Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo – 2007

Cache Oblivious Computation of Shortest Paths: Theoretical and Practical Issues

Luca Allulli

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIX Ciclo - 2007

Luca Allulli

Cache Oblivious Computation of Shortest Paths: Theoretical and Practical Issues

Thesis Committee

Prof. Fabrizio d'Amore (Advisor)
Prof. Giorgio Ausiello
Prof. Camil Demetrescu

Author's address:

Luca Allulli
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: allulli@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼allulli

Acknowledgments

It is well-known that research is not an easy task, and that progress in research is discontinuous; nonetheless I always felt a bit uncomfortable in times where progress was slow, because I had an (unjustified) feeling to waste time. Luckily, I found in the Department of Computer Engineering of the University of Rome "La Sapienza" an extremely warm and positive environment.

Firstly, I would like to thank my advisor, prof. Fabrizio d'Amore. He has constantly been following, supporting and advising me, helpfully and tactfully. Working with Fabrizio has always been pleasant, and fun.

I would like to thank Giorgio Ausiello and Camil Demetrescu, members of my thesis committee: they superbly did their job, with support and wise suggestions in the crucial moments.

A special thanks goes to Luigi Laura for being a good professor, friend, co-author, fellow traveller, and... persuader. Luigi is the person that caused everything to begin: he first invited me to do my Master's thesis with him (as a co-advisor) and Giorgio (as the advisor); then he convinced me to begin the doctorate.

I owe a debt of gratitude to Norbert Zeh. Norbert's brilliant and clear way of writing, and the welcoming message I found of his website, struck me and induced me to contact him. When I visited Norbert at Dalhousie University, Halifax, Canada, I actually found a welcoming person, with a brilliant and clear way of explaining things. Furthermore he has been for me a great guide in the field of external algorithms, and a nice (and patient) co-author. Working with him and with Peter Lichodzijewski has been extremely instructive, fun and, considering Chapter 5, productive.

Enrico Puddu wrote some modules of the platform presented in Chapter 6, as part of his Master's thesis. He is a very nice person, and I am pleased to have collaborated with him.

I want to thank the reviewers of this thesis, that read two drafts and helped me in improving them. I wish I could have more time to implement all of their suggestions.

I am grateful to my roommates at the department, that let me relax and discover new fields of computer science: Vincenzo Bonifaci, Antonella

Chirichiello, Marco Fratarcangeli, Adnan Noor Mian, Andrea Ribichini. In particular, talking with Vincenzo has always been stimulating (he always has some cute problem to propose), and led to a scientific collaboration.

Last but not least, I would like to thank two professors of the department that always showed a special interest in my work: Maurizio Lenzerini, who coordinates the Doctorate program, and Roberto Baldoni. I am glad that the collaboration with Roberto yielded a scientific work.

Outside the university a lot of friends rendered my life brighter; a big thanks goes to Andrea, Daniele, lucAkela, Marco, Michele, and Vincenzo. And the biggest thank goes to God an to my relatives, especially my parents "papà" Giorgio and "mamma" Giosi, for their love. To them I dedicate this work.

Luca Allulli
Rome, December 2007

Contents

Chapter 1

Introduction

1.1 Massive data sets and memory hierarchies

This thesis deals with algorithms designed to process very large data sets. If we want to solve a problem on a large data set, such that the input instance or the intermediate data we need during the computation do not fit main memory, basically we have two choices: either we use (consciously or unconsciously) the virtual memory system of the machine, or we explicitly store some of the data on disk. In both cases the Random Access Machine (RAM) model is not adequate to help us in designing and analyzing algorithms.

If we wish to use the virtual memory system, we could still use the RAM model to design algorithms, but not to analyze them. In fact virtual memory is seen by algorithms as a unique, large memory level, like the memory algorithms for the RAM model work with. However, analyzing the cost of an algorithm using the RAM model does not yield meaningful results: it is not reasonable to assume that every access to a memory element costs the same. Memory accesses that involve a disk transfer are orders of magnitude slower than accesses that find the requested element in the main memory.

If, on the other hand, we want to explicitly transfer data between the main memory and the disk, with the RAM model we cannot even describe algorithms; the model must be extended with an abstraction of the disk, and of its interaction with the main memory.

Even when the input instance is not massive, and the whole computation can be carried on in main memory, one could desire to analyze the cost of an algorithm using more accurate models than the RAM model, because the memory architecture of modern computers is hierarchical: it consists of several levels, increasingly larger but slower, ranging from processor registers to several cache levels (L1, L2, sometimes L3), to the Dynamic Random Access Memory, and to the disk. Thus, again, it is not reasonable to assume that

every memory access has the same cost.

We can look at our issue from another perspective. The use of a hierarchical memory organization is justified by the observation that most software exhibits *locality*: informally, after an access to a memory element with address a, the probability that the same element or an element belonging to some neighborhood of a will be accessed in the near future is high. Thus, it makes sense to keep a neighborhood of the memory location with address a in a fast memory level for some time. Now, if we give a formal definition of locality, or (alternatively) we define an abstract machine model where locality is implicitly taken into account, we can transform an empirical observation into a design goal, i.e., we can build algorithms that are provably local, and thus perform well on machines with hierarchical memory organization.

1.2 Modelling memory hierarchies: the Ideal Cache and the External Memory models

The *Ideal Cache* model, introduced by Frigo *et al.* [39] (see [34] for a tutorial), allows to measure how much "non-local" an algorithm for the RAM is. The model is an abstraction of a virtual memory system: an algorithm for the Ideal Cache model can see and work with only one, unlimited memory level, the *disk*, as if it were an algorithm for the RAM. The model manages another memory level, the *cache*, with a limited size M, divided into blocks of size B. The disk is divided into blocks of size B, too; a subset of them are contained in cache. When an algorithm performs a memory access, if the block containing the accessed element is not in cache a *memory transfer* occurs: one of the cache blocks is evicted, using the optimal evicting strategy [54], and replaced by the block of the accessed element. Clearly, the fewer memory transfers an algorithm causes, the more local the algorithm is. Hence we can measure the quality of an algorithm by measuring its *I/O cost*, i.e. the number of memory transfers it performs as a function of the size of its input, in the worst case.

In order to analyze algorithms which explicitly transfer blocks between the main memory and the disk we can use the *External Memory* model by Aggarwal and Vitter [5]. The memory organization of this model is identical to that of the Ideal Cache model. But this time algorithms explicitly access the cache, rather than the disk, and can explicitly issue block transfers between the cache and the disk. Again, the I/O cost of an algorithm is defined as the number of memory transfers it performs as a function of the input size, in the worst case.

An algorithm for the Ideal Cache model which ignores the cache parameters B and M is called *cache-oblivious*. In contrast, an algorithm for the External Memory model must known B and M to manage memory transfers, and is

thus called *cache-aware*.

It can be shown [39] that, under weak assumptions that usually hold, the asymptotic cost of a cache-oblivious algorithm does not change if we manage cache with a physically feasible paging algorithm such as Least Recently Used (LRU) or First In First Out (FIFO) [54] instead of the optimal algorithm. Hence, consider an I/O-optimal cache-oblivious algorithm A: A can be run on any two-level cache system managed by LRU or FIFO, and will optimize transfers between levels, without needing to be configured with respect to M and B. In other words, A is *portable*. As a further consequence, when A is run on a machine with hierarchical memory organization, A optimizes transfers between any pair of adjacent memory levels [39], if every level is managed by LRU or FIFO. In other words, good cache-oblivious algorithms are local, and then they make a good use of the whole memory system.

But working without the knowledge of cache parameters is not an easy task: in general, cache-aware algorithms are strictly more powerful than cache-oblivious ones [24]. Even for problems where the asymptotic cost of cache-oblivious and cache-aware algorithms is the same, cache-aware algorithms are often conceptually simpler than their cache-oblivious counterparts, and this simplicity translates into smaller constant factors hidden by the asymptotic analysis of I/O costs.

1.3 I/O-efficient shortest paths

Suppose we want to compute the single-source shortest paths of a massive graph: we are given an undirected graph $G = (V, E)$, a positive edge length function $\ell : E \to \mathbb{R}^+$, and a special vertex s, the *source*. We want to compute, for every vertex $v \in V$, its *distance* $D(v)$ from s, i.e. the length of the minimum-length path connecting s to v. We may solve the problem using Dijkstra's algorithm, but it is easy to see that in the worst case we would incur at least one memory transfer per edge, and one memory transfer per vertex: too much, considering that with one memory transfer a whole block of B elements (containing, for example, $\Omega(B)$ edges) can be brought from disk to cache. Let us see where Dijkstra's algorithm is not local.

The algorithm visits vertices in increasing order of their distance from s. For every unvisited vertex v, it maintains a tentative distance $d(v)$ from s. When the algorithm visits a vertex v, the one with the smallest tentative distance among the still unvisited vertices, it learns that $d(v)$ is actually v's distance $D(v)$ from s. Then, it can improve the tentative distance of every unvisited neighbor w of v to $\min\{d(w), D(v) + \ell(vw)\}$ (this operation is called *edge relaxation*). How many memory transfers does this operation require, in the worst case?

First of all, the adjacency list of v must be accessed; since the blocks where it is stored could be anywhere on disk, in the worst case a memory transfer is needed to access it. Then, the algorithm must test, for every neighbor w of v, if w has already been visited. Again, if we store this boolean information with w, and w is in a random position on disk, a memory transfer for every neighbor is required in the worst case. Summing up over all the vertices we get the claimed cost of $\Omega(n+m)$ memory transfers, with $n = |V|$ and $m = |E|$.

We could improve the $\Omega(n)$ term if we managed to store the graph on disk locally with respect to the order we visit vertices. In other words, if we could (efficiently) preprocess the graph and store adjacency lists on disk so that adjacency lists that are close to each other tend to be accessed within a short time interval, we would reduce the number of accesses that cause a memory transfer.

To get rid of the $\Omega(m)$ term, when we visit a vertex v we can avoid to test whether its neighbors have been visited: we relax *all* the edges incident to v. The problem with this approach is that we could end up visiting vertices more than once; we must set up some mechanism to detect vertices that, in spite of having been visited, got a new tentative distance due to a spurious edge relaxation. This detection must occur at some time before such vertices are visited again.

Algorithms using the techniques hinted above have been presented in literature. The problems of undirected breadth-first search (BFS) and single-source shortest paths (SSSP) have been solved using cache-aware algorithms which perform less than one memory transfer per vertex, if the graph is sparse; undirected BFS has been solved cache-obliviously, too. In this work we describe the mentioned techniques in details, and present a new cache-oblivious algorithm for undirected SSSP.

1.4 Engineering cache-oblivious algorithms

The best known cache-oblivious and cache-aware algorithms for many problems share the same asymptotic I/O cost. Cache-oblivious algorithms optimize the usage of the whole memory hierarchy, whereas cache-aware algorithms are simpler and greatly benefit from the knowledge of cache parameters. An experimental comparison of the approaches would be desirable. However, there is not much literature about experimental analysis of cache-oblivious algorithms: only the basic algorithms and data structures have been implemented and tested (see, for example, [26, 16, 28]).

Possibly one reason for this is the poor support offered by general-purpose programming environments (i.e., high-level programming languages and libraries) to cache-efficiency. For cache-oblivious algorithms it is crucial to con-

trol the memory layout of data; whilst standard algorithms make an extensive usage of pointers to structure data. Pointers are supported or implicitly used by every programming language; complex memory layouts are not.

In this thesis we introduce CO_2, a C++ platform that aims at facilitating the implementation, engineering and experimentation of cache-oblivious algorithms.

1.5 Outline of the thesis

The rest of this work is organized as follows.

In Chapter 2 we explain how the memory system of modern computers is organized, and why. We present theoretical models that take into account the presence of several levels of memory, with different access costs. We consider the strong and the weak points of every approach, with a particular focus towards the Ideal Cache model and cache-oblivious algorithms.

In Chapter 3 we show the main techniques for designing cache-oblivious (and often cache-aware) algorithms. We present a set of basic cache-oblivious algorithms and data structures, extensively used in this work and in literature as building blocks for more complex algorithms.

In Chapter 4 we deal with graph traversal algorithms. In particular, we considers the problems of breadth-first search (BFS) and single-source shortest paths (SSSP) on undirected graphs. We first present the basic algorithms of Munagala and Ranade [50] and of Kumar and Schwabe [44], that incur at least one memory transfer per vertex. We then discuss how it is possible to improve this cost if the input graph is sparse, using the *clustering technique*. We introduce this technique in its original context, cache-aware BFS [46], and then we show how it is possible to extend it to cache-aware SSSP [48] and to cache-oblivious BFS [27]. Each of the extensions requires a hierarchical data structure with one dimension, the *hot pool structure*.

In Chapter 5 we combine all the techniques described in Chapter 4 to develop a novel algorithm for cache-oblivious SSSP on undirected graphs. Our algorithm can be considered as the perfect marriage between the algorithms for cache-aware SSSP and cache-oblivious BFS, because its cost reduces to that of cache-oblivious BFS when every edge has a unit length. By combining the extensions previously described we obtain a bi-dimensional hierarchical hot pool structure. The most challenging issue we cope is organizing and dynamically maintaining the memory layout of the hot pool.

In Chapter 6 we present CO_2, a platform for the implementation, experimentation and engineering of cache-oblivious algorithms, written in C++. We motivate our choice of implementing a platform for cache-oblivious algorithms (no other library existed so far); we expose the architecture of CO_2, showing

the layers it is constituted of; and we describe each layer in details.

Chapter 2

Complex memories: from empirical tricks to systematical exploitation

Contemporary computers and EDVAC, one of the first computers designed by John von Neumann, share the same high-level logical organization, namely the *von Neumann architecture* [40]. However, technological advances caused macroblocks of von Neumann's machine (CPU, memory, bus, etc.) to increase in complexity. The leading principle was to preserve the overall simplicity of the model, while concentrating development efforts on single parts of it in order to improve overall performance. The simplicity of the high-level model is essential to provide programmers with a clean, easy to understand and to use environment. Also from a theoretical point of view simplicity is desirable, because it makes it possible to model system with simple mathematical models. The universally adopted model of computers is the Random Access Machine (RAM) model [6]. It is taught and used since the first year of every computer science university course.

Nevertheless, these models are valid under the assumption that every "unit" operation (execution of a CPU instruction, memory access, etc.) is always performed with the same, unit cost. The complexity of the subsystems of recent computers invalidates, in general, this assumption. In fact, hardware designers empirically analyzed typical behaviors of programs (e.g. typical memory access patterns, typical flows of CPU instructions) and employed a number of ideas and tricks in order to improve system average performance. Thus in some cases RAM-like models are not accurate enough to give a good estimate of the cost of a program, or of an algorithm:

- Some algorithms do not exhibit the typical behavior on which the engineering efforts of subsystems have been targeted, and thus their actual

performance can be considerably worse than expected.

