not an endpoint.} \end{cases}$$

Finally, we define the potential of a list as:

$$\Phi(L) = \begin{cases} f_1(|L|), \text{ if } L \text{ consists of only one bucket;} \\ f_2(|L|), \text{ if } L \text{ consists of two buckets;} \\ f_3(|L|), \text{ if } L \text{ consists of three buckets.} \\ \sum_{b \in L} \Phi(b), \text{ if } L \text{ consists of four or more buckets.} \end{cases}$$

Computation of $\Delta_i^{(1)}$ and $\Delta_i^{(2)}$

From the definitions in the previous sections, it is trivial that $\Delta_i^{(2)} \leq 6/K$. Let us now bound $\Delta_i^{(1)}$ for every case.

Large lists

1. Normal split: $\Delta_i^{(1)} = 4 \cdot f(\frac{3}{4}) - 3 \cdot f(1) = -1$.

2. Normal merging: Let b_i, b_j, b_k be the initial buckets. Denote $x = |b_i|, y = |b_j|, z = |b_k|$ and $\varphi = f(x) + f(y) + f(z)$. It holds that $x + y + z = 2$. Further, suppose w.l.o.g. that $x \leq y \leq z$. Then, $\Delta_i^{(1)} = 2 \cdot f(1) - \varphi = 2 - \varphi$. We devise 3 cases according to the values of x, y and z: (a) $x \leq \frac{2}{6}$, $y \geq \frac{5}{6}$, and $z \geq \frac{5}{6}$, (b) $x < \frac{5}{6}$, $y < \frac{5}{6}$, and $z \geq \frac{5}{6}$ and (c) $x < \frac{5}{6}$, $y < \frac{5}{6}$, and $z < \frac{5}{6}$.

   For any of them, the reader can easily check that $\varphi \geq 3$, and so, $\Delta_i^{(1)} \leq -1$.

3. Equal redistribution of elements when inserting on a bucket i, s.t. $|\ell^2(i)| < \frac{1}{2}$ (Case 3b in Section 3.4). Let $x_2 = |\ell^2(i)|$, then: $\Delta_i^{(1)} = 2 \cdot f(\frac{1+x_2}{2}) - (f(1) + f(x_2)) = -2$.

4. Equal redistribution of elements when inserting on a bucket i, s.t. $|\ell^3(i)| + |\ell^2(i)| < \frac{5}{4}$ (Case 3c in Section 3.4). Let $x_2 = |\ell^2(i)|$ and $x_3 = |\ell^3(i)|$, then: $\Delta_i^{(1)} = 3 \cdot f(\frac{1+x_2+x_3}{3}) - (f(1) + f(x_2) + f(x_3)) = 8 - 6(x_2 + x_3) - (f(x_2) + f(x_3))$.

   We devise two cases according the value of x_2: (a) $x_2 \leq \frac{5}{6}$ and (b) $x_2 \geq \frac{5}{6}$. In both cases, the reader can easily check that $\Delta_i^{(1)} \leq -2$.

5. Removing the first bucket when the second bucket is at $\frac{5}{12}$ or less: Let b_3 be the third bucket and $x = |b_3|$, then: $\Delta_i^{(1)} = 2(\frac{5}{12} - (1 - x)) + f(1) - (2 \cdot 0 + f(\frac{5}{12}) + f(x)) = -\frac{8}{3} + 2x - f(x)$. We devise two cases according to the value of x: (a) $x \geq \frac{5}{6}$ and (b) $x \leq \frac{5}{6}$. In any of them, it can be checked that $\Delta_i^{(1)} \leq -1$.

6. Splitting at the first bucket: $\Delta_i^{(1)} = 2 \cdot 0 + f(1) - (2 \cdot 1) = -1$.

7. Splitting at the second bucket: $\Delta_i^{(1)} = 2 \cdot \frac{1}{2} + f(1) + 2 \cdot f(\frac{3}{4}) - (2 \cdot 1 + 2 \cdot f(1)) = -1$.

8. Splitting at the third bucket: Let b_1 be the first bucket, $x = |b_1|$, $y = \min(x, \frac{1}{4})$, and $z = x - y$, then: $\Delta_i^{(1)} = 2z + f(\frac{3}{4} + y) + 3 \cdot f(\frac{3}{4}) - (2x + 3 \cdot f(1)) = -\frac{3}{2} - 2y + f(\frac{3}{4} + y)$. There are two cases according to the value of y: (a) $y \geq \frac{1}{12}$ and (b) $y \leq \frac{1}{12}$. In both cases, it can be checked that $\Delta_i^{(1)} \leq -1$.

Small lists

1. Split 1–2: $\Delta_i^{(1)} = f_2(1) - f_1(1) = -2$.

2. Merging 2–1: $\Delta_i^{(1)} = f_1(\frac{1}{2}) - f_2(\frac{1}{2}) = -2$.

3. Split 2–3: $\Delta_i^{(1)} = f_3(2) - f_2(2) = -2$.

4. Merging 3–2: $\Delta_i^{(1)} = f_2(1) - f_3(1) = -2$.

5. Split 3–4: $\Delta_i^{(1)} = 2(2 \cdot \frac{1}{2}) + 2 \cdot f(1) - f_3(3) = -2$.

6. Merging 4–3: Let b_2 and b_3 be the middle buckets of the original 4-bucket list. Let $x = |b_2|$ and $y = |b_3|$, where $y = \frac{3}{2} - x$. Then, $\Delta_i^{(1)} = f_3(\frac{3}{2}) - (2(2 \cdot 0) + f(x) + f(y)) = -(f(x) + f(\frac{3}{2} - x))$. Let us assume that $x \geq y$ without loss of generality. There are two cases: (a) $x \geq \frac{5}{6}$ and (b) $x \leq \frac{5}{6}$. In both of them, it can be checked that $\Delta_i^{(1)} \leq -1$.

7. Splitting 4–5 at a non-endpoint when all buckets are full: $\Delta_i^{(1)} = 2(2 \cdot \frac{5}{8}) + 2 \cdot f(1) + f(\frac{3}{4}) - (2(2 \cdot 1) + 2 \cdot f(1)) = -1$.

8. Extra reorganization after splitting 5–6 at the third bucket: Let b_5 be the fifth bucket, $x = |b_5|$, $y = \min(x, \frac{1}{4})$, and $z = x - y$, then: $\Delta_i^{(1)} = 2z + f(\frac{3}{4} + y) - (2x + f(\frac{3}{4})) = -\frac{1}{2} - 2y + f(\frac{3}{4} + y)$.

   There are two cases according to the value of y: (a) $y \geq \frac{1}{12}$ and (b) $y \leq \frac{1}{12}$. In any of them, it can be checked that $\Delta_i^{(1)} \leq 0$.

□

Note that the bound $\Delta_i^{(2)} \leq 6/K$ is not necessarily tight. It remains as an open problem showing that this bound is tight or that, by contrast, a tighter bound exists.

3.6 Experimental analysis

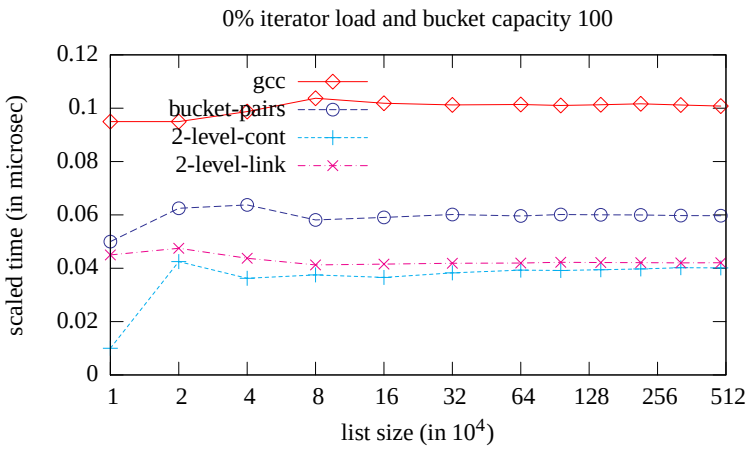
In this section, we experimentally analyze the performance of our implementations and show their competitiveness in several common settings.

We developed three implementations. Two of them use contiguous bucket arrangement, one of which uses the *bucket of pairs* iterator solution and the other the *2-level* solution. The third implementation uses a linked bucket arrangement and the *2-level* iterator solution. Notice that in contrast to a flat doubly linked list, our operations deal with several cases and each of them with more instructions. This makes our code 3 or 4 times longer (in code lines).

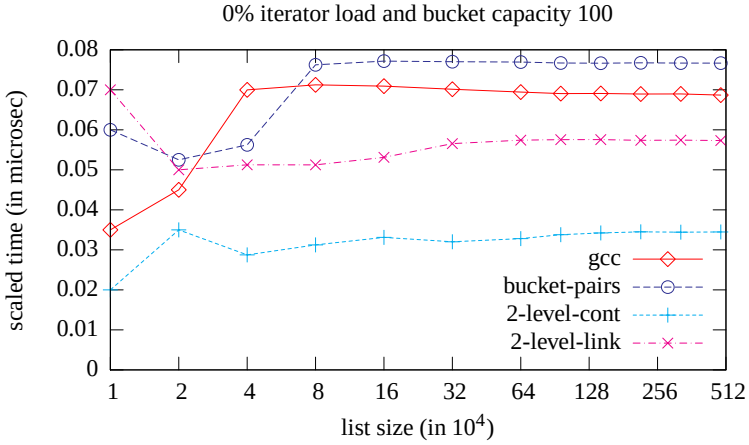
The results are shown for a Sun workstation with Linux and an AMD Opteron CPU at 2.4 GHz, 1 GB main memory, 64 KB + 64 KB 2-associative L1 cache, 1024 KB 16-associative L2 cache and 64 bytes per cache block. The programs were compiled using the GCC 4.0.1 compiler with optimization flag `-O3`. Comparisons were made against the current STL GCC implementation and LEDA 4.0 (in the latter case the compiler was GCC 2.95 for compatibility reasons).

All the experiments were carried with lists of integers considering several list sizes that fit in main memory. Besides, all the plotted measurements are scaled to list size for a better visualization.

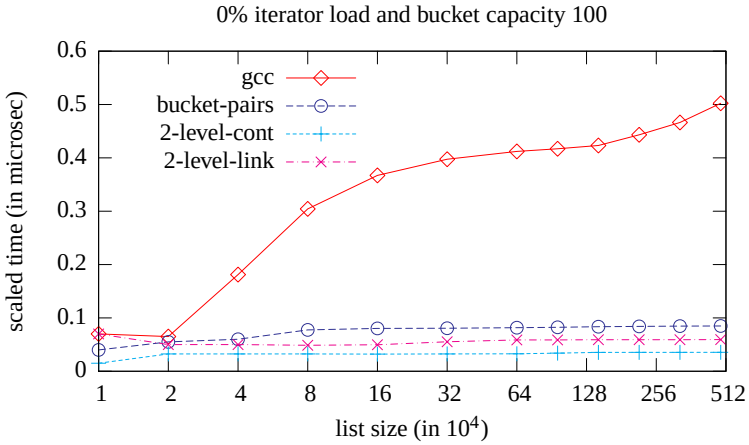
With regard to performance measures, we collected wall-clock times, that were repeated enough times to obtain significative averages (variance was always observed to be very low).



(a) `push_front`



(b) Traversal before shuffling



(c) Traversal after shuffling

Figure 3.13: Performance results for basic operations with no iterator load.

Furthermore, to get significant insight on the behavior of the cache, we used Pin [61], a collection of tools for the dynamic instrumentation of programs. Specifically, we have used a Pin tool that simulates and gives statistics of the cache hierarchy (using typical values of the AMD Opteron).

In the following we present the most significant results. Firstly, we analyze the behavior of lists with no iterators involving basic operations and common access patterns. Then, we consider lists with iterators. Finally, we compare our implementations against LEDA, and consider other hardware environments.

3.6.1 Basic operations with no iterator load

Insertion at the back and at the front

Given an initially empty list, we wish to compare the time to get a list of n elements by successively applying n calls to either the `push_back` or `push_front` methods.

The results for `push_front` are shown in Figure 3.13(a); a similar behavior was observed for `push_back`. In these operations, we observe that our three implementations perform significantly better than GCC. This must be due to manage memory more efficiently: firstly, the allocator is called only once for all elements in a bucket and not for every element. Secondly, our implementations ensure that buckets get full or almost full in these operations, and so, less total memory space is allocated.

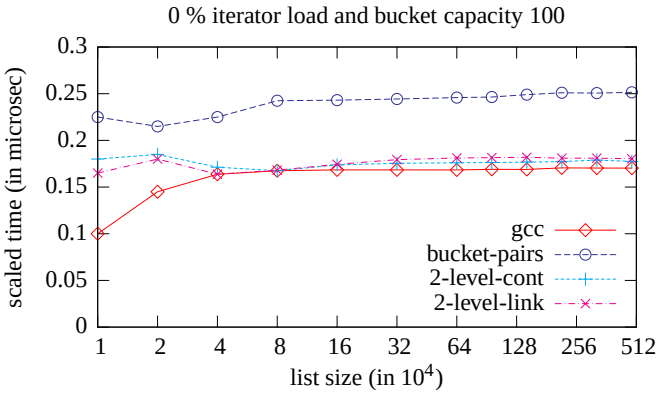
Traversal

Consider the following experiment: First, build a list; then, create an iterator at its begin and advance it up to its end four times. At each step, add the current element to a counter. We measure the time taken by all traversals.

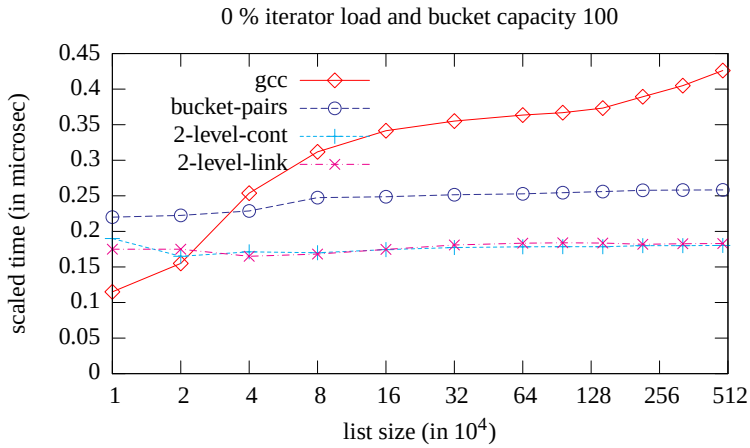
Here, the way to construct the list plays an important role. If we just build the list as in the previous experiment, the traversal times are those summarized in Figure 3.13(b). These show that performance does not depend on list size and that our 2-level contiguous list implementation is specially efficient even compared to the other 2-level implementation. Our linked bucket implementation is slower than the contiguous implementation because, firstly, its buckets are bigger for the same capacity and so, there are more memory accesses (and misses). Secondly, the increment operation of the linked implementation requires more instructions.

Rather, if we sort this list before doing the traversals, and then measure the time, we obtain the results shown in Figure 3.13(c). Now, the difference between GCC's implementation and ours becomes very significant and increases with list size (our implementation turns out to be more than 5 times faster). Notice also that there is a big slope just beginning at lists with 20000 elements.

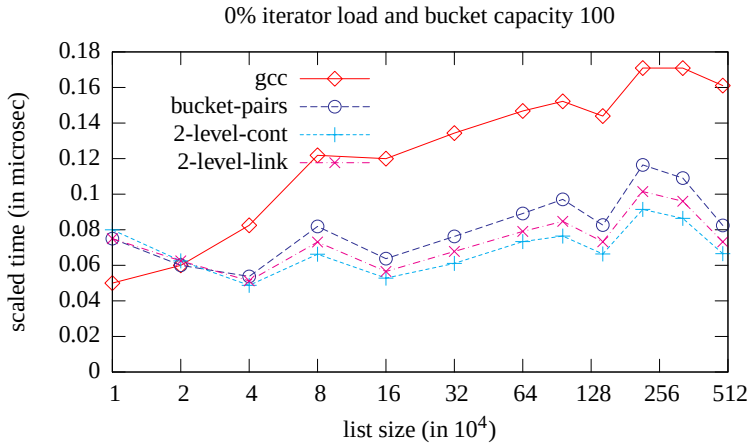
The difference in performance is due to the different physical arrangement of elements in memory (in relation to their logical positions). To prove this claim, we repeated the same experiment using the Pin tool, counting the number of instructions, as well as, L1 and L2 cache accesses and misses. Some of these results are given in Figure 3.15(a). Firstly, these show that indeed our implementations incur in less caches misses (both in L1 and L2). Secondly, the scaled ratio of L1 misses is almost constant because even small lists do not fit



(a) Insertion before shuffling



(b) Insertion after shuffling



(c) Intensive insertion

Figure 3.14: Performance results for basic operations with no iterator load.

in L1. Besides, the big slope in time performance for the GCC implementation coincides with a sudden rise in L2 cache miss ratio, which leads to a state in which almost every access to L2 is a miss. This transition also occurs in our implementations, but much more smoothly. Nevertheless, the L2 access ratio (that is, L1 miss ratio) is much lower because logically close elements are in most cases in the same bucket and so, already in the L1 cache (because bucket capacity is not too big).

Insertion

In this experiment, we deal with insertions at arbitrary points. First, a list is built (using the two abovementioned ways). Then, it is forwardly traversed four times. At each step, with probability $\frac{1}{2}$, an element is inserted before the current. We measure the time of doing the traversal plus the insertions.

Results are shown in Figures 3.14(a) and 3.14(b), whose horizontal axis corresponds to the initial list size.

Analogously to plain traversal, performance depends on the way the list is built. However, as in this case the computation cost is greater, the differences are smoother. In fact, when the list has not been shuffled, the bucket of pairs list performs worse than GCC's. Our two other implementations perform similarly to GCC's though. On the other hand, when the list has been shuffled, GCC's time highly increases, while ours is almost not affected.

It is interesting to observe that the linked arrangement implementation does not outperform the contiguous ones even though it does not require shifting elements inside the bucket. This must be due to the fact that more memory accesses (and misses) are performed and this is still dominant. This was confirmed performing the same experiment under the Pin environment. As more insertions per element are done, this gain starts to be rather noteworthy. This is shown in Figure 3.14(c).

Internal sort

The STL requires a $O(n \log n)$ `sort` method that preserves the iterators on its elements. Our implementations use a customized implementation of merge sort in order to keep with the validity of iterators.

Results of executing the sort method are given in Figure 3.16. These show that our implementations are between 3 and 4 times faster than GCC. Taking into account that GCC also implements a merge sort, we claim that the significant speedup is due to the locality of data accesses inside the buckets. To confirm this, Figure 3.15(b) shows the Pin results. Indeed, GCC does about 30 times more cache accesses and misses than our implementations.

Effect of bucket capacity

The previous results were obtained for buckets with capacity of 100 elements. Anyway, this choice did not appear to be critical. Specifically, we repeated the previous tests with lists with other capacities, and observed that once the bucket capacity is not very small (less than 8-12 elements), a wide range of values behaved neatly. Note that a bucket of integers with capacity of 8 elements is already 40-80 bytes long (depending on the implementation and address length) and a typical cache block is 64 or 128 bytes long.

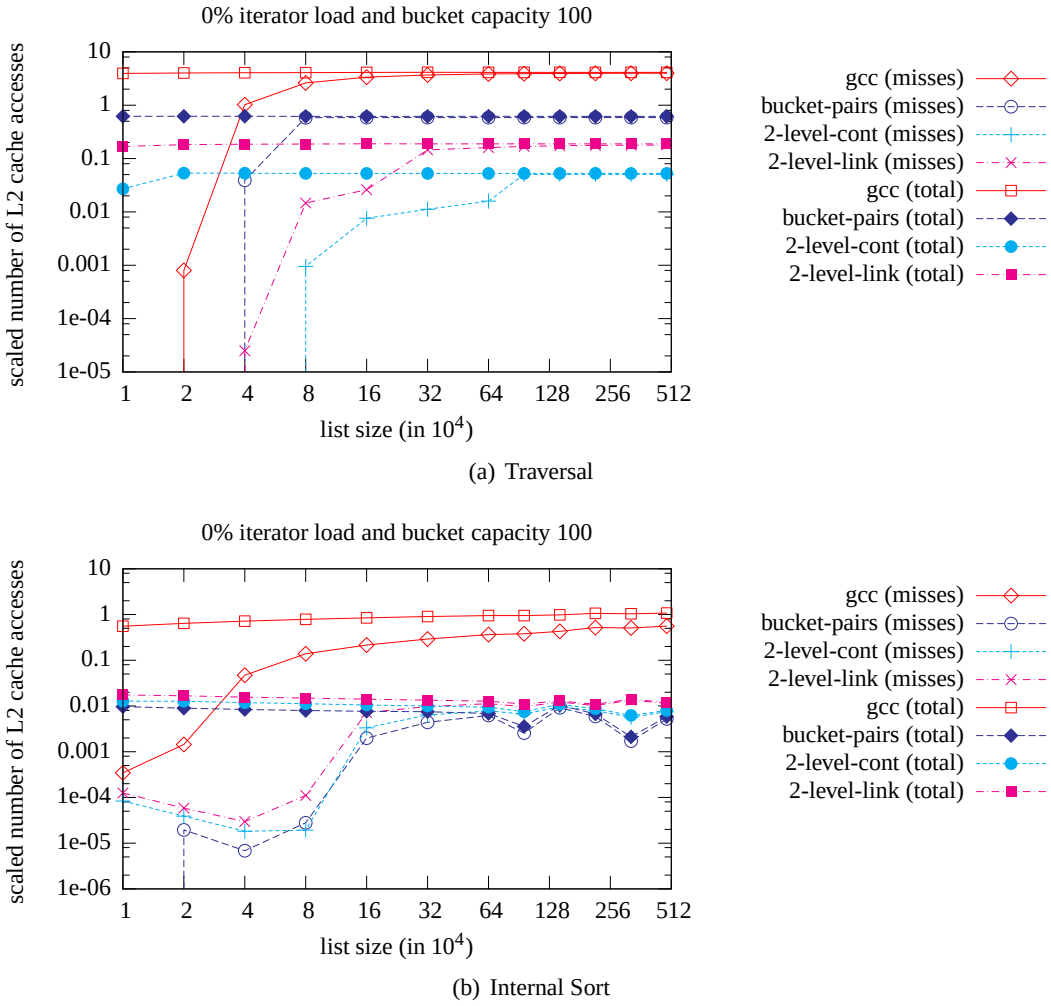


Figure 3.15: Simulation results on the cache performance (the vertical axis is logarithmic).

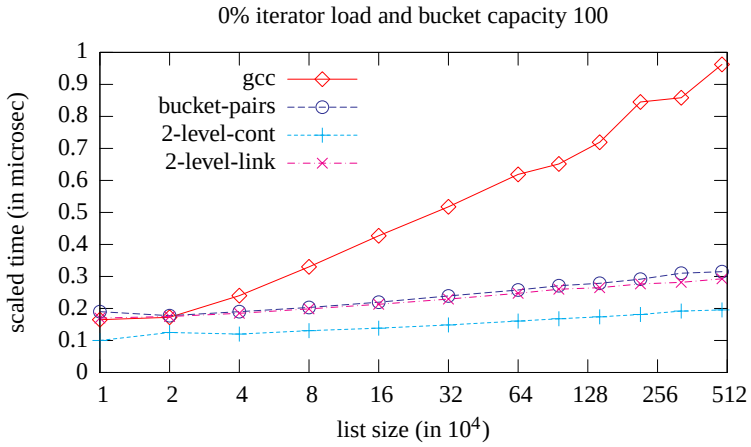


Figure 3.16: Performance results for `sort` with no iterator load.

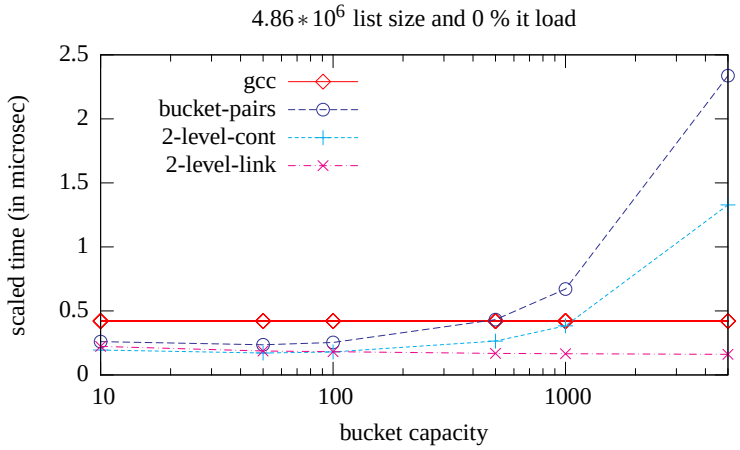


Figure 3.17: Performance results for insertion after shuffling, with variable bucket capacity (list size 486000).

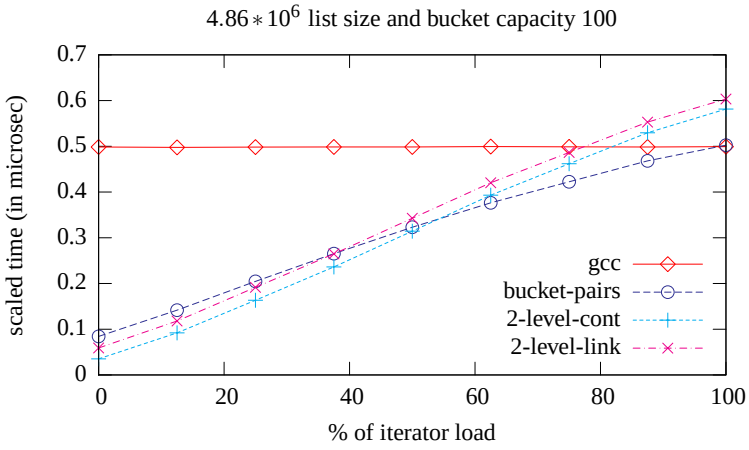
To illustrate the previous claims, we show in Figure 3.17 insertion results on a shuffled list with initially roughly 5 million elements. These show that for contiguous arrangement implementations, time decreases until a certain point and then starts to increase. In these cases, increasing the bucket size increases the intrabucket movements which finally results more costly than the achieved locality of accesses. In contrast, the linked arrangement implementation seems to be not affected because no such operations are performed, accesses of a bucket do not interfere between them, and our insert reorganization algorithm takes into account at most three buckets at a time.

If we perform the previous test with several instances at the same time, a smooth rise in time for all implementations can be seen, in particular for big bucket capacities. In fact, it is common dealing with several data structures at the same time. In this case, some interferences within the different objects accesses can occur, which are more probable as the number of instances grows. Therefore, it is advisable to keep a relatively low bucket size.

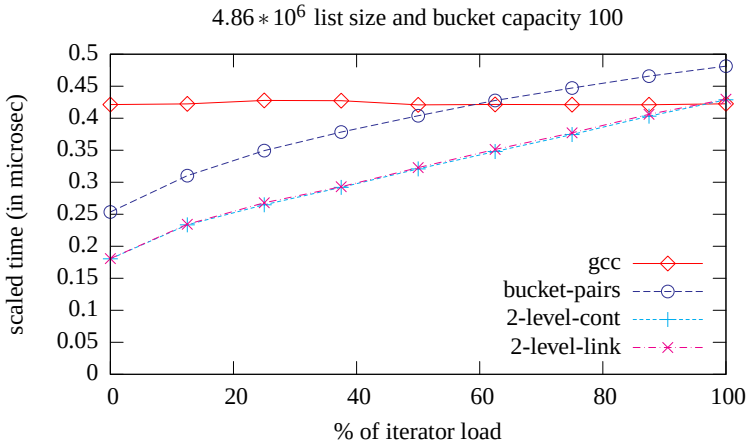
3.6.2 Basic operations with iterator load

Now, we repeat the previous experiments on lists that do have iterators on their elements. We use the term *iterator load* to refer to the percentage of elements of a list that have one or more iterators on them. Given how our iterators are implemented, this is what really can affect performance and not the exact number of iterators.

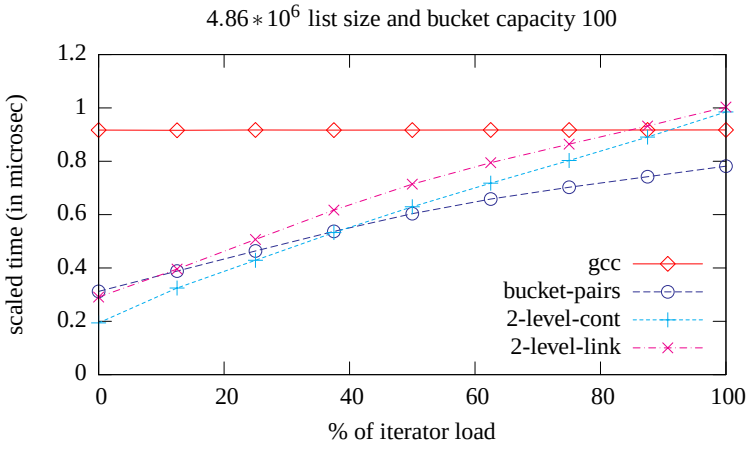
Results are shown for tests in which elements have already been shuffled, iterator loads range from 0% to 100% and a big list size is fixed (about 5 million elements) because then, it is crucial to manage data in the cache efficiently.



(a) Traversal after shuffling



(b) Insertion after shuffling



(c) Internal sort

Figure 3.18: Performance results depending on the iterator load.

Traversal

When there are no iterators on the list, our implementations perform a list traversal very fast because the increment operation is simple and elements are accessed with high locality. However, when there are some iterators, it may turn slower because the increment operation depends whether there are other iterators pointing to the element or its successor. In contrast, the increment operation on traditional doubly linked lists is independent of it, and so, performance must be not affected. When the list has not been shuffled, this is exactly the case.

In contrast, when the elements are shuffled, which changes iterators logical order, iterators accesses may score low locality. Results for this case are shown in Figure 3.18(a), which show indeed that the memory overhead become the most important factor in performance. Nevertheless, the good locality of accesses to the elements themselves makes our implementations more competitive than GCC's up to 80% iterator load even for relatively small lists (about 100000 elements).

Insertion

When an element is inserted in a bucket with several iterators, some extra operations must be done but are much less than in the traversal case in relative terms. Therefore, performance should be less affected.

Insertion results are shown after the elements have been shuffled in Figure 3.18(b).

The results are analogous to the traversal test but with smoother slopes, as happened with no iterator load. Specifically, when the list has been shuffled, our implementations are more convenient up to 80% iterator load.

Internal sort

Guaranteeing iterators consistency in our customized merge sort is not straightforward, specially in the case of 2-level approaches that need some (though small) auxiliary arrays. Performance results are shown in Figure 3.18(c).

The results indeed show that the 2-level implementations are more sensitive to the iterator load. Anyway, all of our implementation are faster than GCC for iterators loads lower than 90%.

3.6.3 Comparison with LEDA

We compare now our lists with LEDA lists. The interface of LEDA lists is very similar to STL `lists`. Besides, they use a doubly linked list implementation. In addition, LEDA lists use a customized memory allocator.

In Figure 3.19, we show the results for traversal operation after shuffling. These make evident the limitations in performance of using a doubly linked list compared to our cache-conscious approach. LEDA's times are just slightly better than GCC's, but remain worse than our implementations.

We omit the rest of plots with LEDA, because its results are just slightly better than GCC. The only exception is its internal sort (a quicksort) which is very competitive. Nev-

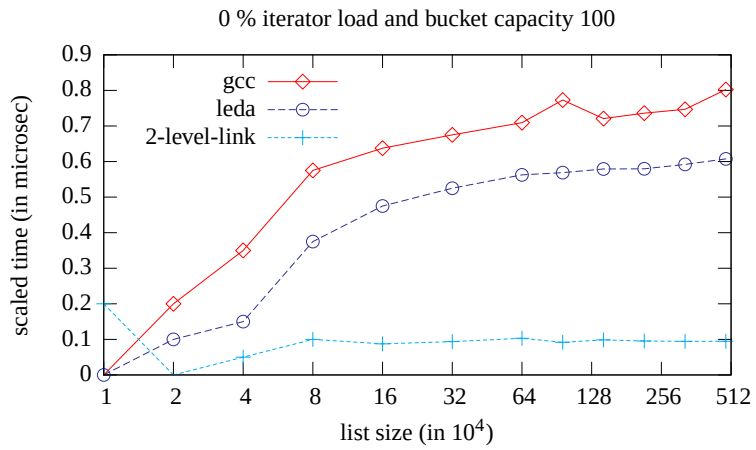


Figure 3.19: Performance results for traversal after shuffling (LEDA).

ertheless, it requires linear extra space, does not keep iterators (items in LEDA jargon) and is not faster than ours.

3.6.4 Other environments

The previous experiments have been run in a AMD Opteron machine. We have verified that the results we claim also hold on other environments. These include an older AMD K6 3D Processor at 450 MHz with a 32 KB + 32 KB L1 cache, 512 KB L2 off-chip (66 MHz) and a Pentium 4 CPU at 3.06 GHz, with a 8KB + 8KB L1 cache and 512 KB L2 cache. On both machines, similar results are obtained in relative terms. Indeed, the results were better on newer machines and compilers.

3.7 Conclusions and future work

In this chapter we have presented three cache-conscious lists implementations that are compliant with the C++ standard library. Cache-conscious lists were studied before but did not cope with library requirements. Indeed, these goals enter into conflict, particularly preserving both constant costs and iterators requirements.

This chapter shows that it is possible to efficiently and effectively combine cache consciousness with STL requirements. On the one hand, our reorganization algorithm guarantees a minimum average occupancy of $2/3$ and almost maximum occupancy when the list is built adding elements at the endpoints. Besides, we have shown optimal bounds with respect to the number of created and destroyed buckets for some general sequences of operations.

On the other hand, a wide range of experiments has shown that our implementations are useful in many situations. The experiments compare our implementations against doubly linked list implementations such as GCC and LEDA. These show for instance that our lists can offer 5-10 times faster traversals, 3-5 times faster internal sort and still be competitive with an (unusual) big load of iterators. Besides, in contrast to doubly linked lists, the

performance of our data structure does not degenerate when the list is shuffled.

Further, the experiments show that the 2-level implementations are specially efficient. In particular, we would recommend using the linked bucket implementation, although its benefits only evince when the modifying operations are really frequent, because it can make more profit of bigger cache blocks.

Our approach to STL `lists` could be extended to more complicated data structures. For instance, several cache-conscious implementations of dictionaries use B-trees. See e.g., implementations in the CPH STL [25]. As far as we are concerned, they do not implement though iterator functionality. In enhancing iterators in B-trees, the 2-level implementation of iterators seems that should also translate into a tree data structure to keep up with costs. Thus, the bucket of pairs implementation of iterators might turn out to be more attractive in this context. Overall, the extension is not straightforward.

Chapter 4

On the number of string lookups in BSTs with digital access, and related algorithms

Efficient string algorithms and data structures are obtained in essentially two ways. On the one hand, *ad hoc* string algorithms and data structures can be defined. On the other hand, comparison-based data structures and algorithms can be specialized so that less (or not at all) redundant character comparisons are performed. See Section 2.3 for more details on these approaches.

In this chapter, we deepen into the properties of the latter approach. Comparisons in such specialized algorithms and data structures take advantage of the digital structure of strings, and the order and the outcome of other comparisons. To efficiently reuse this information, some extra data per element must be kept. In some cases, the resulting comparison method does not access the actual strings, but only their extra data instead. Assuming the usual pointer implementation of strings, this is relevant from a performance perspective, because avoiding chasing a pointer saves potential cache misses.

We analyze the number of string lookups in so-enhanced BSTs, quicksort and quickselect. In the following, we call them respectively *augmented BSTs (*ABSTs*)*, *augmented quicksort (*AQSORT*)* and *augmented quickselect (*AQSEL*)*. In Section 4.1, we describe them in more detail. In order to compute the number of string lookups of ABSTs, AQSORT and AQSEL, it is key to relate them with TSTs, which are *ad hoc* string trees. See Section 2.3.1 for further details on *ad hoc* string trees. In Section 4.2, we introduce some useful notation. Then, in Section 4.3, we describe precisely the relationship between ABSTs and TSTs, and present theoretical analysis on the number of string lookups. Following from this analysis, we add some redundancy to ABSTs, which avoids one specific kind of string lookups. Its analysis is on Section 4.4. Then, in Section 4.5, we analyze AQSORT and AQSEL relating them to ABSTs. Finally, we sum up some conclusions in Section 4.6.

4.1 Previous work

Combining fast string comparisons with comparison-based data structures and algorithms has been tackled in some previous works (see e.g., [79, 45, 28]). In this section, we review how to enhance string comparisons in BSTs, quicksort and quickselect.

ABSTs. We denote by *augmented BST (*ABST*)* a generic BST that is modified as described in [79, 45] for fast string comparisons. The name choice illustrates the fact that the tree nodes must be augmented with some extra piece of information. Specifically, ABSTs are obtained from generic BSTs as follows. Consider a BST node storing a string y and pointers to its children (and possibly to its parent). Let ℓ_p and ℓ_s be respectively the length of the maximum common prefix of y with its predecessor and its successor. Each node in an ABST additionally stores ℓ, defined as the maximum of ℓ_p and ℓ_s, and a boolean μ to indicate which of the two is the maximum. Searching for a string w works as follows: while going down the tree, two lengths are kept: l_p and l_s; they are the maximum common prefix of w with the current predecessor and the current successor respectively (thus, initially they are 0). The new comparison function works as shown in Figure 4.1. According to μ, ℓ is compared against l_p or l_s. Let l be the length to which ℓ is compared to. If $l = \ell$ and l is the biggest of l_p and l_s, the characters of w must be compared against those of y, starting at the $\ell+1$-th position, which has the first significant character of y, until a difference is found. Besides, if at least the first pair of characters are equal (i.e., $y[\ell+1] = w[\ell+1]$), ℓ_p or ℓ_s must be updated accordingly. If $l \neq \ell$ or l is the smallest of l_p and l_s, the next node to visit is merely determined on the values of ℓ, μ, and l, there is no need to look up the string y, and there is no change in ℓ_p or ℓ_s. Using this method, searching for a string of length m in a BST of height h takes time $O(m+h)$.

Concerning storage needs, the overall increment in storage becomes less important as longer is the string pointed to from the node. Besides, looking up the fields ℓ and b in a search only increases the number of local memory accesses, which are handled efficiently by memory hierarchies.

AQSORT and AQSEL. We denote by *augmented quicksort (*AQSORT*)* a generic quicksort algorithm that is modified as described in [79] for fast string comparisons. Analogously, we denote by *augmented quickselect (*AQSEL*)* a generic quickselect algorithm that is so-modified (note that this case is not explicitly tackled in [79]). Similarly as with BSTs, some extra piece of information must be kept per each element.

The application of these techniques relies in the following properties of quicksort and quickselect. First, if the pivot is chosen without comparing elements, comparisons are solely made in the partitioning process. In every partition, each element is compared exactly once against the pivot. Besides, the recursive calls of quicksort have the implicit structure of a BST. Each pivot choice (and partitioning) constitutes a node and each recursion base case corresponds to a leaf. Note that in the case of quickselect, the structure of the recursive calls corresponds to a path in a BST.

Given the correspondence of quicksort and quickselect with BSTs, we can enhance these algorithms with fast string comparisons in an analogously as BSTs. Specifically, generic quicksort and quickselect are modified as follows. For each string x, the length of

```
resultComparison compare(nodePtr t, stringType& key, int& l_p, int& l_s) {
    if (t->b) {
        if (l_p > t->l) return SMALLER;
        if (l_p < t->l or l_p < l_s) return GREATER;
        return look_string (t, key, l_p, l_s );
    }
    else {
        if (l_s > t->l) return GREATER;
        if (l_s < t->l or l_s < l_p) return SMALLER;
        return look_string (t, key, l_p, l_s );
    }
}

// Case stringType is char*
resultComparison look_string (nodePtr t, stringType& key, int& l_p, int& l_s) {
    int l = t->l;
    while (key[l] == t->key[l] and key[l] ≠ NULL) ++l;
    if (key[l] < t->key[l]) {
        l_s = l;
        return SMALLER;
    }
    l_p = l;
    if (key[l] > t->key[l]) return GREATER;
    return EQUAL;
}
```

Figure 4.1: Specialized compare function for ABSTs.

the maximum common prefix of x with its predecessor and successor in the implicit BST structure is kept. We denote them respectively $\gamma_p(x)$ and $\gamma_s(x)$. The following changes must be done while partitioning. Let v be the string acting as pivot, and let w be any string in the array to be partitioned. Then, the comparison function of w against v is analogous to that in ABSTs: v acts as the string in the node and w acts as the string to be searched. Besides, the swap function of two given strings x_1 and x_2 must be specialized so that, in addition, $\gamma_p(x_1)$ is swapped with $\gamma_p(x_2)$, and $\gamma_s(x_1)$ is swapped with $\gamma_s(x_2)$.

Note that the comparisons use the results of previous partitioning steps (comparisons inside each partitioning step are independent). The first partition must compare all the strings from the beginning, as in the non-specialized partitioning algorithm, and, in addition, it must update $\gamma_p(x)$ and $\gamma_s(x)$. However, from the second call on, the computed prefixes $\gamma_p(x)$ and $\gamma_s(x)$ are used and accordingly updated to avoid redundant character comparisons. Thus, this specialized partition is only beneficial if it is going to be repetitively used (as in quicksort and quickselect), because of the information about the prefixes gathered in previous iterations.

Finally, we present parallel versions of AQSORT and AQSEL in Chapter 9.

4.2 Notation

In this section, we present the notation that will be used throughout the chapter. Let a be an array of strings. Then, $S(a)$ denotes the set of strings in a. Analogously, let u be a search tree (e.g., a TST, an ABST). Then, $S(u)$ denotes the set of strings stored in u. Let w be a string. Then, $C(u,w)$ denotes the sequence of pairwise character comparisons made in u when searching for w. We say that each pairwise character comparison in u is *matched* to the node whose string is looked up to make the comparison. $I(u,w)$ denotes the sequence of subsequences resulting from partitioning $C(u,w)$ after each inequality comparison.

Besides, the following notation on TST elements is used, in addition to standard notation on TSTs (see Section 2.3.1). A node with at least one not null comparison pointer is a *decision node*. A node that is not reached chasing a descent pointer is a *descent root*. A path that begins in a descent root, ends either in a decision node or in a leaf, and is made exclusively of descent pointers, is called a *descent path*. It is *proper* if it contains two or more nodes (joined by one or more descent pointers). Note that intermediate nodes can also be decision nodes. Thus, there may be several descent paths beginning at the same node, but with different lengths. For instance, in Figure 2.1(b) there are three descent paths beginning from the right child of the root: `tr`, `tre` and `tree#`. Let x be a decision root or a leaf in a TST; $D(x)$ denotes the sequence of nodes in the descent path that ends in x.

The following notation regards to the searching path for a string w in a TST. A *decision taken node* for w is a decision node in the searching path of w such that one of its comparison pointers is chased. For instance, when searching for `tree` in the TST in Figure 2.1 (b), the root is a decision taken node, whilst its right child is a decision node but not a decision taken node. A *search descent path* for w is a descent path ending in a decision taken node or a leaf. Alternatively stated, it is a maximal descent path in the searching path for w. It is *proper* if it contains at least two nodes. For instance, the searching path for `tree` in the TST in Figure 2.1 (b) contains only one search descent path: `tree#`. The searching path for `trie` contains two search descent paths: `tre` and `ie#`. Note that the last character in the

search descent paths are not included in the searched word; for instance, `trie` is build up concatenating `tr` and `ie`.

4.3 On the number of string lookups in ABSTs

In this section, we analyze the number of string lookups in ABSTs. First, we present some independent properties on TSTs and ABSTs. Then, we relate both kinds of search trees. Finally, we relate the number of string lookups in ABSTs with known properties of TSTs. This relationship with TSTs was already sketched in [79], in order to compute the number of digit comparisons.

Lemma 4.3.1. Let t be a TST and let w be a string. Each of the subsequences in $I(t,w)$ is matched to a search descent path in t for w, and each search descent path in t for w is matched to a subsequence.

Proof. $I(t,w)$ is obtained from $C(t,w)$ by partitioning after each inequality comparison. In t inequality happens only if a comparison pointer is chased. Thus, the nodes that are matched to a subsequence in $I(t,w)$ must be joint by descent pointers. Moreover, the first must be a descent root, and the last must be a decision taken node. This is exactly a search descent path. On the other hand, each search descent path must be matched to a subsequence in $I(t,w)$, because otherwise $I(t,w)$ would not represent the sequence $C(t,w)$. □

From the previous lemma, we have the following corollary:

Corollary 4.3.2. Let t be a TST. The number of search descent paths in t when searching for a string w is $|I(t,w)|$. The cumulative number of search descent paths in t when searching for every string in $S(t)$ is $\sum_{w\in S(t)} |I(t,w)|$.

Now, we analyze ABSTs.

Lemma 4.3.3. Let b be an ABST and let w be a string. Each of the subsequences in $I(b,w)$ is matched to a node in b. Moreover, if a node in the searching path in b for w is not matched to any subsequence in $I(b,w)$, then no character is compared in that node.

Proof. $I(b,w)$ is obtained from $C(b,w)$ by partitioning after each inequality comparison. In b inequality happens only if a pointer is chased. Thus, each of the subsequences in $I(b,w)$ must be matched to exactly one node. On the other hand, if a node is not matched to any subsequence, no character is compared in the node, because otherwise $I(b,w)$ would not represent the sequence $C(b,w)$. □

From the previous lemma, we have the following corollary:

Corollary 4.3.4. Let b be an ABST. The number of string lookups when searching for a string w is $|I(b,w)|$. The cumulative number of string lookups in b when searching for every string in $S(b)$ is $\sum_{w\in S(b)} |I(b,w)|$.

Using the previous properties, we relate TSTs and ABSTs.

Definition 4.3.5. A TST t and an ABST b are *equivalent* iff for any string w, $I(t,w)$ equals $I(b,w)$.

Note that if t and b are equivalent, then $S(t)$ and $S(b)$ must be equal. Also, the number of digit comparisons in t and in b when searching for any w must be equal.

Lemma 4.3.6. Let S be a set of strings. If an ABST b and a TST t are built inserting the strings in S in the same order (and applying no rotations), t and b are equivalent.

Proof. The proof is by induction on the number of strings. Before any string is inserted, when searching for any string y in b and in t, results in $I(b,y)$ and $I(t,y)$ being empty (i.e., no character comparison is made). Thus, t and b are equivalent.

Before inserting the string w_{i+1} in t and in b, i strings have already been inserted in both trees, which are equivalent. Therefore, $I(t,w_{i+1})$ equals $I(b,w_{i+1})$. Let r be the leaf in t and let s be the leaf in b that is reached when searching for w_{i+1}. Let x be the length of the longest common prefix of w_{i+1} with the strings already inserted. Inserting w_{i+1} in t and in b produces the following modifications: In t a descent path of $|w_{i+1}|-x+1$ nodes is added as a child of r containing each of the characters in w starting at the $(x+1)$-th position (in sequence order), plus the special end character. In b, only one node is added as a child of s, containing w_{i+1}. In any case, given a searched string y, $I(t,y)$ and $I(b,y)$ change only if nodes r and s are reached, respectively. In particular, $I(t,y)$ and $I(b,y)$ contain one additional subsequence. Besides, the resulting new subsequence is the same in t and in b, because the same substring is reached from r and s, respectively. □

Note that an ABST has a unique equivalent TST, but a TST may have several equivalent ABSTs.

Using the previous lemmas, the following theorem relates the number of string lookups in ABSTs with properties in TSTs.

Theorem 4.3.7. Let t be a TST and let b be an equivalent ABST. Let w be any string. Then,

- The number of string lookups in b when searching for a string w coincides with the number of search descent paths in t when searching for w.
- The number of string lookups in b when searching for all the strings in b coincides with the cumulative number of search descent paths in t when searching for every string in $S(t)$.

Proof. From Corollary 4.3.2, $|I(t,w)|$ is the number of search descent paths in t for w. From Corollary 4.3.4, $|I(b,w)|$ is the number of string lookups in b when searching for w. Given that t and b are equivalent, $I(t,w)$ equals $I(b,w)$. Thus, $|I(t,w)|$ equals $|I(b,w)|$ and $\sum_{w\in S(t)}|I(t,w)|$ equals $\sum_{w\in S(b)}|I(b,w)|$. □

In particular, the number of string lookups in ABSTs is never greater than in plain BSTs. Recall that searching a string in a plain BST requires as many string lookups as nodes in the searching path. In the case of TSTs, there are no string lookups at all (because nodes contain a character instead of a string). By contrast, the number of visited nodes in TSTs, which are also accessed through pointers, can be very large. Specifically, it depends on the string length, the alphabet cardinality and the insertion order of the strings.

Now, the problem of determining the number of string lookups in ABSTs is reduced to determining the number of search descent paths in TSTs. We do so using some known results on TSTs. First, we need some additional definitions on TSTs.

Definition 4.3.8 ([17])**.** Let t be a TST and let w be a string. The *(comparison) search cost* $R(t,w)$ is the number of comparison pointers in the searching path of w in t. The *(comparison) path length* $L(t)$ is the sum of the distances of all external nodes to the root of the tree, where distance is measured as the number of comparison pointers.

An exact mathematical expression for $R(t,w)$ and $L(t)$ is given in [17, Theorem 2.1]. An asymptotic expression for $R(t,w)$ and $L(t)$ is given in [17, Theorem 2.2].

Besides, $R(t,w)$ and $L(t)$ are strongly related with the number of search descent paths. Note that $R(t,w)$ is equivalent to counting the number of search descent paths except for the last one. The following corollary gives the exact relationship.

Corollary 4.3.9. Let t be a TST and let w be a string. The number of search descent paths in t for w is equal to $R(t,w)+1$. The cumulative number of search descent paths in t when searching for every string in $S(t)$ is $L(t)+S(t)$.

Finally, from Theorem 4.3.7 and Corollary 4.3.9, we characterize in the following corollary the number of string lookups in ABSTs.

Corollary 4.3.10. Let t be a TST and let b be an equivalent ABST. The number of string lookups in b when searching for a string w is $R(t,w)+1$. The number of string lookups in b when searching for every string in $S(b)$ is $L(t)+S(t)$.

4.4 On the number of string lookups in CABSTs, an extension of ABSTs

In the following, we present *character augmented BSTs (*CABSTs*)*, a variant of ABSTs in which some redundancy is added (see Section 2.3.2 for more examples on redundancy) in order to avoid the string lookups due to binary searching for a character position. We relate the number of string lookups in CABSTs with some properties of TSTs, and in turn, with some properties of Patricia tries.

We define CABSTs as follows.

Definition 4.4.1. A CABST β is an extension of an ABST b in which a character field is added to each node. This field stores the first significant character of the string stored in the node (i.e., that at the position after the maximum common prefix). We say that β *corresponds* to b.

Comparisons are accordingly modified to take this character into account. The sequence of character comparisons, though, remains the same, and so, it can be related to TSTs analogously as ABSTs are related to TSTs.

Definition 4.4.2. A character stored in a node r in a CABST is *useful* when searching for a string w, iff it avoids from looking up the string in r.

In the following, the benefits of CABSTs when searching for a string w are precisely described taking into account the relationship between ABSTs and TSTs. We use the following definition.

Definition 4.4.3. Let t be a TST and let b be an equivalent (C)ABST(thus $I(t,w) = I(b,w)$ for any string w). Let r be a node in b and let s be a node in t. We say that r and s are *search related* if the same subsequence in $I(t,w)$ is matched in t and in b when searching for any string w.

Now, we can describe the benefits of CABSTs in terms of the previous property.

Lemma 4.4.4. Let t be a TST and let b be an equivalent CABST. Let r be a node in β search related to a node s in t when searching for w, a character stored in r is useful when searching for w iff $D(s)$ is not a proper search descent path.

Proof. If a character stored in r is useful when searching for w, the string in r does not need to be looked up. Thus, exactly one pairwise character comparison is made in r. Given that s in t is search related to r, only one character is compared in $D(s)$. This implies that $D(s)$ is not a proper search descent path.

The opposite implication is proved as follows. If $D(s)$ is a not a proper search descent path, only one character is compared in $D(s)$. Given that s is search related to r, also only one character must be compared in r. Besides, according to CABSTs definition, this character is stored in r. Given that comparing the stored character suffices, the character in r is useful. □

That is, from Lemma 4.4.4 it follows that CABSTs need at most one string lookup to determine the matching value for a character position. Specifically, the looked up string is the one with the coinciding character value (corresponding to a proper search descent path). We enunciate this relationship more precisely in the following theorem:

Theorem 4.4.5. Let t be a TST and let β be an equivalent CABST. Let w be any string. The number of proper search descent paths in the searching path of t for w coincides with the number of strings looked up in β when searching for w. The cumulative number of search descent paths in t when searching for every string in $S(t)$ coincides with the total number of strings looked up in β when searching for every string in $S(b)$.

Proof. From Theorem 4.3.7, the number of nodes in b whose string is looked up when searching for w coincides with the number of search descent paths in t. In β, that number of nodes is decremented by the number of useful nodes. According to Lemma 4.4.4, a node in β is useful if it is search related to a node s in t, s.t. $D(s)$ is not a proper search descent path. Thus, a string in a node in β is looked up in β when searching for w only if it is search related to a node s in t, s.t. $D(s)$ is a proper search descent path. Then, the theorem follows. □

However, as far as we know, the exact number of proper search descent paths in TSTs has not been analyzed. By contrast, we can give some straightforward upper bounds. The first one is obtained relating CABSTs to Patricia tries using the following fact and lemma. The fact follows from the definition of a proper descent path.

Fact 4.4.6. The number of proper descent paths in a TST t completely included in the searching path of t for any string w is greater than or equal to the number of proper search descent paths for w.

Lemma 4.4.7. The number of proper descent paths in a TST t completely included in the searching path of t for any string w corresponds with the search cost in a Patricia trie storing $S(t)$. The cumulative number of descent paths in t for searching every string in $S(t)$ corresponds with the external path length in a Patricia trie storing $S(t)$.

Proof. Let p be a Patricia trie and let q be a trie such that $S(t) = S(p) = S(q)$. Given the aforementioned equality between the sets of strings, the nodes in t where a proper descent path ends are related to the nodes in q in which two or more pointers go out from them and in turn, these are exactly the nodes in p (p has no other nodes).

Therefore, the number of descent paths traversed in t when searching for w corresponds with the search cost in p and similarly, the cumulative number of descent paths corresponds with the external path length. □

Useful references on the analysis of Patricia tries are [23, 14, 24]. Finally, from Theorem 4.4.5, Fact 4.4.6 and Lemma 4.4.7, we have the following corollary:

Corollary 4.4.8. The number of strings looked up in CABST β when searching for any string w is upper bounded by the search cost in a Patricia trie storing the same set of strings. The total number of strings looked up in β when searching for every string in $S(\beta)$ is upper bounded by the external path length of a Patricia trie storing the same set of strings.

In particular, the cost when searching in a Patricia trie is never greater than the length of the searched string as it is the case with tries. On the other hand, the number of string lookups in β and b is upper bounded by the number of string lookups in plain BSTs. Specifically, the number of string lookups in a BST of height h is $O(h)$. Putting all together, we have the following corollary:

Corollary 4.4.9. The number of string lookups in a CABST of height h when searching for any string w of length m is $O(min(m,h))$.

Finally, we note that string lookups in CABSTs could mostly be avoided for many common datasets, by storing the k first relevant digits, where $k \geq 1$ is a small constant. This way, most (if not all) memory accesses are limited to accessing to the tree nodes.

4.5 On the number of string lookups in AQSORT and AQSEL

In this section, we analyze the number of string lookups in AQSORT and AQSEL, by relating them to ABSTs. Besides, we present and analyze CAQSORT and CAQSEL. These are variants of AQSORT and AQSEL, respectively, in which some redundancy is added. They are obtained analogously as CABSTs are obtained from ABSTs.

In the following, we assume a pivot selection method that does not need to compare elements. Besides, a denotes an array of strings that is used as input for the algorithms that we analyze.

First, we formally define CAQSORT and CAQSEL.

Definition 4.5.1. CAQSORT and CAQSEL are extensions of respectively AQSORT and AQSEL, in which a character field is added per each element in a. This field stores the first significant character of the string.

Now, thanks to the relationship of quicksort and quickselect with BSTs, we characterize the number of string lookups. First, we consider (C)AQSORT.

Theorem 4.5.2. Let b be the ABST corresponding to the execution e of AQSORT on a, and let β be the CABST corresponding to the execution f of CAQSORT on a. The number of string lookups and digit comparisons during e and f coincide with the number of string lookups and digit comparisons in b and β, respectively, when searching for every string in $S(a)$.

Proof. There is an isomorphism between a quicksort execution and a plain BST built by inserting the strings as they are used as pivots. Since augmenting the BST and quicksort does not change the order in which comparisons are made, the theorem follows. □

In particular, the number of string lookups in (C)AQSORT is never greater than in plain quicksort. Recall that quicksort requires as many string lookups as comparisons (regarded as atomic).

Besides, from Corollary 4.3.10 and Theorem 4.5.2, we have the following important result:

Corollary 4.5.3. Let t be a TST equivalent to the ABST corresponding to the execution e of AQSORT on a. The number of string lookups in e is $L(t)+S(t)$.

Furthermore, the following lemma relates the number of string lookups in AQSORT and in multikey quicksort. Multikey quicksort is a sorting algorithm isomorph to a TST built by inserting the strings as they are used as pivots. See Section 2.3.1 for further details on multikey quicksort.

Lemma 4.5.4. Let t be an equivalent TST to the ABST corresponding to the execution e of AQSORT on a. Let m be an execution of multikey quicksort isomorph to t. The number of string lookups in e is never greater than the number of string lookups in m.

Proof. Given the equivalence of m and e with t, exactly the same digit comparisons must be made in m and e. In multikey quicksort, the number of string lookups coincides with the number of digit comparisons. By contrast, in (C)AQSORT the number of string lookups is at most the number of digit comparisons. Thus, the lemma follows. □

In return, multikey quicksort uses no (explicit) additional space (recall that (C)AQSORT needs linear additional space with respect to the number of elements being sorted).

Now, we consider (C)AQSEL.

Theorem 4.5.5. Let p be an ABST built by inserting the strings as they are used as pivots in the execution e of AQSEL on a, let π be a CABST built by inserting the strings as they are used as pivots in the execution f of CAQSEL on a. The number of string lookups and digit comparisons in e and f coincide with the number of string lookups and digit comparisons in p and π, respectively, when searching for every string in $S(a)$.

Proof. There is an isomorphism between a quickselect execution and the sequence of comparisons resulting from searching every string in $S(a)$ in a BST built in the same order as pivots are selected (note that the BST is actually a path). So happens with the augmented versions, because the fact of augmenting the BST and quickselect does not change the order in which comparisons are made. Thus, the theorem follows. □

In particular, the number of string lookups in (C)AQSEL is never greater than in plain quickselect. Recall that quickselect requires as many string lookups as element comparisons.

Besides, from Corollary 4.3.10 and Theorem 4.5.5, we have the following corollary:

Corollary 4.5.6. The number of string lookups in an execution of AQSEL is $\sum_{w \in S(a)} (R(t,w)+1) = n + \sum_{w \in S(a)} R(t,w)$.

It does not seem straightforward, though, to obtain a close formula for the number of string lookups in (C)AQSEL. Also, it is not possible to relate it to the number of string lookups in multikey quickselect. See Chapter 5 for a detailed description of multikey quickselect. Analogously to multikey quicksort, the number of string lookups in multikey quickselect coincides with the number of digit comparisons. However, the sequence of digit comparisons in (C)AQSEL and in multikey quickselect cannot be related. Specifically, we have seen in Theorem 4.5.5 that a (C)AQSEL execution is related to a (C)ABST built by inserting the strings as they are used as pivots, and thus, in turn, a (C)AQSEL execution can be related to a TST. By contrast, the obtained TST does not describe the sequence of digit comparisons in an execution of multikey quickselect on the same array.

4.6 Conclusions and future work

In this chapter, we have analyzed comparison-based data structures and algorithms enhanced for fast string comparisons using a cache-conscious approach. We have stressed that the outcome of some of the comparisons can be determined without looking up any string. This is relevant with regard to performance because saving string lookups saves random memory accesses, which are not handled efficiently by modern memory hierarchies.

First, we have characterized the number of string lookups in so-enhanced BSTs, which we have called ABSTs. Then, we have added some redundancy (we have called the resulting trees, CABSTs) to avoid string lookups due to binary searching when determining the next value for a character position. Doing so, the number of string lookups per search in the worst-case is also reduced. The analysis of (C)ABSTs is done relating them with some kinds of tries. Specifically, we can obtain the precise number of string lookups when searching in an ABST using already known analytical results for TSTs. For the number of string lookups when searching in a CABST, we have provided an upper bound relating TSTs in turn, this time with Patricia tries.

Additionally, we have analyzed the number of string lookups in so-enhanced comparison-based algorithms, namely quicksort and quickselect (which we have called respectively (C)AQSORT and (C)AQSEL) relating them to (C)ABSTs. However, neither the number of string lookups nor the number of digit comparisons in (C)AQSEL can be directly expressed in terms of properties of TSTs or of multikey quickselect.

The theoretical work in this chapter could be complemented with implementations and experimentation. In particular, the proposed enhancements for fast string comparisons are amenable to be coded in the top of common comparison-based STL implementations for dictionaries, sorting and selection. For instance, the practicability of implementing CABSTs on the top of a red-black tree implementation for STL dictionaries was shown in the master thesis of F. Martínez [66], directed by the author of this PhD thesis. That thesis

contributed a thorough implementation and several performance experiments. The results there support the ones in [20] for ABSTs, in which the performance is always improved with respect to plain BSTs. In particular, CABSTs obtain further considerable speedups compared to ABSTs, in particular, for big data sets. Overall, the results in [66] show that as expected, avoiding string lookups does improve the performance in practice.

The experimental work for CABSTs could also be extended to draw an experimental comparison against burst tries [48]. Burst tries have been shown to be very fast (the fastest) in practice for common data sets. Yet, as any kind of tries, they do not handle skewed strings well (in particular long strings sharing long prefixes). CABSTs should be ideally be implemented as well on the top of a cache-conscious data structure (e.g., B-trees), for the comparison.

Finally, the ideas in this chapter could be combined with parallel algorithms and data structures. We follow this approach in Chapter 9.

Chapter 5

Multikey quickselect

Selection is an important problem closely related to sorting. Given an unsorted array of n elements, a rank r between 1 and n and an order function, a selection algorithm returns the r-th element in the given order. Quickselect, the counterpart of quicksort for selection, is a widely used efficient algorithm for generic keys. For a thorough analysis of the expected number of (atomic) comparisons and swaps see, e.g., [56, 64, 65].

In this chapter, we consider the selection problem on strings using lexicographical order. Note that quickselect is not very efficient for this particular case, because it considers keys as atomic, which implies making redundant character comparisons. For the expected number of character comparisons see e.g., [100]. Note also that enhancing quickselect for strings, as shown in Chapter 4, needs extra linear space with respect to the array size. Radixselect, the counterpart of radixsort for selection (see [63] for an analysis) also requires extra space, or alternatively, two passes on the data per iteration.