- Conversely, algorithm designers, deeply aware of how the subsystems are organized, can take advantage of their knowledge to develop more efficient algorithms.

In order to systematize this effort, and to provide algorithm designers with concrete algorithmic paradigms they can adopt, it is essential to build new, more sophisticated theoretical models of computers, that retain as much as possible the simplicity of the RAM model, while giving more accurate information on the complexity of algorithms on real-world machines.

This thesis is focused on the memory subsystem of computers. We first describe the way recent memories are organized, and why. Several theoretical models have been introduced in literature in order to capture the essential features of memory subsystems: we present a selection of them. We give particular emphasis to the recent Ideal Cache (or "Cache-Oblivious") model. Within this model we review a set of basic algorithms and data structures: we emphasize their algorithmic paradigms in order to collect a set of elementary tools, useful for the development of more sophisticated algorithms. We concentrate particularly on algorithms for graph traversal problems: we describe the so-called *clustering paradigm*, and adopt it in a novel, improved cache-oblivious algorithm for the single-source shortest paths problem. Finally, we deal with practical issues: we present a platform which aims at facilitating the implementation and the experimental analysis of cache-oblivious algorithms. Our platform is as general as possible with respect to the algorithms it can host (we implement many of the basic cache-oblivious algorithms and data structures, with the final purpose of running our shortest paths algorithm); to the methodology of analysis (it can be used to run experiments on *real* and *simulated* hardware), and to the scope of the implemented algorithms (usable in real-world applications). The original contribution of this thesis are the novel cache-oblivious single-source shortest paths algorithm and the experimental platform.

2.1 Hierarchical memory organization

At a high level, the main memory of computers can be seen as an array of *memory locations*, addressed with natural numbers, which contain symbols (natural numbers); every location can be individually, randomly accessed either to read or to modify its content. In the RAM model, every memory access and the execution of every CPU instruction are performed at unit cost. Since most CPU instructions operate on a small number of memory elements, one would like to perform every memory access in a time comparable to the

CPU cycle period. However, it is impossible to reach this goal for memories of arbitrary size [40]. First, fast memories are expensive. The faster the adopted technology, the more expensive the memory unit element. Second, and most important, there are physical limits on the speed of large memory devices, because the speed of electric signals propagating within devices is limited. An electric signal must leave the processor, reach the destination memory element, and come back to the processor within the memory access time τ; thus τ is proportional to the diameter D of the memory device. On the other hand, for a given technology (i.e. for a given density of memory elements) in a d-dimensional space, memory size s is proportional to D^d. Thus we obtain the bound $\tau = \mathrm{O}(\sqrt[d]{s})$ on the memory access time [21, 30].

To overcome these limitations and construct memory systems that, though large in size, have a small *average* access time, an empirical observation proves to be crucial: most computer programs exhibit high degrees of locality. Informally, a program is *local* if every access to memory is followed, with high probability, by new accesses to the same memory location (temporal locality) or to memory locations not far away (spatial locality). This observation yields the following idea: keep memory elements in a large, slow memory device. When it is necessary to access a memory element a for the first time, copy a and its neighborhood on a faster memory device. Subsequent memory accesses will likely find the accessed element in the fast device.

To explain why a's neighborhood is brought to the fast device together with a, another observation is necessary. Once a memory element has been accessed, on most memory devices it is possible to access neighboring elements at a considerably lower cost. It is possible to divide memory into contiguous *blocks* of elements such that the cost of accessing and transferring an entire block is comparable to (a small constant times) the cost of accessing and transferring a single element. Hence, thanks to spatial locality it makes sense to transfer an entire block to the fast memory device.

This idea can be exploited at different levels of granularity, leading to the *hierarchical memory organization*. A hierarchical memory system consists of h *memory levels* $L_1, \ldots, L_h$, stored in h different memory devices. Level L_{i+1} is larger, but slower, than level L_i. Level L_h contains all the memory elements addressable by the CPU.[1] Consider any pair of adjacent levels L_i, L_{i+1}. L_i and L_{i+1} are divided into blocks of size B_i; B_i is a parameter of the interface between adjacent levels, and in general interfaces between larger levels have larger block size. Every block of L_i is the exact copy of a block of L_{i+1}; thus level L_i contains a subset of the memory locations contained in L_{i+1}. We now describe an operation, QUERY(L_i, a), whose purpose is to enforce that

[1] For the sake of simplicity, our discussion assumes that the *subset-inclusion property* holds. We will discuss this property later.

the memory block containing location a is contained in L_i. If a is already contained in a block of L_i, nothing is done. Otherwise, a is recursively queried to L_{i+1} with a QUERY(L_{i+1}, a). The block b of L_{i+1} containing a is transferred to L_i, where it replaces another block. A *block replacement algorithm* (also known as *paging algorithm*) is responsible for choosing the block of L_i to *evict*, that is, to replace with b (see [54]). If the evicted block has been modified in L_i, it must be written to level L_{i+1} before being overwritten. Important examples of block replacement algorithms are FIFO (First In, First Out), which evicts the block that has been least recently brought to level L_i, and LRU (Least Recently Used), that evicts the block of L_i that has been least recently accessed.

Now, when the CPU needs to access a memory element a, it simply issues a QUERY(L_1, a) operation; and the memory system guarantees that a will be found in L_1. Notice that, if most QUERY$(L_1, \cdot)$ operations issued by the CPU find the requested element already in level L_1, then the average memory access time will be close to the access time of level L_1, the fastest one. But in the worst case, if the program is not local at all, every CPU access will cause level L_h to be recursively queried: then the hierarchical organization becomes completely useless.

When a block b is transferred from level L_{i+1} to level L_i, in general it cannot occupy an arbitrary slot of size B_i of L_i (even if L_i is completely empty). We say that level L_i is *k-way associative* if, for every block b, there are k slots of L_i which can hold b. If $k = |L_i|/B_i$, that is, if b can occupy an arbitrary slot of L_i, we say that level L_i is *fully associative*.

In our discussion we have assumed that, for $i \in \{1, \ldots, h\}$, every block contained in level L_i is also contained in level L_{i+1}. This assumption, known as the *subset inclusion property* assumption, does not hold in real systems in general: for example in commonly used operating systems, such as Linux and Microsoft Windows, *memory pages* (i.e. blocks into which main memory and the disk are partitioned) loaded into main memory may not be stored on disk. This behavior occurs, for example, when a new memory page is allocated: it is not written to disk until it is evicted from memory (if it ever is). However from a theoretical point of view we can always assume, and we will, that the subset inclusion property holds, by extending level L_i, for $i \in \{1, \ldots, h\}$ to include all memory blocks in levels $L_1, \ldots, L_{i-1}$; that is, we set $\widetilde{L_i} = \cup_{k=1}^{i} L_k$, and work with levels $\widetilde{L_1}, \ldots, \widetilde{L_h}$.

An alternative memory organization scalable with respect to memory size has been proposed by Bilardi *et al.* [17, 18, 19]: the *pipelined memory* organization. Pipelined memory allows the CPU to instantiate $\Theta(1)$ memory accesses per cycle, independently from memory size. At any time, a number of requests may be traveling on the bus, concurrently. Each request individu-

ally travels toward its destination memory element and back to the processor, taking a number of CPU cycles that depend on how far the destination location is from the CPU. Notice that, differently from the classic hierarchical memory organization, the pipelined memory organization is not compatible with von Neumann's high-level machine model; machine subsystems (including the CPU) and software must be appropriately designed to work with such a memory.

2.2 Using mass memory devices as additional memory space

While hierarchical memory organization is a clean, but sophisticated way to provide a large quantity of memory trying not to sacrifice speed, a simpler but less elegant solution has been adopted since the early years of computers. The idea is to use a mass storage device, like a hard disk, as an additional storage for data actively used by computer programs. Most DBMS do not load DB's in memory, where they only hold some buffers, and perform most operations directly on disk. In the 70's and 80's it was a common practice to split even machine code into segment (modules), and explicitly load from disk only those modules that were about to be executed, discarding other modules to free up memory. For the programmer this approach is, in general, considerably less practical than hierarchical memory, but it gives her a complete control over transfers between mass and main memory: better performance can be achieved, but only if the programmer — or the program — is aware of system properties, such as access time to single and to bunches of elements on main and mass memories, as well as memory size. Another advantage of this approach is that internal work and disk transfers sometimes can be parallelized: while the mass storage is loading the next bunch of information to process, the CPU can work on the current one. Finally observe that virtually every recent computer uses a hierarchical memory [40]: one should not neglect software locality, even for programs that handle explicitly mass memory.

2.3 Modeling locality

In this section we review mathematical models which allow to quantify software locality. A standard method to measure the efficiency of an algorithm is to compute its cost on an abstract machine model, on suitably chosen input instances (most notably, the worst case instance for every input size) and to compare it to the cost of other algorithms for the same problem and, if it is available, to a lower bound of the problem itself.

As we discussed earlier, the RAM model is not suitable for evaluating locality. Other machine models have been proposed to this end. The logarithmic-cost RAM [6] is identical to the RAM, except that an access to memory element at address a costs $\log a$. The model is similar to memory hierarchies in that it provides some fast memory (the very first memory locations), while increasingly larger areas of memory are increasingly slower. However, it diverges from real-world machines to a large extent:

- The choice of the logarithmic growth of access cost is arbitrary, and in general does not reflect real access costs. To mitigate this problem, the Hierarchical Memory Model (HMM) has been proposed [3], where the cost of accessing a memory element a is given by a non-decreasing positive function $f(a)$. Notice, however, that under the HMM it is difficult to design *portable* algorithms, in the sense that an algorithm that is efficient under a certain cost function can become inefficient under other functions. In other words, with the HMM one can only hope to develop efficient algorithms for a specific machine (or class of machines), rather than efficient algorithms for any machine with a hierarchical memory organization. For further information, see [20].

- Logarithmic-costs RAM and HMM do not model block memory transfers: if an algorithm wants to transfer a block of consecutive memory elements between different memory areas, it has to pay the full access cost for every element of the block. In the Hierarchical Memory with Block Transfer Model [4], instead, it is possible to transfer a whole memory block between different areas paying a full memory access cost for the first accessed locations of each block, plus a fixed cost (say 1) for every subsequent access to the same blocks.

- In the logarithmic-costs RAM and similar models, the algorithm must explicitly handle memory transfers in order to obtain good performance. In contrast, it would be desirable a model where a good locality of programs implies a good cost on the model, and a good performance on real-world hierarchical memory systems, where memory transfers are automatically managed.

2.3.1 The Parallel Disk Model

The Parallel Disk Model of Aggarwal and Vitter [5] approaches the problem of the unknown relative cost of different memory areas in a simple and elegant way: not making any assumption, and counting separately the number of accesses to *each* memory area.

The machine in the model has two memory levels: a fast, small one, usually called the *memory*, of size M, whose elements can be directly accessed by the processor by their address, as in the RAM model; and a slow, large level, composed of D arrays called *disks*, each divided into *blocks* of size B. Also the memory is divided in M/B slots of size B. At any time, the processor can issue an *I/O operation*, consisting in the transferral of $d \leq D$ blocks from d distinct disks to d memory slots, or vice versa from memory slots to disks. The cost of an algorithm A in the Parallel Disk Model is given as two functions of the parameters D, M, B and of the size N of the input instance: the *CPU complexity* A_{CPU}, i.e. the number of instructions executed by the CPU (as in the RAM model); and the *I/O complexity* A_{IO}, i.e. the number of I/O operations issued by the CPU on the worst input instance of size N.

Trivially, for a problem P any lower bound on the RAM model is also a lower bound for the CPU complexity in the Parallel Disk Model. For what concerns the I/O complexity, some lower bounds have been given in literature, most notably lower bounds for the fundamental problems of scanning, sorting and permuting an array of N elements [5]. The former two lower bounds have been asymptotically matched by corresponding algorithms; as we will discuss later (see Section 3.3), their high efficiency justifies the fact that they are used as building blocks in most algorithms and data structures in this model. Here we want to observe that, assuming that an I/O operation is k times more costly than a CPU operation, the total cost of an algorithm A is $A_{TOT} = A_{CPU} + kA_{IO}$. If A is asymptotically optimal in both the CPU and the I/O complexity, then it is portable, because A_{TOT} is asymptotically optimal for any value of k. However, notice that in real-world machines, where the Random Access Memory is modeled by the fast memory level and disks by the slow level, the value of k depends on block size, which should be chosen *a posteriori* for the considered algorithm A in order to optimize A_{TOT}. On the other hand, consider a problem P for which no asymptotically optimal algorithm both in the CPU and in the I/O cost is known. Here the cost functions given by the model can be used less directly: in order to choose a good algorithm for P, if many algorithms are available, a good trade-off between the CPU and the I/O complexity must be found; the trade-off depends on k, which in turn depends on B (and of course on the machine). As a consequence, it is not possible in general to identify a theoretically winning (and thus portable) algorithm for P.

In the literature, most problems have been faced in the special case where $D = 1$. In this case the Parallel Disk Model is more properly known as the *External Memory* (EM) model. Notice that external memory algorithms (algorithms with $D = 1$) can still take advantage of multiple disks, if disks are *striped*, i.e., if each block is split into several chunks, and each chunk is stored on a different disk. Disk striping essentially allows to increase the value of B

without sacrificing the access time, because all the chunks of a block can be read in parallel. In the rest of this thesis, we will consider the Parallel Disk Model only when $D = 1$.

2.3.2 The Ideal Cache model

The External Memory model is useful to develop algorithms that explicitly transfer data between two levels of memory. In order to manage memory transfers explicitly, the algorithm must know parameters M and B. The model provides no help in using efficiently a deep hierarchical memory.

Inspired by the External Memory model, Frigo *et al.* [39] proposed the *Ideal Cache* model to address this issue. As in the EM model, the memory of the Ideal Cache model consists of two levels, divided in blocks of size B: a fast, limited level of size M, and a slow, unlimited level. The CPU has no visibility of the fast level: it can only address, directly, elements of the slow level. In other words, any algorithm for the RAM model is an algorithm for the Ideal Cache model, and vice versa.

As in the EM model, the complexity of an algorithm A is given as two functions of M, B, and N: the CPU complexity, that is the number of instructions executed by the CPU, and the I/O complexity, that is the number of memory transfer between the two memory levels, in the worst case on input of size N. Data is always transferred in blocks between levels. At any time, the fast level contains a subset of the memory blocks of the slow level. When the CPU accesses a memory element a, if the block of a is contained in the fast level, no memory transfer occurs. Otherwise, the block of a is copied from the slow to the fast level, replacing another block. The block to evict from the fast memory is chosen by the optimal paging strategy, that is, it is chosen such that the total number of memory transfers is minimized. This behavior justifies the name of the model.