We propose another algorithm: *multikey quickselect* (MKQSEL, in the following), the counterpart of multikey quicksort [9] (MKQSORT, in the following) for selection. See Section 2.3 for an overview on MKQSORT and other efficient string sorting algorithms. Analogously to MKQSORT, MKQSEL benefits from the internal representation of strings to avoid redundant comparisons. Besides, it partitions its input into three sets according to a given pivot value, and then, it recursively proceeds using a divide-and-conquer approach. As a result, it makes almost as few character comparisons as radixselect, it is in-place and it is as easy to implement as quickselect. Thus, like MKQSORT, it is expected to be rather fast in practice.

This chapter is organized as follows. In Section 5.1, we describe MKQSEL in more detail. Then, in Section 5.2, we analyze its average number of comparisons and swaps for two flavors of the selection problem. Furthermore, we propose some algorithmic enhancements that also apply to MKQSORT. We present them in Section 5.3, and we analyze them in Section 5.4. Finally, in Section 5.5, we compare all the algorithms presented in this chapter and provide some hints on implementation. Section 5.6 sums up some conclusions, and provides some hints on future work.

```
// Pre: 1 ≤ r ≤ n
int mkqsel(arrayType<stringType>& v, int n, int j, int r) {
    if (n < N) {
        Select and return the r-th smallest string of v using any algorithm.
    }
    else {
        Pick a partitioning value p (the so-called pivot).
        Ternary partition v on the j-th character w.r.t. p to form v<, v=, and v>.
        Let n<, n=, and n> be respectively the sizes of v<, v=, and v>.
        if (r ≤ n<) return mkqsel(v<, n<, j, r);
        else if (r > n< + n=) return mkqsel(v>, n>, j, r − n< − n=);
        else if (p ≠ EOS) return mkqsel(v=, n=, j + 1, r − n<);
        else return v[r];
    }
}
```

Figure 5.1: Pseudocode description for ternary MKQSEL.

5.1 MKQSEL algorithm

We call ternary MKQSEL the natural derivation of (ternary) MKQSORT for the selection problem. A pseudocode description can be found in Figure 5.1. There, v is a linear array (or subarray) with n strings. In our description, we consider the common pointer representation where the string end is marked with a special symbol (we denote this symbol by EOS) like C-like strings (i.e., `char*`). Assuming that all the strings in v are identical in the first $j-1$ characters, the algorithm returns the r-th lexicographically smallest string of v. Initially, $j = 1$.

We consider two variants of the selection problem. In the *partitioned output* variant, the final array must become binary partitioned with respect to $v[r]$, that is, the elements in $v[1..r-1]$ must be less than or equal to $v[r]$, and the elements in $v[r+1..n]$ must be greater than or equal to $v[r]$ (this coincides with the definition of STL `nth_element`, see the C++ Standard [50] for further reference). In the *only selection* variant, we do not impose any condition in the final order of the elements in the array.

Furthermore, as with sorting, the pivot can be chosen in many ways. We can pick p as the j-th character from the first (say) of the remaining strings. Alternatively, the string from which to extract p could be chosen at random. The former choice is simpler, while the latter is safer when the input is biased.

Also analogously to sorting, it might be worth in practice to turn to another selection algorithm when $n < N$ for some constant N. Our analysis is valid for any algorithm and choice of N.

We now adapt the *split-end* partitioning, presented in [8] and used in [9], to MKQSEL. The first step of this method has the invariant shown in Figure 5.2. Initially, $a = b = 1$ and $c = d = n$. While a string $v[b]$ with $v[b][j] > p$ is not found, we increment b. Then, while a string $v[c]$ with $v[c][j] < p$ is not found, we decrement c. We then swap $v[b]$ with $v[c]$. This is

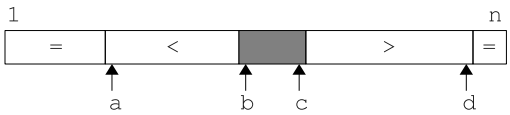


Figure 5.2: *split-end* partitioning.

done until $b < c$. Besides, whenever a string x with $x[j] = p$ is found, it is swapped towards the ends, either incrementing a (if x is on the left side) or decrementing d (otherwise).

At the beginning of the second step, the array consists of $v^{\ell}_{=}$, $v_{<}$, $v_{>}$, and $v^{r}_{=}$ in this order. What is left now is different from the sorting case, and depends on the variant of the selection problem that we are considering. In the *partitioned output* variant, $v^{\ell}_{=}$ must be moved after $v_{<}$ only when selection follows either the branch with $v_{<}$ or the branch with $v_{=}$. Symmetrically, $v^{r}_{=}$ must be moved before $v_{>}$ only when selection follows either the branch with $v_{>}$ or the branch with $v_{=}$.

In the *only selection* variant, we only ask for the value of the sought element, so less movements are required: $v^{\ell}_{=}$ and $v^{r}_{=}$ must be joined only when selection follows the branch with $v_{=}$. Besides, it is enough that the smaller subarray of the two is moved next to the other. No other moves are needed.

5.2 Analysis of ternary MKQSEL

In this section, we compute the cost of MKQSEL. Here we assume that n infinite-long strings are drawn independently from a random uniform distribution on the universe of strings Γ^{∞}, where Γ is the character alphabet and $C \geq 2$ is its cardinality. Considering infinite strings is a common technical convenience; e.g., this is used in the analysis of Ternary Search Trees (TSTs) [17], whose construction is isomorphic to MKQSORT.

Specifically, we calculate the asymptotic expected number of comparisons and swaps for the *partitioned output* and the *only selection* cases, under the usual consideration that each rank is equally likely to be selected. We assume that all swap operations have the same cost, but the analysis could be smoothly adapted to batch-swap operations [55] (batch swap operations save roughly a third of the moves for consecutive ranges by interleaving the operations). Finally, we consider that the pivot is chosen as the first element of the remaining array. Under our hypothesises, this pivot strategy leads to asymptotically equivalent costs as choosing the element randomly from the remaining array.

5.2.1 Notation

Let $T_k(n)$ denote the expected cost of a call to MKQSEL with n strings, where k is the number of possible values left for the current character after the previous calls. We thus have $1 \leq k \leq C$, and $k = C$ initially.

Let $t_k(n)$ be the so-called toll function, i.e., the non-recursive cost of $T_k(n)$. It includes the constant cost of picking the pivot, plus the cost of partitioning.

For every $0 \leq \ell \leq n$, let $P(n,\ell,p) = \binom{n}{\ell} p^{\ell}(1-p)^{n-\ell}$ be the probability that a binomial random variable with n Bernoulli trials, each with independent probability p of success, achieves exactly ℓ successes.

In the following, we use i to denote the cardinal position (starting at 0) of the value of the pivot in a subalphabet of cardinality k. For instance, if we knew from previous calls

that the current n strings could only have values from the range ['e','h'] in the current (j-th) character, then i would range from 0 to 3.

For the proofs, the following notation is useful. Assume $T_0(n) = 0$. The expected contribution of the calls to $v_<$ when the character i is chosen as pivot is

$$S_1\{T\}(i,k,n) \quad = \quad \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot T_i(\ell) \; .$$

Also, $S_1\{Y\}(i,k,n)$ would denote the above expression replacing each $T_i(\ell)$ by $Y_i(\ell)$, and $S_1\{|Z|\}(i,k,n)$ would denote the same expression replacing each $T_i(\ell)$ by $|Z_i(\ell)|$.) Similarly, for the calls to $v_>$ we have

$$S_2\{T\}(i,k,n) \quad = \quad \sum_{r=0}^{n} P(n,r,(k-i-1)/k) \cdot \frac{r}{n} \cdot T_{k-i-1}(r) \; ,$$

and for the calls to $v_=$ (which do not depend on i) we have

$$S_3\{T\}(k,n) \quad = \quad \sum_{m=0}^{n} P(n,m,1/k) \cdot \frac{m}{n} \cdot T_C(m) \; .$$

Altogether, by linearity of expectations,

$$T_k(n) \quad = \quad t_k(n) \; + \; \frac{1}{k} \sum_{i=0}^{k-1} \Big(S_1\{T\}(i,k,n) \; + \; S_2\{T\}(i,k,n) \; + \; S_3\{T\}(k,n) \Big) \; .$$

Let

$$S\{T\}(k,n) \quad = \quad S_3\{T\}(k,n) \; + \; \frac{2}{k} \sum_{i=1}^{k-1} S_1\{T\}(i,k,n) \; .$$

Then, by symmetry, we get

$$T_k(n) \quad = \quad t_k(n) \; + \; S\{T\}(k,n)$$

for every $n \geq N$, with $T_k(n)$ equal to any value for $1 \leq n < N$.

5.2.2 A recurrence for the cost of ternary MKQSEL

Let us consider the cost due to a recursive call into $v_<$ in Algorithm 5.1. Under our hypotheses, any i between 0 and $k-1$ is equally likely to be the pivot. For every $0 \leq \ell \leq n$, the probability that exactly ℓ strings have a j-th character smaller than i is $P(n,\ell,i/k)$. Moreover, the probability that the sought string belongs to $v_<$ is ℓ/n. In that case, we must perform a recursive call with $n' = \ell$ (and implicitly $k' = i$, note that Algorithm 5.1 does not know nor use k).

The cost of the second recursive call in Algorithm 5.1 is absolutely symmetrical. Regarding the third call, the probability that exactly m strings have a j-th character equal to i is $P(n,m,1/k)$, and the probability to follow that branch is m/n. In that case, we must perform a recursive call with $n' = m$ (and $k' = C$).

Putting all this together, and by linearity of the expectations, we get

$$T_k(n) \quad = \quad t_k(n) + \sum_{m=0}^{n} P\Big(n,m,\frac{1}{k}\Big) \frac{m T_C(m)}{n} + \sum_{i=0}^{k-1} \sum_{\ell=0}^{n} P\Big(n,\ell,\frac{i}{k}\Big) \frac{2\ell T_i(\ell)}{kn} \tag{5.1}$$

for every $n \geq N$, with $T_k(n)$ equal to some value for $1 \leq n < N$.

5.2.3 Toll functions

Here we list and compute the expected value of the toll functions $t_k(n)$ used in (5.1) under several cost measures. We start stating some useful properties of the resulting partitioning.

Lemma 5.2.1. Consider the pivot value at a fixed position i. The expected size of $v_<$ is $n_< = in/k$. The expected size of $v_>$ is $n_> = (k-i-1)n/k$. The expected size of $v^\ell_=$ is $n^\ell_= = in/(k(k-1))$ if $k > 1$, with $n^\ell_= = n$ for $k = 1$. The expected size of $v^r_=$ is $n^r_= = (k-i-1)n/(k(k-1))$ if $k > 1$, with $n^r_= = 0$ for $k = 1$.

Proof. The probabilities for a value to be equal to, smaller than or greater than the pivot are $1/k$, i/k and $(k-i-1)/k$ respectively, so we get the first two results. Moreover, we have $n^\ell_= = (n_< + n^\ell_=)/k$ and $n^r_= = (n_> + n^r_=)/k$. Solving the equations, we get the last two results. □

Corollary 5.2.2. Fix the pivot position to i. The expected size of $v^\ell_=$ plus $v_<$ is $n_\le = in/(k-1)$ if $k > 1$, with $n_\le = n$ for $k = 1$. The expected size of $v^\ell_=$ plus $v^r_=$ is $n_= = n/k$.

Each element is compared exactly once in partitioning. The high-level description of the partitioning method uses ternary character comparisons, i.e., the outcome of a comparison is 'less than', 'equal to' or 'greater than'. The next fact holds for any string distribution.

Fact 5.2.3. The number of ternary character comparisons in one partitioning is $n + O(1)$.

However, in practice, ternary comparisons are usually implemented with two binary comparisons. Therefore, for the sake of completeness, we count whenever a second binary comparison would be needed after a first binary comparison. This happens in the left side when a value is found to be less than or equal to the pivot, and analogously in the right side.

Lemma 5.2.4. The expected number of *second* binary character comparisons is $c_k(n) = 2(k+1)n/(3k) + O(1)$ if $k > 1$, with $c_1(n) = n$.

Proof. Consider a fixed pivot i. *Second* binary comparisons are performed in the left side with probability $(i+1)/k$. By symmetry, and using Corollary 5.2.2, $c_k(n) = 1/k \sum_{i=0}^{k-1} 2(i+1)/k \cdot in/(k-1)$, and the lemma follows. □

The partitioning algorithm is made of two steps. Let $p_k(n)$ be the expected number of swaps of the first one, and let $v_k(n)$ be the expected number of swaps of the second one.

Lemma 5.2.5. $p_k(n) = (k+4)n/(6k) + O(1)$ if $k > 1$, with $p_1(n) = n + O(1)$.

Proof. Fix a pivot i. Each value greater than i that before partitioning had a rank less than $n\le$ causes one swap. Using Corollary 5.2.2, the expected number of swaps due to this source is $(k-i-1)/k \cdot in/(k-1)$. On the other hand, every string in $v^\ell_=$ and $v^r_=$ causes one swap. Averaging over the i's, the lemma follows. □

Note that $v_k(n)$ depends on whether *partitioned output* or *only selection* variant is considered, while $p_k(n)$ is independent of this fact.

Lemma 5.2.6. Assuming *partitioned output* variant, $v_k(n) = 2(k+1)n/(3k^2)$ if $k > 1$, with $v_1(n) = 0$.

Proof. When $k = 1$ no swap is done. Otherwise, fix a pivot i. If the $v_<$ or the $v_=$ branch is followed, which happens with probability $(i+1)/k$, each string in $v^\ell_=$ must be swapped to the middle. By symmetry with $v^r_=$, using Lemma 5.2.1 for the expected size of $v^\ell_=$, and averaging over all i's, the lemma follows. □

In the following, for any boolean expression b, let $[b]$ be the Iverson bracket for b, i.e., $[b]$ evaluates to 1 when b is true and to 0 when b is false.

Lemma 5.2.7. With *only selection* variant, $v_k(n) = n/(4k^2) - n/(4k^2(k - [k \text{ is even}]))$.

Proof. When $k = 1$ no swap is done. Otherwise, swaps are performed only when the $v_=$ branch is followed, which happens with probability $1/k$. In that case, the smallest of $v^\ell_=$ and $v^r_=$ must be moved next to the other. Fix a pivot i. If k is even, we must sum (twice by symmetry) all $n^\ell_=$ corresponding to $i = 0..k/2-1$. If k is odd, we must sum (again twice by symmetry) all $n^\ell_=$ corresponding to $i = 0..(k-3)/2$, plus just once the $n^\ell_=$ corresponding to the middle $i = (k-1)/2$. In both cases we must multiply by $1/k$, the probability of every pivot. Using the value for $n^\ell_=$ in Lemma 5.2.1 and arranging things, the lemma follows. □

Finally, putting the results in this section together, we obtain the toll functions corresponding to the expected number of swaps in one partitioning.

Corollary 5.2.8. The expected number of swaps for *partitioned output* variant is $t_k(n) = (1/6 + 4/(3k) + 2/(3k^2))n + O(1)$ if $k > 1$, with $t_1(n) = n + O(1)$.

Corollary 5.2.9. The expected number of swaps for *only selection* variant is $t_k(n) = (1/6 + 2/(3k) + 1/(4k^2) - 1/(4k^2(k - [k \text{ is even}])))n + O(1)$, with $t_1(n) = n + O(1)$.

5.2.4 Solving the recurrence for ternary MKQSEL

In this section, we solve (5.1) for several toll functions. We first present a couple of technical lemmas needed later.

Lemma 5.2.10. Suppose $t_k(n) = O(n)$ for every $1 \le k \le C$. Then, $T_k(n) = O(n)$ for every $1 \le k \le C$.

Proof. Using the hypothesis, for every $1 \le k \le C$ there exist $n_k \ge N$ and B_k large enough such that $|t_k(n)| \le B_k n$ for every $n \ge n_k$. Let $N' = \max\{2(C+1), n_1, n_2, \ldots, n_C\}$, let $B' = \max\{2k^2 B_k/(C-1) : 1 \le k \le C\}$, let $B'' = \max\{|kT_k(n)|/(Cn) : 1 \le k \le C, 1 \le n < N'\}$, and let $B = \max\{B', B''\}$. With all these definitions, we get that for all $1 \le k \le C$, if $n \ge N'$, then $|t_k(n)| \le B(C-1)n/(2k^2)$; and if $n < N'$, then $|T_k(n)| \le BCn/k$. These properties will be useful in some steps below.

Now let $Y_k(n) = |T_k(n)| - BCn/k$ for all $1 \le k \le C$. Substituting into (5.1) and using the definitions of $S\{T\}(k,n)$, $S_3\{T\}(k,n)$ and $S_1\{T\}(i,k,n)$, for $n \ge N'$ we have $Y_k(n) \le I_k(n) + S\{Y\}(k,n)$, where

$$\begin{aligned} I_k(n) &= B(C-1)n/(2k^2) + \sum_{m=0}^{n} P(n,m,1/k) \cdot \frac{m}{n} \cdot BCm/C \\ &\quad + \frac{2}{k} \sum_{i=1}^{k-1} \sum_{\ell=0}^{n} P(n,\ell,i/k) \cdot \frac{\ell}{n} \cdot BC\ell/i - BCn/k\,. \end{aligned}$$

Therefore,

$$\begin{aligned} 2k^2 I_k(n)/B &= (C-1)n + \frac{2k^2}{n}\sum_{m=0}^{n} P(n,m,1/k)\cdot m^2 \\ &\quad + \frac{4Ck}{n}\sum_{i=1}^{k-1}\frac{1}{i}\sum_{\ell=0}^{n} P(n,\ell,i/k)\cdot \ell^2 - 2Ckn\,. \end{aligned}$$

For the next step, we use the well-known identity

$$\sum_{c=0}^{n} P(n,c,p)\,c^2 = pn(pn+1-p) \tag{5.2}$$

for any $0 \le p \le 1$ to get

$$\sum_{m=0}^{n} P(n,m,1/k)\cdot m^2 = \frac{n(n+k-1)}{k^2}\,,$$

and also

$$\sum_{i=1}^{k-1}\frac{1}{i}\sum_{\ell=0}^{n} P(n,\ell,i/k)\cdot \ell^2 = \sum_{i=1}^{k-1}\frac{n(in+k-i)}{k^2} = \frac{(k-1)(n+1)n}{2k}\,.$$

Putting all this together produces

$$2k^2 I_k(n)/B = (1-C)n + 2(C+1)(k-1) \le (C-1)(2(C+1)-n) \le 0\,,$$

which implies $I_k(n) \le 0$ and $Y_k(n) \le S\{Y\}(k,n)$ for every $n \ge N'$. Furthermore, we have $Y_k(n) \le 0$ for every $n < N'$. Hence, a trivial proof by induction on n shows that $Y_k(n) \le 0$ for every n, from which we deduce $|T_k(n)| \le BCn/k = O(n)$ for every n and every k. □

Lemma 5.2.11. Suppose $t_k(n) = o(n)$ for every $1 \le k \le C$. Then, $T_k(n) = o(n)$ for every $1 \le k \le C$.

Proof. For a binomial random variable $R(n,p)$ with n trials and probability p, and for every $\varepsilon > 0$, a Chernoff bound yields

$$\Pr\{R(n,p) \ge (p+\varepsilon)n\} \le e^{-\varepsilon^2 n/2}\,.$$

For every $1 \le k \le C$ and every $1 \le i < k$, the probabilities in the expression for $S_1\{T\}(i,k,n)$ are those for $R(n,i/k)$. Let $f = 1-1/(2C)$. Substituting above with $\varepsilon = f-p$, we get the bound $\Pr\{R(n,p) \ge fn\} \le e^{-(f-p)^2 n/2} \le e^{-n/(8C^2)}$.

On the other hand, for every i we can split the recurrence for $S_1\{|T|\}(i,k,n)$ into two sets of recursive calls, like this:

$$\begin{aligned} S_1\{|T|\}(i,k,n) &= \sum_{0\le\ell\le fn} P(n,\ell,i/k)\cdot\frac{\ell}{n}\cdot|T_i(\ell)| \\ &\quad + \sum_{fn<\ell\le n} P(n,\ell,i/k)\cdot\frac{\ell}{n}\cdot|T_i(\ell)|\,. \end{aligned}$$

Taking into account that $\ell \leq n$, and $|T_i(\ell)| = O(n)$ (by Lemma 5.2.10), the second sum above can be bounded by $e^{-n/(8C^2)}O(n) = o(n)$. The same bound can be proved for the recursive calls to $|T_C(m)|$ with $m \geq fn$ in the expression for $S_3\{|T|\}(k,n)$.

Now, define $M(n) = \max\{|T_k(n)| : 1 \leq k \leq C\}$. Let a_n be any index between 1 and fn where $M(a_n)$ is maximum. Then

$$\begin{aligned} M(n) &\leq \max\{\ |t_k(n)| + S\{|T|\}(k,n) : 1 \leq k \leq C\ \} \\ &\leq o(n) + M(a_n) \max_{1\leq k\leq C} \left\{ \sum_{0\leq m< fn} P(n,m,1/k)\cdot \frac{m}{n} \right. \\ &\qquad \left. + \frac{2}{k}\sum_{i=1}^{k-1} \sum_{0\leq \ell< fn} P(n,\ell,i/k)\cdot \frac{\ell}{n} \right\}, \end{aligned}$$

because $|T_C(m)| \leq M(m) \leq M(a_n)$ and also $|T_i(\ell)| \leq M(\ell) \leq M(a_n)$. Taking into account that the sum of weights for each k is bounded by 1, we get $M(n) \leq o(n) + M(a_n)$. Now, the solution to the recurrence $M(n) = o(n) + M(a_n)$ is $M(n) = o(n)$ (see e.g., [80, Theorem 5.3 and Lemma 5.4]). Since $|T_k(n)| \leq M(n)$, this finishes the proof. □

Lemma 5.2.12. Suppose $t_k(n) = f_k \cdot n + o(n)$ for every $1 \leq k \leq C$. Then, $T_k(n) = A_k \cdot n + o(n)$ for every $1 \leq k \leq C$, where A_k is defined by

$$A_k = f_k + \frac{A_C}{k^2} + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 A_i \,. \tag{5.3}$$

Proof. Let $Z_k(n) = T_k(n) - A_k \cdot n$ for all $1 \leq k \leq C$. Substituting into (5.1) and using the definitions of $S\{T\}(k,n)$, $S_3\{T\}(k,n)$ and $S_1\{T\}(i,k,n)$, for $n \geq N'$ we have $Z_k(n) = J_k(n) + S\{Z\}(k,n)$, where

$$\begin{aligned} J_k(n) &= f_k \cdot n + o(n) \ + \ \sum_{m=0}^{n} P(n,m,1/k)\cdot \frac{m}{n}\cdot A_C \cdot m \\ &\quad + \ \frac{2}{k}\sum_{i=1}^{k-1}\sum_{\ell=0}^{n} P(n,\ell,i/k)\cdot \frac{\ell}{n}\cdot A_i \cdot \ell \ - \ A_k \cdot n \,. \end{aligned}$$

Using (5.2), the first sum above is equal to $A_C/k^2(n+k-1)$, and the second sum above is equal to $2/k^3 \sum_{i=1}^{k-1} i^2 A_i(n+k/i-1)$. Therefore, by the definition of the A_k's in (5.3),

$$J_k(n) = o(n) + \frac{A_C(k-1)}{k^2} + \frac{2}{k^3}\sum_{i=1}^{k-1} iA_i(k-i) = o(n) \,.$$

Thus, we have $Z_k(n) = o(n) + S\{Z\}(k,n)$, which is $o(n)$ by Lemma 5.2.11. The lemma follows. □

Lemma 5.2.13. Let $C \geq 2$ be any integer constant, and let f_k be any function over $[1..C]$. Then, the solution to (5.3) is

$$A_k = \frac{(k+1)E_k}{k^2} + \frac{(C+1)E_C}{C(C-1)k} \,, \tag{5.4}$$

where E_k is defined by the recurrence

$$E_k = \frac{k^3 f_k - (k-1)^3 f_{k-1}}{(k+1)k} + E_{k-1} \tag{5.5}$$

for $k \geq 2$, with $E_1 = f_1/2$. In particular,

$$A_C = \frac{(C+1)E_C}{C(C-1)} . \tag{5.6}$$

Proof. First, note that every A_k depends on A_C. To remove this dependence, define $B_k = A_k - A_C/k$, which yields

$$\begin{aligned} B_k &= f_k + \frac{A_C}{k^2} + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 \left(B_i + \frac{A_C}{i}\right) - \frac{A_C}{k} \\ &= f_k + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 B_i + \left(1 - k + \frac{2}{k}\sum_{i=1}^{k-1} i\right)\frac{A_C}{k^2} \\ &= f_k + \frac{2}{k^3}\sum_{i=1}^{k-1} i^2 B_i . \end{aligned}$$

This is a recurrence that can be solved by more or less standard manipulations. To begin with, define

$$D_k = k^3 B_k = k^3 f_k + 2\sum_{i=1}^{k-1} \frac{D_i}{i} .$$

Then, we have $D_1 = f_1$, and for every $k \geq 2$,

$$D_k - D_{k-1} = k^3 f_k - (k-1)^3 f_{k-1} + \frac{2D_{k-1}}{k-1} ,$$

that is,

$$D_k = k^3 f_k - (k-1)^3 f_{k-1} + \frac{(k+1)D_{k-1}}{k-1} .$$

As a last step, let

$$E_k = \frac{D_k}{(k+1)k} = \frac{k^3 f_k - (k-1)^3 f_{k-1}}{(k+1)k} + E_{k-1}$$

for $k \geq 2$, with $E_1 = f_1/2$. Now, we observe that $D_k = (k+1)kE_k$, $B_k = (k+1)E_k/k^2$, and $A_k = (k+1)E_k/k^2 + A_C/k$. Hence, we can deduce that $A_C = (C+1)E_C/C^2 + A_C/C$, which yields $A_C = (C+1)E_C/(C(C-1))$. □

Let $H_k = \sum_{1 \leq i \leq k} 1/k$ denote as usual the k-th harmonic number.

Lemma 5.2.14. Let $C \geq 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \leq k \leq C$, where α, β and γ are any constants. Then, the solution to (5.3) is

$$A_C = 3\alpha + \frac{f_1 + 38\alpha - 10\beta + 2\gamma}{3(C-1)} + \frac{6(\beta - 3\alpha)(C+1)H_C + f_1 - \alpha - \beta - \gamma}{3C(C-1)} ,$$

$A_1 = f_1 + A_C$, and

$$A_k = 3\alpha + \frac{3A_C + f_1 + 29\alpha - 10\beta + 2\gamma}{3k} + \frac{6(\beta - 3\alpha)(k+1)H_k + f_1 - \alpha - \beta - \gamma}{3k^2}$$

for $2 \le k < C$. (This last expression also holds for $k = C$.)

Proof. We use Lemma 5.2.13 with $f_k = \alpha + \beta/k + \gamma/k^2$ for every $k \ge 2$. Substituting into (5.5), we get $E_2 = (f_1 + 4\alpha + 2\beta + \gamma)/3$, and

$$\begin{aligned} E_k &= \frac{\alpha(3k^2 - 3k + 1) + \beta(2k-1) + \gamma}{(k+1)k} + E_{k-1} \\ &= 3\alpha + \frac{2\beta - 6\alpha}{k} + \frac{7\alpha - 3\beta + \gamma}{(k+1)k} + E_{k-1} \end{aligned}$$

for $k \ge 3$. Iterating, the solution of this recurrence is

$$E_k = 3\alpha(k-2) + (2\beta - 6\alpha)\left(H_k - \frac{3}{2}\right) + (7\alpha - 3\beta + \gamma)\left(\frac{1}{3} - \frac{1}{k+1}\right) + E_2 \,.$$

Note that this last equality holds for $k \ge 2$. Now, it is enough to plug this expression and the corresponding one for E_C into (5.4) and (5.6) and make some simplifications to finish the proof. □

We also need the following specific lemma to compute the expected number of swaps of *only selection* variant. Its proof is similar to that of Lemma 5.2.14.

Lemma 5.2.15. Let $C \ge 2$, $f_1 = 0$, $f_k = 1/(k^2(k-1))$ for all even $2 \le k \le C$, and $f_k = 1/k^3$ for all odd $2 \le k \le C$. Then, the solution of (5.3) for A_C is

$$A_C = \frac{4(C+1)(H_C - H_{\lfloor C/2 \rfloor}) - 2}{3C(C-1)} + \frac{(2C-1)[C \text{ is even}]}{3C(C-1)^2} - \frac{(2C+1)[C \text{ is odd}]}{3C^2(C-1)} \,.$$

Proof. Again, we use Lemma 5.2.13, this time with $f_1 = 0$, $f_k = 1/(k^2(k-1))$ for even k, and $f_k = 1/k^3$ for odd $k > 1$. Substituting into (5.5), we get $E_1 = 0$, $E_2 = 1/3$, and

$$E_k = \frac{k/(k-1) - 1}{(k+1)k} + \frac{1 - (k-2)/(k-3)}{k(k-1)} + E_{k-2}$$

for even $k > 2$. From this, and after a lengthy manipulation, we can get

$$E_k = \frac{4(H_k - H_{k/2}) - 2}{3} + \frac{2k-1}{3(k+1)(k-1)}$$

for even $k > 2$. (Alternatively, this can be proved by induction.) Now, for odd $k > 1$, it is enough to use

$$E_k = \frac{1 - (k-1)/(k-2)}{(k+1)k} + E_{k-1}$$

and simplify the resulting expression to get

$$E_k = \frac{4(H_k - H_{\lfloor k/2 \rfloor}) - 2}{3} - \frac{2k+1}{3(k+1)k} \,.$$

From here, it is easy to obtain the result claimed for A_C. □

Finally, the following theorem characterizes the cost of ternary MKQSEL putting together the previous results.

Theorem 5.2.16. The cost of ternary MKQSEL is described by the following statements:

- The expected number of ternary comparisons is:
  $(3+\frac{13}{(C-1)}-\frac{6(C+1)H_C}{C(C-1)})n+o(n)$
- The expected number of *second* binary comparisons is:
  $(2+\frac{59}{9(C-1)}-\frac{24(C+1)H_C+1}{9C(C-1)})n+o(n)$
- The expected number of swaps considering the *partitioned output* variant is:
  $(\frac{1}{2}-\frac{14}{9(C-1)}+\frac{30(C+1)H_C-7}{18C(C-1)})n+o(n)$
- The expected number of swaps considering the *only selection* variant is:
  $(\frac{1}{2}+\frac{7}{18(C-1)}+\frac{4(C+1)H_{\lfloor C/2\rfloor}+3}{12C(C-1)}-\frac{(2C-1)[C \text{ is even}]}{12C(C-1)^2}+\frac{(2C+1)[C \text{ is odd}]}{12C^2(C-1)})n+o(n)$

Proof. $t_k(n)$ is given in Fact 5.2.3, Lemma 5.2.4 and Corollaries 5.2.8 and 5.2.9. In all cases, $t_k(n)$ is as required in Lemma 5.2.12. The resulting f_k's are of the form in Lemmas 5.2.14 or 5.2.15. Combining everything, the theorem follows. □

5.3 MKQSEL algorithm revisited

In the previous analyses, k is the cardinality of the remaining alphabet for the current character. But Algorithm 5.1 does not actually use k in the algorithm. In this section, we show how multikey algorithms can benefit from the information of the value of k.

Let the global constants F and L be respectively the first and last values of the alphabet. Thus, $C = L - F + 1$. To keep track of k, we add two parameters, f and l, which are respectively the first and last possible alphabetic values for the j-th character. This way, we have $k = l - f + 1$. Initially, $f = F$ and $l = L$.

The recursive calls are modified as follows. If the $v_<$ branch is followed, then $f' = f$ and $l' = p - 1$. If the $v_>$ branch is followed, then $f' = p + 1$ and $l' = l$. If the $v_=$ branch is followed, then $f' = F$ and $l' = L$.

5.3.1 Specializations for small k

When $k = 1$, all the strings are equal with respect to the current character. Thus, we can avoid partitioning and directly proceed by the $v_=$ branch. This roughly saves n redundant ternary comparisons, and reduces the A_C term in Lemma 5.2.12 by $\frac{C+1}{3C(C-1)}$. So, as intuition suggests, this specialization is of special interest when C is small. Note that this improvement also applies to MKQSORT.

When $k = 2$, one of the subarrays produced by the ternary partition will be empty. Therefore, using a binary exchange partition [67] would reduce the cost. But instead of computing the savings of this enhancement, we propose another algorithm for selecting strings which, in some way, directly incorporates those specific improvements. Certainly, ternary partitioning can be replaced by binary partitioning when k is known.

```
// Pre: 1 ≤ r ≤ n
int mkqsel_binary(arrayType<stringType>& v, int n, int j, int r, int f, int l) {
    if (n < N) {
        Select and return the r-th smallest string of v using any algorithm.
    }
    else if (l − f ≥ 0 and there exists a partitioning value p > f) {
        Binary partition v on the j-th character w.r.t. p to form v< and v≥.
        Let n< and n≥ be respectively the sizes of v< and v≥.
        if (r ≤ n<) return mkqsel_binary(v<, n<, j, r, f, p − 1)
        else return mkqsel_binary(v≥, n≥, j, r − n<, p, l);
    }
    else if (p ≠ EOS) return mkqsel_binary(v, n, j + 1, r, F, L);
    else return v[r];
}
```

Figure 5.3: Pseudocode description for binary MKQSEL.

5.3.2 Binary MKQSEL and MKQSORT

Algorithm progress in MKQSEL and MKQSORT is guaranteed trivially by using ternary partitioning (i.e., there is at least one element equal to the pivot value). In return, ternary partitioning needs to swap some strings twice (those in $v_=$) and the comparisons are more expensive than with binary partitioning.

As stated above, ternary partitioning can be replaced by binary partitioning when k is known. We call the resulting algorithms *binary* MKQSEL (see Algorithm 5.3) and *binary* MKQSORT. Algorithm progress is guaranteed by choosing a pivot $p > f$, and incrementing j when $k = 1$.

A possible pivot selection algorithm is: scan the array from left to right, until a value greater than f is found. If it is not found, proceed as when $k = 1$. Otherwise, partition the unexamined part of the array, as the strings that have been discarded as pivots are already in a correct place. Note that binary MKQSEL, as it is, produces a binary partitioned output.

5.4 Analysis of binary MKQSEL

In this section, we analyze binary MKQSEL using very similar steps to those in Section 5.2. Therefore, we only point out the main differences.

Let $X_k(n)$ denote the expected cost of a call to binary MKQSEL with n elements, where k is the current number of possible characters. Let $x_k(n)$ be the toll function for $X_k(n)$. To start with, $X_1(n) = X_C(n)$. For $k \geq 2$, any i between 1 and $k-1$ is equally likely to be the pivot. A call into $v_<$ will have $k' = i$, and a call into $v_\geq$ will have $k' = k - i$. Taking into account symmetry, this time we get the recurrence

$$X_k(n) \quad = \quad x_k(n) + \sum_{m=0}^{n} P\left(n, m, \frac{1}{k}\right) \frac{2mX_C(m)}{(k-1)n} + \sum_{i=2}^{k-1} \sum_{\ell=0}^{n} P\left(n, \ell, \frac{i}{k}\right) \frac{2\ell X_i(\ell)}{(k-1)n} \tag{5.7}$$

for $2 \leq k \leq C$ and every $n \geq N$, with $X_k(n)$ equal to some value for $1 \leq n < N$.

5.4.1 Toll functions

Here, we compute the expected value of the toll functions $x_k(n)$ used in (5.7). Note that they hold for $k > 1$. As for ternary partitioning, the following fact holds for any string distribution and pivot picking method.

Fact 5.4.1. The number of (binary) character comparisons in one partitioning is $n + O(1)$.

Lemma 5.4.2. The expected number of swaps is $x_k(n) = (k+1)n/(6k) + O(1)$.

Proof. Consider a fixed pivot $i > 0$. Each element that before partitioning had a rank less than $n_<$ and it is greater or equal than i causes one swap. The probability of this event is $(k-i)/k$. Besides, the probability that an element is smaller than i is i/k, so $n_< = in/k$. As a result, the expected number of swaps is $(k-i)in/k^2$. Averaging over all i's, the lemma follows. □

5.4.2 Solving the recurrence for binary MKQSEL

Here we solve (5.7) for several toll functions. The proofs of the lemmas for the cost of binary MKQSEL are very similar, in fact, a bit easier than the corresponding proofs for the cost of ternary MKQSEL (see Section 5.2). Therefore, we only give (or sketch) the few proofs that have some interest.

Lemma 5.4.3. Suppose $x_k(n) = o(n)$ for every $1 \le k \le C$. Then, $X_k(n) = o(n)$ for every $1 \le k \le C$.

Lemma 5.4.4. Suppose $x_k(n) = f_k \cdot n + o(n)$ for every $1 \le k \le C$. Then, $X_k(n) = A_k \cdot n + o(n)$ for every $1 \le k \le C$, where A_k is defined by

$$A_k = f_k + \frac{2A_C}{k^2(k-1)} + \frac{2}{k^2(k-1)} \sum_{i=2}^{k-1} i^2 A_i \tag{5.8}$$

for $k \ge 2$, with $A_1 = A_C$.

Lemma 5.4.5. Let $C \ge 2$ be any integer constant, and let f_k be any function over $[2..C]$. Then, the solution to (5.8) is $A_C = E_C/(C-1)$, where E_k is defined by the recurrence

$$E_k = k f_k - \frac{(k-1)(k-2) f_{k-1}}{k} + E_{k-1} \tag{5.9}$$

for $k \ge 3$, with $E_2 = 2 f_2$.

Proof. We follow exactly the same steps as in Lemma 5.2.13, this time with $B_k = A_k - A_C/k$, which yields $B_k = f_k + \frac{2}{k^2(k-1)} \sum_{i=2}^{k-1} i^2 B_i$; $D_k = k^2(k-1)B_k$, which yields $D_k = k^2(k-1) f_k - (k-1)^2(k-2) f_{k-1} + kD_{k-1}/(k-2)$; and $E_k = D_k/(k(k-1))$, which yields (5.9). □

Lemma 5.4.6. Let $C \ge 2$, and let $f_k = \alpha + \beta/k + \gamma/k^2$ for every $2 \le k \le C$, where α, β and γ are any constants. Then, the solution of (5.8) is

$$A_C = 3\alpha + \frac{2(\beta - \alpha)(H_C - 1)}{C-1} + \frac{\gamma}{C},$$

and

$$A_k = 3\alpha + \frac{2(\beta-\alpha)H_k - \alpha - 2\beta}{k} + \frac{\gamma(k-1)}{k^2} + \frac{A_C}{k}$$

for $2 \leq k < C$. (This last expression also holds for $k = C$.)

Proof. We use Lemma 5.4.5 with $f_k = \alpha + \beta/k + \gamma/k^2$ for every $k \geq 2$. Substituting into (5.9), we get $E_2 = 2\alpha + \beta + \gamma/2$, and

$$E_k = 3\alpha + \frac{2\beta - 2\alpha}{k} + \frac{\gamma}{k(k-1)} + E_{k-1}$$

for $k \geq 3$. Iterating, the solution of this recurrence is

$$E_k = 3\alpha(k-2) + (2\beta - 2\alpha)\left(H_k - \frac{3}{2}\right) + \gamma\left(\frac{1}{2} - \frac{1}{k}\right) + E_2 \,.$$

From this, it is easy to obtain the values for A_k. □

Theorem 5.4.7. On the average, binary MKQSEL performs $n/2 + o(n)$ swaps and $(3 - \frac{2(H_C-1)}{C-1})n + o(n)$ comparisons.

5.5 Comparison of MKQSEL algorithms and implementation issues

The analysis presented in this chapter, for uniformly random inputs, allow us to quantitatively compare ternary and binary MKQSEL between them, and also against other algorithms. Figure 5.4 draws this comparison in graphical form. We also include the results for the number of character comparisons in binary quickselect (in particular, we use the expression in [100, Theorem 2 and Figure 1]).

First, we consider the number of character comparisons. As Figure 5.4 (left) shows, both the average number of ternary comparisons (regarded as atomic) in ternary MKQSEL and of binary comparisons in binary MKQSEL are smaller than $3n$, and tend to it when the alphabet cardinality grows. Indeed, $3n$ is also the average number of *element* comparisons in quickselect (see e.g., [64] for an analysis). Also, not surprisingly, the average number of character comparisons in quickselect is a decreasing function with respect to the alphabet cardinality. In quickselect, the number of character comparison is directly related to the length of common prefixes. Indeed, as the alphabet cardinality grows, the length of common prefixes becomes smaller and smaller, resulting for large alphabets in the number of character comparisons tending to the number of element comparisons, that is, $3n$.

With respect to swaps, as Figure 5.4 (right) shows, binary MKQSEL is clearly advantageous. This is specially the case for small alphabet cardinalities and when a partitioned output is required. The average number of swaps, for all variants, tends to $n/2$ when the alphabet cardinality grows. Analogously to comparisons, this coincides with the average number of swaps for quickselect (see e.g., [65] for an analysis).

The computations presented in this chapter assume a uniformly random input. But arguably, other sources should be considered, in order to make a better comparison between

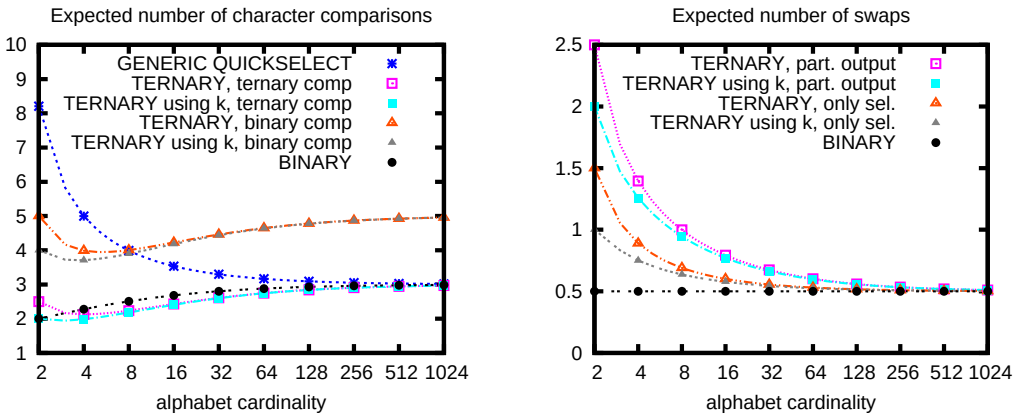


Figure 5.4: Summary of the analysis results for MKQSEL (the values are scaled to n).

selection algorithms for strings. However, note that a random source is among the less advantageous models for our algorithms, which are designed to adapt well to the existence of long common prefixes in many common real datasets.

Furthermore, some of our algorithms need the alphabet cardinality to compute k, i.e., the cardinality of the remaining alphabet for the current character. When the alphabet cardinality is not known, an upper bound can always be deduced from the value data type. As actual pivots are obtained from the input, the algorithm progress is guaranteed. The number of comparisons and swaps in this case would be only slightly higher (unless the alphabet cardinality is very small) than the one computed in this chapter.

Also, alternative algorithms can be described reconsidering the pivot choice. In our algorithms, it is picked at random from the input. By contrast, we could ignore the actual input and pick a randomly chosen character from the current range. Under our source model, this method would have the same cost. However, it would not be robust for general inputs. In fact, if the source was known to be random, better (but ad-hoc) selection algorithms for strings should be used.

Finally, we provide C++ implementations of the algorithms presented in this chapter following from the MKQSORT implementation in [9].

5.6 Conclusions and future work

We have presented multikey quickselect, the analog to multikey quicksort for the selection problem. Similarly to multikey quicksort, multikey quickselect combines radix access with quickselect-like partitioning. As a result, it performs efficient string comparisons, it is in-place, and it is easy to implement. We have described several variants of the main algorithm, including ternary and binary partitioning. Also, we have provided a detailed analysis of those variants for the expected number of comparisons and swaps for uniformly random inputs. In general, binary algorithms perform fewer binary comparisons and fewer swaps than the original (ternary) variant. We have also shown how to specialize the basic algorithms to avoid useless operations. Overall, our results suggest that ternary partition should be replaced by binary partition when the character cardinality is known and it is small.

Multikey quicksort could also take advantage of the ideas presented in this chapter. Besides, the analysis for the number of swaps is also missing for the original description of (ternary) multikey quicksort in [9], and the specialized ternary variant for many equal elements in [55].

Our implementation of multikey algorithms could be the starting point of an experimental analysis. In particular, it would be worth to compare all the variants of multikey quicksort and multikey quickselect, as well as other algorithms. For instance, in the case of selection, radixselect and (C)AQSEL (presented in Chapter 4) would be worth considering. However, as far as we know, no implementations of radixselect are available. Making a thorough experimental comparison on string selection would be a contribution in itself. We expect multikey quickselect to be rather competitive in practice, as multikey quicksort is for sorting. In the case of sorting, all the variants of multikey quicksort, radixsort (see e.g., [54]), Burstsort [87] and (C)AQSORT would be worth considering. Moreover, multikey quicksort and the binary partitioning variant of multikey quickselect could be used as an specialization for STL `sort` and `nth_element`, respectively.

Furthermore, redundancy could be added to multikey quickselect and multikey quicksort to boost their cache efficiency. Indeed, adding redundancy to radixsort has been showed to be useful in practice in [54]. Also, we have analyzed its advantages when combined with enhanced comparison-based algorithms and data structures for strings (see Chapter 4). In addition, cache efficiency could be improved by considering a superalphabet (i.e., considering several symbols at once).

Finally, it would be worth devising parallel versions of multikey quicksort and quickselect, and compare them experimentally against the parallel counterparts of (C)AQSORT and (C)AQSEL (see Chapter 9). In [77], multikey quicksort is used as a building block of a parallel sorting algorithm, but multikey quicksort itself is not parallelized. Anyway, parallelizing the divide-and-conquer work in multikey quicksort is rather straightforward. The challenge is to parallelize the partition. Besides, the latter is the only way to parallelize multikey quickselect. In this thesis, we review, but also present, several efficient parallel partitioning algorithms. All of them concern binary partition, in fact, we are not aware of parallel ternary ones. The study of parallel ternary partitioning algorithms would be interesting from a practical and theoretical perspective. But without parallel ternary partitioning algorithms, our binary partitioning variants of multikey quicksort and multikey quickselect seem the most plausible for eventual parallelization.

Chapter 6

Parallelization of bulk operations for STL dictionaries

In this chapter, we consider the parallelization of red-black tree based STL dictionaries, focusing on current widely available multi-core architectures.

Red-black trees are a kind of balanced binary search trees commonly used to implement STL dictionaries. In particular, they are used in the STL implementation of the GCC compiler. Fine-grained basic operations, i. e., insertion, query and deletion of a single element, are not worth parallelizing because parallelization overhead would dominate their logarithmic running time. But often, many elements are inserted at a time, or a new dictionary is constructed and initialized by a large sets of elements. Of course, the latter is a special case of the former, namely inserting into an empty dictionary. Nevertheless, we treat both cases explicitly, for best performance.

Specifically, we propose new parallel bulk construction and insertion algorithms for red-black trees. The implementation is done extending the GCC implementation, using OpenMP and is part of the MCSTL [86], a parallel implementation of the STL. See Section 2.4.3 for an overview on the MCSTL and other STL implementations. Our implementation is generic, comprises the four dictionary types available in the STL, and it is engineered to provide best performance for a variety of possible input characteristics. Besides, the performance measurements show their practicability on multi-core computers.

This chapter is organized as follows. We describe the proposed algorithms in Section 6.1, relating to previous work. Section 6.2 considers the special requirements for STL compliance, and gives rationale for the taken decisions. We state the results of our experiments in Section 6.3 and provide an analysis. Finally, Section 6.4 sums up some conclusions.

6.1 Algorithms

Red-black trees [46] are balanced binary search trees whose nodes can either be colored red or black, the root always being black. The following invariant is maintained at all times: Each path from the root to a leaf must contain the same number of black nodes. This limits the tree height to $2\lceil \log n \rceil$. By performing so-called *rotations*, the invariant can efficiently

be held up for all operations.

Several pieces of work describe parallel algorithms for red-black trees. On the one hand, parallel red-black tree algorithms have been proposed for the PRAM model (see e.g., [75, 70]). They are highly theoretical, use fine-grained pipelining etc., and are thus not suitable for multi-core or alike processors. Nonetheless, some of the high-level ideas can be transferred to practical algorithms. On the other hand, the STAPL [4] library provides a parallel implementation of red-black trees. However, its implementation is based on a quite rigid partitioning of the tree (mostly meant for distributed-memory systems) that can lead to all the elements being inserted by one processor, in the worst-case.

In contrast, we present a practical implementation for multi-core computers. We implement parallel bulk operations for red-black trees on the top of a sequential data structure core, which stays unaffected. Our partitioning of the data is temporary for one operation and flexible, e. g., multiple threads can work on a relatively small part of the tree. In particular, the cost of splitting the tree is negligible compared to the cost of creating/inserting the elements into the tree, for the common case.

Bulk construction can be seen as a special case of insertion. Indeed, we treat it the other way around here. Bulk construction (of a subtree) is the base case for bulk insertion. This way, we also get good performance for the case where many elements are inserted between two already contained elements (or conceptually, $-\infty$ and $+\infty$ for bulk construction).

The rest of this section is organized as follows. First, in Section 6.1.1 we present the common setup for both bulk insertion and construction operations. Then, in Sections 6.1.2 and 6.1.3, we present the parallel bulk construction and insertion algorithms, respectively. Later, we analyze them in Section 6.1.4. Finally, we show in Section 6.1.5 how to enhance the parallel bulk insertion algorithm taking advantage of load-balancing techniques.

In the following, our description focus in the `set` container, as it is one of the more general and more complicated ones, at the same time. In Section 6.2, we give hints for the rest of containers. We use the following notation: p denotes the number of used threads, n denotes the size of the existing tree, and k denotes the number of elements to insert or to construct the tree from, respectively.

6.1.1 Common setup

We present the common setup for both bulk insertion and construction operations, i. e., preprocessing the data to enable parallelization, and allocating node objects.

Preprocessing for parallelization

The first step is to make sure the input sequence S is sorted, which also is a precondition for the PRAM algorithms in [75]. If two wrongly ordered elements are found, checking the order is aborted, and the input sequence is sorted stably. Sorting stably is needed for unique dictionaries (i.e., `set` and `map`) to keep up with the operations semantics in the C++ Standard [50]. Sorting is done using the parallel sorting algorithms in the MCSTL. In particular, because the actual input sequence must not be modified, we need linear temporary storage here, namely an array.

Then, the resulting sorted sequence S' is divided into p subsequences of (almost) equal size. If S' is a random access sequence (in particular, it is so if it has been obtained from

sorting S), diving the sequence is straightforward. Otherwise, we use the SINGLEPASS algorithm presented in Chapter 7 to partition sequences whose size is unknown and random access is not provided. In fact, the required processing for the partitioning is done whilst the sequence is checked for being sorted.

Allocation and initialization

Each thread t allocates and constructs the nodes for its subsequence S_t, which are still unlinked. The tree nodes are stored in an array of pointers, shared among the threads. This allows the algorithm to easily locate the nodes for linking, later on. If we wanted to avoid the array of pointers, we would need a purely recursive algorithm where allocation and initialization of the pointers are intertwined.

We also deal with equal elements in this step. In parallel, the surplus elements are removed from the sequence. A gap might emerge at the end of each S_t. Since we use stable sorting, we can guarantee that the first equal element becomes the one inserted, as demanded by the STL specification.

6.1.2 Parallel tree construction

Once the common setup has been performed, the tree is constructed as a whole by setting the pointers and the color of each node, in parallel. For each node, its pointers can be calculated independently, using index arithmetic. This takes into account the gaps stemming from removed equal elements, as well as the incompleteness of the tree. Each node is only written to by one thread, there is no need for synchronization. Thus, this is perfectly parallelized.

The algorithm constructs a tree that is complete except for the last level, which is filled partially from left to right. The last level is colored red, all other nodes are colored black.[1] Another option would be to also color every ℓ^{th} level red. The smaller ℓ, the more favored are deletions in the subsequent usage of the dictionary. The larger ℓ, the more favored are insertions.

6.1.3 Parallel bulk insertion

The problem of inserting elements into an existing tree is much more involved than construction. The major question is how to achieve a good load balance between the threads. There are essentially two ways to divide the work into input subsequences S_t and corresponding subtrees T_t.

1. We divide the tree according to the input sequence, i. e., we take elements from the sequence and split the tree into corresponding subtrees.
2. We divide the input sequence according to the tree, i. e., we split S_t by the current root element, recursively.

Both approaches fail if very many elements are inserted at the end of the path to the same leaf. The first approach gives guarantees only on $|S_t|$, but not on $|T_t|$. The second approach does not guarantee anything about $|S_t|$, and bounds $|T_t|$ only very weakly: One of the subtrees might have twice the height as the other, so only $|T_1| < |T_2|^2$ holds. We use the first

[1] We could omit this if the tree is complete, but this is a rare special case. For a tree containing only one element, the root node is colored black anyway.

option in the initial step, and the second approach to compensate for greatly different $|T_t|$ dynamically.

The split and concatenate operations on red-black trees [96, 103] are the major tools in our parallelization of the bulk insertion operation. Specifically, we consider the following operations.

Split into Multiple Subtrees. A red-black tree is split into p red-black trees, such that the elements will be distributed according to $p-1$ pivots.

Concatenate. Two range-disjoint red-black trees and a node "in-between" are concatenated to form a single red-black tree. Rebalancing might be necessary.

The whole algorithm consists of four phases. For coordinating the work, *insertion tasks* and *concatenation tasks* are generated, and processed at a later stage.

Phase 0. Common setup, as described above.

Phase 1. Split the tree into p subtrees, the leftmost elements of the subsequences (except the first one) acting as splitters. This is performed in a top-down fashion, traversing at most $p-1$ paths, and generating $p-1$ concatenation tasks. With each concatenation task, we associate a node which is greater than all elements in the left subtree, but smaller than all elements in the right subtree. The algorithm chooses either the greatest element from the left subtree, or the least element from the subsequence to insert. This node will be used as tentative new root when concatenating the subtrees again, but might end up in another place, due to rebalancing rotations. The resulting subtrees and the corresponding subsequences form p independent insertion tasks, one for each thread.

Phase 2. Process the insertion tasks. Each thread inserts the elements of its subsequence into its subtree, using an advanced sequential bulk insertion algorithm together with load-balancing techniques (see Section 6.1.5).

Phase 3. Process the concatenation tasks to rebuild and rebalance the tree. This phase is not actually temporally separated from Phase 2, but only conceptually. As soon as one thread has finished processing an insertion task, it tries to process its parent concatenation task, recursively. However, this can only happen if the sibling subtask is also already done. Otherwise, the thread quits from this task and continues with another. It takes that one from its local queue or steals from another thread, if the own queue is empty. The root concatenation task does not have any parent, so the algorithm terminates after having processed it.

6.1.4 Analysis

We analyze now the parallel time complexity of our algorithms. Assume for simplicity that the input sequence has already been preprocessed. Besides, as in the C++ Standard, we consider each element as atomic (i.e., the construction of one element has constant cost). To get rid of lower-order terms, we assume the number of elements in the tree being relatively large with respect to the number of threads. Specifically, we assume that $n > p^2$.

Our construction algorithm takes $O(k/p)$ parallel time because each thread is responsible for constructing $O(k/p)$ elements and the construction of each element is independent.

Calculating the worst-case cost for the bulk insertion algorithm is more involved. Splitting the tree sequentially into p parts takes $O(p \log n)$ time. Note that, in the worst-case, one of the partitions contains a tree of size (almost) n, and the others contain (almost) empty trees. Splitting the tree in parallel using a divide and conquer approach takes $O(\log p \log n)$ parallel time. However, preliminary experiments showed that, our parallel algorithm was not faster than the sequential one (in our experiments, $p = 8$, which is relatively small), so we resorted to the sequential version.

Inserting a subsequence of size k/p sequentially into a tree of constant size takes $O(k/p)$ time. Inserting a sequence of size k/p sequentially into a tree of size n is upper bounded by the cost of inserting the elements one by one, i. e., $O(\frac{k}{p} \log n)$. Therefore, doing k insertions in p disjoint trees takes $O(\frac{k}{p} \log n)$ parallel time.

Finally, the p trees must be concatenated. Note that the concatenation operation itself is done sequentially but concatenations of disjoint trees will happen in parallel. A concatenation takes $O(\log \frac{n_1}{n_2})$ sequential time, where n_1 is the size of the larger tree and n_2 is the size of the smaller subtree. Besides, the trees to be concatenated can differ in size by at most n elements. Therefore, one concatenation takes at most $O(\log n)$ time. Given that there are $O(\log p)$ levels of concatenations in total, the cost of concatenating all the trees is $O(\log p \log n)$.

As a result, the total cost of the operation is dominated by the insertion itself and therefore, it takes $O(k/p \log n)$ parallel time.

6.1.5 Adding dynamic load-balancing

The sequence of the elements to insert is perfectly divided among the threads. However, the corresponding subtrees might have very different sizes, which of course affects (wall-clock) running time negatively. Even worse, it is impossible to predict how elaborate the insertion will be, since this depends on the structure of the tree and the sequence. To counter this problem, we add *dynamic load-balancing* to Phase 2, using the *randomized work-stealing* [12] paradigm.

Each thread breaks down its principal insertion task into smaller ones. It goes down the subtree and divides the subsequence recursively, with respect to the current root. For each division, it creates the appropriate concatenation task, in order to reestablish the tree in Phase 3. The insertion task for the right subtree is pushed to the thread's work-stealing queue, while the recursion continues on the left subtree. However, the queue is only used if S_t is still longer than a threshold s. Otherwise, the algorithm works on the problem in isolation (i.e., sequentially), to avoid the overhead of maintaining the queue. When there is nothing left to split, the bulk construction method is called for the remaining elements, and the new tree is concatenated to the leaf. If the subsequence has only a few elements left, the single-element insertion algorithm is called. As a side-effect, this gives us a more efficient sequential bulk-insertion algorithm, for s conceptually set to ∞.

To implement work-stealing, we use the lock-free double-ended queue provided by the MCSTL. It allows efficient synchronization of the work using hardware-supported atomic operations. On the downside, its size must be limited at construction time. Fortunately, the

size of all queues is bounded by the height of the tree, namely $2\lceil \log n \rceil$. As a whole, the approach is similar to the parallelized quicksort in the MCSTL [86, p. 6].

6.2 Interface and implementation aspects

We provide an implementation for the four dictionary types in the STL: `set`, `map`, `multiset` and `multimap`. All of them are parameterized by a `Key` type, which must always be provided, and `Comparator` and `Allocator` types, which are optional. In the following, we discuss implementation issues regarding the choice of dictionary type and its parameters.

6.2.1 (multi)set or (multi)map

The elements in `set` and `multiset` only consist of the dictionary key, whilst the elements in `map` and `multimap` consist of both the dictionary key and some additional data (the latter dictionary types are additionally parameterized by a `Data` type). By joining the dictionary key and its associated data into an appropriate helper data structure, the set data types can easily be generalized to the map data types. This approach is also taken in the libstdc++. However, some optimizations could be applied if we split the dictionary key from the additional data.

6.2.2 set/map or multiset/multimap

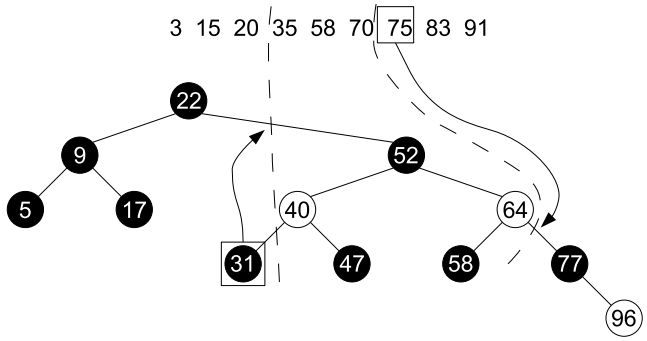
Our algorithms are built on the assumption of unique dictionaries, which is actually the most complicated case. This is extended for non unique dictionaries by just introducing a new comparator object `less_equal` and using it when appropriate. The new comparator behaves as the class comparator for unique dictionaries while for non unique dictionaries works as follows. If `less` is the class comparator we define `less_equal` as: `less_equal(a,b) = not(less(b,a))`. That is, it also returns true for equal elements.

6.2.3 Memory management

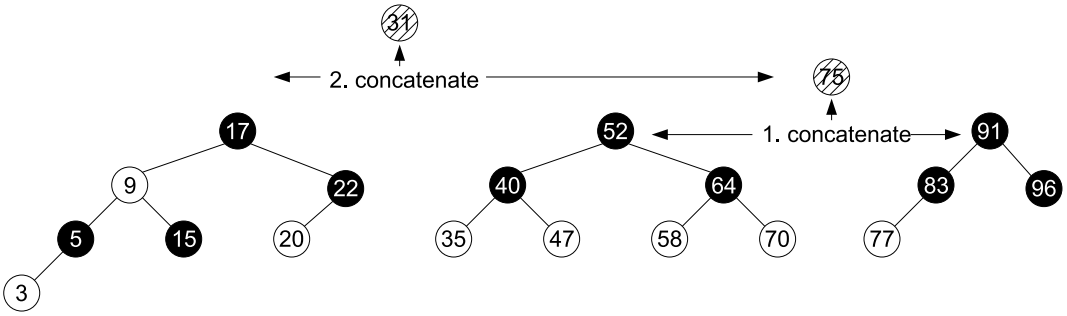
For both bulk operations, allocating the node objects constitutes a considerable share of the work to be done. We cannot allocate several of them with one function call, since we need to be able to deallocate them one by one in case the user deletes elements from the data structure. The memory management concept of C++ does not allow such an asymmetric usage. Tests have shown that allocating memory concurrently scales quite well, but imposes some work overhead compared to the strictly sequential implementation. There also exist memory managers designed for this specific problem, e. g., Hoard [10]. We successfully used it on the Sun machine. However, for our 64-bit Intel platform, Hoard could not even match the performance of the built-in allocator, so we did not use it there.

6.2.4 Trade-offs for different input data and comparator types

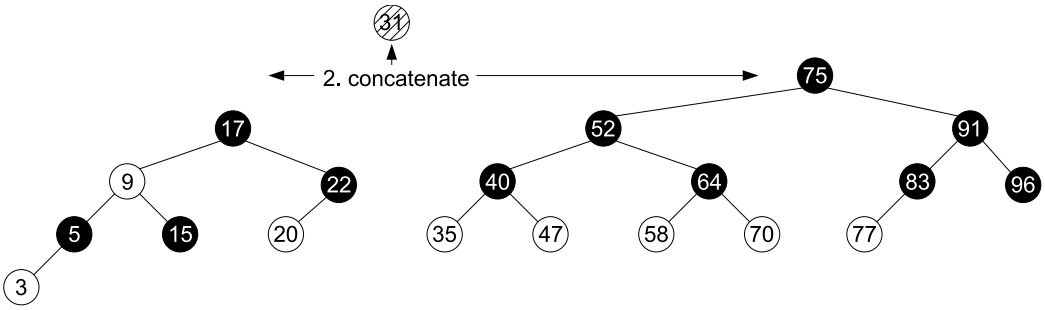
The cost for comparing and copying an element depends greatly on the type of the elements and its comparator. We also have to relate this to the cost of copying the node as a whole, which contains three pointers and the node color flag, in addition to the actual payload.



(a) Phase 0/1. The input sequence is divided equally, the tree is split accordingly.



(b) Phase 2. The two elements that will serve as the tentative roots in the future concatenation tasks are found to be 31 and 75. They are excluded from the subtree and not inserted to it, respectively. 58 is not inserted either, since it is a duplicate.