Traditionally, the fast memory level is known as *cache*, the slow one as *memory*. In order to avoid confusion with the homonymous term of the EM model, in the remainder of this thesis we will refer to the fast level of both models as *cache*, and to the slow level as *disk*.

To understand how this model relates to real-world machines, some issues should be addressed:

- The Ideal Cache model uses the optimal paging strategy, which is not physically feasible, since it needs to know future memory accesses in order to decide the page to evict (in other words, it is an *off-line* strategy).

- Memory hierarchies are composed of several levels of memory, while the Ideal Cache model has only two levels.

- Some levels in memory hierarchies have limited associativity, while the Ideal Cache model is fully associative.

If LRU, FIFO, or any other *marking* on-line paging strategy [54] is used instead of the optimal off-line strategy, then the asymptotic I/O complexity of an algorithm A does not change, provided that the I/O complexity of A satisfies a very weak regularity condition, namely that $A_{IO}(N, M, B) = \mathrm{O}(A_{IO}(N, 2M, B))$, i.e. that the I/O complexity of A gets worse by at most a constant factor if memory is halved [39]. Consequently an algorithm A can be conveniently analyzed in the Ideal Cache model, where any assumption on the behavior of the paging strategy yields an upper bound on its cost; and this cost is asymptotically valid when real-world paging strategies such as LRU are used.

In any multilevel hierarchical memory, consider any level L_i. Transfers between L_i and L_{i+1} can be modeled using a two-level cache with parameters $M = |L_i|$ and $B = B_i$, managed by the same paging algorithm that manages level L_i; cache represents L_i, while disk represents the slowest level in the hierarchy (i.e. the entire virtual memory space). It is easy to see that cache always contains the same blocks as level L_i (or levels $L_1, \ldots, L_i$, if the subset inclusion property does not hold); every memory access finds the requested element a in a level $L_{i'}$, $i' \leq i$, if and only if the block containing a is in cache. Then, the number of memory transfers incurred by the model is exactly the number of memory transfers between level L_i and L_{i+1} of the hierarchy. As a consequence, an efficient algorithm for the Ideal Cache model will handle efficiently memory transfers between any pair of adjacent levels in the hierarchy, if its definition does not depend on cache parameters M and B. An algorithm not depending on cache parameters is called *cache-oblivious*. This is perhaps the most theoretically appealing feature of cache-oblivious algorithms: they can optimize the usage of any multi-level hierarchy, even if they are analyzed on a two-level one [39]. Due to the importance of cache-oblivious algorithms, the Ideal Cache model is sometimes referred to as the cache-oblivious (CO) model. Conversely, external memory algorithm are often called *cache-aware* algorithms.

A satisfactory theoretical justification of the validity of the Ideal Cache model for not fully associative caches is missing. In [39], Frigo *et al.* point out that a fully associative cache can be simulated on an External Memory machine with the expected cost of $O(1)$ I/O's per memory transfer in the Ideal Cache model. But this result does not imply that an efficient cache-oblivious algorithm runs efficiently on memory levels that, *as a matter of fact*, are not fully associative. However, it is intuitive that a good cache-oblivious algorithm features a good locality and will likely perform well (apart from pathological cases) even on not fully associative memory levels.

2.3.3 Comparison and relations between models

As we have seen in the previous section, in [39] Frigo *et al.* showed that it is possible to simulate a cache-oblivious algorithm with an algorithm for the EM model, with the same cache parameters M and B, at the expected cost of one I/O per original memory transfer. This means that the EM model is more general than the CO one: in particular, any lower bound for a problem in the EM model is also valid in the CO model.[2]

Shortly after the introduction of the CO model, an astonishing number of cache-oblivious algorithms and data structures have been presented which often match the I/O cost of their EM counterparts. Considered problems include sorting, basic matrix operations, FFT [39], dictionary operations (B-trees and string dictionaries) [15, 26, 16, 25], priority queues [11, 23, 27, 32, 48], graph connectivity and traversal [11, 32, 27, 48], and several computational geometry problems [22, 2, 12].

Many of these results require an additional assumption of the form $M = \Omega\left(B^{1+\varepsilon}\right)$, either for a specific value of ε or for any fixed ε. Such an assumption is generically called *tall cache assumption*. Tall cache assumptions are commonly used also in external memory algorithms. For reasonable values of ε, e.g. for $\varepsilon = 1$ ($M = \Omega\left(B^2\right)$), these assumptions are satisfied by common real-world caching systems.

A natural and important question is whether the external memory model is *strictly* more powerful than the cache-oblivious one. This question has a positive answer: Brodal and Fagerberg in [24] show that (1) for the problem of permuting, no cache-oblivious algorithm can match the external memory lower bound; and, more significantly, (2) no cache-oblivious sorting algorithm attains the external memory lower bound without some tall cache assumption, whereas many optimal external memory sorting algorithms do not require it. We will discuss this result in more details in Section 3.3.2.

Since for many problems a cache-oblivious and a cache-aware algorithm with the same (or very similar) bound exist, another natural question is how their performance relate, in practice. The question is not a simple one, because both the models present advantages and disadvantages. Some points in favor of cache-aware algorithms are the following:

- Cache-aware algorithms are generally simpler than their cache-oblivious counterparts, because the latter must be general with respect to M and B. Cache-oblivious algorithms are often based on techniques, such as operating at several levels of granularity simultaneously, that allow them to "guess" the right value of cache parameters. These techniques are rather

[2] To be rigorous, any lower bound on the I/O complexity of *randomized* algorithms for the EM model is valid for any (randomized or deterministic) cache-oblivious algorithm.

expensive, even in cases when the asymptotic cost of the corresponding external memory algorithms is preserved.

- I/O operations performed by external memory algorithms are managed explicitly, while memory transfers incurred by cache-oblivious algorithms are performed by (not optimal) paging strategies. This worsens the actual number of memory transfers of cache-oblivious algorithms by another constant factor. (Notice, however, that most cache-oblivious algorithms actually have a good cache performance when LRU is used as a paging strategy.)

- When adopting an external memory algorithm, the value of B can be chosen to maximize performance of that particular algorithm. In a memory hierarchy, in contrast, the various block sizes B_i are general purpose values, chosen by hardware and/or operating system vendors. Notice, however, that tuning system parameters is not an easy task; a theoretically optimal tuning requires the knowledge of system parameters (access times, latencies, block transferral times, etc.) which not always are publicly disclosed by hardware vendors, and are influenced by factors difficult to control, like various levels of caching (operated, for instance, by the operating system and the disk controller).

- Some external memory algorithms use *prefetching* to parallelize internal and external work: while the algorithm is working on internal memory, it explicitly issues parallel I/O operations to write results from the last batch of internal work, and load into memory data for the next batch. On the other hand, cache-oblivious algorithms must rely to prefetching caching algorithms in order to parallelize work. Prefetching is currently done through hardware heuristics, and can speed-up memory transfers only between the uppermost levels of memory hierarchies (such as between L2 cache and main memory). In [51] Pan *et al.* empirically study the effectiveness of hardware prefetching with respect to cache-oblivious algorithms.

The main advantage of cache-oblivious algorithms is that they make a good use of the whole memory hierarchy, while cache-aware algorithms only handle efficiently transfers between memory and disk. Thus, the time spent by a cache-oblivious algorithm on "internal" operations, i.e. operations that only involve internal memory (excluding disk), may be competitive with that of cache-aware algorithms, because a better use of cache compensates the larger constant factors hidden by asymptotic analysis. This effect is particularly prominent with hierarchies with a large number of internal levels, and

makes some cache-oblivious algorithms the best choice for medium-sized input instances, instances i.e. whose size is comparable to the size of main memory.

An empirical study is hence fundamental to test the practical viability of a cache-oblivious algorithm. Some interesting experimental analysis of cache-oblivious algorithms have been presented [26, 16, 28]. In particular, in [28] Brodal *et al.* show that a carefully engineered implementation of a cache-oblivious sorting algorithm beats, in internal memory, production-quality implementations of QUICKSORT, on many platforms. If the arrays to be sorted do not fit internal memory, highly optimized cache-aware sorting algorithms prove to be the best option, even if the performance gap with respect to cache-oblivious sorting is not dramatic.

It would be unfair to conclude our discussion without citing other important benefits of cache-oblivious algorithms, not directly related to their performance:

- Cache-oblivious algorithms can be easily ported on any system because they do not need to be tuned on system parameters.
- In spite of being more involved than their cache-aware counterparts, cache-oblivious algorithms are generally easier to implement, because the programmer does not need to get into low level details regarding block transfers, neither directly nor indirectly (i.e. through the use of libraries). Moreover:
  - the programmer does not have to take care of packing information into disk blocks;
  - it is straightforward to use cache-oblivious algorithms and data structures as subroutines of other cache-oblivious algorithms, because the programmer does not have to take care of memory management; whilst the programmer of cache-aware algorithms must explicitly allocate segments of memory to every cache-aware subroutine or data structure in use.

Chapter 3

Basic cache-efficient algorithms and data structures

The goal of this chapter is twofold. On one side we present the main cache-oblivious algorithms and data structures, and their relationship with the corresponding cache-aware ones. On the other, we use them as a basis to discuss techniques and paradigms for designing cache-efficient algorithms.

Throughout this chapter, we assume to work with a cache of size M divided into blocks of size B.

3.1 Scanning

A very common operation in cache-efficient algorithms is scanning a number of consecutive elements in an array. Trivially, this can be done in external memory by using one or two cache blocks as a buffer:[1] when the block containing the element we are interested in is not in the buffer, we load it from disk; the next $\Omega(B)$ scanned elements will be found in the buffer (this may not happen only when the first, partially empty block is loaded). We use the big-omega notation to underline that the array does not need to be full; we only need that a constant fraction of it contains elements we are actually interested in. As we will see in this and in the next chapters, most dynamic data structures strive to keep their data into arrays such that, when a subarray is scanned, a constant fraction of its elements is actually used by the algorithm.

In this way, scanning N elements requires $1 + \mathrm{O}(N/B)$ I/O's. $\Omega(N/B)$ is a trivial lower bound on the I/O cost of accessing N distinct disk elements,

[1] We use two memory blocks if we want to be able to scan memory forward and backward.

because a disk access can bring in cache up to B distinct elements. In general a scan requires at least one I/O, in order to bring in cache the first block to be scanned; this cost cannot be amortized if $o(B)$ array elements are read. After the first scan of an array, it is possible to perform further scans without paying the non-amortizable cost by keeping the scanning buffer in cache.

Notice, finally, that a cache-oblivious algorithm which simply scans a region of N consecutive elements of its memory space incurs $1 + \mathrm{O}(N/B)$ memory transfers for its first scan, and optimally $\mathrm{O}(N/B)$ memory transfers for any subsequent scan, provided that the optimal paging strategy does not evict the cache pages used as a buffer. Since cache is made up of M/B blocks, the paging algorithm can allocate up to $\Theta(M/B)$ cache blocks to be used as buffers. In particular, under the tall-cache assumption $M = \Omega\left(B^{1+\varepsilon}\right)$, we have that $\Theta(M/B) = \Omega(B^{\varepsilon})$ arrays can be scanned without incurring non-amortizable I/O's. This property will be crucial in our cache-oblivious shortest paths algorithm, described in Chapter 5.

3.2 The divide-and-conquer paradigm

In their seminal paper [39], Frigo *et al.* identify recursion, coupled with a suitable data layout in memory, as one of the basic paradigms for designing cache-efficient algorithms. A divide-an-conquer algorithm partitions its input into one or more chunks, recurses on each chunk, and recombines the output of its recursive activation. Now, suppose that data of each chunk are stored consecutively on disk, and that a recursive activation of the algorithm needs no more space than the chunk itself (apart from some auxiliary space of constant size, such as local variables). Then, whenever the input chunk of a recursive activation fits cache, the whole recursive activation can be carried out without loading any block in cache more than once (and thus the optimal paging algorithm will incur in no more than one I/O per cache block). Similarly, whenever an input chunk fits $O(1)$ blocks, the whole recursive activation can be carried out paying $O(1)$ I/O's. In other words, our algorithm is able to partition the input instance into subinstances that fit cache (or cache blocks), and work internally on them. This approach works if there is a level of recursion where chunk size is sufficiently close to cache/block size, and if the total cost of recombining solutions at higher recursion levels is not too large.

Examples of optimal cache-oblivious recursive algorithms given in [39] include the deterministic median-finding algorithm (see [33]), blocked matrix multiplication and matrix transposition.

It is often useful to organize data in memory to favor locality. A good example is the *van Emde Boas* layout of binary trees, presented by Prokop [53] drawing inspiration from van Emde Boas [55, 56], which is the following

one: Store a complete tree T with N nodes, and height $h = \log_2(N+1)$, in an array a of size N. Consider the $2^{\lceil h/2 \rceil}$ nodes at height $\lceil h/2 \rceil + 1$: they are roots of $2^{\lceil h/2 \rceil}$ *bottom* disjoint subtrees $S_1, \ldots, S_{2^{\lceil h/2 \rceil}}$ of T. Removing these subtrees from T, we are left with a *top* (sub)-tree S_0 of T, consisting of all the nodes at height not greater than $\lceil h/2 \rceil$. In a, store S_0 first, then $S_1, \ldots, S_{2^{\lceil h/2 \rceil}}$. Notice that every subtree S_i has size bounded by $2^{\lceil h/2 \rceil} - 1 = \Theta\left(\sqrt{N}\right)$, and that the number of subtrees in a is $1 + 2^{\lceil h/2 \rceil} = \Theta\left(\sqrt{N}\right)$. For every subtree S_i, store it recursively using the van Emde Boas layout.

The van Emde Boas layout is profitably used by many algorithms. As an example we consider an iterative algorithm, binary search. Binary search traverses the searched tree T on a root-to-leaf path p. In the van Emde Boas layout, there is a recursion level k such that trees at level k have height between $\sqrt{B}$ and B. We call trees at recursion level k *basic trees*. As soon as path p enters a basic tree (from its root), it does not leave it until it reaches one of its leaves. Since a basic tree fits a constant number (1 or 2) of cache blocks[2], it costs $\mathrm{O}(1)$ memory transfers to traverse a basic tree. A basic tree has height at least $\Omega\left(\log \sqrt{B}\right) = \Omega(\log B)$. Then path p traverses at most $\mathrm{O}(h/\log B) = \mathrm{O}(\log_B N)$ basic trees, for a total cost of $\mathrm{O}(\log_B N)$ memory transfers, which matches the lower bound for external memory searching.