(c) Phase 3. Eventually, the concatenation tasks are processed. First, the right two subtrees are concatenated, using 75 as the new root. Then, the resulting tree is concatenated with the left one using 31 as the new root. In this case, a rotation is necessary for rebalancing purposes, so 75 ends up at the top position.

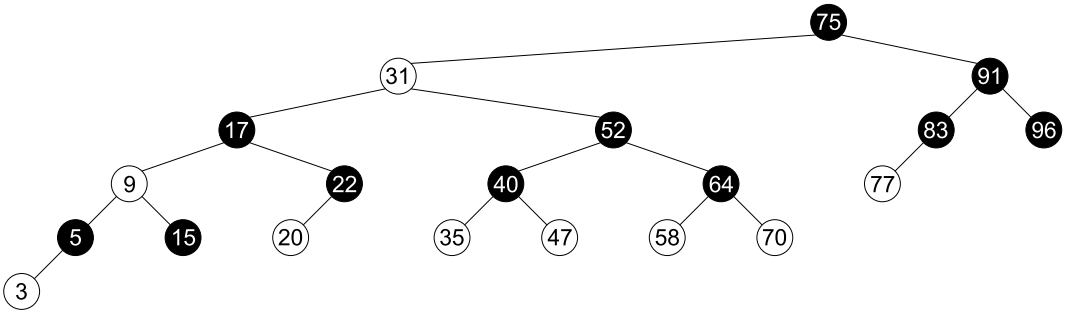


Figure 6.1: Inserting into a tree, using three threads. “Red” nodes are shown transparent.

In particular, we have taken this trade-off into account in sorting the elements when processing. In general terms, for large elements, it is better to sort pointers only, to avoid costly copying. For small elements, it is more cache-efficient to sort the elements directly. We contrasted this fact with some preliminary experiments. A fine-grained tuning is out of the scope of this study.

6.2.5 Task data structures

For distributing the subtasks among the threads, we need some extra data structures to describe them. An *insertion task* consists of the tree to be inserted into (represented by its root), the range of elements to insert, and a pointer to the parent concatenation task.

A *concatenation task* description contains a reference to the root t of the new tree, and also its black height (number of black nodes on each path down to a leaf). Furthermore, there are pointers to the children and the parent concatenation task of t. Very importantly, a concatenation task must not be tackled before both of its child tasks are processed. To handle it, a boolean variable initially false is set atomically (so a race condition is avoided) to true when the first child task is solved. The thread processing the second subtask then knows that it can continue up to the top.

6.3 Experimental analysis

We tested our program on two multi-processor machines with the following processors:

1. Sun T1 (1 socket, 8 cores, 1.0 GHz, 32 threads, 3 MB shared L2 cache),
2. Intel Xeon E5345 (2 sockets, 2×4 cores, 2.33 GHz, $2 \times 2 \times 4$ MB L2 cache, shared among two cores each).

For the Intel machine, we used the Intel C++ compiler 10.0.25, on the Sun machine, we took GCC 4.2.0, always compiling with optimization (`-O3`) and OpenMP support. In both cases, we used the libstdc++ implementation coming with GCC 4.2.0. On the Xeon, compiling using the Intel compiler lead to more consistent results, due to the better OpenMP implementation, which is causing less overhead.

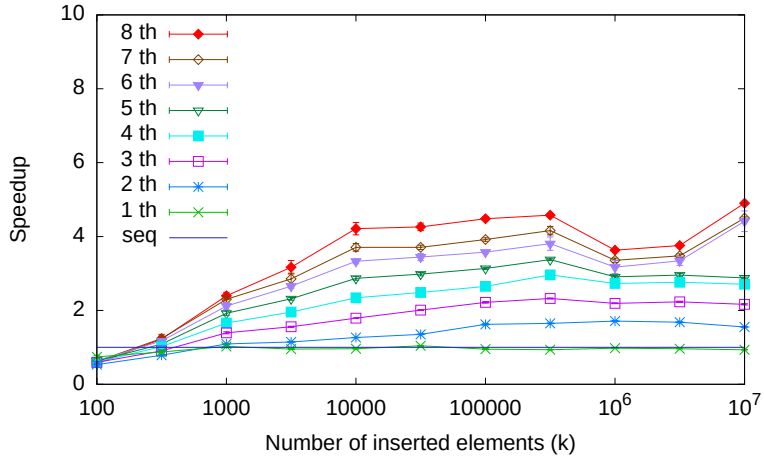
We have run each test at least 30 times and taken the average values for the plots, accompanied by the standard deviation range (very small in all cases). The following parameters concerning the input were considered:

Tree size/insertion sequence length ratio. Let n be the size of the existing dictionary/tree. Let $r = n/k$. Thus, $r = 0$ is equivalent to bulk construction.

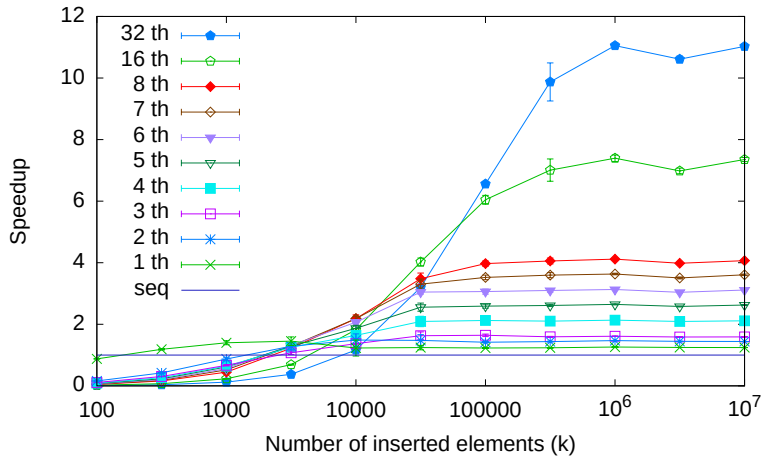
Sortedness of the insertion sequence. The input sequence can be presorted or not. If the input is not presorted, our algorithms are much faster than the libstdc++ implementation, which inserts the elements one after the other. We focus on presorted insertion sequences here because parallel sorting is a separate issue.

Key data type. We show experiments on 32-bit signed integers.

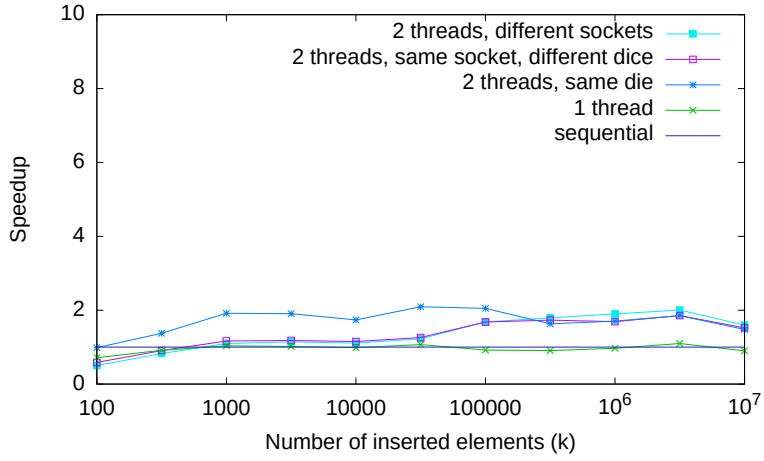
Randomness of the input sequence. The input sequence by default consists of values in the range {`RAND_MIN`...`RAND_MAX`}, but can optionally be limited to a smaller range,



(a) On the Xeon



(b) On the T1



(c) On the Xeon, fixing two cores

Figure 6.2: Performance results for constructing a set of integers.

$\{\texttt{RAND\_MIN}/100 \ldots \texttt{RAND\_MAX}/100\}$. This poses a challenge to the load-balancing mechanisms.

Both load-balancing mechanisms were switched on by default, but deactivated for some experiments, to show their influence on the running time.

For the insertion tests, we first build a tree of the appropriate size using the sequential algorithm, and then insert the elements using our parallel bulk insertion algorithm. This guarantees fair initial conditions for comparing the sequential and the parallel implementation. Actually, the shape of the existing tree does affect performance, though the sequential insertion is more affected than our algorithms.

In the following, we first consider the results for the sequential case, and then, we consider the parallel case.

6.3.1 Sequential results

Our sequential algorithm is never substantially slower and in some cases substantially faster. In particular, our specialized algorithms are very competitive when having random inputs limited to a certain range, being more than twice as fast (see Figures 6.4(a) and 6.4(b)) as the standard implementation. This is because of saving many comparisons in the upper levels of the tree, taking advantage of the input characteristics.

6.3.2 Parallel results

Figures 6.2(a) and 6.2(b) show the results for dictionary construction. These show that our algorithms scale quite well. In particular, on the 8-core Sun T1, even the absolute speedup slightly exceeds the number of cores, culminating in about 11. This shows the usefulness of per-core multithreading in this case.

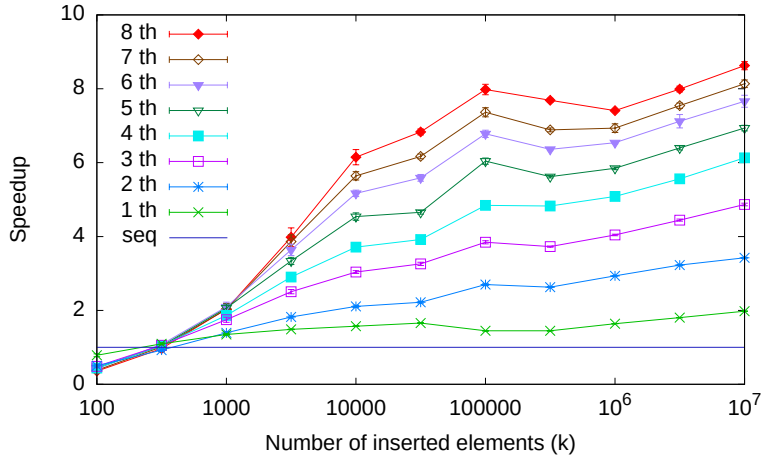
For insertion, our algorithm is most effective when the existing tree is smaller than the data to insert (see Figures 6.3(a) and 6.3(b)). This is also due to the fast (sequential) insertion procedure, compared to the original algorithm. Splitting the input sequence is most effective in terms of number of comparisons because the subsequences left when reaching a leaf still consist of several elements.

In all cases, remarkable speedups are achieved: speedup of at least 3 is achieved for four threads and speedup of at least 5 is achieved for eight threads. The break-even is already reached for as little as 1000 elements, the maximum speedup is usually hit for 100000 or more elements.

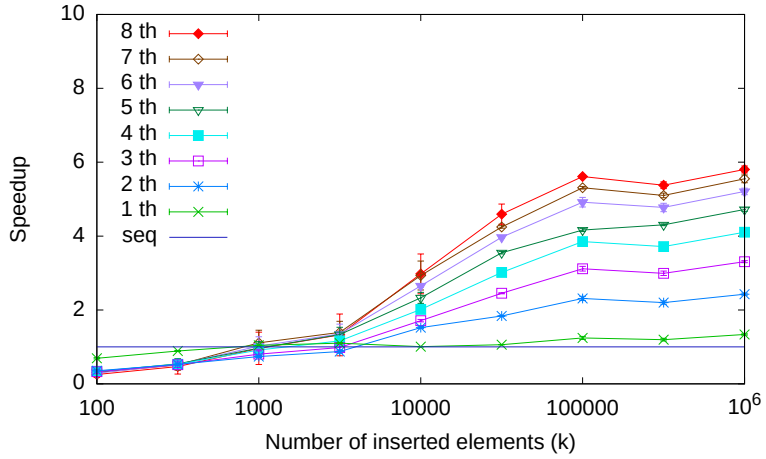
The tree splitting step, used to make an initial balancing in the case of insertion, is shown to be really effective. For instance, compare Figures 6.3(b) and 6.3(c), which only differ in whether Phase 1 of the algorithm is run. We can see that both the speedup and the scalability are far better when the initial splitting is activated.

On top of that, the parallelization scales nicely, specifically when using dynamic load-balancing. Switching load-balancing off hurts performance for large inputs, while for small inputs it does not create considerable overhead. Dynamic load-balancing makes the algorithm more robust, comparing Figures 6.4(a) and 6.4(b).

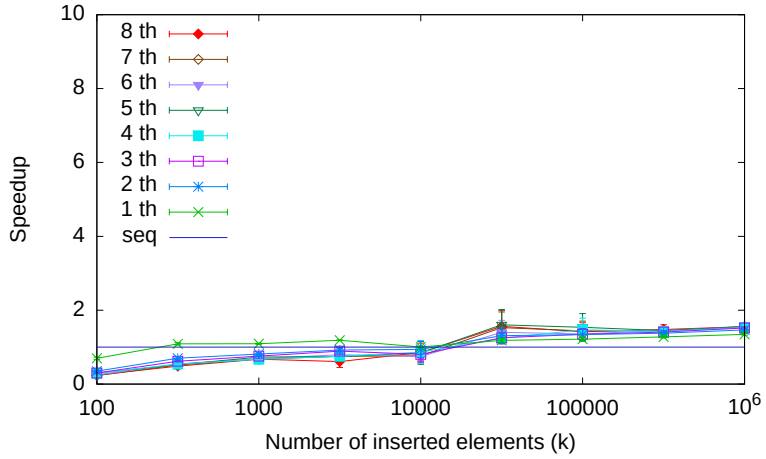
As a by-product, we show the effects of mapping two threads to cores in different ways (see Figure 6.2(c)). For small inputs, the most profitable configuration is sharing the cache,



(a) $r = 0.1$

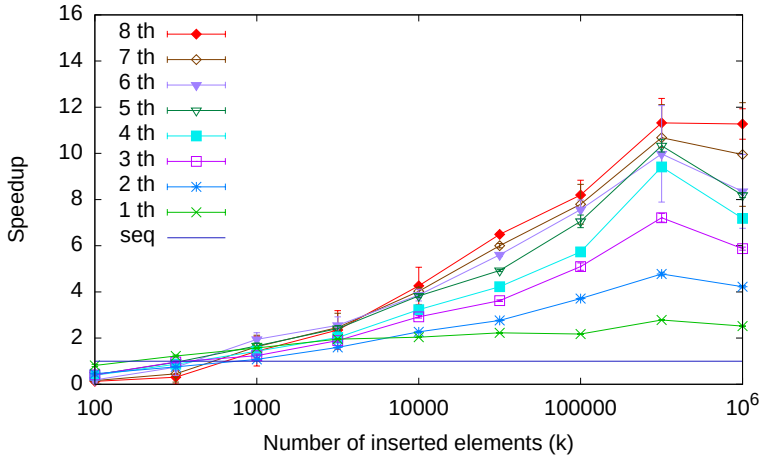


(b) $r = 10$

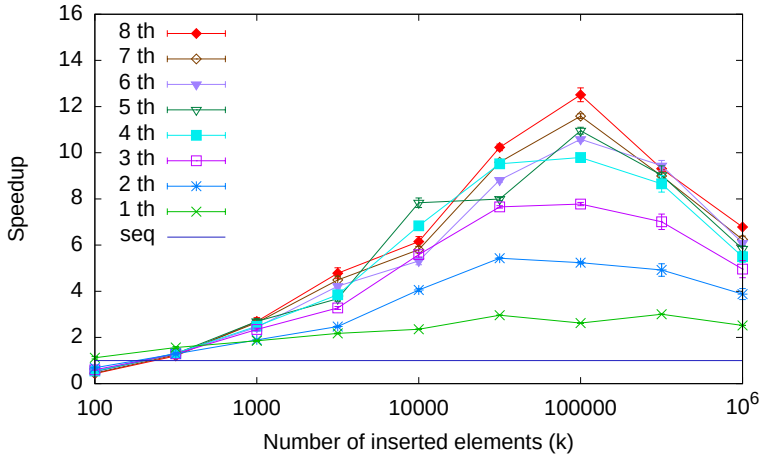


(c) $r = 10$, without using initial splitting of the tree

Figure 6.3: Performance results for inserting integers into a set on the Xeon.



(a) $r = 10$



(b) $r = 10$, without using dynamic load-balancing

Figure 6.4: Performance results for inserting integers from a limited range into a set on the Xeon.

i. e., both threads running on the same die. As the input data gets larger, memory bandwidth becomes more important, so running the threads on different sockets wins. Running both threads on the same socket, but on different dice is in fact worst, because it combines both drawbacks. For the other tests, we have refrained to explicitly bind threads to specific cores, however, because the algorithms should be allowed to run on a non-exclusive machine in general.

6.4 Conclusion and future work

In this chapter, we have shown that construction of and bulk insertion into a red-black tree can be effectively parallelized on multi-core machines, on top of a sequential implementation which remains unaffected. Our construction bulk operation shares some ideas with the theoretical algorithm in [75], giving a practical implementation. The code has been released in the MCSTL [86], version 0.8.0-beta.

Our bulk insertion operation divides the work using two approaches: First without considering the tree structure, then dynamically, depending on the subtree structure. This way, we can ensure good scalability even if only a limited part of the tree is affected. Also, we provide dynamic load-balancing on the subtrees to rule out external factors.

We discussed some of the main practical obstacles for the actual implementation, including memory allocation, which can take a large part of the running time, if one is uncareful.

Our experiments take into account several kinds of input data distribution as well as different variants of our algorithms. The results of our experiments show that good speedups can be obtained, in particular for small input sizes, and if the existing tree is larger than the input sequence. Also, the sequential version of our insertion algorithm is usually much faster than the libstdc++ version because we benefit from structure in the input.

To speed up programs that do not use bulk operations explicitly, we could use lazy updating. Each sequence of insertions, queries, and deletions could be split into consecutive subsequences of maximal length, consisting of only one of the operations. This approach could transparently be covered by the library methods. Then, for example, a loop inserting a single element in each iteration, would also benefit from the parallelized bulk insertion.

Another enhancement to the functionality is bulk *deletion* of elements. Since it is easy to remove a consecutive range from a tree, it would be rather interesting to tackle the problem posed by a `remove_if` call, specializing it for the dictionary types.

Chapter 7

Single-pass list partitioning

In many parallel algorithms, the input must be first divided into (independent) parts of similar size so that parallel computation is effective. Most parallel algorithm descriptions disregard how the input is actually divided (or assume that the input can be divided by index computations). If we want to use such algorithms in practice, we have to deal with sequences whose elements can only be accessed one-by-one and whose length is unknown, e.g., a single linked list. Also, this definition corresponds with STL forward access sequences. See Section 2.4.1 for an overview on STL iterators and sequences. Indeed, most STL algorithms are defined on forward access sequences.

In this chapter, we consider the problem of the division of such sequences into parts of similar length to allow effective parallel computation. We call this problem *list partitioning*. Solving it efficiently is of utmost importance because the speedup of parallel programs is limited by the sequential portion (according to Amdahl's law).

Given that the length of the sequence is unknown, one could think of first traversing the sequence to determine its length, and then traversing it a second time to actually divide the sequence. We call this algorithm TRAVERSETWICE. However, traversing the sequence can be expensive, so we do not want to pay for it twice. The elements can be spread in memory in a cache-unfriendly way, and/or calculating the next element may be costly. To avoid this, one could also think of using a dynamic array, storing the pointers to the elements there, effectively converting the sequence to a randomly accessible one. We call this algorithm POINTERARRAY. However, this is very costly in terms of additional space. We subsume both algorithms as the *trivial* solutions.

In contrast, we present a list partitioning algorithm using only sublinear additional space, and accessing each element exactly once. Besides, we provide a C++ implementation and draw an experimental comparison: we evaluate the algorithms by themselves as well as their impact in parallel performance. The experiments show that our list partitioning algorithm is effective and fast in practice.

This chapter is organized as follows. In Section 7.1, we formally define the list partitioning problem. Then, in Section 7.2 we present our single-pass list partitioning algorithm. Next, in Section 7.3 we present the experimental results. Finally, we sum up some conclusions in Section 7.4.

7.1 Problem definition

A linearly traversable sequence of unknown length n is to be divided into p parts of almost equal length. Let the ratio r be the quotient of the length of the longest part and the length of the shortest part. It is a good quality measure for the partitioning, since it correlates to the efficiency of processing the parts in parallel, given that processing time is proportional to a part's length. Thus, to guarantee good efficiency, r should be upper bounded by a constant R at any time, only depending on a tuning parameter, namely the oversampling factor $\sigma \in \mathbb{N} \setminus \{0\}$.

Without loss of generality, we assume that the input sequence has length at least σp, i. e., $n \geq \sigma p$. Otherwise, if $p \leq n$, we can lower σ down so that $\sigma p \leq n$. If $p > n$, we reduce p to n to avoid that any part is empty (and therefore, $r = \infty$), which would not be sensible for our purposes.

This can be considered as an online problem since the input is given one element at a time, without information about the whole problem. Thus, we can define a competitive ratio between the optimal offline algorithm and our online algorithm. For the optimal offline algorithm, the difference in part lengths is at most 1, which gives a ratio $r_{\mathrm{OPT}} = \lceil n/p \rceil / \lfloor n/p \rfloor \overset{n\to\infty}{\to} 1$.

7.2 The SINGLEPASS algorithm

Let L be a forward linearly traversable input sequence. Our single-pass algorithm, denoted SINGLEPASS, keeps a sequence of boundaries $S[0 \ldots p]$, where $[S[i-1], S[i])$ defines the i^{th} subsequence of L. Inserting a subsequence into S means storing its boundaries in the appropriate places. A boundary is identified by its rank in L.

The basic SINGLEPASS algorithm works as follows:

1. Let $k := 1$, $S := \{\}$.
2. Iteratively append to S at most $2\sigma p$ 1-element consecutive subsequences from L.
   $S := \{0, 1, 2, \ldots, 2\sigma p\}$
3. While L has more elements do:
   Invariant: $|S| = 2\sigma p$, $S[i+1] - S[i] = k$
   (a) Merge each two consecutive subsequences into one subsequence.
   $S[0, 1, 2, \ldots, \sigma p] := S[0, 2, 4, \ldots, 2\sigma p]$
   This results in σp subsequences of length $2k$.
   (b) Let $k := 2k$.
   (c) Iteratively append to S at most σp consecutive subsequences of length k from L.
   $S := \{0, k, \ldots \sigma pk, (\sigma p + 1)k, (\sigma p + 2)k, \ldots, l\}$, $\sigma pk < l \leq 2\sigma pk$
4. The $\sigma p \leq |S| \leq 2\sigma p$ subsequences are divided into p parts of similar lengths as follows. The $|S|$ mod p rightmost parts are formed by merging $\lceil |S|/p \rceil$ consecutive subsequences each, from the right end. The remaining $p - (|S| \bmod p)$ leftmost parts are formed by merging $\lfloor |S|/p \rfloor$ consecutive subsequences each, from the left end.

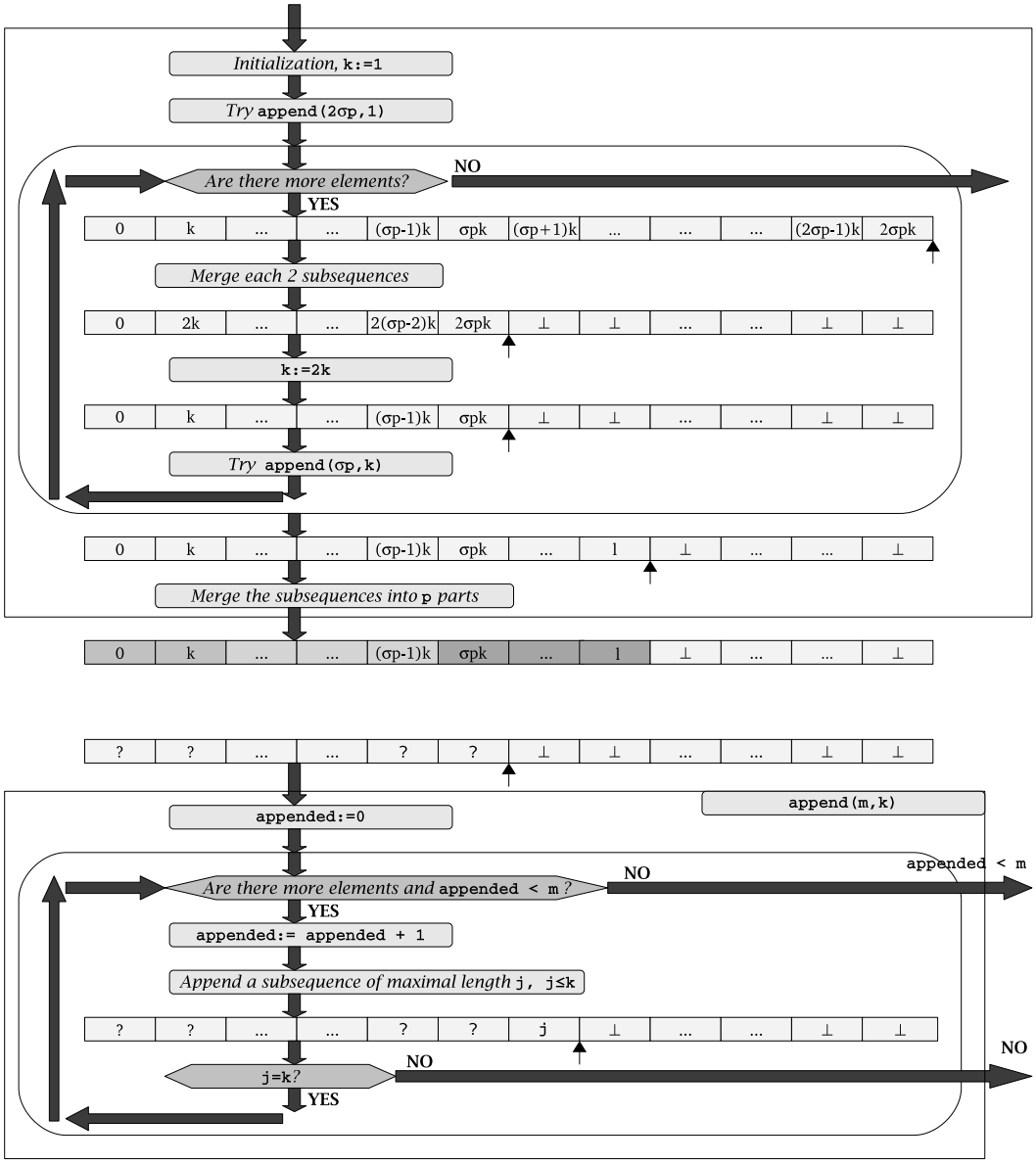


Figure 7.1: Basic SINGLEPASS list partitioning algorithm scheme.

The algorithm (visualized in Figure 7.1) takes special care of the rightmost subsequence E. If L runs empty prematurely in Step 2 or 3c, the last subsequence is shorter than k, which might be shorter than the others, i. e., $|E| \leq k$. Let T be the part containing E, there is no part that consists of more subsequences than T. So, if exactly one part is longer than all the others (i. e., $|S| \bmod p = 1$), this is specifically T. In this case, T differs from the other parts in $|E|$ elements. As a whole, the algorithm guarantees that in the worst-case, two parts differ at most in one complete subsequence (i. e., in at most k elements).

The basic SINGLEPASS algorithm needs $\Theta(\sigma p)$ additional space to store S. The time complexity is $\Theta(n + \sigma p \log n)$. This is proved as follows: We need to traverse the whole sequence, taking $\Theta(n)$ time. In addition, Step 3 visits $\Theta(\sigma p)$ elements of S in $\Theta(\log n)$ iterations.

The worst-case ratio of r is bounded by $\frac{\sigma+1}{\sigma}$. The worst-case occurs when just one complete subsequence was appended after reducing the list. Without loss of generality, to analyze the average ratio, we consider only complete subsequences, therefore $\sigma p \leq |S| < 2\sigma p$. The average ratio of r (denoted by $\mathbb{E}r$) is upper bounded by

$$\begin{aligned}
\mathbb{E}r &< \frac{1}{\sigma p} \sum_{\ell=\sigma p}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \\
&= \frac{1}{\sigma p} \left(\sigma + \sum_{\ell=\sigma p, p \nmid \ell}^{2\sigma p-1} \frac{\lceil \ell/p \rceil}{\lfloor \ell/p \rfloor} \right) \\
&= \frac{1}{\sigma p} \left(\sigma + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{\sigma+\ell+1}{\sigma+\ell} \right) \\
&= \frac{1}{\sigma p} \left(\sigma p + (p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma+\ell} \right) \\
&= 1 + \frac{1}{\sigma p} \left((p-1) \sum_{\ell=0}^{\sigma-1} \frac{1}{\sigma+\ell} \right) \\
&= 1 + \frac{1}{\sigma p} \left((p-1)(\Psi(2o) - \Psi(\sigma)) \right) \\
&\approx 1 + \frac{1}{\sigma p} \left((p-1)(\ln(2o) - \ln(\sigma)) \right) \\
&= 1 + \frac{1}{\sigma p} \left((p-1) \ln(2) \right).
\end{aligned}$$

E. g., for $\sigma = 10$ and $p = 32$, using the previous expression for the worst-case and average ratio, respectively, we get that the longest subsequence is at most 10% longer than the shortest one, and expectedly 7% longer.

A generalization of this algorithm performs Step 3a and 3b only every m^{th} loop iteration. In the remaining iterations of the main loop, S is doubled in size so that space for additional subsequences is needed. This is equivalent to increasing the oversampling factor to σn^{γ} with $\gamma = 1 - 1/m$.

The generalized SINGLEPASS algorithm needs as many iterations of Step 3 as the basic algorithm, i. e., $\Theta(\log n)$ iterations. The additional space increases, but sublinearly, growing with $O(\sigma p n^{\gamma})$. The time complexity of this algorithm is $\Theta(n + \sigma p(n^{\gamma} + \log n))$.

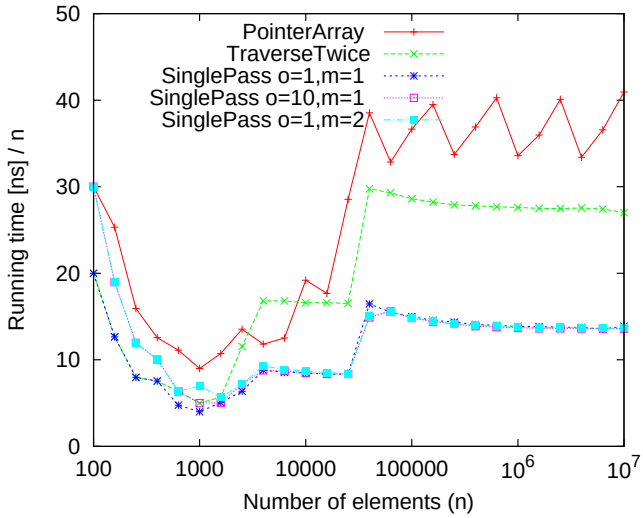


Figure 7.2: Running times of the list partitioning algorithms for $p = 4$.

In the worst-case, the longest sequence and the shortest sequence have length $(n^\gamma + 1)k$ and $(n^\gamma)k$, respectively. It holds $\sigma p n^\gamma k \approx n$, so $k \approx \frac{n^{1/m}}{\sigma p}$. Subsuming this, the lengths of the subsequences do at most differ by $\frac{n^{1/m}}{\sigma p}$ elements, i. e., the difference decreases relatively to n, as n grows. Therefore, the bound for r also converges to 1.

Generally speaking, the choice of m trades of time and space versus solution quality. The larger m, the more memory and time is used, but r becomes better. This is the same effect as would be caused by a dynamically growing σ. Choosing $m = 2$ appears to be a good compromise.

7.3 Experimental analysis

We have implemented our SINGLEPASS algorithm in the general form, so it subsumes the two variants. Besides, we have implemented the two naïve algorithms, namely TRAVERSETWICE and POINTERARRAY algorithms. The implementation has been done in C++ and as part of the MCSTL [86], a parallel implementation of the STL. See Section 2.4.3 for an overview on the MCSTL and other STL implementations. Dynamic arrays have been implemented using STL `vector`.