3.3 Sorting

In computing, sorting is such a fundamental operation that computers, in French, are called *ordinateurs* (sorters). In the area of cache-efficient algorithms sorting is even more important, because it can be done fairly efficiently. Consider two arrays a, b of tuples: let $a_i = (a_{i,1}, \ldots, a_{i,h})$ be the i-th element of a, $b_j = (b_{,1}, \ldots, b_{j,k})$ be the j-th element of b. If we sort both a and b by (say) first item of the tuples (i.e. by $a_{*,1}$ and $b_{*,1}$ respectively), we can scan a and b "in parallel": at any time, the first item of the currently scanned tuple of a corresponds to the first element of the currently scanned tuple of b. This technique is often used to efficiently exchange data between a and b.

The lower bound for sorting an array of N elements in external memory is $\Omega(\mathrm{sort}(N))$ I/O's [5], where $\mathrm{sort}(N) = (N/B)\log_{M/B}(N/B)$. This lower bound is attained by many different algorithms, that mostly adhere to one of the following paradigms: merging subarrays that have been recursively sorted (like MERGESORT); or distributing elements to subarrays which hold elements from disjoint intervals, recursively sorting every subarray, and concatenating them (like QUICKSORT). Before briefly describing the simplest of these algo-

[2] A basic tree is not larger than B, but, because of alignment, it may be spread over two cache blocks.

rithms, Multi-Way Mergesort (MWMS), we make some considerations on the sorting bound.

Consider the main memory of a typical computer: its size could be of $M = 1\text{GB} = 2^{30}\text{B}$, divided into blocks of size $B = 4\text{KB} = 2^{12}\text{B}$ (see [41]). We want to sort a huge array of size $128\text{GB} = 2^{37}\text{B}$. With these values, $\log_{M/B}(N/B) = 25/18 < 2$: it means that $\text{sort}(N) \approx N/B$, and then the I/O cost for sorting the array is comparable to the cost for scanning it. Even considering the interface between the L1 and L2 cache, with the typical values of $M = 32\text{KB} = 2^{15}\text{B}$ and $B = 64\text{B} = 2^{6}\text{B}$ (see [52, 36]), the logarithmic term remains limited: $\log_{M/B}(N/B) = 31/9 < 4$. In any case of practical relevance we can safely assume that $\text{sort}(N) \ll N$. Whenever possible, it is better to sort and scan an array than to access its element randomly, incurring, in the worst case, in 1 memory transfer per access.

3.3.1 A simple cache-aware algorithm

Mergesort has good locality properties (and is cache-oblivious). It is not hard to see that its I/O complexity is $\text{O}((N/B)\log_2(N/B))$. Notice that M does not appear in this bound: this is due to the fact that Mergesort does not use all the available cache effectively; it only employs 3 blocks to buffer, during Merge steps, the two input streams and the output stream. In order to improve its performance and use cache more effectively we can divide the input array into $\Theta(M/B)$ subarrays, one per cache block; sort them recursively; and merge them. (In fact, it is easy to merge up to $M/B - 1$ sorted arrays I/O-efficiently: it suffices to scan them "in parallel", allocating one cache block as a buffer for each input array, and one block for the output array. Each step of the merging algorithm consists in copying one element from one of the input arrays to the output array. The smallest uncopied element is selected. A binary heap can be used to select the smallest element CPU-efficiently.) We stop recursion as soon as we obtain a subarray that fits one cache block: at this point we sort the subarray internally. What we have described is the Multi-Way Mergesort algorithm.

To analyze the cost of MWMS, consider its recursion tree. For every level i, the overall I/O cost of merging elements at level i of the recursion tree is $\text{O}(N/B)$, because, at every node, merging up to $\text{O}(M/B)$ subarrays can be done with the scanning cost; and the total size of the subarrays to merge at all the nodes of level i is N, because the concatenation of these subarrays is a permutation of the array to sort. It remains to compute the height of the recursion tree. Since the fan-out of every node is $\Theta(M/B)$, the input array gets divided into subarrays of size $\text{O}(B)$ (base case of recursion) at level $h = \log_{M/B}(N/B)+1$. The I/O complexity of MWMS is thus $\text{O}((N/B)\cdot h) =$

$\mathrm{O}\Big((N/B)\log_{M/B}(N/B)\Big) = \mathrm{O}(\mathrm{sort}(N))$ (see also [57]).

3.3.2 Sorting cache-obliviously

MWMS is cache-aware; in [39] two cache-oblivious algorithms have been presented, based on the paradigms exposed above. One of these, FUNNELSORT, is essentially a version of MWMS which guesses the right fan-out of nodes of the recursion tree using a recursive approach. We overview the simplified version of FUNNELSORT by Brodal and Fagerberg [22, 28], called LAZY FUNNELSORT.

The algorithm splits the input array, of size N, into $N^{1/d}$ subarrays, for a fixed d; it recursively sorts each subarray, and merges them. The fan-out of nodes in the recursion tree does depend on input size, and then varies across levels. The idea is that there is a recursion level where the fan-out is roughly the optimal one of MWMS, i.e. $\Theta(M/B)$. For deeper recursion levels, input is so small that it fits cache, and can be sorted with no more memory transfers than those required to load it into cache. For less deep recursion levels, fan-out of nodes is $\omega(M/B)$: then it is not possible to use the classic merge algorithm of MWMS, because cache is not large enough to hold $\omega(M/B)$ memory buffers. The solution is to merge subarrays using a cache-oblivious merging algorithm, called FUNNEL MERGER, which itself uses recursion to adapt to cache parameters. For details, see [28].

FUNNELSORT is optimal under the tall cache assumption $M = \Omega\left(B^2\right)$. LAZY FUNNELSORT is parametric with respect to a parameter $\varepsilon \in \mathbb{R}^+$; for any *fixed* value of the parameter, the resulting algorithm requires the tall-cache assumption $M = \Omega\left(B^{1+\varepsilon}\right)$, which can be weaker than that of FUNNELSORT. However, the I/O complexity of LAZY FUNNELSORT is proportional to $1/\varepsilon$: it means that, in order to be more general (i.e. applicable to a broader class of caches), LAZY FUNNELSORT must be more costly; a behavior which clashes against the spirit of cache-obliviousness. But nothing can be done to avoid this: in [24] this behavior is proved to be the best possible in the cache-oblivious model, thus establishing a separation result from the external memory model. In absence of any tall-cache assumption, MERGESORT turns out to be the best possible cache-oblivious sorting algorithm.

We finally observe that neither (LAZY) FUNNELSORT nor the other cache-oblivious sorting algorithm of [39] is in-place. In [38] Franceschini presents the first cache-oblivious in-place optimal sorting algorithm.

3.4 Basic data structures

In this section we present two examples of cache-efficient data structures, namely priority queues and dictionaries. We underline that they have a very

different theoretical impact: priority queues are recurrent ingredients of cache-efficient algorithms, while dictionaries are more of independent interest.

3.4.1 Priority queues

A priority queue stores a set of elements from an ordered universe. It provides at least two operations, INSERT(p), which inserts element (or *priority*) p, and DELETEMIN(), which retrieves and removes the minimum priority from the priority queue. Observing that N invocations of INSERT($\cdot$), followed by N invocations of DELETEMIN() suffice to sort an array of size N, we infer that the total I/O cost of these $2N$ operations cannot be smaller than the sorting lower bound $\Omega\Big((N/B)\log_{M/B}(N/B)\Big)$. Thus there is a lower bound of $\Omega\Big((1/B)\log_{M/B}(N/B)\Big)$ on the amortized number of memory transfers per priority queue operation, where N is the total number of operations performed on the priority queue.

The first cache-oblivious priority queue matching this bound was presented by Arge *et al.* in [11]. It also supports a DELETE(p) operation, allowing to remove an arbitrary element from the priority queue. Some algorithms require a different kind of priority queue, whose elements have not only a priority p, but also a *key* x, drawn from an ordered universe. Each key may appear at most once in the priority queue. An additional operation DECREASEKEY(x, p) decreases the priority of element with key x to p, if p is less than the current priority of x; otherwise it does nothing. DECREASEKEY($\cdot$) operation is used in Dijkstra-like shortest paths algorithms. Chowdhury and Ramachandran [32] and, independently, Brodal *et al.* [27] developed priority queues providing DELETE($\cdot$), DELETEMIN() as well as UPDATE($\cdot$) operations. UPDATE($\cdot$) is a combination of the INSERT($\cdot$) and the DECREASEKEY($\cdot$) operations: it acts as the latter if element with key x is already present in the priority; as the former otherwise. These data structures do not match the complexity bound of [11], because the amortized cost of the execution of any of the above mentioned operations is $O((1/B)\log_2(N/B))$ memory transfers. Notice that in the external memory model exactly the same bounds hold: there are priority queues with an optimal amortized cost of $O\Big((1/B)\log_{M/B}(N/B)\Big)$ I/O's per operation, but they do not support DECREASEKEY($\cdot$) (see [10]), while the best cache-aware priority queue supporting DECREASEKEY($\cdot$) costs $O((1/B)\log_2(N/B))$ I/O's per operation, amortized [44]. The problem of closing this gap, or proving that priority queues supporting DECREASEKEY($\cdot$) are inherently less efficient, is open.

Meyer and Zeh in [48] propose a cache-aware priority queue supporting the UPDATE($\cdot$) and the BATCHEDDELETEMIN() operation. The latter retrieves all the priorities which differ less than 1 from the minimum

priority. The amortized cost per element involved in an operation is $O\Big((1/B)(\log_2 C + \log_{M/B}(N/B))\Big)$, where C is a constant such that, at any time, the maximum and the minimum priority stored in the priority queue differ by at most C. Notice that for small enough values of C this priority queue is more efficient than that of [32] and [27]. The priority queue, developed in a cache-aware context, is actually cache-oblivious because its building blocks (essentially sorting, scanning and stacks) can be implemented cache-obliviously.

Cache-oblivious priority queues of [11], [32], [27] and [48] are based on similar ideas: essentially, the data structure is composed of several levels, of increasing size. Every level contains a *data buffer*, stored in an array. Data buffers of lower levels contain elements with smaller priorities, that are closer to being retrieved by DeleteMin() operations. Insert($\cdot$), Delete($\cdot$) and Update($\cdot$) operations are performed lazily: they insert a signal in a *signal buffer*, associated with the downmost level. Every level has an associated signal buffer. A DeleteMin() operation works on the smallest possible set of levels from the bottom: it applies updates contained in signal buffers to data buffers; if necessary, signals are propagated upward; while data elements propagate downward, until they are retrieved from the downmost data buffer.

A priority queue that slightly deviates from this model is given in [23] by Brodal and Fagerberg; its "levels" are constituted by Funnel Mergers of increasing size, linked together by binary mergers. It supports the same operations, at essentially the same costs, of the priority queue of [11].

3.4.1.1 Applications

Priority queues provide a convenient way of structuring computations, because they allow to "suspend" a part of the current computation, and to restore it (with all the data it needs) at the right time. To exemplify this concept, we describe the use of priority queues in Dijkstra-like shortest paths algorithms and in the so-called time-forward processing.

Dijkstra's single-source shortest paths algorithm [35] (see Fig. 3.1) maintains a priority queue which stores, for any vertex of a graph $G = (V, E)$, an upper bound on its distance from a special vertex s, the *source*. The algorithm visits vertices in increasing order of their distance from s. When a vertex v is extracted from the priority queue, it is *visited* (lines 5-8): the algorithm learns its distance d from s. Hence the algorithm learns a new upper bound on the distance from s of every neighbor w of v: namely, the sum $d + \ell(vw)$, where $\ell(vw)$ is the length of edge vw. If this upper bound improves the previous best upper bound on w's distance from the source, stored in the priority queue, the priority associated to w is updated (line 8). Thus the priority queue structures the computation, in the sense that it allows to visit vertices in the desired order

```
DIJKSTRA(G, s)
    ▷ G = (V, E) is a graph, s ∈ V
    ▷ Let Q be a priority queue
1   INSERT(Q, s, 0)
2   for each v ∈ V − {s}
3         do INSERT(Q, v, +∞)
4   while ¬ ISEMPTY(Q)
5         do (v, d) ← DELETEMIN(Q)
               ▷ We are visiting vertex v, whose distance from s is d
               ▷ Let A(v) be the adjacency list of v
6              for each vw ∈ A(v)
7                    do if w has not been visited yet
8                          do DECREASEKEY(Q, w, d + ℓ(vw))
```

Figure 3.1: Dijkstra's algorithm

of their distance from the source. Notice, however, that an efficient implementation of the priority queue is not sufficient to render Dijkstra's algorithm cache-efficient, because the algorithm often needs to read information stored with vertices, such as the current best bound on their distance from s and, when they are visited, their adjacency lists. Every such access may cause a memory transfer, because this information could be located anywhere on disk; it is easy to see that in the worst case Dijkstra's algorithm incurs $\Omega(n+m)$ such accesses, where n and m denote the number of vertices and edges of the graph, respectively. In the next chapter we deal with techniques to avoid these random accesses as much as possible.

The time-forward processing [10, 31] is a technique for evaluating functions on a DAG, provided that a topological sort of the DAG is given. The problem is the following one: every vertex v of the graph has an *input* label $\ell(v)$. For every vertex v, compute an *output* label $\ell'(v)$ expressed as a function of $\ell(v)$ and of the output labels of v's direct predecessors $\ell'(v_1), \ldots, \ell'(v_k)$. An application is the evaluation of boolean circuits, where source vertices represent inputs, and other vertices represent logic ports.

To evaluate a DAG $G = (V, E)$, we assume that every vertex $v \in V$ has an identifier $i(v) \in D$, where D is a totally-ordered domain (typically, $D = \mathbb{Z}$); we also assume that the order of identifiers is consistent with the graph topology, that is, it is a topological order. We visit vertices in the order of their identifiers. We make the inductive assumption that, while visiting a vertex $v \in V$, both $\ell(v)$ and all the output labels of v's direct predecessors

$\ell'(v_1), \ldots, \ell'(v_k)$ are known. With this information we compute the output label $\ell'(v)$, and "send" it to every direct successor of v in G. This is simple to achieve with a priority queue Q: for every direct successor w of v, we insert the item $(i(w), \ell'(v))$ with priority $i(w)$ in Q. In this way every time we visit a new vertex w, the items with smallest priorities in Q are those items sent by w's predecessors, which carry their output labels. The inductive assumption is thus satisfied. Notice that, in order to retrieve efficiently the direct successors of a vertex v when v is visited, it is sufficient to store graph edges in an array; for every edge uw we store the pair $(i(u), i(w))$. Pairs are sorted by source vertex identifier. It is easy to verify that the complexity of the time-forward processing is $\mathrm{O}(\mathrm{sort}(|V| + |E|))$ memory transfers.