We have performed two kinds of experiments. First, we present the evaluation of the list partitioning algorithms isolatedly. Then, we present the impact of using list partitioning algorithms in the parallelization of two STL algorithms.

7.3.1 Comparison of list partitioning algorithms

We have compared all the algorithms both measuring the running time as well as the quality of the results. Concerning quality, we have computed both the worst-case ratio r and its

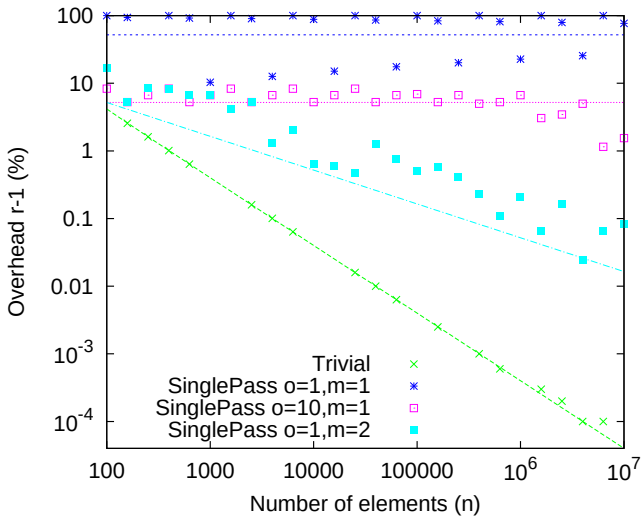


Figure 7.3: Quality of the list partitioning algorithms for $p = 4$. We show the worst-case overhead ratio $h = r - 1$, as well as its expectancy. The results are in %. Note that the missing points are actually 0.

expectation. For a better plot reading, we have rescaled these results using the *overhead* ratio h, where h is defined from the ratio r as $h = r - 1$.

It must be noted that the actual quality of the results is deterministic with respect to the problem size. That is, the quality of our solution does not depend on the specific input data but only on its size.

Setup. We have tested our program on an AMD Opteron 270 (2.0 GHz, 1 MB L2 cache). We have used GCC 4.2.0 as well as its libstdc++ implementation, compiling with optimization (`-O3`).

Parameters for testing. We have repeated our experiments at least 10 times and report the average running time (variance was observed to be small). The focus is on results for $p = 4$. Recall that as p grows, r becomes smaller.

For SINGLEPASS, we consider the following parameter combinations:

1. $\sigma = 1$, $m = 1$
2. $\sigma = 10$, $m = 1$
3. $\sigma = 1$, $m = 2$

Therefore, it uses $\Theta(p)$, $\Theta(10p)$, and $\Theta(p\sqrt{n})$ additional space, respectively.

Results. Figure 7.2 summarizes the performance results, and Figure 7.3 the quality results. They show that the performance of the SINGLEPASS algorithm is very good. In particular, it takes only half the time compared to the TRAVERSETWICE algorithm, and even less compared to the POINTERARRAY algorithm. That is, we can divide a sequence into parts using virtually the same time as for only traversing it once. Further, the varying

running times for POINTERARRAY must be due to the amortization of the vector doubling cost (i. e., depending on how much of the allocated memory has been actually used by the vector).

The quality of the solution for our algorithm improves with the amount of additional space allowed for the auxiliary sequence S (i. e., increased σ or m). The simplest variant ($\sigma = 1$, $m = 1$) has a worst-case ratio of 2 and an average ratio of 1.5. In addition, the overhead $r - 1$ is divided by σ (in our case, $\sigma = 10$). When setting m to 2, our algorithm behaves very well, converging to zero overhead. The average overhead ratio decreases exponentially with the input size n. Note that the (optimal) worst-case ratio achieved by the naïve algorithms also decreases exponentially (and even at a faster ratio than SINGLEPASS).

7.3.2 Parallelization using list partitioning

After having evaluated the isolated performance for the list partitioning algorithms, we investigate their impact in their intended domain, i.e., data-parallel algorithms. Two interesting examples are STL `accumulate` and STL `list::sort` operation.

STL `accumulate` is a STL algorithm that computes the sum (or some other reduction based on an associative binary operation) of a sequence of elements, starting with some initial value. Here, we consider a forward access input sequence.

STL `list::sort` stably sorts a list instance. The implementation in the libstdc++ consists on a doubly linked list. See Section 3.2 for more details on this implementation. In the following, we first present parallel implementations using list partitioning for both algorithms. Then, we evaluate the impact in performance of different list partitioning algorithms.

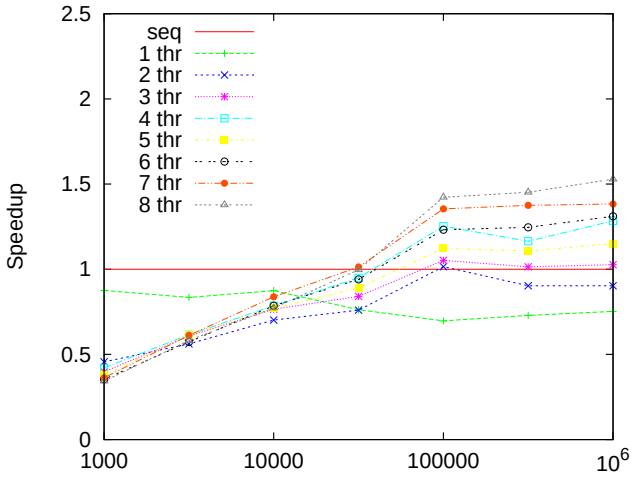
Parallel implementations. Parallelizing STL `accumulate` is straightforward. Let the desired number of threads be p. The list is partitioned into p parts, using one of the described algorithms. In parallel, each thread accumulates its part. After that, the p results are combined sequentially.

For STL `list::sort`, we partition the list as above and split it into the respective sublists. These are sorted by one thread each, in parallel. Recursive tree-shaped merging combines the results in $\log_2 p$ steps, each merge per se being done sequentially.

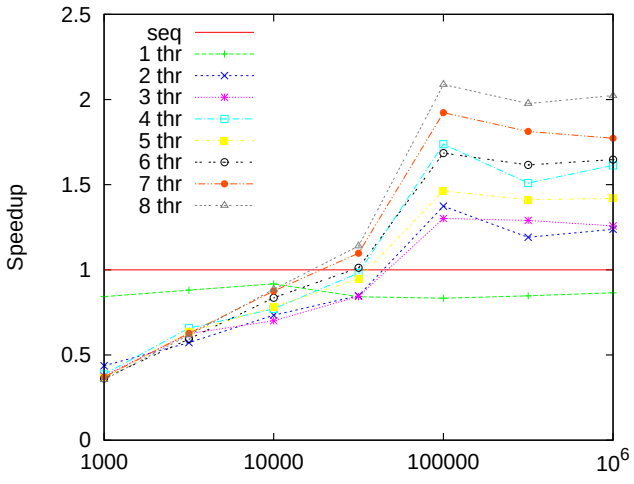
Parameters for testing. We have repeated our experiments at least 10 times and report the average running time.

For SINGLEPASS, we have used the parameters $\sigma = 1$ and $m = 2$. Deduced from the results in Section 7.3.1, this choice produces a high quality partitioning at a low overhead in running time and additional space. The data type for STL `list::sort` is a 16-byte element containing an 8-byte integer key. The values are randomly generated. As use case for accumulate, we approximately multiply double-precision floating-point numbers by summing up their logarithms.

Setup. We have run these experiments on a dual-socket AMD Opteron 2350 (2×4 cores, 2.0 GHz, 2×2 MB L3 cache).



(a) Using TRAVERSETWICE partitioning



(b) Using SINGLEPASS partitioning

Figure 7.4: Performance results for STL `list::sort`.

Results. Figures 7.4(a) and 7.4(b) show the speedup for parallel STL `list::sort` using the TRAVERSETWICE and SINGLEPASS list partitioning algorithms, respectively. The POINTERARRAY variant is not compared here since it has a linear space overhead.

In both shown cases, the achieved speedups in absolute terms are not particularly good. This is probably mostly due to the bad collective memory bandwidth caused by the random accesses to traverse the more and more scattered sublists. Nonetheless, for a large number of elements, the SINGLEPASS list partitioning algorithm is significantly better than TRAVERSETWICE because the sequential fraction is speed up by a factor of 2, and the parts are of very similar size.

Figures 7.5(a) and 7.5(b) show the speedup for parallel STL `accumulate` using the TRAVERSETWICE and SINGLEPASS list partitioning algorithms, respectively.

In this case, the achieved speedups in absolute terms are better because summing the logarithm of floating-point numbers is computationally intensive. Again, the performance using SINGLEPASS is much better than with TRAVERSETWICE, in particular for a large number of elements.

Overall, for large inputs, SINGLEPASS obtains much better speedups than TRAVERSETWICE both for STL `list::sort` and STL `accumulate`.

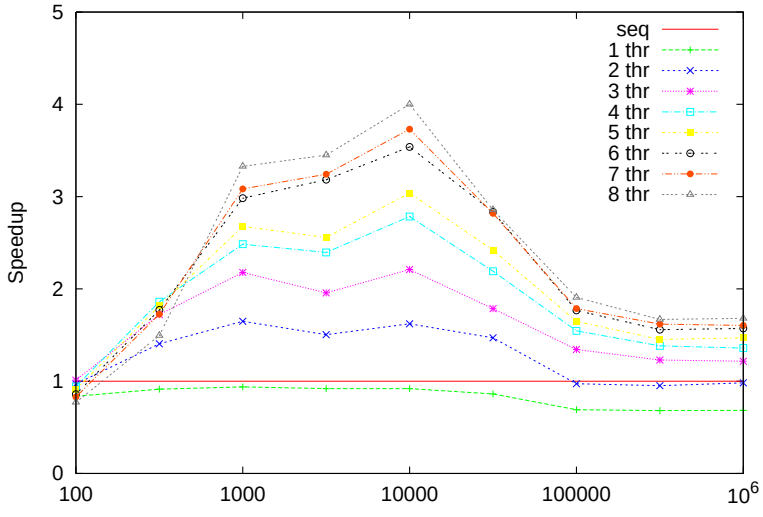
7.4 Conclusions and future work

We have presented a simple, though non-trivial, algorithm to solve the list partitioning problem using only one traversal and sublinear additional space. Our algorithm divides a linearly traversable sequence of unknown length n in time $\Theta(n + \sigma p(n^{1-1/m} + \log n))$ using $O(\sigma p n^{1-1/m})$ additional space. σ and m are tuning parameters of the algorithm. We have implemented all the algorithms presented in this chapter. The code was first released in the MCSTL [86], version 0.8.0-beta. Later, this code has been included in the *libstdc++ parallel mode* (see Section 2.4.3 for further details on these libraries), and thus, it is automatically installed as a standard component of major Linux distributions and other Unix-like operating systems.

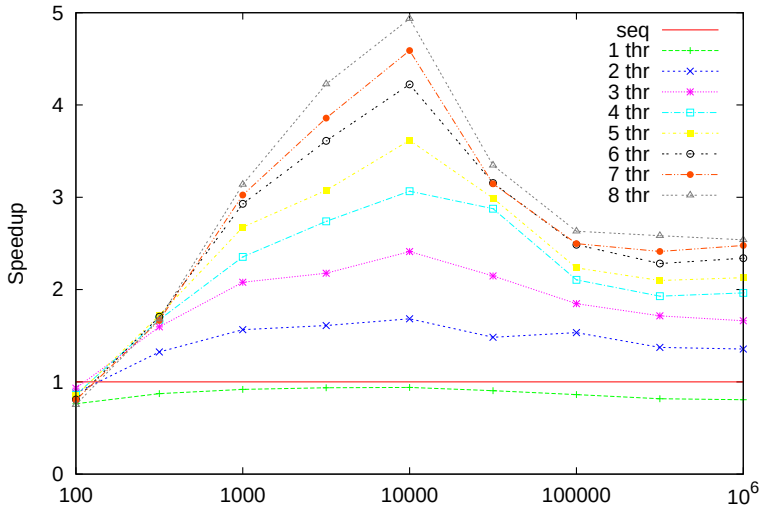
Our experiments have shown that our algorithm is very efficient in practice. It takes almost the same time as if the list was just traversed, without any processing. Besides, very high quality solutions can be obtained. The larger m, the better the quality, trading off memory. If $m = 1$, the worst-case overhead ratio 1 is divided by the oversampling factor σ. If $m > 1$, the worst-case overhead ratio decreases exponentially with n.

Therefore, for large input instances and in most practical situations, processing perfectly equal parts and almost equal parts should take about the same time, because the time to process each of the parts fluctuates in the same order of magnitude. In addition to this, our approach computes the partitioning twice as fast as the naïve approach.

As a result, using our list partitioning algorithm as a substep of parallel algorithms, the overall performance is significantly improved compared to a naïve implementation.



(a) Using TRAVERSETWICE partitioning



(b) Using SINGLEPASS partitioning

Figure 7.5: Performance results for STL `accumulate`.

Chapter 8

Parallel partition revisited

The partitioning of an array is a basic building block of many key algorithms, as quicksort and quickselect. Partitioning an array with respect to a pivot x consists of rearranging its elements such that, for some splitting position v, all elements at the left of v are smaller than x, and all other elements are greater or equal than x. It is well known that an array of n elements can be partitioned sequentially and in-place using exactly n comparisons and m swaps, where m is the number of elements greater than x whose original position is smaller than v.

In this chapter, we consider the problem of partitioning an array in parallel. Several algorithms have been proposed to partitioning in parallel [47, 29, 98, 59, 86]. We focus on suitable algorithms for current widely available multi-core architectures. Specifically, we consider a simple algorithm by Francis and Pannan [29], a fetch-and-add based algorithm by Tsigas and Zhang [98] and a variation of the former in the MCSTL [86], a parallel STL implementation (see Section 2.4.3 for an overview on the MCSTL and other STL implementations). All these algorithms, though very different in nature, can be divided into three main phases:

1. A sequential setup of each processor's work
2. A parallel main phase in which most of the partitioning is done
3. A cleanup phase, which is usually sequential.

Besides, in order to avoid too much synchronization, they perform more than n comparisons and m swaps.

In this chapter, we show how the former parallel partitioning algorithms can be modified so that exactly n comparisons are made, which is optimal. The original algorithms disregard part of the work done in the main parallel phase when cleaning up. By contrast, we propose an alternative parallel cleanup phase that uses the whole comparison information of the parallel phase. A small static order-statistics tree is used to efficiently locate the elements to be swapped and to swap them in parallel. We provide a detailed analysis.

Furthermore, we show that the resulting parallel partitioning algorithms are also scalable and competitive. To do so, we have implemented and drawn an experimental comparison between all the previous algorithms, both with their original cleanup and with our cleanup. Besides, the implementation is provided according to the specification of STL `partition`. Previously, only the MCSTL implementation was available.

This chapter is organized as follows. In Section 8.1, we present the considered algorithms. Then, in Section 8.2, we present our cleanup algorithm. Then, we present our implementation of the previous algorithms in Section 8.3. It follows the experimental comparison in Section 8.4. Finally, we sum up some conclusions in Section 8.5.

8.1 Previous work and a variant

In this section, we give an overview of several partitioning algorithms suitable for multicore implementation. In the following, the input consists of an array of n elements and a pivot. Besides, p processors are available and we assume that $p \ll n$. Also, we disregard some details as rounding issues for the sake of simplicity.

STRIDED algorithm

The STRIDED algorithm by Francis and Pannan [29] works as follows:

1. Setup: The input is (conceptually) divided into p pieces of size n/p. The pieces are not made of consecutive elements, but one of every p elements instead. That is, the i-th piece is made up of elements $i, i+p, i+2p, \ldots$.

2. Main phase: Each processor i, in parallel, gets a piece, applies sequential partitioning on it, and returns its splitting position v_i.

3. Cleanup: Let $v_{min} = \min\{v_i : 1 \leq i \leq p\}$ and $v_{max} = \max\{v_i : 1 \leq i \leq p\}$. It holds that all the elements at the left of v_{min} and at the right of v_{max} are already well placed with respect to the pivot. In order to complete the partition, sequential partition is applied to the range (v_{min}, v_{max}).

The main phase takes $\Theta(n/p)$ parallel time. For random inputs, the cleanup phase is expected to take constant time. However, it was not stated in [29] that in the worst-case the algorithm takes $\Theta(n)$ time and thus, there is no speedup. E.g., if the pieces are made exclusively of either smaller or greater elements than the pivot and these are alternated, then, $v_{min} \leq p$ and $v_{max} \geq n-p$, and $|(v_{min}, v_{max})| = \Theta(n)$.

BLOCKED algorithm

Accessing elements with stride p as in the STRIDED algorithm, can cause a high cache miss ratio. We propose the BLOCKED algorithm to overcome this problem. This algorithm uses blocks of b elements instead of individual elements. Each block in the piece is separated by stride p blocks. Note that if $b = 1$, BLOCKED is equal to STRIDED.

F&A algorithms

Heidelberger *et al.* [47] proposed a parallel partitioning algorithm in the PRAM model in which elements from both ends of the array are taken using fetch-and-add instructions. Fetch-and-add instructions (atomically increment a variable, and return its original value)

were introduced in [44] and are useful, for instance, to implement synchronization and mutual exclusion.

In a first approach, exactly one element is taken at a time and so, at the end of the parallel phase, the array is already partitioned. In this case, n fetch-and-add operations are used. In a second approach, the algorithm is generalized to blocks: a block of b elements is acquired at each fetch-and-add instruction. So, the number of fetch-and-add instructions is n/b. However, in this case, some sequential cleanup remains to be applied after the parallel phase.

Later, Tsigas and Zhang [98] presented a variant of the second approach for multiprocessors. More recently, a further variant has been proposed as part of the MCSTL [86]. In the latter, the cleanup phase is partially done in parallel.

Let us now briefly describe these F&A algorithms:

1. Setup: Each processor takes two blocks, one from each endpoint of the array. Namely, one left block and one right block.

2. Main phase: While there are blocks, each processor applies the so-called *neutralize* method to its two blocks. The neutralize method consists in applying the sequential partitioning algorithm to the array made by (conceptually) concatenating the right block to the left. However, the left and right pointers to the current elements cannot cross the borders of a block. When a left (right) block is completely processed (i.e., neutralized), a fresh left (right) block is acquired and processed.

3. Cleanup: At this point, at most p blocks remain unneutralized. Each author presents a different cleanup algorithm:

— In [98], while unneutralized blocks remain, one block is taken from each endpoint and neutralization is applied to them. Then, the unneutralized blocks are placed between the neutralized blocks. At most p blocks need to be swapped and this is done sequentially. Finally, sequential partition is applied to the range of blocks with unprocessed elements.

— In [86], all unneutralized blocks are placed between the neutralized blocks. Then, the parallel partitioning algorithm is applied recursively to this range. The number of processors is divided by two in each call until there is only one processor or block. Finally, the remaining range is partitioned sequentially.

The main parallel phase takes $\Theta(n/p)$ parallel time. The cleanup phase takes $\Theta(bp)$ sequential time in [98]. Rather, in [86], it takes $\Theta(b \log p)$ parallel time.

8.2 The new parallel cleanup phase

In this section, we present our cleanup algorithm. It avoids extra comparisons, and swaps the elements fully in parallel. We have applied it on the top of STRIDED, BLOCKED and F&A. First, we introduce the terminology. Then, we present the data structure on which the algorithm relies. It follows the cleanup algorithm itself. Finally, we analyze the resulting STRIDED, BLOCKED and F&A algorithms.

Terminology

In the following, we shall use the following terms to describe our algorithm. A *subarray* is the basic unit of our algorithm and data structure. The *splitting position v* of an array is the position that would occupy the pivot after partitioning. A *frontier* separates a subarray in two consecutive parts that have different properties. A *misplaced element* is an element that must be moved by our algorithm. We denote by m the total number of misplaced elements and by M the total number of subarrays that may have misplaced elements.

The Case of BLOCKED. In this case, subarrays correspond with exactly one of the p pieces. Moreover, v can be easily known after the parallel phase. The frontier of a subarray corresponds with the position that would occupy the pivot after partitioning this subarray. Thus, a frontier defines a left and a right part. A misplaced element corresponds either to a smaller element than the pivot that is on the right of v (misplaced on the right) or to a greater element than the pivot that is on the left of v (misplaced on the left). The total number of subarrays that may have misplaced elements (M) is at most p.

The Case of F&A. In this case, a subarray corresponds to one block. The frontier separates a processed part from an unprocessed part. The processed part of left blocks is the left part and the processed part of right blocks is the right part. Though v is unknown after the parallel phase, it holds that v is in some range $V = [v_{\text{beg}}, v_{\text{end}}]$. A misplaced element corresponds either with a processed element that is in V or with an unprocessed element that is not in V.

We consider now the array of elements made by left blocks and the array of elements made by right blocks. We denote by β the global border that separates the left and right blocks (i.e., the point where no more blocks could be obtained).

In left blocks, a misplaced element on the left is an unprocessed element in a smaller position than v_{beg} and a misplaced element on the right is a smaller element than the pivot in a greater or equal position than v_{beg}. Let μ_l be the total number of misplaced elements in left blocks, and rank those misplaced elements starting to count from the leftmost towards the right. In right blocks, a misplaced element on the left is a greater element than the pivot in a smaller or equal position than v_{end}, and a misplaced element on the right is an unprocessed element in a greater position than v_{end}. Let μ_r be the total number of misplaced elements in right blocks, and rank those misplaced elements starting to count from the rightmost towards the left. Thus, $m = \mu_l + \mu_r$. Besides, M is at most $2p$: there are at most p blocks that may contain unprocessed elements and there are at most p blocks that may contain misplaced processed elements.

8.2.1 The data structure

We use a complete binary tree with M leaves (or the next power of two if M is not a power of two) to know which pairs of elements must be swapped. This tree is shared by all processors and is stored in an array (like a heap, which provides easy and efficient access to the nodes).

Each leaf stores information of the i-th subarray. Specifically, how many elements are misplaced to the left and to the right of its frontier (m_l^i and m_r^i) and how many elements are in the left and in the right to its frontier (n_l^i and n_r^i). The internal nodes accumulate the

information of their children but do not add any new information. In particular, the root stores the information of the array made of all the subarrays in the leaves.

So, our tree data structure can be considered as a special kind of order-statistics tree in which the internal nodes have no information by themselves. An *order-statistics tree* (see e.g., [19, Section 14]) performs rank operations efficiently using the information of the size of the subtrees.

Figure 8.1 shows two instances of our tree data structure.

8.2.2 The algorithm

Tree initialization phase. In this phase, the tree is initialized. Specifically, two bottom-up traversals of the tree are needed. Only the first initialization of the leaves depends on the partition algorithm used in the main parallel phase.

1. **First initialization of the leaves.** In the case of BLOCKED, the leaves values n_l^i and n_r^i for each subarray i can be trivially computed during the parallel phase. In the case of F&A, the left (right) blocks that contain unprocessed elements can be easily known after the parallel phase. The left (right) blocks that contain misplaced elements but have already been processed can only be located between the left (right) unprocessed blocks. In order to locate the latter efficiently, we sort the unneutralized blocks with respect to the block position in the array. Then, we iterate on left (right) blocks (sequentially) to the left (right) of the border β until p neutralized blocks have been found, or the leftmost (rightmost) unneutralized block has been reached.

2. **First initialization of the non-leaves.** Using a parallel reduce operation, each internal node computes its n_l^j and n_r^j values from its children. As a result, the root stores the number of left and right elements in the whole array. Thus, the splitting position v can be directly deduced.

3. **Second initialization of the leaves.** The leaves get the values m_l^i and m_r^i using n_l^i, n_r^i and v. At this point, it may turn out that some subarrays have no misplaced elements. This fact does not disturb the correctness of our algorithm.

4. **Second initialization of the non-leaves.** The number of misplaced elements for the internal nodes are computed using a second parallel reduce operation on m_l^j and m_r^j fields.

Parallel swapping phase. In this phase, our tree data structure is queried so that the misplaced elements can be swapped in parallel and no comparisons are needed. This phase is independent of the specific partitioning algorithm.

The total number of misplaced elements is used to distribute the work equally among the processors. A range $[r_i, s_i)$ of ranks of misplaced elements to swap is assigned to each processor. The elements are swapped in ascending rank. Specifically, the j-th misplaced left element is swapped with the j-th misplaced right element. To locate the first pair of elements to swap, respective rank queries are made to the tree. That is, a query is made for the r_i left misplaced element and another for the r_i right misplaced element. Misplaced elements are swapped as long as the rank s_i is not reached. If the rank s_i has not yet been reached but

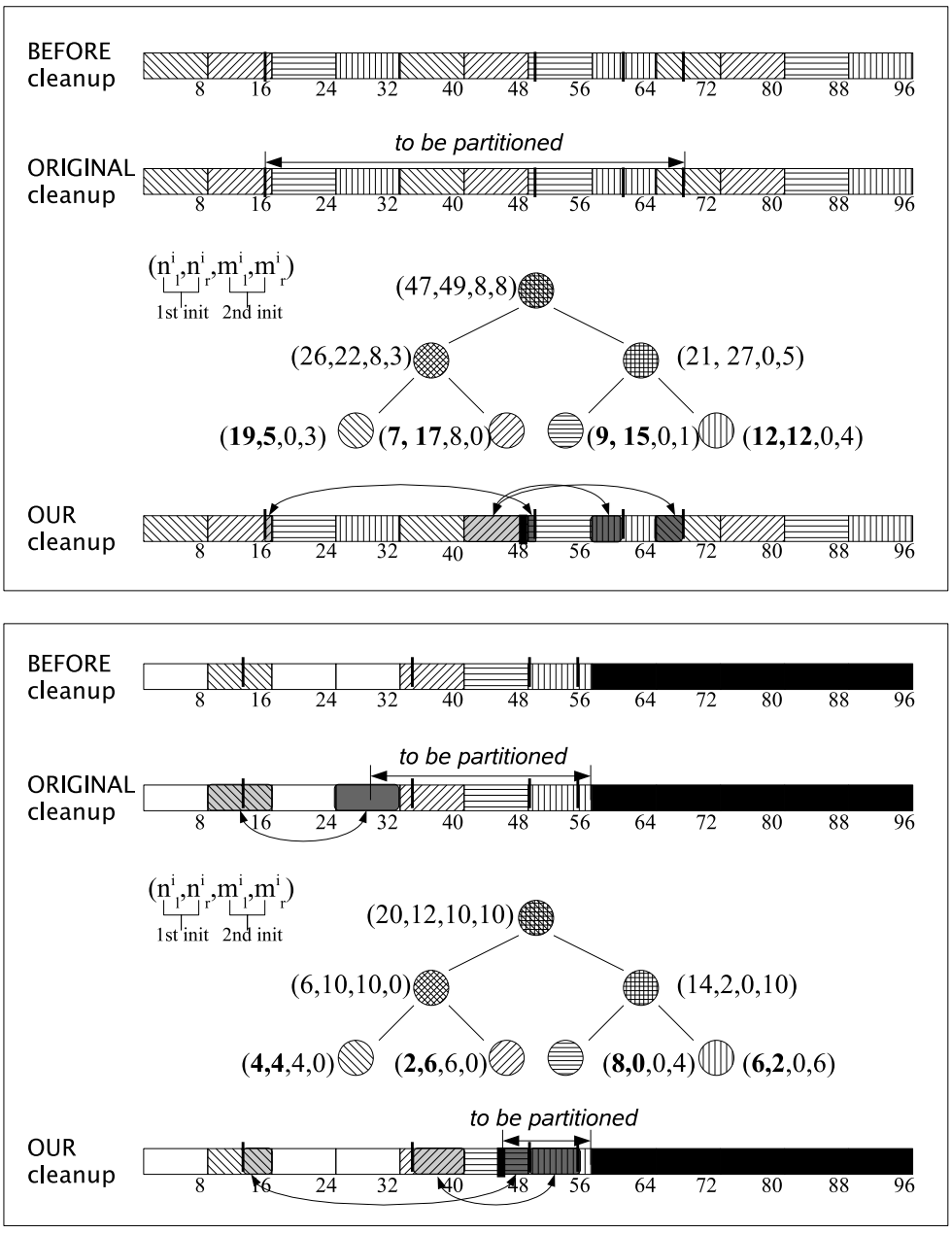


Figure 8.1: Example of our data structure.

the current subarray has no more misplaced elements, the next subarray is fetched. Let c_i be the position in the tree corresponding to the current subarray. Then, the next subarray of left misplaced elements is in $c_i + 1$ and the next subarray of right misplaced elements is in $c_i - 1$.

Completion phase. This phase depends on the specific partitioning algorithm. In the case of BLOCKED, the whole array has already been partitioned and we are done. In the case of F&A, some unprocessed elements may remain. When this happens, V is not empty and includes exclusively all the unprocessed elements. In order to obtain a valid partition, we apply our parallel partitioning algorithm $\log p$ times and recursively in V. Specifically, the block size is divided by 2 in each recursive call, and as a result, also the size of the problem in each recursive call is at most half the previous (because at least p blocks have been fully processed). Finally, sequential partition is applied to the remaining elements.

8.2.3 Cost analysis

In this section we analyze the cost of our algorithms.

The following theorem shows that our cleanup phase achieves an optimal number of comparisons:

Theorem 8.2.1. BLOCKED and F&A perform exactly n comparisons when using our cleanup algorithm.

Proof. The tree initialization and the parallel swapping phases perform no comparisons for both BLOCKED and F&A.

In the case of BLOCKED, the completion phase is empty. Therefore, no comparisons are performed during cleanup for BLOCKED and thus, comparisons are only performed during the main parallel phase, which are exactly n.

In the case of F&A, after the first main parallel phase $n - |V|$ elements have been compared and $|V|$ have remained unprocessed. In the next recursive step, V is the input. Besides, elements can only be compared during a certain parallel phase and at most once. All the elements must be eventually compared because our algorithm produces a valid partition. Thus, our cleanup algorithm makes exactly $|V|$ comparisons, and n comparisons are needed as a whole. □

The following results bound the parallel time of BLOCKED and F&A:

Lemma 8.2.2. The tree initialization phase takes $\Theta(\log p)$ parallel time for BLOCKED and F&A.

Proof. The algorithm-independent part takes $\Theta(\log p)$ parallel time because all the work is done in parallel, and is dominated by the two parallel reduces, which can be performed in logarithmic parallel time (see e.g., [57]).

In the case of BLOCKED, the algorithm-dependent part takes constant parallel time because each leaf can be initialized trivially and in parallel.

In the case of F&A, the algorithm-dependent part takes $\Theta(\log p)$ parallel time, because $2p$ elements are sorted and this takes $\Theta(\log p)$ parallel time using p processors (see e.g., [52]).

		BLOCKED		F&A	
		original	tree	original	tree
COMPS	main	n		$n-\|V\|$	
	cleanup	$v_{max}-v_{min}$	0	$\leq 2bp$	$\|V\|$
	total	$n+v_{max}-v_{min}$	n	$\leq n+2bp$	n
SWAPS	main	$\leq n/2$		$\leq \frac{n-\|V\|}{2}$	
	cleanup	$m/2$	$m/2$	$\leq 2bp$	$\leq m/2+\|V\|$
	total	$\leq \frac{n+m}{2}$	$\leq \frac{n+m}{2}$	$\leq \frac{n-\|V\|}{2}+2bp$	$\leq \frac{n+m}{2}+\|V\|$
PTIME	main	$\Theta(n/p)$		$\Theta(n/p)$	
	cleanup	$\Theta(v_{max}-v_{min})$	$\Theta(m/p+\log p)$	$\Theta(b\log p)$	$\Theta(\log^2 p+b)$
	total	$O(n)$	$\Theta(n/p+\log p)$	$\Theta(n/p+b\log p)$	$\Theta(n/p+\log^2 p)$

Table 8.1: Summary of costs for BLOCKED and F&A algorithms.