3.4.2 B-trees

The van Emde Boas layout allows to store a *static* tree of N elements so that searches have an optimal cost of $\mathrm{O}(\log_B N)$ memory transfers. The external memory data structure *B-tree* [14] and its variants allow to maintain a *dynamic* dictionary with a worst case cost of $\mathrm{O}(\log_B N)$ I/O's for any search and update operation (such as insertions and deletions); moreover, *range query* operations (which, given an interval $\mathcal{I}$, ask the set of the dictionary keys belonging to $\mathcal{I}$) are performed at an optimal cost of $\mathrm{O}(\log_B N + k/B)$ I/O's, where k denotes the output size.

The so-called *cache-oblivious B-trees* are dynamic dictionaries with costs similar to those of B-trees. The first cache-oblivious B-tree, proposed by Bender *et al.* [15], was a combination of a packed memory structure and a static binary tree, laid out with the van Emde Boas layout. A *packed memory structure* allows to dynamically maintain N ordered elements in an array of size $\mathrm{O}(N)$, with density $\Theta(1)$, thus allowing to scan any subarray of k elements in $\mathrm{O}(k/B)$ memory transfers. An insertion or deletion of an element at a specified position of the array costs $\mathrm{O}\big((1/B)\log^2 N\big)$ memory transfers, amortized. The static binary tree is used to index elements of the packed memory structure. Every dictionary operation (search, range query or update) first uses the binary tree to find the interested element on the packed memory structure; then it works on the packed memory structure; and finally, if it is an update operation, it either modifies the binary tree locally or rebuilds it from scratch.

Searches and range queries have a worst-case optimal cost of $\mathrm{O}(\log_B N + k/B)$ memory transfers, if the output size is k; an update costs $\mathrm{O}\big(\log_B N + (1/B)\log^2 N\big)$ memory transfers amortized, where the first term is due to the tree search, the last to the maintenance cost of the packed memory structure.

In order to get rid of the $\mathrm{O}\big((1/B)\log^2 N\big)$ term an indirection artifice can be used: we store elements in a number of buckets; in the packed memory

structure we only store pointers to buckets. In this way the number of elements in the packed memory structure and the number of updates to it are reduced, and an amortized cost of $O(\log_B N)$ memory transfers per operation, including updates, is achieved. However, we lose the possibility of performing efficient range queries. The problem of performing updates in $O(\log_B N)$ and range searches in $O(\log_B N + k/B)$ memory transfers amortized is still open.

In [16] Bender *et al.* present a simplified cache-oblivious dictionary based on the same ideas and with the same asymptotic costs of the aforementioned structure; the authors show its practical viability by simulating a hierarchical memory system running their data structure and a classic B-tree.

Brodal *et al.* in [26] propose another simplified dictionary with the same costs, that differs from the one of [15] mainly in that in [26] the packed memory structure, instead of being explicitly stored in an array, is embedded in the static binary search tree itself.

3.5 Untangling pointers

Pointer-based structures are difficult to manage cache-efficiently. Following a pointer could mean performing a disk access. For a particular pointer-based structure, the linked list, a technique allows to avoid this issue. The *list ranking* algorithm by Chiang *et al.* [31] transforms a linked list with N elements into an ordered array, with a cost of $O(\text{sort}(N))$ memory transfers. List ranking is handful in those cases when a list is constructed step-by-step by an algorithm which associates each list element with its successor: in order to organize the list so that it can be accessed simply by scanning, we can rank it. In this sense, ranking plays for lists the same role that sorting plays for arrays. In this section we first describe the list ranking algorithm, and then present, as an application, the Euler tour technique, which allows to handle efficiently another pointer-based structure, the tree. For a deeper discussion of the list ranking technique you can refer to [45].

3.5.1 List ranking

Given a linked list with N elements, a link ranking algorithm associates with every element its *rank*, a natural number denoting its position in the list. List head has rank 1 and, if $b = \text{succ}(a)$, then $\text{rank}(b) = \text{rank}(a) + 1$.

For the sake of exposition we consider the *prefix sum problem*, which is more general than list ranking but can be easily reduced to it. We are given a list L, whose elements are labeled with natural numbers. Let $\ell(a) \in \mathbb{N}$ be the label of element a. We want to compute, for every vertex a, the new label $\ell'(a) = \sum \{\ell(x) \mid \text{rank}(x) \leq \text{rank}(a)\}$. In other words, we want to compute for every vertex the sum of the labels of all its predecessors in L. It is easy to see

that, for every a such that $\mathrm{succ}(a)$ exists, $\ell'(\mathrm{succ}(a)) = \ell'(a) + \ell(\mathrm{succ}(a))$. To rank a list it suffices to solve the prefix sum problem with input $\ell(a) = 1$ for every list element a.

Our generalized list ranking algorithm is recursive. If the input list L has at least 2 elements, we first find a maximal independent set I of L, seen as an undirected graph. We will discuss how to efficiently find a maximal independent set later. We remove I from L, charging the labels of the removed elements to their successors:[3] for all $a \in I$, we let $\ell(\mathrm{succ}(a)) \leftarrow \ell(a) + \ell(\mathrm{succ}(a))$, and create the shortcut $\mathrm{succ}(\mathrm{pred}(a)) \leftarrow \mathrm{succ}(a)$. We recursively run the algorithm on the remaining list $L - I$, and then we re-incorporate elements of I, updating their labels: for all $a \in I$, $\ell(a) \leftarrow \ell(\mathrm{pred}(a)) + \ell(a)$.

Assuming for now that a maximal independent set of a list with k elements can be found in $\mathrm{O}(\mathrm{sort}(k))$ memory transfers, we compute the cost of the algorithm. Since a maximal independent set of a list of size k contains at least $k/3$ elements, input size of the i-th recursive activation is at most $(2/3)^{i-1}N$. Every activation consists in computing a maximal independent set, and in exchanging data (labels, pointers) between list elements and their successors or predecessors. This operation can be done sorting and scanning copies of the list. We exemplify this very common technique. Assume that a list item has the form $(a, \mathrm{succ}(a), \ell(a))$,[4] and that we want to charge labels of I to L. We sort L by first term (by element), I by second term (by successor), and perform a parallel scan of the lists, maintaining that the successor of the current list item $(c, a, \ell(c))$ of I corresponds to the element of the current item $(a, b, \ell(a))$ of L; then we let $\ell(a) \leftarrow \ell(c) + \ell(a)$, as desired.

The total complexity of the h recursive activations of the algorithm is thus $\mathrm{O}\Big(\sum_{i=0}^{h} \mathrm{sort}\big((2/3)^i N\big)\Big)$ memory transfers; we have that $\sum_{i=0}^{h} \mathrm{sort}\big((2/3)^i N\big) < \sum_{i=0}^{\infty} (2/3)^i (N/B) \log_{M/B}(N/B) = \mathrm{O}(\mathrm{sort}(N))$, which hence is the I/O complexity of our algorithm.

In order to compute a maximal independent set of an undirected graph G, Zeh [58] uses the time-forward processing. We first transform G into a DAG, directing edges in some way. Since we want vertex identifiers to be topologically ordered, we direct every edge from its lowest to its highest-id endpoint. Then we apply the time-forward processing, with the following rule: source vertices are always selected for the independent set; any other vertex is selected if and only if none of its direct predecessor has been selected. The correctness of the algorithm is easy to prove.

[3] Instead of computing a new labeling ℓ', we modify the labeling ℓ.

[4] To represent successors of list elements we use the so-called *soft-pointers*, i.e. identifiers of the successor elements themselves, rather than *hard-pointers* to the physicals locations where they reside. Soft pointers are more friendly with respect to cache-efficiency, because they do not need to be altered when data is relocated. It is possible to replace hard with soft pointers simply via sorting and scanning.

3.5.2 Euler tours of trees

A tree is another example of pointer-based structure for which there exists a useful and efficient-to-compute linearization: the Euler tour. An *Euler tour* of a tree T is an Euler tour of the digraph G obtained by T replacing every edge of T with a pair of opposite-directed edges.

We construct an Euler tour of G as a linked list as follows. For every node v, we consider its neighbors $w_0, \ldots, w_{k-1}$ in an arbitrary order; let $w_k = w_0$ by definition. For every neighbor w_i of v, $i \leq k-1$, we set the successor of edge $w_i v$ to be $v w_{i+1}$. In other words the Euler tour, after entering node v by edge $w_i v$, leaves v by $v w_{i+1}$. It is easy to construct this linked list by sorting and scanning copies of the edge list of G.

In this way we obtain a circular linked list; by removing an arbitrary element from the list we transform it in an ordinary list, corresponding to a cycle in G, the tour. Then we can apply the list ranking algorithm to linearize the tour. The cost of the whole outlined algorithm is $\mathrm{O}(\mathrm{sort}(N))$ memory transfers if the input tree has N nodes.

The Euler tour technique finds numerous applications in cache-efficient tree and graph related algorithms (see [45]. Examples are the various flavors of depth-first search (DFS) of trees: by traversing the Euler tour of a tree and determining when we are visiting a node for the first or for the last time it is easy to obtain, respectively, a pre-order and a post-order visit of the tree.

3.6 Graphs have a skeleton: the minimum spanning tree

Now that we know how to efficiently handle trees, it is time to take advantage of this ability to work with graphs. These objects are much more difficult to treat. A minimum spanning tree catches some structural properties of a graph, and can "guide" a systematic exploration of it. For this reason minimum spanning trees are helpful also in other fields of algorithmics: for instance, many approximation algorithms on graphs are based on properties of the minimum spanning tree.

Borůvka's algorithm for the minimum spanning tree problems grows a minimum spanning forest; at every step components grow in "parallel". For this reason it is suitable to be transformed into a cache-efficient algorithm.

The cache-efficient version of Borůvka's algorithm is recursive. Let $G = (V, E)$ be the input graph. Every vertex selects its lightest incident edge. We add selected edges to the minimum spanning tree. Let $F \subseteq E$ be the set of selected edges. We compute the connected components of (G, F); and contract every component into a single vertex, throwing away, for each pair

of components, all the edges that connect them but the lightest one. We thus get a contracted graph $G_c = (V_c, E_c)$; we remove isolated vertices[5] from G_c and we recurse on it.

Selecting the lightest edges can be done via sorting and scanning. To find connected components, we observe that (V, F) is a forest with some edges appearing twice; more precisely, exactly one edge per component of (V, F) appears twice, i.e., the lightest edge in the component. (In fact, if vw is the lightest edge of a component, both its endpoints v and w select it; no other edge of the component can appear twice, because a connected component with k vertices has $k-1$ distinct edges.) We compute the Euler tour of this forest; for every component we select a special vertex, the *leader*, and associate a unique component identifier to it (to select leaders we can select, for every duplicated edge in (V, F), one of its endpoints); and we "push" component identifiers to every other vertex of components using list ranking. Component contraction can be done via sorting and scanning. Thus every recursive activation costs $\mathrm{O}(\mathrm{sort}(|E|))$. Since every contraction step halves (at least) the number of nodes, the total cost of the algorithm is $\mathrm{O}(\mathrm{sort}(|E|) \log |V|)$.

The efficiency of the algorithm can be improved by dividing the execution in phases: each phase is composed of several contraction steps, and works on a reduced version of the graph. In this way we obtain the best known cache-oblivious algorithm for the minimum spanning tree, which has a complexity of $\mathrm{O}(\mathrm{sort}(|E|) \log \log |V|)$ memory transfers, by Arge *et al.* [11]. The algorithm is based on the best known cache-aware algorithm for the minimum spanning tree, by Arge *et al.* [13], which manages to reduce the complexity to $\mathrm{O}\left(\mathrm{sort}(|E|) \log \log \frac{|V|B}{|E|}\right)$ I/O's by halting recursion and switching to another minimum spanning tree algorithm (essentially, a cache-efficient version of Prim's algorithm) when the input graph has less than $|E|/B$ vertices. Notice that the algorithm of [13] is inspired, in turn, by a similar algorithm by Munagala and Ranade for computing connected components [50]. Finally, we mention a faster randomized algorithm for the minimum spanning tree, by Abello *et al.* [1], which solves the problem with an expected cost of $\mathrm{O}(\mathrm{sort}(|E|))$ I/O's. Originally conceived for the external memory model, the algorithm, based on scanning and sorting, is in fact cache-oblivious.

[5]There may be isolated vertices only if G is not connected; in this case the algorithm actually compute a minimum spanning forest of G.

Chapter 4

Cache-oblivious graph traversal algorithms

In this chapter we present cache-efficient algorithms and techniques for two fundamental graph traversal problems, namely breadth-first search (BFS) and single-source shortest paths (SSSP) on undirected graphs. As we mentioned in Section 3.4.1, it is not easy to explore a graph cache-efficiently, because vertices and edges are visited in an order that is hard to predict [42]. A formal definition of the problems we are interested in follows.

Problem 4.1 (Single-source shortest paths) *Given an undirected graph $G = (V, E)$, a special vertex $s \in V$ (the* source*), and a positive labeling of its edges $\ell : E \to \mathbb{R}^+$ (the* edge lengths*), find, for every vertex $v \in V$, its* distance *$D(v) \in \mathbb{R}^+$ from the source; that is, the length of the minimum-length path connecting s to v.*

Problem 4.2 (Breadth-first search) *Given an undirected graph $G = (V, E)$ and a special vertex $s \in V$ (the* source*), find, for every vertex $v \in V$, its distance from the source $D(v) \in \mathbb{R}^+$, assuming that every edge $e \in E$ has length* 1*.*

The classic internal algorithm for the BFS (see, for instance, [33]) uses a queue Q to visit vertices in increasing order of distance (or *level*) from the source. In this context, a vertex v is *visited* when the algorithm determines its distance $D(v)$ from s. At the beginning, s is inserted in Q with its distance $D(s) = 0$. Then the algorithm repeatedly extracts the current visited vertex v from Q and, for every adjacent unvisited vertex w of v, it inserts w in Q with distance $D(w) = D(v) + 1$.[1]

[1] Actually, distances are not explicitly stored in Q; they are implicitly maintained by the algorithm.

In the CO model the algorithm incurs $\Theta(n+m)$ memory transfers, where $n = |V|$ and $m = |E|$. $\Theta(n)$ memory transfers are needed to load, for every visited vertex v, its adjacency list, in order to enqueue v's unvisited neighbors in Q. A total of $\Theta(m)$ memory transfers are performed because, when a vertex v is visited, the algorithm checks whether every neighbor w of v has been already visited, performing an access to w.