Thus, in both cases, the total cost is $\Theta(\log p)$ parallel time. □

Lemma 8.2.3. The parallel swapping phase performs exactly $m/2$ swaps and requires $\Theta(m/p)$ parallel time. In particular, in the case of F&A, the swapping phase takes $O(b)$ parallel time, where b denotes the block size.

Proof. There are m misplaced elements after the main parallel phase. The parallel swapping phase swaps pairs of misplaced elements so that their final position is not misplaced. Therefore, $m/2$ swap operations are needed. Besides, the pairs are evenly divided among the p processors. Thus, swapping all of them takes $\Theta(m/p)$ parallel time. In the case of F&A, $m \leq 2bp$, thus parallel swapping takes $O(b)$ parallel time. □

Theorem 8.2.4. The cleanup phase takes $\Theta(m/p+\log p)$ parallel time for BLOCKED and the whole partition takes $\Theta(n/p+\log p)$ parallel time.

Proof. From Lemmas 8.2.2 and 8.2.3 follows that the cleanup phase takes $\Theta(m/p+\log p)$ parallel time for BLOCKED. Given that $m = O(n)$, the whole BLOCKED algorithm takes $\Theta(n/p+\log p)$ parallel time on the average and in the worst-case. □

Theorem 8.2.5. Consider $p \leq b$. The cleanup phase takes $\Theta(\log^2 p+b)$ parallel time for F&A and the whole partition takes $\Theta(n/p+\log^2 p+b)$ parallel time.

Proof. F&A takes $T(n,p) = \Theta(n/p)+C(b,p)$, where $C(b,p)$ is the cost of our cleanup algorithm. $C(b,p) = b+\log p+T'(b/2,\log p)$ parallel time, and T' is defined by the following recurrence:

$$T'(\beta,i) = \begin{cases} O(3\beta+\log p)+T'(\beta/2,i-1) & \text{if } i>1, \\ O(2\beta) & \text{otherwise.} \end{cases}$$

There are $2\beta p$ blocks at the beginning of each recursive step and $\log p-1$ recursive steps are needed. Thus, $C(b,p) = O(\log^2 p+b)$ parallel time. □

We would like to highlight that Theorem 8.2.5 improves previous theoretical bounds for F&A (provided $\log p \leq b$, which is of practical relevance).

Finally, Table 8.1 summarizes worst-case results for BLOCKED and F&A algorithms.

8.3 Implementation

Our parallel algorithms are implemented using OpenMP. Besides, they follow the specification of STL `partition`, so that eventually the could be used instead of the sequential implementation.

Since implementations of STRIDED were not available, we have resorted to implement it ourselves. We have also implemented our BLOCKED variant, which improves STRIDED cache performance. In particular, we have implemented enhanced iterators on the top of them so that (sequential) STL `partition` can transparently be applied to the *virtual* sequences in which the input sequence is divided. We say *virtual* because (logically) contiguous elements in these resulting sequences are not contiguous in the input sequence.

For F&A, we have taken its implementation from MCSTL 0.7.3-beta, but we have also implemented our own version ourselves. In particular, our F&A implementation calls STL `partition` when falling back to the sequential case. Our implementation of F&A and the one in MCSTL differ in the following: a) ours statically assigns the initial work; therefore, it avoids mutual exclusion here; b) ours does not use `volatile` variables, and critical regions are slightly simpler; and c) ours avoids redundant comparisons using some book-keeping. The implementation in the *libstdc++ parallel mode*, the successor of the MCSTL, improves locking mechanisms and some other low-level details, but the algorithm itself remains the same.

Furthermore, we have implemented our cleanup algorithm on the top of the previous four algorithms.

8.4 Experimental analysis

We have analyzed STRIDED, BLOCKED, and F&A with and without our cleanup algorithm.

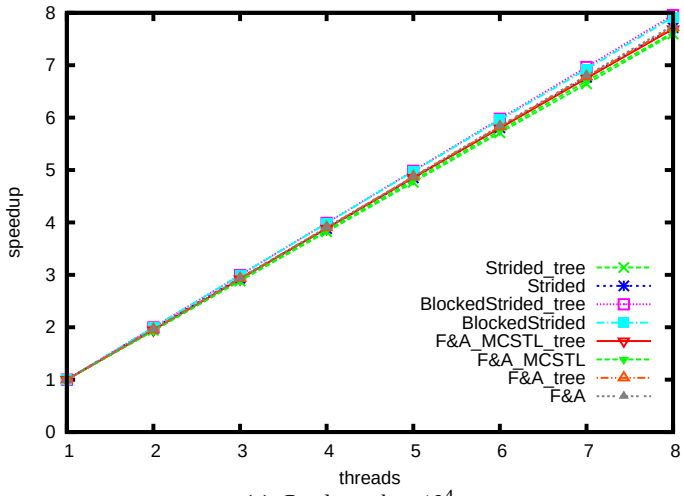
The tests have been run on a machine with 4 GB of main memory and two sockets, each one with an Intel Xeon quad-core processor at 1.66 GHz with a L2 cache of 4 MB, shared among two cores. Thus, there are 8 cores in total. We have used the GCC 4.2.0 compiler with the `-O3` optimization flag.

All tests have been repeated 100 times; figures show averages (variance was small).

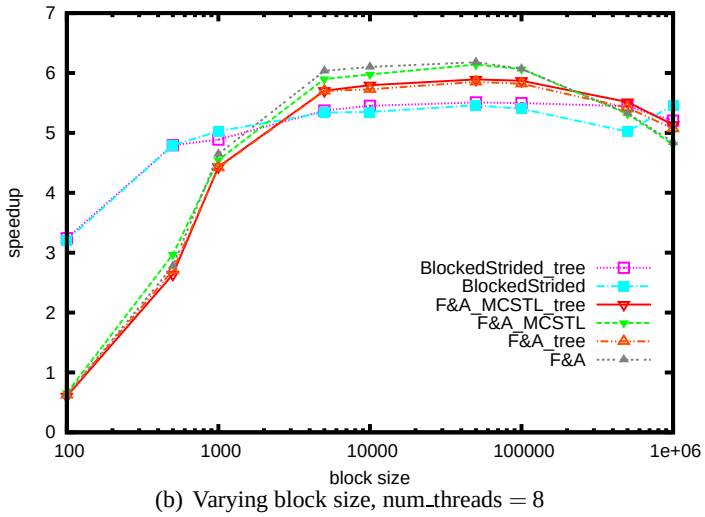
Basic evaluation. We have first analyzed the speedup of partitioning in parallel a large number of random integers. The speedup is always measured with respect to the sequential STL `partition` algorithm. The block size b has been set to 10^4 (this decision will be justified later in this section).

Figure 8.2(c) shows the results. In this figure, Strided refers to our implementation of STRIDED, BlockedStrided refers to our implementation of BLOCKED, F&A_MCSTL refers to the MCSTL implementation of F&A, F&A refers to our own implementation of F&A. We add the suffix _tree to the previous labels to refer to the algorithm with our modified parallel cleanup phase.

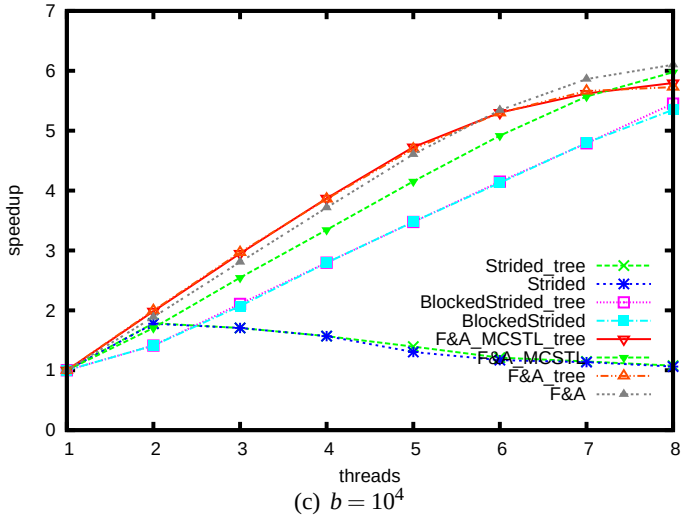
These results show that F&A is better than BLOCKED, which is better than STRIDED. Whereas the speedup of STRIDED is nonexistent for more than two threads, BLOCKED performs reasonably well, and our F&A implementation achieves some better results than the MCSTL F&A. However, it makes almost no difference whether our cleanup step is used



(a) Costly $<$, $b = 10^4$



(b) Varying block size, num_threads $= 8$



(c) $b = 10^4$

Figure 8.2: Performance results for parallel partition for integers, $n = 10^8$.

or not. Only in the case of F&A, the achieved speedup is slightly better, making it almost perfect for up to 4 threads.

The awful performance of STRIDED is due to its high cache miss ratio; its behavior clearly contrasts with BLOCKED (which uses blocks of elements rather than individual elements). On the other hand, the shown experimental results do not correspond with worst-case instances (see Section 8.1). Indeed, we have also constructed and evaluated *ad hoc* worst-case instances for STRIDED and BLOCKED. As expected, the running time is significantly worse than for sequential partition.

Finally, in order to understand the loss of performance when using more than 5 threads, we have devised two new experiments. The first one reproduces the previous experiment but uses a slower comparison function. Its results are shown in Figure 8.2(a). In this case, all algorithms show similar behavior and excellent speedups with up to 8 threads. Again, there is not much of a difference whether our cleanup phase is used or not. Our second experiment has consisted in measuring the speedup of a trivial parallel program to compute the sum of two arrays. The resulting speedups (not shown) are also not optimal for the biggest number of threads. So, we can conclude that memory bandwidth is limiting the efficiency of the partitioning algorithms, which are demanding with regard to I/O.

Influence of block size. Several algorithms rely on a block size parameter b. In order to determine its optimal value, we have run various tests, varying the number of threads and the value of b. The results in Figure 8.2(b) (where the number of threads is fixed to 8) show that, except for very small block sizes, the performance is not much affected. Besides, given that, for smaller input sizes, big block sizes are not convenient, we have fixed b to 10^4.

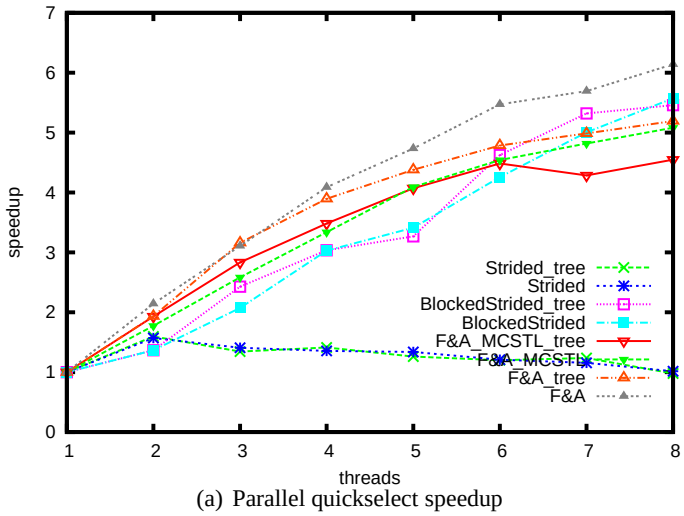
Operations Count. In order to analyze the behavior of the cleanup phase in more detail, we have counted swap and comparison operations. Figures 8.3(b) and 8.3(c) show respectively the number of extra comparisons and swaps with respect to the sequential implementation. They are depicted divided by the block size.

Figure 8.3(b) gives an experimental proof of Theorem 8.2.1. Besides, combining our cleanup algorithm with the original MCSTL algorithm does not achieve the optimality in the number of comparisons, because this implementation makes extra comparisons whenever a new block is fetched in the main parallel phase. Specifically, our experiments show that two comparisons are repeated per block in the average.

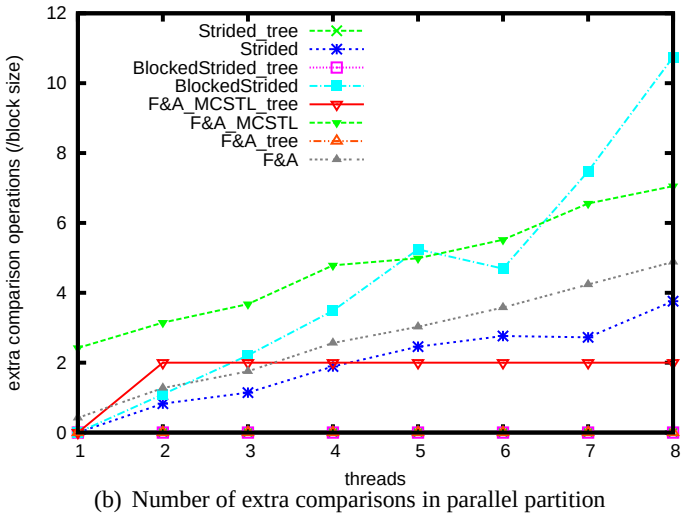
Figure 8.3(c) shows that our cleanup algorithm does not need more swaps than the original cleanup algorithms. Essentially, the same number of extra swaps are needed. In the case of F&A, we could not give such an equality analytically.

As a by product, these results show that the number of misplaced elements resulting from the parallel phase (given random uniformly generated inputs) is really small, no matter the partitioning algorithm. In particular, STRIDED is the algorithm that performs less extra operations. However, its performance is the worst because of its bad cache usage.

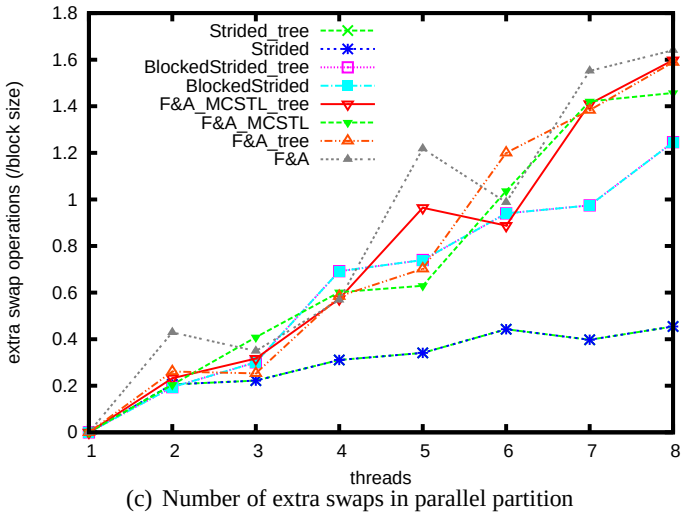
Application: Quickselect. Quicksort and quickselect are typical applications of partitioning. As quicksort offers two (not exclusive) ways to be parallelized —parallelizing the partitioning and parallelizing the independent work by divide and conquer—, we found more interesting to analyze quickselect. Quickselect is a common implementation for STL



(a) Parallel quickselect speedup



(b) Number of extra comparisons in parallel partition



(c) Number of extra swaps in parallel partition

Figure 8.3: Performance results and operations count for integers, $n = 10^8$, $b = 10^4$.

`nth_element`. We have implemented STL `nth_element` using the parallel partitioning algorithms in this chapter. Only if the array is small, sequential partitioning is called.

The results are shown in Figure 8.3(a). These are coherent with those of partition but they differ because the relative behavior between the algorithms changes slightly with the input size. First, our F&A implementation advantage increases. Second, our cleanup algorithm increases the runtime of the F&A based quickselect. Indeed, in our experiments we have observed that for a big number of threads and as input gets smaller, using our cleanup algorithm with F&A is counterproductive. Figure 8.3(a) also shows that the simple BLOCKED algorithm performs quite well.

8.5 Conclusions and future work

In this chapter, we have presented, implemented and evaluated several parallel partitioning algorithms suitable for multi-core architectures.

From an algorithmic point of view, we have described a novel cleanup parallel algorithm that does not disregard comparisons made during the parallel phase. This cleanup has been successfully applied to three partitioning algorithms: STRIDED, BLOCKED (a cache-conscious implementation of the former) and F&A. In the case of STRIDED and BLOCKED, a benefit of our cleanup is reducing its parallel time in the worst-case from $\Theta(n)$ to $\Theta(n/p+\log p)$. In the case of F&A, we have shown how to modify it to reduce its parallel time from $\Theta(n/p+b\log p)$ to $\Theta(n/p+\log^2 p)$. Unlike their original versions, these algorithms perform the minimal number of comparisons when using our cleanup phase.

From an engineering perspective, we have contributed carefully designed implementations of the aforementioned algorithms that are compliant with the specification of STL `partition`.

Finally, and from an experimental point of view, we have conducted an evaluation to compare those algorithms and implementations. According to our experiments, the partitioning algorithm of choice is F&A, because it scales nicely. However, our results show that, in practice, the benefits of our cleanup algorithm are limited. This happens because the number of misplaced elements after the parallel phase is very small.

Our experiments also show that I/O between the memory and the processor limits the performance achieved by parallel implementations as the number of threads increases. It remains to be further investigated how these results change, if more cores and/or memory bandwith is available.

Chapter 9

Combining digital access and parallel partition for quicksort and quickselect

Comparison-based data structures and algorithms, like quicksort and quickselect, can be specialized for strings to perform efficient digital comparisons. These techniques rely on the order and outcome of previous comparisons, as well as on the digital structure of strings. See Chapter 4 for further details.

In this chapter, we apply these existing techniques for strings on the top of parallel algorithms. To the best of our knowledge, this is a novel application. Specifically, we describe the case of parallel partitioning-based algorithms, namely, parallel quicksort and quickselect.

Parallelizing quicksort can be done in two (not exclusive) ways: parallelizing the partition and parallelizing the divide and conquer work. By contrast, parallelizing quickselect can only be done by parallelizing the partition. We reuse here the parallel partitioning algorithms in Chapter 8, whose properties regarding comparisons are crucial for the application of the aforementioned string techniques. The resulting algorithms are provided as specializations of the STL `sort` and `nth_element` algorithms for strings. Indeed, most common implementations of these STL algorithms are based on quicksort and quickselect, respectively. Besides, we draw an experimental comparison considering several string datasets. The results show the effectiveness of our approach in practice.

This chapter is organized as follows. In Section 9.1, we present our solution in detail, including some additional precautions while choosing the pivot and handling repeated elements. In Section 9.2 we present the experimental results for parallel quicksort and quickselect. Finally, we sum up some conclusions in Section 9.3.

9.1 Our solution

Combining efficient digital access to strings with partitioning-based algorithms is based in the following specific properties (see Section 4.1 for further details). First, each element is compared exactly once against the pivot during a partition. This fact is precisely guaranteed

by the parallel partitioning algorithms presented in Chapter 8. Second, the implicit structure defined by the order of the recursive calls corresponds to a BST. Since the parallel execution of recursive calls in the case of quicksort keeps up with the relative order in sequential quicksort, we can enhance string comparisons on the aforementioned parallel partitioning algorithms to build efficient parallel quicksort and quickselect implementations.

In the following, we describe some of the most relevant implementation issues.

9.1.1 Basic implementation

We describe our implementation based on the parallel partitioning algorithms in Chapter 8. They are provided according to the specification of STL `partition`. In particular, their input consists of a random access input sequence, e.g., an array. See Section 2.4.1 for an overview on STL iterators and sequences.

Enhancing (parallel or sequential) partitioning with efficient digital access to strings requires storing some extra data per element. This data regards the outcomes from previous comparisons, and it consists of the length of two prefixes. Besides, comparison and swap algorithms must be specialized, so that this data is used and updated. See Section 4.1 for further details on these algorithms. Nonetheless, the specification of STL `partition` and their based algorithms, namely STL `sort` and `nth_element`, only considers providing a customized comparator. Given that the STL specification does not consider providing a customized swap functor, we need to provide a full replacement to specialize the parallel partitioning algorithms for strings. Similarly, every call to sequential STL `partition` must be replaced for a call to a specialized sequential partitioning algorithm for strings. See Section 8.3 for a detailed explanation about when falling back to sequential partitioning is required.

Furthermore, in order to provide a solution as least intrusive as possible (minimal changes in existing code), we store the prefixes apart from the data itself. Let A denote the array in which the prefixes are stored. Note that A must be created once at the beginning of parallel quicksort and quickselect, and it can be initialized in parallel. Then, A is accessed and updated by the specialized swap and comparison algorithms, which in our case have random access iterators as input parameters. Note that the relative position of an element in the sequence can be computed in constant time, given its random access iterator. In addition, we encapsulate the specialized comparison and swap algorithms, together with the pivot and the rest of attributes, in a class that is passed as an input parameter to the partitioning algorithms. Thus, our approach provides a flexible framework to deal with different implementations of the comparison and swap algorithms. In particular, one could define a trivial implementation of them that considers keys as atomic.

9.1.2 Additional precautions

Many implementations of sequential and parallel quicksort and quickselect exist. Crucial aspects in their performance are the pivot choice, and also how repeated elements are handled by the sequential partitioning algorithm. See e.g., [8, 19] for an overview of possible approaches. In the following, we consider the main precautions that must be taken when dealing with specialized partitioning algorithms for strings.

The main issue arises when choosing the pivot for partitioning: if elements need to be compared in the process (e.g., using median of 3), these comparisons must keep the stored prefixes consistent. One possibility is to provide an additional comparison algorithm that does use the prefix information but does not update it. Or alternatively, given that in some cases we are not that interested in the exact result, we can merely rely on the prefix information to make a decision (to the detriment of the quality of the partitioning). We call the later technique *approximate comparison*. Similarly, we must not compare the pivot against itself, to avoid corrupting the prefix information. A typical approach to avoid this self-comparison is to place the pivot at an endpoint of the input.

With respect to handling repeated elements, popular approaches, such as always swapping repeated elements and 3-way partition (see e.g., [8]), guarantee that each element is compared exactly once in each partitioning, and the implicit structure defined by the order of recursive calls corresponds to a BST.

9.2 Experimental analysis

In the following, we evaluate in practice our solution to enhance parallel partitioning-based algorithms with fast string comparisons. Specifically, we present performance results for parallel quicksort and quickselect.

The tests have been run on a machine with 4 GB of main memory and two sockets, each one with an Intel Xeon quad-core processor at 1.66 GHz with a L2 cache of 4 MB shared among two cores. Thus, there are 8 cores in total. We have used the GCC 4.2.0 compiler with `-O3` optimization.

All tests have been repeated 20 times; figures show averages.

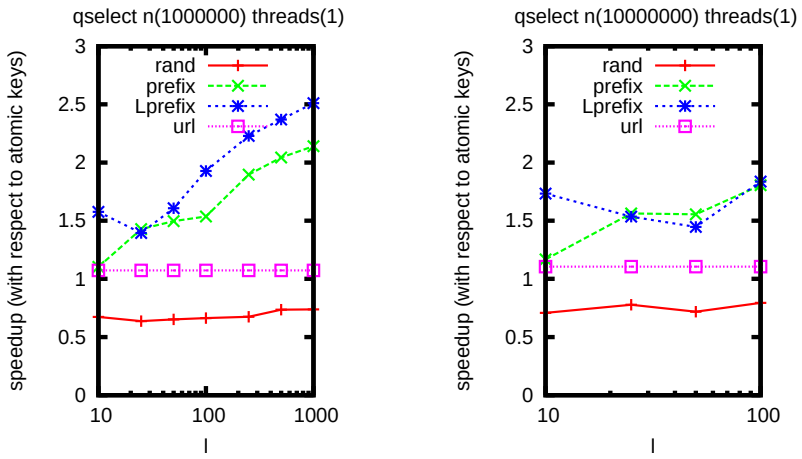
This section is organized as follows. First, we present our implementation. Then, we describe the tested datasets. It follows an overview of sequential results. Finally, we discuss parallel results.

9.2.1 Tested implementation

We have specialized the parallel and sequential partitioning algorithms in a generic way with respect to the specific string type. In particular, our implementations can tackle C-like strings, namely `char*`, and standard C++ `string`s. In this sense, applying our approach on the top of C++ `string`s has less relative overhead in terms of additional space needed. However, we present our results only for `char*` because most existing experimental studies and implementations are keyed for `char*`. Among these are the implementations in [20] about enhancing search trees with fast string comparisons, and burstsort (see e.g., [88]), a cache-efficient (burst)trie-based sorting algorithm.

The implementation is based on that for the parallel partitioning algorithms in Chapter 8. There, OpenMP is used for parallelization and we stick to it. In particular, we have implemented a simple static distribution to parallelize the independent work in quicksort. Anyway, any further enhancement that does not imply comparisons, such as load-balancing techniques (see e.g., [86, 97]), are perfectly compatible. It is not our aim here analyzing how quicksort can be parallelized best.

Besides, we have implemented sequential 3-way partitioning.



(a) $n = 10^6$ (b) $n = 10^7$

Figure 9.1: Performance results for sequential quickselect with enhanced string comparisons.

9.2.2 Datasets

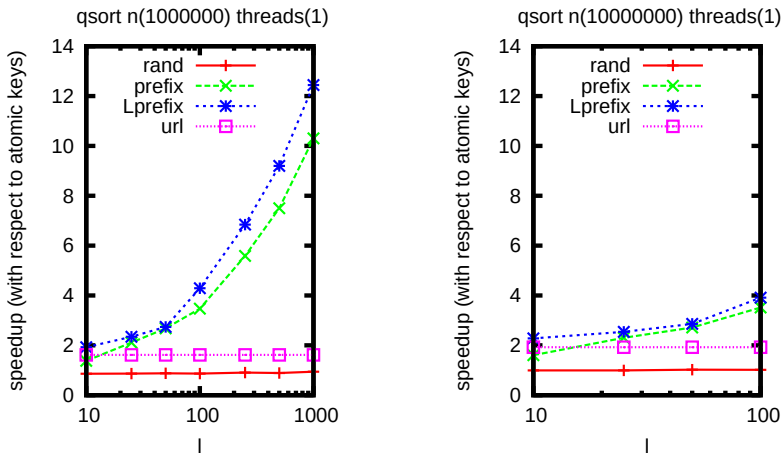
We consider string datasets that depend on several parameters (such as string length and alphabet size) and on the distribution of the string instances. We have run experiments with synthetic and real datasets. In all cases, once the data is generated, it is shuffled so that most accidental memory locality got in the generation is broken.

With respect to synthetic datasets, we have considered the following parameter combinations:

- string length (denoted by l): 10, 100, 1000
- alphabet: binary, A-Z (ASCII)
- distribution: uniformly random (denoted by `rand`), common prefix of uniformly random length (denoted by `prefix`), long prefix of fixed size (denoted by `lprefix`).
  In the case of `prefix` and `lprefix` the strings are made of two parts: a prefix of length u containing only one distinct character, and a suffix of length $l - u$ randomly generated. In the case of `prefix`, u is chosen randomly, in the case of `lprefix`, u is the same for all strings.

With respect to real datasets, we have considered the ones in [88] for burstsort. Among them, we focus on an url dataset (denoted by `url`) because the average string length is the highest. Recall from Section 2.3 that enhancing digital access on the top of comparison-based algorithms is specially useful for long strings and prefixes.

Finally, we have considered input sizes of 1 and 10 million elements. In the latter case, the longest string length considered is 100 because, otherwise, the dataset does not fit in main memory (in the considered experimental setting).



(a) $n = 10^6$ (b) $n = 10^7$

Figure 9.2: Performance results for sequential quicksort with enhanced digital comparisons.

9.2.3 Sequential performance

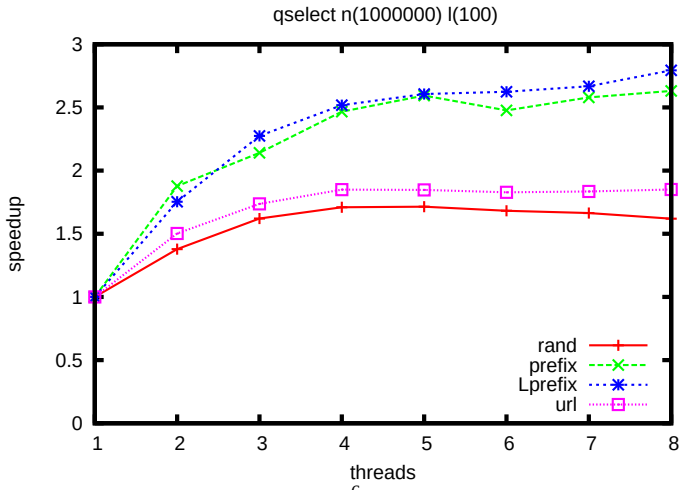
Figures 9.1 and 9.2 show, respectively, some performance results on sequential quickselect and quicksort for strings with enhanced digital comparisons. The results are shown as speedups with respect to the generic sequential quicksort and quickselect implementations in which keys are considered atomic. We consider several combinations of string length and number of elements. Specifically, we show results for words in the A-Z alphabet generated following `rand`, `prefix` and `lprefix` distributions. Moreover, we show the results compared to the `url` dataset (the assigned string length is arbitrary, just for plotting the results together). Finally, the pivot is selected using *approximate comparison*s (not significative differences where shown in performance with respect to choosing the pivot using exact comparisons).

In the case of quickselect (Figure 9.1), enhancing efficient digital comparisons harms if the common prefixes are scarce (i.e., random strings). Instead, in the case of quicksort (Figure 9.2), applying this technique is always beneficial. As we have seen in Section 4.1, the first partitioning always does more work than regular partitioning, and it is as we go deeper into recursion, that the gathered information on comparisons becomes really useful. In the case of quickselect, only part of the work done in one partitioning is used later, and so, the overhead of the first recursive calls is more notorious.

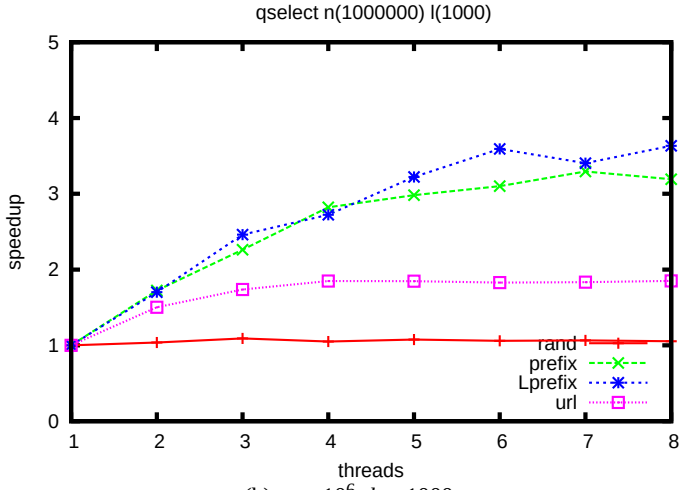
Logically, the longer the common prefixes, the better the performance with enhanced digital comparisons. Also, the results for binary alphabets are generally better, because it is much more likely to generate common prefixes when making random choices.

9.2.4 Parallel performance

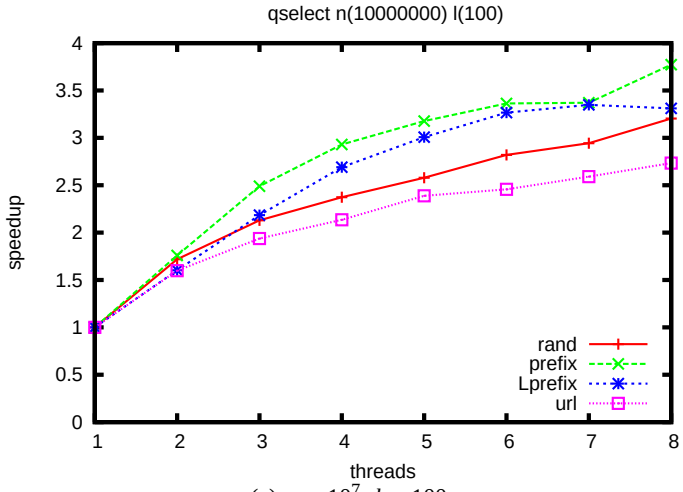
In the following, we describe performance results for parallel quickselect (Figures 9.3(a) to 9.4(c)) and parallel quicksort (Figures 9.5(a) and 9.5(b)). The results are shown both for the generic versions and for enhancing digital comparisons. Besides, we analyze the effect in



(a) $n = 10^6$, $l = 100$

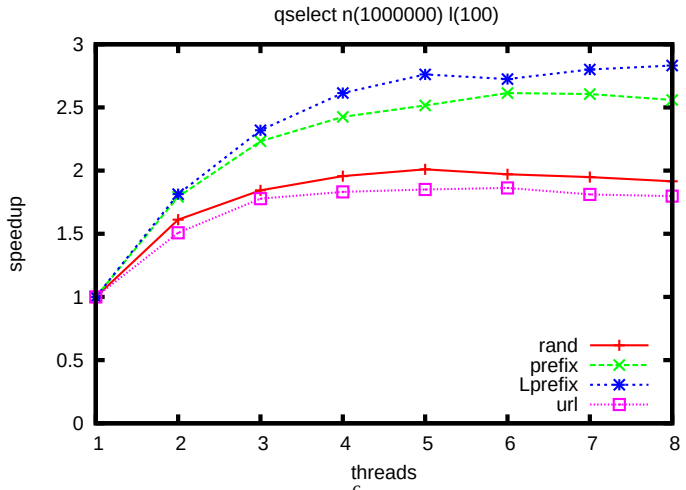


(b) $n = 10^6$, $l = 1000$

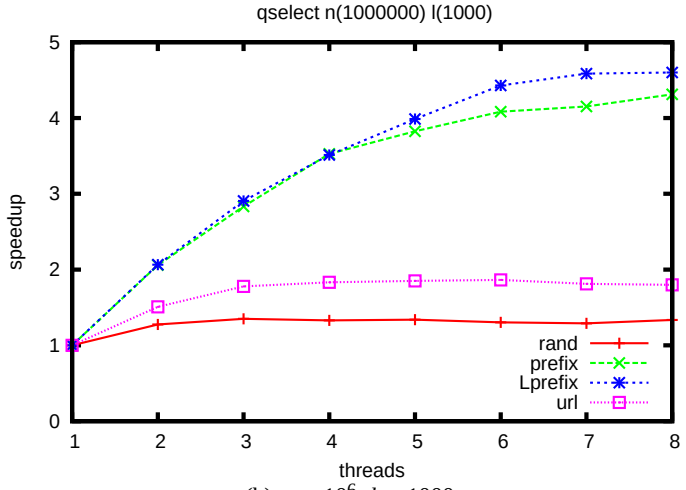


(c) $n = 10^7$, $l = 100$

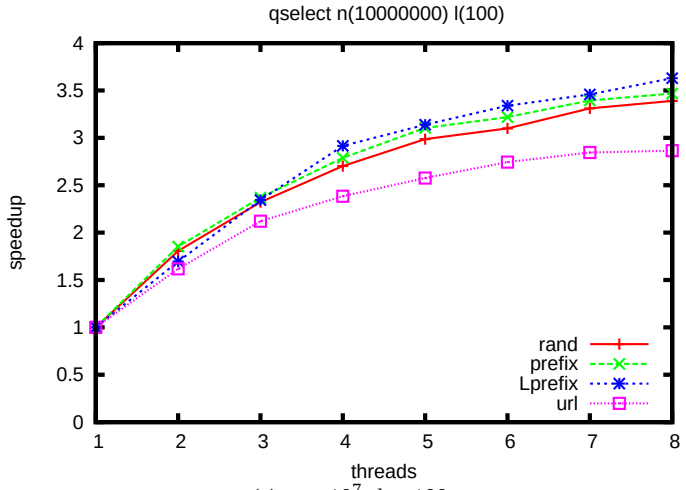
Figure 9.3: Performance results for generic parallel quickselect.



(a) $n = 10^6$, $l = 100$



(b) $n = 10^6$, $l = 1000$



(c) $n = 10^7$, $l = 100$

Figure 9.4: Performance results for parallel quickselect with enhanced digital comparisons.

parallel performance of several combinations of string length and number of elements. We focus in parallel quickselect results because the only source of parallelism is partitioning.

In all cases, the results are shown as speedups with respect to the respective sequential counterpart. The words are generated in the A-Z alphabet following `rand`, `prefix` and `lprefix` distributions. Moreover, we show the results compared to the `url` dataset (the assigned string length is arbitrary, just for plotting the results together). Finally, the results are shown for the F&A partitioning algorithm. We also run experiments for the BLOCKED partitioning algorithm. We do not tackle them here because they are very similar in relative terms, but slightly worse speedups are obtained.

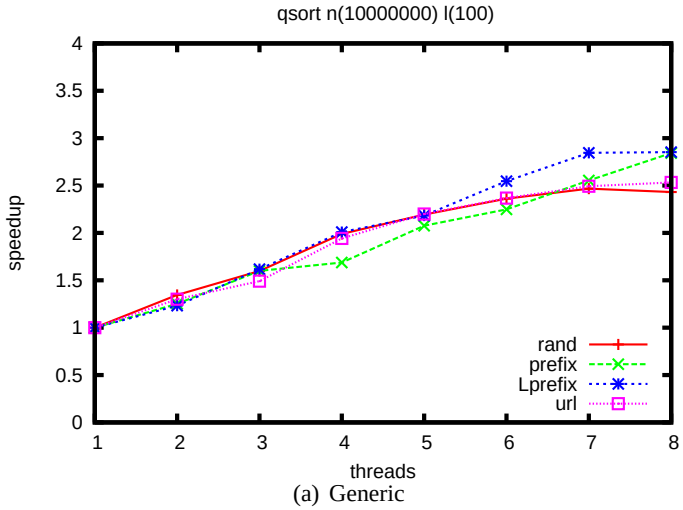
For a fixed set of string parameters, the greatest speedups are obtained when enhancing digital comparisons (compare Figures 9.3(a), 9.3(b), 9.3(c), 9.5(a) considering keys as atomic against Figures 9.4(a), 9.4(b), 9.4(c) and 9.5(b) considering digital keys, respectively). Therefore, enhancing fast digital comparisons for parallel partitioning-based algorithms not only does not hurt scalability, but generally improves it.

Comparing our parallel quicksort and quickselect implementations, quickselect achieves higher parallelism. However, recall that our parallel quicksort distributes the divide and conquer work statically, which is definitely not an optimal strategy. Nonetheless, the absolute speedups for quickselect are worse than those obtained in Chapter 8 for integers. First of all, the number of tested elements is smaller (because otherwise the data does not fit in main memory) and thus, parallelism is not so effective. Indeed, we can see comparing performance results for quickselect (Figures 9.3(a) against 9.3(c), or 9.4(a) against 9.4(c)), that for a fixed string length, the speedups are better as larger is the number of elements. This is also the case for quicksort. On the other hand, partitioning strings is more memory intensive than partitioning integers, because pointers to arbitrarily far locations must be followed. Indeed, the best speedups, both for the generic versions and for enhancing fast digital comparisons, are obtained for the longest common prefixes, because the cost of arithmetic operations (comparing characters and comparing prefix lengths, respectively) is more balanced with the cost of memory accesses. This is particularly true for quickselect (see Figures 9.3(b) and 9.4(b)).

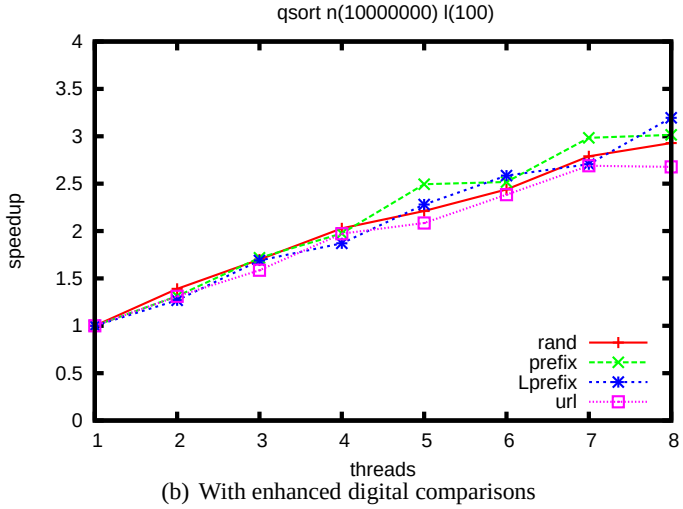
9.3 Conclusions and future work

In this chapter, we have shown how existing techniques to enhance string comparisons on the top of comparison-based algorithms can be applied in combination with parallel algorithms. Specifically, we have described how to enhance parallel partitioning-based algorithms, namely, quicksort and quickselect. This novel application of string techniques is independent of parallelism itself, as long as the properties on the relative order of comparisons are kept. In particular, in the case of parallel quicksort and quickselect, the main issue is that each element must be compared at most once in each partition. This can be achieved relying on our parallel partitioning algorithms presented in Chapter 8.

From an engineering perspective, we have described how the C++ design and implementation of these parallel partitioning algorithms can be refined to provide a non intrusive specialization for strings. In particular, we have considered both standard `string` and (C-like) `char*`. Besides, we have discussed additional precautions that must be taken, for instance, when choosing the pivot.



(a) Generic



(b) With enhanced digital comparisons

Figure 9.5: Performance results for parallel quicksort ($n = 10^7$, $l = 100$).

Finally, we have presented some experimental results that indicate the practicability of this approach. Indeed, the scalability of parallel quicksort and quickselect for strings is generally better when combined with efficient digital comparisons. However, the absolute speedup results are far to be optimal.

As a next step, our parallel partitioning algorithms enhanced for strings could be used together with highly-tunned and tested implementations of parallel quicksort and quickselect. We expect that doing so would not damage the scalability of the original algorithms. In particular, it could be interesting to integrate our parallel partitioning algorithms into some existing parallel implementations of the STL. However, to the best of our knowledge, no such string specialization exists for a sequential implementation, and so, that should be done in the first place.

Furthermore, the experimental work in this chapter could be extended by adding redundancy in parallel partitioning algorithms so as to improve cache efficiency (see Section 4.5). Indeed, our implementations already support this feature.

As further future work, the techniques for fast string comparisons could be combined with other parallel comparison-based algorithms and data structures. That is interesting from a practical perspective, because many parallelizations of comparison-based algorithms exist, while those are scarce for *ad hoc* algorithms.

For instance, our work in Chapter 6 for parallel bulk operations in red-black trees could be specialized for fast string comparisons. The parallel algorithms should be modified as follows. The construction algorithm should compute the common prefix of each string key with its ancestors in the searching path and store the result (recall that the nodes are stored in an array and so, the ancestors can be located in constant time by index computations). The common prefixes for an element can be computed most efficiently by also using this information from the ancestors. However, in a parallel context, that can only be done safely if both nodes are computed by the same thread. Besides, the rest of modifying operations, including split and union, should be specialized to use and update the associated information regarding common prefixes.

Chapter 10

Conclusions

This thesis has studied several problems that arise from combining theory and practice in algorithmics, in particular when considering requirements in software libraries. The well-known Standard Template Library (STL) of the C++ programming language has been the common thread of this research. The main targets have been sequences, dictionaries, and the sorting and selection problems. Making improvements in such fundamental problems and in such a well-established library as the STL was a challenge. This goal has been achieved through a synergy of computational techniques and research approaches. Specifically, I have combined techniques from analysis and design of algorithms and data structures, high performance computing, experimental algorithmics, and algorithm engineering. [1] Besides, in all cases, I provide an implementation (Appendix A gives the exact links). With respect to the approaches, I have taken advantage of the particularities of string keys and the capabilities of modern computers; namely, memory hierarchies and parallel cores. In the following, whilst summing up the contributions in this thesis, I highlight some of their deriving future research lines and difficulties.

Most of the contributions in this thesis concern generic data structures and algorithms that take advantage of hardware capabilities. First, three cache-conscious implementations of STL `lists` have been presented. In contrast to conventional cache-conscious approaches, these lists fully support iterator functionalities while providing worst-case guarantees about their cache efficiency. According to experimental results, my implementations are much faster than a doubly linked list implementation, in particular for traversals. Obtaining big performance improvements for such a basic component as `list` was particularly rewarding. I want to stress that the actual implementation was rather involving, even requiring attention to low-level details. In particular, if iterator operations are not properly inlined by the compiler, that forecloses any gain.

Second, a parallel implementation of bulk operations in STL dictionaries practical for multi-core computers has been presented. This has been done extending the GCC red-black tree implementation. The proposed parallel algorithms take advantage of constant-cost index computations to access the elements, as in algorithms described for the PRAM model, and split and union operations in red-black trees; as well as of load-balancing techniques.

[1] Throughout this document I have used an impersonal style; namely the use of the “we” pronoun should be understood as “the reader and me” or “my collaborators and me”. By contrast, in this chapter I rather use “I” to emphasize my personal perspective on the work done in this thesis.

The experiments have showed that great speedups can be obtained. Nonetheless, the benefits are limited by suboptimal allocator performance (memory allocators have been considered as black boxes in this thesis). From my point of view, the explosion of practical parallel data structures requires effective and flexible parallel allocators to be available and, in particular, integrated into standard libraries.

Third, an elegant, yet powerful, algorithm to partition linearly traversable sequences of unknown size into even-sized independent pieces has been presented. This problem arises, for instance, in preprocessing the input for parallelization of STL algorithms (most of them require only forward access on the elements). In particular, its study was motivated by its application in parallelizing the aforementioned bulk operations for dictionaries. The proposed algorithm is online (i.e., it only needs one pass on the data) and uses sublinear additional space. Besides, experimental evidence of its practicability for parallelization has been provided. In addition, this algorithm has been included in the *libstdc++ parallel mode*, which is a parallel implementation in the GCC compiler of many STL algorithms.

Furthermore, new variants of algorithms by exploiting the reuse of computations have been devised. Specifically, the first practical parallel partitioning algorithms for multi-core computers that perform an optimal number of comparisons, i.e., one per element have been presented. That is, all the previous existing practical implementations needed to compare some elements twice. The resulting implementations can be used to parallelize STL `partition`. In turn, parallel quicksort and quickselect based on this parallel partitioning algorithms can be used to parallelize STL `sort` and `nth_element`, respectively. However, when considering keys as atomic, no significant performance gains were obtained with respect to other implementations. By contrast, when considering string keys and their digital structure, these new algorithms can be neatly combined with existing techniques to avoid redundant character comparisons for strings. Indeed, it was specially gratifying to ascertain the feasibility of this novel combination, namely parallel algorithms and techniques to enhance digital comparisons for strings on the top of comparison-based algorithms. Besides, these implementations are amenable. In particular, some great speedups have been shown for parallel quicksort and quickselect. Overall, I consider that this is a powerful approach because there is a large body of parallel comparison-based algorithms, in contrast to the case of parallel *ad hoc* algorithms for strings.

In addition, so-enhanced comparison-based algorithms and data structures for strings have been considered from a cache-conscious perspective. The motivating observation was that the outcome of some comparisons can be determined by merely using information from other comparisons. Thus, under the common assumption of a string implementation through pointers, costly memory accesses are saved. The number of string lookups in ABSTs (that is, in so-enhanced BSTs for strings) has been analyzed relating string lookups to known properties in tries. Besides, the benefits of adding redundancy to avoid one common case of string lookups have been analytically shown. Finally, the analysis has been extended for so-enhanced quicksort and quickselect for strings. Once the correspondence between properties of TSTs and ABSTs was set, obtaining the results was rather straightforward because existing results for TSTs could be reused. A priori, though, the analysis seemed rather involving. In addition, other authors have shown the practicability of so-enhanced red-black trees for strings (in particular, adding redundancy) to specialize STL dictionaries. Note that, in contrast, tries or alike cannot keep up with the requirements of STL dictionaries.

Finally, *multikey quickselect*, the analogous of multikey quicksort for selection, has been presented, analyzed and implemented. In particular, it can be used to specialize STL `nth_element`. The proposed algorithm is rather straightforward. In fact, it was rather surprising to find out that it had not been formally described before. Nonetheless, there were some decisions to be made which only arose when carefully defining and analyzing the algorithm. In particular, a variant has been proposed that uses binary partitioning instead of ternary partitioning. For all variants, a detailed analysis for the number of comparisons and swaps has been provided. Besides, enhanced algorithms have been described and analyzed that avoid a source of useless comparisons and swaps by using information from the alphabet. In addition, analogous variants can be obtained for multikey quicksort, for which also implementations have been provided.

From my point of view, a specially interesting target that arises from this thesis is taking advantage of the particularities of common data types, like strings, to obtain more efficient STL components. To the best of our knowledge, no widely known STL (or alike) implementation has specialized components for the `string` datatype. In fact, most of efficient implementations of string algorithms and data structures in the literature are only provided for C-like `char*` strings or/and cannot cope with STL requirements. By contrast, generic string implementations have been supplied in this thesis. In particular, some of them take advantage of multi-core computers and memory hierarchies. Also, efficient string selection algorithms have been proposed, to which not much attention had been paid before. Still, some further work needs to be done to make the proposed implementations completely generic and STL compliant (e.g., considering character traits). Besides, in the new C++ forthcoming Standard, new character types for improved Unicode support are going to be introduced. Thus, it will be relevant, from a practical perspective, reconsidering existing *ad hoc* algorithms and data structures under the assumption of a big alphabet cardinality. In turn, this can lead to interesting theoretical problems.

This thesis has also arisen some of the issues that hamper algorithmic research in the context of data structures libraries. First, some of the requirements in data structures libraries are very tied to an specific implementation. In some cases, proposing alternatives keeping up with all requirements is not feasible. For instance, the definition of STL dictionaries is very tied to red-black tree properties. In this sense, I definitely favor more flexible libraries, in which a smaller set of tight requirements is stated component-wise, and besides, where mechanisms to choose among several alternatives are provided (as it happens in LEDA or with STL container adaptors). In other cases, the limitations come from the language (or the associated tools) themself. For instance, memory allocators do not allow an asymmetric usage that would help in parallelization and cache efficiency. Additionally, OpenMP, which has been used to implement the parallel algorithms in this thesis, does not allow throwing exceptions over parallel region boundaries. Another issue is the lack of statistical data on the usage of the STL. We can easily argue than the STL is widely extended. In particular, several other research projects take the STL as a focus (CPH-STL, STXXL, MCSTL...). Still, I could not find any data on the actual usage of the components, i.e., how frequently each method is used, which methods are used in combination, etc. This piece of information would help in the design of more efficient data structures and algorithms and would strengthen some of the possible decisions.

Overall, this thesis has aimed at narrowing the gap between theory and practice in algo-

rithmics, providing new efficient algorithms and data structures in the context of the STL. My approaches have been diverse and evolving, also guided by fast and radical changes in available technology. Namely, at the starting point of this thesis, parallel computation was not much of an issue in every-day programming. Nowadays, with multi-core computers all around, there is no doubt about the need of parallel computation to obtain more efficient programs, and the importance of tools and libraries that ease this task.

Appendix A

Implementations

All the algorithms and data structures in this thesis have been implemented in C++ and in the context of the STL. Links to every implementation can be found through my webpage at the *Departament de Llenguatges i Sistemes Informàtics* of the *Universitat Politècnica de Catalunya* (`www.lsi.upc.edu/~lfrias`). The source code used for experimentation is also included. Typically, the later consists of a test program in C++, plus several Perl scripts to run the experiments and plot the obtained results. In order to guarantee the availability and persistence of the code after the end of this thesis, all the implementations that are not hosted anywhere else will also be hosted in SourceForge. SourceForge.net is the world's largest open source software development web site and eases external collaboration. The following table gives the exact links for each implementation.

Contribution	Link to implementation
Cache-conscious STL `list`	`http://sourceforge.net/projects/cachelists`
Parallelization of bulk operations for STL dictionaries	`http://algo2.iti.kit.edu/singler/mcstl` (0.8.0-beta version)
Single-pass list partitioning	`http://gcc.gnu.org` (4.3 version on)
Generic parallel partition	`http://sourceforge.net/projects/parpartition`
CAQSORT and CAQSEL (sequential and parallel)	`http://sourceforge.net/projects/stringbsts`
Multikey quicksort and multikey quickselect	`http://sourceforge.net/projects/mkqsel`

Bibliography

[1] D. Abrahams. Exception-safety in generic components. In *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 69–79, Berlin, Heidelberg, 2000. Springer.

[2] A. Aggarwal and S. J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1-3, 2001. Revised Papers*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208, Berlin, Heidelberg, 2003. Springer.

[5] A. Andersson and S. Nilsson. Implementing radixsort. *Journal on Experimental Algorithmics*, 3(7), 1998.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[7] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–242, New York, NY, USA, 2006. ACM Press.

[8] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, 1993.

[9] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices, 35(11), pages 117–128, New York, NY, USA, 2000. ACM Press.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[12] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[13] Boost. Boost C++ libraries. `http://www.boost.org`.

[14] J. Bourdon. Size and path length of patricia tries: dynamical sources context. *Random Structures and Algorithms*, 19(3-4):289–315, 2001.

[15] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *STOC'03: Proceedings of the thirty-fifth annual ACM Symposium on Theory of Computing*, pages 307–315, New York, NY, USA, 2003. ACM Press.

[16] G. S. Brodal and R. Fagerberg. Cache-oblivious string dictionaries. In *SODA'06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, New York, NY, USA, 2006. ACM Press.

[17] J. Clément, P. Flajolet, and B. Vallée. Dynamical sources in information theory: A general analysis of trie structures. *Algorithmica*, 29(1):307–369, 2001.

[18] The C++ Standards Committee. The C++ standards committee, 2009. `http://www.open-std.org/JTC1/SC22/WG21/`.

[19] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[20] P. Crescenzi, R. Grossi, and G. F. Italiano. Search data structures for skewed strings. In *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*, volume 2647 of *Lecture Notes in Computer Science*, pages 81–96, Berlin, Heidelberg, 2003. Springer.

[21] E. Demaine. Cache-oblivious algorithms and data structures. In *EEF Summer School on Massive Data Sets*. 2002.

[22] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[23] L. Devroye. A study of trie-like structures under the density model. *The Annals of Applied Probability*, 2(2):402–434, 1992.

[24] L. Devroye. Universal asymptotics for random tries and PATRICIA trees. *Algorithmica*, 42(1):11–29, 2005.

[25] DIKU (Performance Engineering Laboratory). CPH STL (Copenhaguen STL). `http://www.cphstl.dk`.

[26] P. Ferragina and R. Grossi. Fast string searching in secondary storage: theoretical developments and experimental results. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 373–382, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[27] P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[28] G. Franceschini and R. Grossi. A general technique for managing strings in comparison-driven data structures. In *ICALP'04: Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 606–617, Berlin, Heidelberg, 2004. Springer.

[29] R. S. Francis and L. J. H. Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18(5):543–550, 1992.

[30] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[31] L. Frias. Implementació dels maps de la STL usant LBSTs. Master's thesis, Facultat d'Informàtica de Barcelona, July 2004. Codirected by J. Petit and S. Roura. In Catalan.

[32] L. Frias. Extending STL maps using LBSTs. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*, pages 155–166. Society for Industrial and Applied Mathematics, 2005.

[33] L. Frias. On the number of string lookups in BSTs (and related algorithms) with digital access. Technical report LSI-09-14-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, 2009.

[34] L. Frias and J. Petit. Parallel partition revisited. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 142–153, Berlin, Heidelberg, 2008. Springer.

[35] L. Frias and J. Petit. Combining digital access and parallel partition for quicksort and quickselect. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 33–40, Washington, DC, USA, 2009. IEEE Computer Society.

[36] L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. In *Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings*, volume 4007 of *Lecture Notes in Computer Science*, pages 121–133, Berlin, Heidelberg, May 2006. Springer.

[37] L. Frias, J. Petit, and S. Roura. Lists Revisited: Cache Conscious STL Lists. *Journal of Experimental Algorithmics*, 14:3.5–3.27, 2009.

[38] L. Frias and S. Roura. Multikey Quickselect. Technical report LSI-09-27-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes Informàtics, 2009.

[39] L. Frias and J. Singler. Parallelization of Bulk Operations for STL Dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007, Rennes, France, August 28-31, 2007, Revised Selected Papers*, volume 4854, pages 49–58, Berlin, Heidelberg, March 2008. Springer.

[40] L. Frias, J. Singler, and P. Sanders. Single-Pass List Partitioning. In *The Second International Conference on Complex, Intelligent and Software Intensive Systems*, pages 817–821. IEEE Computer Society Press, March 2008.

[41] L. Frias, J. Singler, and P. Sanders. Single-pass list partitioning. *Scalable Computing: Practice and Experience*, 9(3):179–184, 2008.

[42] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.

[43] GNU GCC. GCC C++ Standard Library, 2009.
`http://gcc.gnu.org/onlinedocs/libstdc++/`.

[44] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing a MIMD, shared-memory parallel machine. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 239–254, New York, NY, USA, 1998. ACM Press.

[45] R. Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 372–381, Berlin, Heidelberg, 1999. Springer.

[46] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, 16-18 October, 1978, Ann Arbor, Michigan, USA*, pages 8–21. IEEE, 1978.

[47] P. Heidelberger, A. Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):133–138, 1990.

[48] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[49] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. 1995.

[50] International Standard ISO/IEC 14882. *Programming languages — C++*. American National Standard Institute, 1st edition, 1998.

[51] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981, Proceedings*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431, Berlin, Heidelberg, 1981. Springer.

[52] J. JáJá. *An introduction to parallel algorithms*. Addison-Wesley, Redwood City, CA, USA, 1992.

[53] N. Josuttis. *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley, 1999.

[54] J. Kärkkäinen and T. Rantala. Engineering radix sort for strings. In *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008. Proceedings*, volume 5280 of *Lecture Notes in Computer Science*, pages 3–14, Berlin, Heidelberg, 2008. Springer.

[55] E. Kim and K. Park. Improving multikey quicksort for sorting strings with many equal elements. *Information Processing Letters*, 109(9):454–459, 2009.

[56] D. E. Knuth. *The art of computer programming, volume 3 (3rd ed.): Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[57] V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[58] A. Lamarca. *Caches and algorithms*. PhD thesis, University of Washington, 1996.

[59] J. Liu, C. Knowles, and A. Davis. A cost optimal parallel quicksorting and its implementation on a shared memory parallel computer. volume 3758 of *Lecture Notes in Computer Science*, pages 491–502, Berlin, Heidelberg, 2005. Springer.

[60] Dinkum Ware Ltd. Dinkum. `http://www.dinkumware.com/`.

[61] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[62] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[63] H. Mahmoud, P. Flajolet, P. Jacquet, and M. Régnier. Analytic variations on bucket selection and sorting. *Acta Informatica*, 36(9-10):735–760, 2000.

[64] H. M. Mahmoud, R. Modarres, and R. T. Smythe. Analysis of quickselect: An algorithm for order statistics. *ITA – Theoretical Informatics and Applications*, 29(4):255–276, 1995.

[65] C. Martínez and S. Roura. Optimal sampling strategies in quicksort and quickselect. *SIAM Journal on Computing*, 31(3):683–705, 2001.

[66] F. Martínez. Especialització dels maps de la STL per strings. Master's thesis, Facultat d'Informàtica de Barcelona, 2009. Directed by L. Frias. In Catalan. `http://hdl.handle.net/2099.1/7671`.

[67] P. M. Mcilroy, K. Bostic, and M. D. Mcilroy. Engineering radix sort. *Computing Systems*, 6:5–27, 1993.

[68] K. Mehlhorn and S. Naher. LEDA: A library of efficient data types and algorithms. In *Mathematical Foundations of Computer Science 1989, MFCS'89, Porabka-Kozubnik, Poland, August 28 - September 1, 1989, Proceedings*, volume 379 of *Lecture Notes in Computer Science*, pages 88–106, Berlin, Heidelberg, 1989. Springer.

[69] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[70] X. Messeguer and B. Valles. Synchronized parallel algorithms on redblack trees. In *VECPAR'98: 3er International meeting on Vector and Parallel Processing*, pages 699–704, 1997.

[71] D. R. Morrison. PATRICIA: A practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[72] OpenMP. *OpenMP webpage*, 2009. `http://www.openmp.org`.

[73] OpenMP Architecture Review Board. *OpenMP Application Program Interface v 2.5*, 2005.

[74] OpenMP Multi-Threaded Template Library. *OpenMP Multi-Threaded Template Library webpage*, 2009. `http://spc.unige.ch/mptl`.

[75] H. Park and K. Park. Parallel algorithms for red-black trees. *Theoretical Computer Science*, 262:415–435, 2001.

[76] J. L. Hennessy D. A. Patterson. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[77] R. Pereira, P. Crocket, and G. Dias. A parallel multikey quicksort algorithm for mining multiword units. In *Workshop on Methodologies and Evaluation of Multiword Units in Real-world Applications (MEMURA Workshop) associated with the 4th International Conference On Languages Resources and Evaluation (LREC 2004)*.

Dias, G., Lopes, J.G.L. & Vintar, S. (eds), Lisbon, Portugal. May 25. 2008, Proceedings, pages 17–24, 2004.

[78] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[79] S. Roura. Digital access to comparison-based tree data structures and algorithms. *Journal of Algorithms*, 40(1):1–23, 2001.

[80] S. Roura. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM*, 48(2):170–205, 2001.

[81] C. A. Schaefer. Reducing search space of auto-tuners using parallel patterns. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 17–24, Washington, DC, USA, May 2009. IEEE Computer Society.

[82] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, 3 edition, 1998.

[83] S. Sen, S. Chatterjee, and N. Dumir. Towards a theory of cache-efficient algorithms. *Journal of the ACM*, 49(6):828–858, 2002.

[84] SGI. Thread-safety for SGI STL. `http://sgi.com/tech/stl/thread_safety.html`.

[85] J. Singler and B. Konsik. The GNU libstdc++ parallel mode: software engineering considerations. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 15–22, New York, NY, USA, 2008. ACM Press.

[86] J. Singler, P. Sanders, and F. Putze. The Multi-Core Standard Template Library. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694, Berlin, Heidelberg, 2007. Springer.

[87] R. Sinha and A. Wirth. Engineering burstsort: Towards fast in-place string sorting. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 14–27, Berlin, Heidelberg, 2008. Springer.

[88] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *Journal on Experimental Algorithmics*, 9:1.5, 2004.

[89] R. Sinha and J. Zobel. Using random sampling to build approximate tries for efficient string sorting. *Journal on Experimental Algorithmics*, 10:2.10, 2005.

[90] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *Journal on Experimental Algorithmics*, 11:1.2, 2006.

[91] SourceForge. Sourceforge.net, 2009. `http://www.sourceforge.net`.

[92] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), HP, 1995.

[93] STLPort Consulting. STLPort. `http://www.stlport.org`.

[94] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[95] B. Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1–4–59, New York, NY, USA, 2007. ACM Press.

[96] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF Regional Conference Series on Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

[97] D. Traoré, J. L Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel STL algorithms. In *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*, volume 5168 of *Lecture Notes in Computer Science*, pages 887–897, Berlin, Heidelberg, 2008. Springer.

[98] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003), 5-7 February 2003, Genova, Italy*, pages 372–381. IEEE Computer Society, 2003.

[99] L. G. Valiant. A bridging model for multi-core computing. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, pages 13–28, Berlin, Heidelberg, 2008. Springer.

[100] B. Vallée, J. Clément, J. A. Fill, and P. Flajolet. The number of symbol comparisons in quicksort and quickselect. In *36th International Colloquium on Automata, Languages and Programming (ICALP 2009)*, volume 5555 of *Lecture Notes in Computer Science*, pages 750–763, Berlin, Heidelberg, 2009. Springer.

[101] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[102] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[103] R. Wein. Efficient implementation of red-black trees with split and catenate operations. Technical report, Tel-Aviv University, 2005.

[104] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1998.

[105] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Information and Computation*, 97(2):150–204, 1992.