The classic internal SSSP algorithm is Dijkstra's algorithm [35]. It is essentially a variant of the BFS algorithm where the queue Q is replaced by a priority queue. For every vertex v, the algorithm maintains an upper bound $d(v)$ of v's distance from s, the *tentative distance* of v; initially the tentative distance is $+\infty$ for all vertices except s, for which $d(s) = 0$. The priority of a vertex v in Q is its tentative distance $d(v)$. When a vertex v is extracted from Q, its tentative distance $d(v)$ equals its actual distance $D(v)$ from the source: then v is visited, and the tentative distance of every unvisited neighbor w of v is updated to $\min(d(w), d(v) + \ell(vw))$. We call this operation the *relaxation* of edge vw.

Again, Dijkstra's algorithm incurs $\Omega(n+m)$ memory transfers in the CO model: $\Omega(n)$ to retrieve adjacency lists of visited vertices, $\Omega(m)$ to test the status of the neighbors of visited vertices and to update their tentative distances. Another $\Omega((m/B)\log_2(n/B))$ memory transfers are due to priority queue operations, if the cache-efficient priority queue of [27] or [32] is used. The same bounds hold for an external memory straightforward implementation of the BFS and of Dijkstra's algorithms.

Problem	**Adaptation of IM algorithms**	**Avoiding to remember visited vertices...**	**...and clustering**
EM-BFS	$\Omega(n+m)$	Munagala and Ranade [50] $O(n + \mathrm{sort}(m))$ (Section 4.1.1)	Mehlhorn and Meyer [46] $O\left(\sqrt{\frac{nm}{B}} + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$ (Section 4.2.1)
CO-BFS	$\Omega(n+m)$	Munagala and Ranade [50] $O(n + \mathrm{sort}(m))$ (Section 4.1.1)	Brodal *et al.* [27] $O\left(\sqrt{\frac{nm}{B}} + \frac{m \log n}{B} + \mathrm{MST}_{\mathrm{CO}}(n,m)\right)$ (Section 4.2.3)
EM-SSSP	$\Omega(n+m)$	Kumar and Schwabe [44] $O(n + (m/B)\log_2(n/B))$ (Section 4.1.2)	Meyer and Zeh [48] $O\left(\sqrt{\frac{nm}{B}\log W} + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$ (Section 4.2.2) --- Meyer and Zeh [49] $O\left(\sqrt{\frac{nm}{B}\log n} + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$
CO-SSSP	$\Omega(n+m)$	Kumar and Schwabe [44] $O(n + (m/B)\log_2(n/B))$ (Section 4.1.2)	Allulli *et al.* [9] $O\left(\sqrt{\frac{nm}{B}\log W} + \frac{m \log n}{B} + \mathrm{MST}_{\mathrm{CO}}(n,m)\right)$ (Chapter 5)

Table 4.1: I/O cost of algorithms for visiting undirected graphs

Throughout this and the next chapter we study techniques to reduce the number of memory transfers due to unstructured accesses to edges and vertices. Table 4.1 summarizes the situation; details will be provided in next sections. In Section 4.1 we see how to reduce the I/O cost of dealing with neighbors of visited vertices from $\Theta(m)$ to $\Theta(\mathrm{sort}(m))$. In Section 4.2 we show that a reorganization of the adjacency lists in clusters allows to incur, on sufficiently sparse undirected graphs, less than one memory transfer per vertex. The algorithms presented in this section are constructed incrementally. The first algorithm based on the clustering paradigm was conceived for the external memory BFS problem; later, it was adapted for the cache-oblivious BFS and for the external memory SSSP problems. In Chapter 5 we show how, combining these adaptations and introducing novel ideas, an improved cache-oblivious algorithm for the shortest paths problem on undirected graphs is obtained.

We conclude this overview underlining that on general (directed) graphs the problem of graph traversal seems more difficult, in the sense that known methods for discriminating the already visited vertices are not as efficient as the ones presented in Section 4.1 for the undirected BFS and SSSP. Using a data structure called the *buffered repository tree*, Buchsbaum *et al.* [29] present external memory algorithms for the general BFS and DFS incurring $O((n + m/B)\log n)$ I/Os; the DFS algorithm by Buchsbaum is also the best known algorithm for undirected graphs. Arge *et al.* [11] present a cache-oblivious version of the buffered repository tree, which allows to implement the general BFS and DFS algorithms cache-obliviously. Also the SSSP problem can be solved on general graphs in $O((n + m/B)\log n)$ I/O's (see Vitter [57], Chowdhury and Ramachandran [32] for, respectively, an external memory and a cache-oblivious algorithm with this cost). Understanding the limits of external memory graph traversal is still a major open question, considering that a huge gap divides current best upper bounds from current best lower bounds of $\Omega(\min(n, \mathrm{sort}(n)) + m/B)$ (see, for example, [42]); another interesting problem is the partial dynamization of graph traversal algorithms.

4.1 Avoiding one memory transfer per edge

The first cache-efficient algorithms for the BFS and the SSSP problems on undirected graphs, respectively by Munagala and Ranade [50] and Kumar and Schwabe [44], deviate from classic algorithms in that they adopt the following idea: Every time a vertex $v \in V$ is visited, deal with *every* neighbor w of v, disregarding its status. In other words, do not read information associated with w, such as if w has been already visited or what is its current tentative distance. Instead, insert newly discovered information about w into some

structures, postponing the moment when this information will be processed. In this way the access to information related to neighbors of visited vertices can be structured, improving the total cost of following edges while visiting vertices from $\Theta(m)$ to $\Theta(\text{sort}(m))$. We now describe how this idea is implemented in the BFS and in the shortest paths algorithms.

4.1.1 Munagala and Ranade's BFS algorithm

The BFS algorithm by Munagala and Ranade [50] visits vertices in batches, level by level. Let L_i be the set of the vertices of V which are part of level i of the BFS tree, $i \in \mathbb{N}$, i.e. whose distance from the source is i. Observe that vertices of level L_i are adjacent only to vertices of levels L_{i-1}, L_i and L_{i+1}.[2] This observation permits, while visiting vertices of level L_i, to find efficiently their unvisited neighbors: let X be the set of *all* the neighbors of vertices of L_i. We know that $X = L_{i-1} \cup L_i \cup L_{i+1}$. To obtain L_{i+1} it suffices to remove L_{i-1} and L_i, which are already known at this stage, from X:

CreateNextBfsLevel(L_{i-1}, L_i)
1 $X \leftarrow$ GetNeighbors(L_i)
2 Sort(X)
3 RemoveDuplicates(X)
4 $L_{i+1} \leftarrow X - (L_{i-1} \cup L_i)$
5 **return** L_{i+1}

The total cost of executing line 1 is $O(n + m/B)$ memory transfers, since every adjacency list is randomly accessed and scanned once. Line 2 costs $O(\text{sort}(m))$ memory transfers in total, because the overall size of sets X returned by GetNeighbors() is $2m$, and the $\text{sort}(\cdot)$ function is super-linear: denoting by X_i the value of set X at iteration i, we have that $\sum_i \text{sort}(|X_i|) \leq \text{sort}(\sum_i |X_i|) = \text{sort}(2m)$. The other operations can be performed by scanning X, L_{i-1} and L_i a constant number of times, and thus produce $O(m/B)$ memory transfers in total. Therefore, the total cost of the algorithm is $O(n + \text{sort}(m))$.

For the sake of completeness we underline that the algorithm of Munagala and Ranade was originally conceived for the external memory model; relaying on sorting and scanning only it is in fact cache oblivious.

4.1.2 Kumar and Schwabe's SSSP algorithm

Kumar and Schwabe's algorithm [44] is a variant of Dijkstra's algorithm; it solves the SSSP problem in $O(n + (m/B)\log_2(n/B))$ memory transfers. It

[2] This property only holds for undirected graphs: thus Munagala and Ranade's algorithm will not work on directed graphs.

uses a priority queue supporting the UPDATE($\cdot$) operation: in its original version the priority queue is a tournament tree [44], which can be replaced by the cache-oblivious priority queue of [27] or of [32] in order to render the whole algorithm cache-oblivious.

Differently from Dijkstra's algorithm, when a vertex $w \in V$ is visited, for every neighbor v of w the algorithm performs an UPDATE$(v, d(w)+\ell(vw))$ operation on the priority queue Q. An UPDATE(v, p) operation inserts element v in Q with priority p if v is not already present in Q; otherwise, if v is already present in Q with priority p', the priority of v is changed to $\min(p, p')$. Thus, in Kumar and Schwabe's algorithm an UPDATE(v, p) operation on an unvisited vertex v is equivalent to an edge relaxation in Dijkstra's algorithm. Instead, if v has already been visited the operation is called a *spurious update* because it reinserts in Q a vertex which should not be there. The consequence is that some invocations of DELETEMIN() may return vertices which have already been visited. It is necessary to avoid visiting these vertices again, either discriminating them as they are retrieved from Q, or deleting them from Q before they are retrieved. Kumar and Schwabe adopt the latter approach.

To this aim, a second cache efficient priority queue Q' is used. Some elements of the form $(v, \cdot)$ are inserted in Q' when v is visited, to signal the risk that v could be reinserted in Q by a spurious update, in the future, when v's neighbors are visited. In the case this actually happens, v will pop out from Q' between the time when the spurious update takes place and the time when v would have been retrieved again from Q. Every time a vertex v is extracted from Q', a DELETE(v) operation is issued on Q: this operation prevents the (future) spurious DELETEMIN() of v. Let us go into some details. When v is visited, for every neighbor w of v a pair $e_{vw} = (v, d(v) + \ell(vw))$ is inserted in Q'. Since w will be visited at a time not later than $d(v) + \ell(vw)$, e_{vw} will not pop-out from Q' before w is visited; hence, when v is spuriously updated by w, e_{vw} is still in Q'. On the other hand, the DELETEMIN() that would retrieve the copy of v spuriously inserted by w takes place at time $d(w) + \ell(vw) > d(v) + \ell(vw)$: thus e_{vw} pops out from Q' before this time, and deletes the spuriously updated element from Q. For more details, see [44]. Finally, we observe that Kumar and Schwabe's technique for dealing with spurious updates only works for undirected graphs: the reason is that on directed graphs, when we visit a vertex v, it is hard to get good estimations of the time when v will be spuriously updated, or spuriously retrieved. In fact a spurious update could result from a directed cycle of any length from v to itself.

The I/O complexity of Kumar and Schwabe's algorithm is $\mathrm{O}(n + (m/B)\log_2(n/B))$.

4.2 The clustering technique

4.2.1 Cache-aware BFS

In the worst case the algorithm of Munagala and Ranade performs a memory transfer every time a vertex v is visited, due to the random disk access that loads v's adjacency list $A(v)$. In order to reduce this cost, Mehlhorn and Meyer [46] present an algorithm for the BFS on undirected graphs which divides the vertex set in *clusters*: when a cluster is accessed, the adjacency lists of its vertices are loaded from disk all-at-once. When the adjacency lists of a cluster are loaded from disk, they are temporarily inserted in an array $\mathcal{H}$ called the *hot pool*. When the algorithm visits a vertex v, it first checks whether $A(v)$ is in $\mathcal{H}$: if so, $A(v)$ is read from $\mathcal{H}$ without the need of a random disk access, and then discarded from $\mathcal{H}$. Otherwise, the adjacency lists of all the vertices of the cluster containing v are loaded from disk and inserted in $\mathcal{H}$, at the cost of one random disk access, plus the cost of scanning all the edges in them.

The array $\mathcal{H}$ is stored on disk, because in general it does not fit cache. To look for adjacency lists in $\mathcal{H}$, then, $\mathcal{H}$ must be scanned entirely. It is crucial to keep the cost of scanning $\mathcal{H}$ limited; in particular, a trade-off between the cost of scanning $\mathcal{H}$ and the cost of random accesses to clusters must be found. To keep the scanning cost limited, $\mathcal{H}$ should not be inspected too often; in particular, the number of times and edge e is scanned as part of $\mathcal{H}$ before being discarded should be limited.

The algorithm visits vertices level by level, like the algorithm of Munagala and Ranade. This allows to construct the next level L_{i+1} from L_{i-1}, L_i, and the set X of the vertices adjacent to the current level L_i; and to scan $\mathcal{H}$ only $\Theta(1)$ times per BFS level.

The crucial property that allows to find a trade-off between the scanning and the random access costs is the following: For every $\mu \in [1, n]$, it is possible to partition the input graph G into $O(n/\mu)$ clusters such that every cluster has diameter in G at most μ. If the diameter of a cluster K is k, vertices of K are part of at most k consecutive BFS levels: then, every vertex $v \in K$ will be visited (and $A(v)$ will disappear from $\mathcal{H}$) within k BFS steps from the moment when $A(v)$ is loaded in $\mathcal{H}$. As a consequence, since every cluster has diameter less than μ, every edge is scanned at most $O(\mu)$ times as part of $\mathcal{H}$.

It is now easy to compute the cost of the algorithm, except the clustering procedure. $O(n/\mu + m/B)$ I/O's are performed to load adjacency lists of clusters: the first term accounts for random accesses to clusters, the second for the cost of moving edges. Another $O(\mu m/B)$ I/O's is the cost of scanning the m edges of E $O(\mu)$ times as part of $\mathcal{H}$. $O(\text{sort}(m))$ I/O's are paid to sort the set X once per BFS step, as in Munagala and Ranade's algorithm.

It remains to be described how to partition V in clusters. In [46] two meth-

ods are proposed, a randomized and a deterministic one. A short description of the deterministic method follows. First a spanning tree T of G is computed, and an Euler tour $\mathcal{E}$ of T is constructed. $\mathcal{E}$ is then traversed (starting from an arbitrary vertex) and chopped into pieces of length μ. More precisely, clusters are grown one at a time while traversing $\mathcal{E}$. Every vertex met for the first time during the traversal is added to the current cluster. When μ vertices have been traversed (whether or not they have been added to the current cluster), the current cluster is closed and a new cluster is generated. The number of generated clusters is $\lceil (2n-2)/\mu \rceil$, because $|\mathcal{E}| = 2n-2$; the constraint on cluster diameter is satisfied by construction. The I/O cost of the procedure is the I/O cost of computing a spanning tree of G; currently the most efficient algorithm to compute a spanning tree is that of Section 3.6, which in fact computes a *minimum* spanning tree in $\mathrm{O}(\mathrm{MST}_{\mathrm{EM}}(n,m)) = \mathrm{O}\big(\mathrm{sort}(m)\log\log\frac{nB}{m}\big)$.

Altogether, the I/O complexity of Mehlhorn and Meyer's BFS algorithm is $\mathrm{O}(n/\mu + \mu m/B + \mathrm{MST}_{\mathrm{EM}}(n,m))$ I/O's. In [46] the value $\mu = \sqrt{nB/m}$ is chosen, such that the first two terms of the previous cost are balanced, yielding a cost of $\mathrm{O}\Big(\sqrt{nm/B} + \mathrm{MST}_{\mathrm{EM}}(n,m)\Big)$ I/O's.

Observe that, if $nB/m \in [1,n]$, the choice $\mu = \sqrt{nB/m}$ is asymptotically optimal, because n/μ is a decreasing function of μ, and $\mu m/B$ is increasing. In particular, if the graph is sparse ($m = \mathrm{O}(n)$), the first term of the cost becomes $\mathrm{O}\Big(n/\sqrt{B}\Big)$, with a $\sqrt{B}$ factor improvement over Munagala and Ranade's algorithm. On the other hand, if $nB/m < 1$, the optimal choice for μ is $\mu = 1$, which yields an I/O complexity of $\mathrm{O}(n + \mathrm{sort}(m) + \mathrm{MST}_{\mathrm{EM}}(n,m))$, similar to that of Munagala and Ranade's algorithm (the additional term $\mathrm{MST}_{\mathrm{EM}}(n,m)$ being due to the clustering procedure). This means that, if the input graph is sufficiently dense ($m = \Omega(nB)$) clustering does not help, because optimal clusters contain only one vertex, and essentially we get back to the algorithm of Munagala and Ranade. Notice, finally, that $nB/m > n$ means $m < B$, that is, the graph fits entirely one cache block: an internal algorithm should be used in this case.

The computational study of Mehlhorn and Meyer's algorithm by Ajwani *et al.* [7] fully validates the theoretical analysis.

4.2.2 Cache-aware SSSP

The clustering approach cannot be applied directly to the shortest paths problem on undirected graphs because of edge lengths: simply forming clusters of diameter less than μ for some μ would create more than $\Omega(n/\mu)$ clusters, in general, because long edges would cause clusters to contain a small number of vertices.

Meyer and Zeh [48] solve this issue by observing that clusters of larger

diameter can be tolerated if one manages to scan their adjacency lists less frequently, as they lay in the hot pool. Before digging into details, some terminology is required. Like any algorithm derived from Dijkstra's algorithm, Meyer and Zeh's algorithm visits vertices in order of their distance from the source s (this constraint will be slightly relaxed later). At any time, the set of visited edges form a ball, whose boundary we call the *frontier*. We say that the algorithm performs a *distance step* when the frontier moves forward by 1.

Now, suppose that the hot pool $\mathcal{H}$ is a hierarchy of r *levels* $\mathcal{H}_1, \ldots, \mathcal{H}_r$, which are inspected by the algorithm at a different pace: level $\mathcal{H}_i$ is inspected only every $\Omega\left(2^i\right)$ distance steps. If an edge stays in $\mathcal{H}_i$ for at most $\mathrm{O}\big(\mu 2^i\big)$ distance steps, then it is inspected at most $\mathrm{O}(\mu)$ times as part of $\mathcal{H}_i$. Thus *higher* levels (levels with higher indices) can store clusters of larger diameter.

In order to explain why some levels can be scanned less frequently, we need the following definitions (see Figure 4.1):

Definition 4.3 (Category of an edge) *An edge $e \in E$ has* category *i if $2^{i-1} \leq \ell(e) < 2^i$.*

Definition 4.4 (i-component) *An* i-component *is a connected component of the graph $G_i = (V, E_i)$ obtained from G removing all its edges of category more than i.*

Suppose that an i-component C contains no visited vertices. In order to reach any vertex of C, the frontier must "traverse" one of the edges that connect C to other i-components. Every such edge has category at least $i+1$, that is, length at least 2^i. It takes at least 2^i distance steps to traverse it. Thus, if the adjacency list $A(v)$ of a vertex $v \in C$ is in the hot pool, it can be placed in level at least $\mathcal{H}_{i+1}$ and "forgot" there for a while, until the time of v's relaxation approaches.

To be more precise we need to introduce the priority queue of Meyer and Zeh, that drives the moving of adjacency lists in the hot pool.

4.2.2.1 The batched priority queue of Meyer and Zeh

The priority queue of [48], which we call MZ-PQ, supports Update$(\cdot)$, Delete$(\cdot)$, and BatchedDeleteMin() operations (see Section 3.4.1) in an amortized cost of $\mathrm{O}\Big((1/B)(\log_2 C + \log_{M/B}(N/B))\Big)$ memory transfers per involved element, where C is a constant such that, at any time, the maximum and the minimum priority stored in the priority queue differ by at most C, and N is the total number of elements ever stored in the structure. Assuming that edges of G have length between 1 and W, and that $\mathrm{O}(m)$ priority queue operations are performed (as in Dijkstra's algorithm), we have that $C \leq W$, and

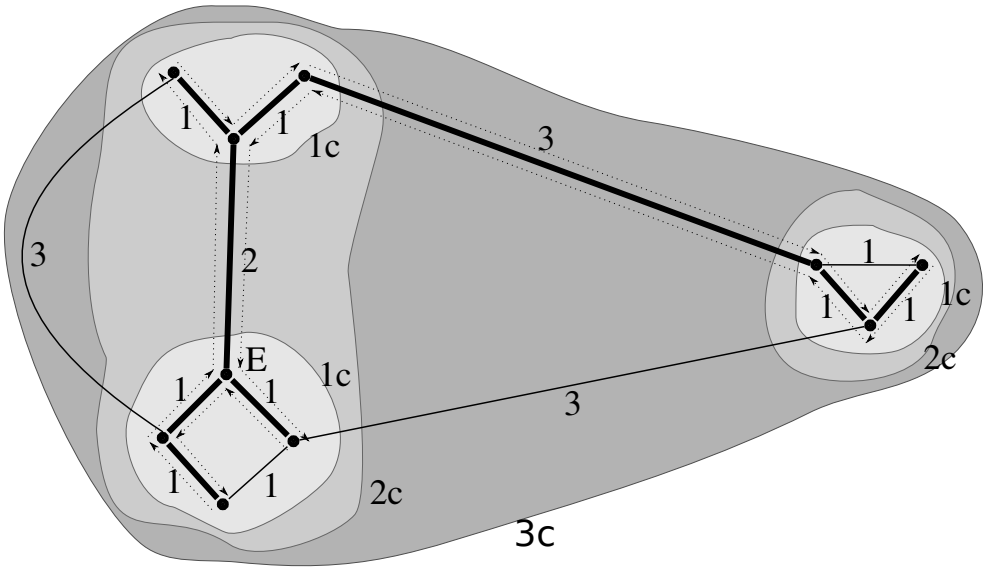

Figure 4.1: Categories and components in a graph. Categories are illustrated by labels located near the appropriate edges. Thick edges define a minimum spanning tree of the graph; notice that thick edges of category not greater than i define a minimum spanning forest of i-components. Dotted tiny lines represent an Euler tour $\mathcal{E}$ of the graph.

then the total cost of priority queue operations in Meyer and Zeh's algorithm is $O((m/B)\log W + \text{sort}(m))$.

In addition to the improved cost of operations when edges are sufficiently short, MZ-PQ is tailored to the shortest paths algorithm for the following reasons:

- Vertices of the vertex set L returned by a BatchedDeleteMin() operation can be visited concurrently, because the tentative distances of any two vertices of L differ by less than 1, i.e. less than the length of any edge. Thus it is not possible, visiting a vertex of L, to improve the tentative distance of another vertex of L.

- The structure of MZ-PQ, coupled with the hot pool structure, allows to move adjacency lists in the hot pool downward as the time of their relaxation approaches. Like the hot pool structure, the priority queue is organized in levels (here called *buckets*): each bucket is associated to a hot pool level. The bucket a vertex v belongs to is chosen according to v's priority, i.e., to the time when its adjacency list is scheduled to be relaxed; as the time of relaxation approaches, v is moved towards *lower* buckets; v's adjacency list is moved accordingly in the hot pool. Details will be provided in next section, but before we need to give an overview on the structure of the priority queue; we underline only those aspects that are directly connected to the shortest paths algorithm.

MZ-PQ consists of $r = \Theta(\log W)$ buckets $\mathcal{B}_1, \ldots, \mathcal{B}_r$, each of which is an array of vertices, sorted by vertex identifier. Buckets $\mathcal{B}_1, \ldots, \mathcal{B}_r$ are stored consecutively in an array $\mathcal{B} = \mathcal{B}_1 \circ \cdots \circ \mathcal{B}_r$. The buckets are defined by priorities $p_0 \le p_1 \le \cdots \le p_r = +\infty$ in the sense that all priority queue entries (x, p) with priority $p_{i-1} \le p < p_i$ are stored in bucket $\mathcal{B}_i$; in particular, no entry has priority less than p_0. Priorities $p_0, \ldots, p_r$ satisfy that, for $1 < i < r$, either $p_i = p_{i-1}$ or $2^{i-2}/3 \le p_i - p_{i-1} \le 2^{i-2}$; for $i = 1$, we have $0 < p_1 - p_0 \le 1$, except at the very beginning of the algorithm. The priorities are initialized to $p_0 = \cdots = p_{r-1} = 0$ and $p_r = +\infty$.

Update(x, p) and Delete(x) operations are implemented by inserting `Update`(x, p) and `Delete`(x) signals into an update buffer $\mathcal{U}_1$. More precisely, there is one such buffer $\mathcal{U}_i$ associated with each bucket $\mathcal{B}_i$; buffers $\mathcal{U}_1, \ldots, \mathcal{U}_r$ are concatenated to form an array $\mathcal{U} = \mathcal{U}_1 \circ \cdots \circ \mathcal{U}_r$. Over time, signals move to higher buffers until they find the bucket which has to be modified. This happens during BatchedDeleteMin() operations.

A BatchedDeleteMin() operation first ensures that $\mathcal{B}_1$ is non-empty. To do so, it scans buckets $\mathcal{B}_1, \ldots, \mathcal{B}_i$, applying for each $1 \le j \le i$ the signals in $\mathcal{U}_j$ to $\mathcal{B}_j$ and then merging the signales into $\mathcal{U}_{j+1}$. It stops when it finds the first bucket $\mathcal{B}_i$ that is non-empty after the updates in $\mathcal{U}_i$ have been applied

to it. It then sets $p_0 = p_{\min}$, where $p_{\min}$ is the minimum priority of any element in $\mathcal{B}_i$ and updates priorities $p_1, \ldots, p_{i-1}$ so that the above constraints on these priorities are satisfied; p_{i-1} equals p_i at the end of this procedure. The elements of $\mathcal{B}_i$ are then distributed over $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$ according to their priorities, which is achieved by moving all elements with priority less than p_{i-1} from $\mathcal{B}_i$ to $\mathcal{B}_{i-1}$, then moving all elements with priority less than p_{i-2} from $\mathcal{B}_{i-1}$ to $\mathcal{B}_{i-2}$, and so on.

Meyer and Zeh prove the following lemma, which basically states that bucket $\mathcal{B}_i$ is scanned every $\Omega\left(2^i\right)$ distance steps of the shortest paths algorithm. Here, we say that a BATCHEDDELETEMIN() operation *empties* bucket $\mathcal{B}_i$ if it distributes the contents of $\mathcal{B}_i$ over buckets $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$.

Lemma 4.5 (Meyer/Zeh [48]) *Let p be the minimum priority of the entries retrieved by a* BATCHEDDELETEMIN() *operation, let $i \geq 2$, and consider the sequence of all subsequent* BATCHEDDELETEMIN() *operations that empty buckets $\mathcal{B}_h$ with $h \geq i$. Let q_k be the minimum priority of the entries retrieved by the k'th operation in this sequence. Then $q_k - p \geq (k-4)2^{i-2}/3$.*

4.2.2.2 Moving components in the hot pool

We are now ready to describe how adjacency lists move in $\mathcal{H}$. The following invariant is maintained:[3]

Invariant 4.6 *If the adjacency list $A(v)$ of a vertex v is in the hot pool, it resides in the lowest level $\mathcal{H}_i$ such that the $(i-1)$-component containing v contains no vertex z that has been visited or is stored at a bucket $\mathcal{B}_{i'}$, $i' < i$, of the priority queue.*

In other words, $A(v)$ is stored in level $\mathcal{H}_i$ if and only if the i-component of v contains a vertex in $\mathcal{B}_i$ or below, but the $(i-1)$-component does not. Intuitively, if a vertex in the $(i-1)$-component of v is going to be visited soon, it is moved downward in the priority queue, and the $(i-1)$-component of v is moved downward in $\mathcal{H}$ in order to maintain the invariant.

The invariant implies that when a vertex v is extracted by a BATCHEDDELETEMIN() operation, if its adjacency list is present in the hot pool, then it is found in the bottom level, $\mathcal{H}_1$. Thus we can give the following high-level description of the shortest paths algorithm:

[3] In order to reduce the *average* I/O complexity of the algorithm, Meyer and Zeh actually use a more complicated rule for moving adjacency lists, with the same worst-case complexity.

```
ShortestPath(G, s, P)
1  Update(Q, s, 0)
2  while ¬ IsEmpty(Q)
3        do L ← BatchedDeleteMin(Q)
4           for each v ∈ L
5                 do if A(v) ∉ H_1
6                       do Load adjacency lists of the cluster of v in H
                           from P
7                    for each vw ∈ A(v)
8                       do Update(Q, w, d(v) + ℓ(vw))
```

$\mathcal{P}$ is a clustering partition of adjacency lists, that we will introduce later. The BatchedDeleteMin() invocation at line 3 causes vertices to be moved in a prefix $\mathcal{B}_1, \ldots, \mathcal{B}_h$ of the priority queue buckets; consequently, in order to maintain the invariant, adjacency lists need to be moved in the corresponding levels $\mathcal{H}_1, \ldots, \mathcal{H}_h$ of the hot pool. In particular, suppose that a BatchedDeleteMin() operation moves a vertex v from some bucket $\mathcal{B}_i$ to bucket $\mathcal{B}_{i-1}$. In order to maintain the invariant, it is sufficient to move the adjacency lists of the vertices of the $(i-1)$-component of v that are located in $\mathcal{H}_i$ to $\mathcal{H}_{i-1}$.

Line 6 loads the adjacency list of a cluster from $\mathcal{P}$ into $\mathcal{H}$, lazily: it copies the whole cluster into an auxiliary bucket $\mathcal{H}'_1$ of an insertion buffer $\mathcal{H}' = \mathcal{H}'_1 \circ \cdots \circ \mathcal{H}'_r$. When a subsequent BatchedDeleteMin() invocation operates on levels $\mathcal{H}_1, \ldots, \mathcal{H}_k$, adjacency lists travel upward in the insertion buffer, until they find the level where they have to be inserted according to the invariant, or they reach bucket $\mathcal{H}'_k$ of the insertion buffer.

A number of auxiliary structures are used for bookkeeping and to perform efficiently the operations of the outlined algorithm; we do not describe them. Instead we sketch a proof of correctness and we analyze the cost of the algorithm.

To prove the correctness of the algorithm, it is sufficient to prove that the invariant is maintained. BatchedDeleteMin() operations move adjacency lists in $\mathcal{H}$ downward correctly. It remains to prove that Update($\cdot$) operations do not mess things up; that is, that BatchedDeleteMin() operations are able to "hook" adjacency lists of vertices inserted by previous Update($\cdot$) operations, and move them downward in the hot pool. It is sufficient to prove the following Lemma.

Lemma 4.7 (Meyer/Zeh [48]) *If an* Update(v, p) *operation inserts vertex v at level $\mathcal{B}_i$ of the priority queue, and if the adjacency list $A(v)$ of v is in $\mathcal{H}$ at that time, then $A(v)$ is in a level $\mathcal{H}_{i'}$ with $i' \leq i$.*

Proof. Consider any UPDATE(v, p_v) operation, performed on v while visiting a vertex u; let h be the level where $A(v)$ is stored, or would have been stored if $A(v)$ were in the hot pool, at that time. We claim that, when the `Update`(v, p) signal reaches its destination level $\mathcal{B}_i$ in the hot pool, if $A(v)$ is in the hierarchy, it is in a level $\mathcal{H}_{i'}$, $i' \leq h \leq i$. It suffices to prove that $h \leq i$, because the level where an adjacency list is stored can only decrease over time, according to Invariant 4.6.

Since $\mathcal{H}_h$ is the proper level of $A(v)$ when u is visited, the $(h-1)$-component C of v contains no visited vertex: then $u \notin C$, which implies that edge uv has category at least h. Thus $p_v = d(u) + \ell(uv) \geq p_{\min} + 2^{h-1}$, where $p_{\min}$ is the minimum priority extracted from Q when u is visited.

At this time, the upper bound on the size of priority queue buckets implies that $p_{h-1} \leq p_{\min}+1+\sum_{x=2}^{h-1}(p_x - p_{x-1}) \leq p_{\min}+1+\sum_{x=0}^{h-3} 2^x = p_{\min}+2^{h-2} < p_v$, that is, the destination bucket of v is at least level h. Then, $i \geq h$. □

Before analyzing the cost of the algorithm, we need to explain how vertices are clustered.

4.2.2.3 The cluster partition

Meyer and Zeh partition the vertex set in a *well-structured clustering*, satisfying the following properties, for a chosen $\mu \in [1, n]$:

(K1) There are $O(n/\mu)$ clusters.

(K2) Every cluster K is a subset of a component. If i is the category of the minimum-category component containing cluster K, we say that K has *category* i, and is an *i-cluster*.

(K3) An i-cluster K does not break $(i-1)$-components; that is, every $(i-1)$-component is either contained in K or disjoint from K.

(K4) The diameter of an i-cluster K is less than $2^i\mu$; the diameter of every i'-component contained in K, $i' < i$, is less than $2^{i'}\mu$.

Property (K1) implies that the cost due to random I/O's to load clusters is $O(n/\mu)$, as in the external memory BFS algorithm.

Properties (K2) and (K3) make it possible to move adjacency lists of components in $\mathcal{H}$ all-at-once, as outlined previously.

Property (K4) allows to bound the time spent by ad edge in the hot pool: more precisely, an edge e stored in bucket $\mathcal{H}_i$ does not linger in $\mathcal{H}$ for more than $O(2^i\mu)$ distance steps. In fact, either e is part of an i-cluster K of diameter $O(2^i\mu)$, which contains a visited vertex (because K has been loaded into the hot pool); or e is part of an i-component C, completely contained in

a loaded cluster, which has diameter $O(2^i\mu)$ and contains a vertex in bucket $\mathcal{B}_i$ or below. In the first case, all the vertices of K will be visited within the next $O(2^i\mu)$ distance steps. In the second case, the vertex in $\mathcal{B}_i$ or below will be visited within $O(2^i)$ distance steps (because of the upper bound on the priorities of priority queue buckets), and thereafter all the other vertices of C will be visited within $O(2^i\mu)$ distance steps.

To compute the cost of inspecting $\mathcal{H}$, we assume that buckets of $\mathcal{H}$ are concatenated in an array $\mathcal{H} = \mathcal{H}_1 \circ \cdots \circ \mathcal{H}_r$, and that upon a BATCHED-DELETEMIN() operation the algorithm *scans* the prefix $\mathcal{H}_1 \circ \cdots \circ \mathcal{H}_i$ of $\mathcal{H}$, where $\mathcal{B}_i$ is the emptied bucket of Q (see the original paper [48] for a complete analysis). By Lemma 4.5, level $\mathcal{H}_i$ is scanned only every $\Omega(2^i)$ distance steps. Thus, an edge e is scanned at most $O((2^i\mu)/2^i) = O(\mu)$ times as part of $\mathcal{H}_i$; and since there are $O(\log W)$ hot pool levels, the total cost of scanning edges in the hot pool is $O((m\mu \log W)/B)$ I/O's.

Choosing the value $\mu = \sqrt{(nB)/(m \log W)}$, which balances the cost of scanning $\mathcal{H}$ with the cost of loading clusters; and adding a cost of $O(\mathrm{MST}_{\mathrm{EM}}(n,m))$ for the clustering algorithm and a cost of $O(\mathrm{sort}(m+n)) = O(\mathrm{MST}_{\mathrm{EM}}(n,m))$ for bookkeeping (see the original paper), the authors prove the following theorem:

Theorem 4.8 (Meyer/Zeh [48]) *The single-source shortest paths problem on an undirected graph with n vertices and m edges, with edge lengths belonging to range $[w, W]$, can be solved in the external memory model with $O\left(\sqrt{\frac{nm\log(W/w)}{B}} + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$ I/O's.*

We conclude by mentioning that in [49] Meyer and Zeh refined their algorithm, removing the dependency of the I/O cost on edge lengths; the resulting algorithm solves the single-source shortest paths problem on undirected graphs incurring in $O(\sqrt{\frac{nm}{B}}\log n + \mathrm{MST}_{\mathrm{EM}}(n,m))$ I/O's.

4.2.3 Cache-oblivious BFS

The BFS algorithm of Mehlhorn and Meyer is based on scanning and sorting; it would be cache-oblivious if it did not have to choose a value of μ that depends on B.

Brodal *et al.* [27] present a cache-oblivious BFS algorithm which achieves a cost close to that of Mehlhorn and Meyer's, using the same conceptual schema and, essentially, guessing a suitable value for μ. The main idea is to replace the hot pool of [46] with a hierarchy of hot pool levels, operating with different, exponentially increasing values of μ. At some level t, μ is close to the value chosen by Mehlhorn and Meyer's algorithm: levels t and below play the role

of the hot pool in Mehlhorn and Meyer's algorithm, whilst higher levels play the role of the cluster repository on disk.

More precisely, the algorithm computes a *nested group partition* with $s = \lceil \log_2 n \rceil$ levels of granularity. At every level $j \in \{1, \ldots, s\}$, the properties of Mehlhorn and Meyer's clustering are replicated with $\mu = 2^j$, i.e.:

(G1) vertices are partitioned into $\mathrm{O}\big(n/2^j\big)$ *j-groups*, and

(G2) every j-group has diameter in G less than 2^j.

The group partition is nested, in the sense that

(G3) every j-group, $j < s$, is a subgroup of a $(j+1)$-group.

There is only one s-group $\mathcal{G}^*$, which contains the whole vertex set. Intuitively, $\mathcal{G}^*$ it is partitioned into two $(s-1)$-groups; each $(s-1)$-group into two $(s-2)$-groups, and so forth. The algorithm computes the nested group partition traversing a spanning tree of the graph, in a similar (but more sophisticated) way than Mehlhorn and Meyer's algorithm.

A hierarchical *hot pool* with s *levels* $\mathcal{H}_1, \ldots, \mathcal{H}_s$ store adjacency lists of unvisited vertices. We call *j-edge-group* the concatenation of all the adjacency lists of a j-group. Level $\mathcal{H}_j$, $j < t$, holds the j-edge-group corresponding to a j-group $\mathcal{G}$ if and only if $\mathcal{G}$ is contained in a $(j+1)$-group $\mathcal{G}'$ with a visited vertex, but $\mathcal{G}$ itself contains no visited vertex. In other words, the first time a vertex of a $(j+1)$-group $\mathcal{G}'$ is visited, $\mathcal{G}'$ is unpacked and its constituent j-groups are moved to level at most j. Level $\mathcal{H}_s$ contains, at the beginning of the algorithm, the whole edge set of the graph, contained in the j-edge-group corresponding to $\mathcal{G}^*$.

The algorithm proceeds in steps: every step visits all the vertices of one BFS level, as in Munagala and Ranade's algorithm. Consider any step, and let L be the set of vertices of the associated BFS level. The algorithm moves down groups in the hot pool, in the following way: For $j = s, s-1, \ldots, 2$, the algorithm extracts from $\mathcal{H}_j$ the j-edge-groups containing vertices in L. Each such edge-group is split into its constituent $(j-1)$-edge-groups, which are then stored in level $\mathcal{H}_{j-1}$. At the end of the iterations, the adjacency lists of all vertices in L are found in level $\mathcal{H}_1$, and extracted from there.

Hot pool levels are stored consecutively on disk; the algorithm dynamically maintains a memory layout where every level $\mathcal{H}_j$ occupies $\Theta\left(|\mathcal{H}_j|\right)$ consecutive disk elements. An index structure allows to access edge-groups performing one pass over $\mathcal{H}$ per BFS step: at every step, for every level $\mathcal{H}_j$, those edge-groups in $\mathcal{H}_j$ containing a vertex in L are directly accessed (following a pointer), in the same order in which they are stored in $\mathcal{H}_j$, starting from the end of the level. This access pattern, which can be seen either as a scan over $\mathcal{H}$ or as

a bunch of random accesses to its groups, is the key to the analysis of the algorithm.

For every fixed value of t, we upper bound the I/O complexity of the algorithm by the sum of two contributions: an upper bound on the cost of accessing j-groups with $j > t$, and an upper bound on the cost of accessing j-groups with $j \leq t$. Intuitively, every time we access a j-group with $j > t$ we perform one random disk access; while we access groups stored in the prefix $\mathcal{H}_t, \mathcal{H}_{t-1}, \ldots, \mathcal{H}_1$ scanning all the prefix. More precisely:

- The access cost to j-groups, $j > t$, can be upper bounded by $\mathrm{O}\big(n/2^t + (m \log n)/B\big)$ memory transfers: the first term is due to the memory transfers that may be necessary to access every such group, while the second term is the cost of scanning the m edges of the graph at most $\log n$ times (once per level), when they are moved across the hot pool. Notice that the total number of j-groups with $j > t$ is $\mathrm{O}\Big(\sum_{j=t+1}^{n} \big(n/2^j\big)\Big) = \mathrm{O}\big(n/2^t\big)$, by Property (G1).

- The I/O cost of accessing all the j-groups with $j \leq t$ during a BFS step can be upper bounded by the I/O cost of a complete scan over the prefix $\mathcal{H}_1, \ldots, \mathcal{H}_t$, with no random accesses (because the paging algorithm can reserve, across BFS steps, O(1) cache block to be used as a scanning buffer). Every edge e sticks around in the prefix $\mathcal{H}_1, \ldots, \mathcal{H}_t$ for at most $\mathrm{O}\big(2^t\big)$ BFS steps, because when e enters $\mathcal{H}_t$ there is a visited vertex in its t-group $\mathcal{G}$, and by Property (G2) $\mathcal{G}$ has diameter at most 2^t. Thus, the total cost of scanning edges as part of $\mathcal{H}_1, \ldots, \mathcal{H}_t$ can be bounded by $\mathrm{O}\big(\big(2^t m\big)/B\big)$ memory transfers.

The value $2^t = \Theta\left(\sqrt{(nB \log n)/m}\right)$ balances the two terms, and yields an upper bound of $\mathrm{O}\left(\sqrt{(nm \log n)/B}\right)$ memory transfers for operations involving $\mathcal{H}$. Adding a cost of $\mathrm{O}(\mathrm{sort}(m))$ memory transfers for sorting elements in X (see Section 4.1.1) and for managing indexes, and a cost of $\mathrm{O}(\mathrm{MST}_{\mathrm{CO}}(n, m)) = \mathrm{O}(\mathrm{sort}(m) \log \log n)$ to compute a nested group partition of the graph, and since $\mathrm{sort}(m) = \mathrm{O}(\mathrm{MST}_{\mathrm{CO}}(n, m))$ we obtain an upper bound of $\mathrm{O}\Big(\sqrt{(nm \log n)/B} + \mathrm{MST}_{\mathrm{CO}}(n, m)\Big)$ on the I/O cost of the algorithm.

If the graph is too dense, i.e. $m = \Omega(nB \log n)$, the optimal value for t, $t = \Theta(1)$, yields an I/O cost of $\mathrm{O}(n + (m/B) \log n + \mathrm{MST}_{\mathrm{CO}}(n, m))$ that, again, is not far away (asymptotically speaking!) from the I/O cost of Munagala and Ranade's BFS algorithm.

Chapter 5

An improved cache-oblivious SSSP algorithm

The content of this chapter is joint work with Peter Lichodzijewski and Norbert Zeh [9].

In the previous chapter we described a clustering paradigm that allows to perform BFS on undirected graphs incurring less than one I/O per vertex, in the external memory model. We showed two extensions of this result, respectively for the shortest paths problem and for cache-oblivious BFS, underlining the techniques that render these extensions possible. In this chapter, we show how to combine these techniques in order to get a cache-oblivious algorithm for the single-source shortest paths problem based on the clustering paradigm. The following theorem summarizes our result:

Theorem 5.1 *Given an undirected graph G with n vertices, m edges, and edge lengths between 1 and W, and provided that $M = \Omega\left(B^{1+\varepsilon}\right)$, the single-source shortest paths problem on G can be solved by a cache-oblivious algorithm that incurs $\mathrm{O}\big(\sqrt{(nm\log W)/B} + (m/B)\log n + \mathrm{MST}_{\mathrm{CO}}(n,m)\big)$ memory transfers.*

As Table 5.1 shows, the complexity of our algorithm can be seen as the perfect marriage between the complexities of the external memory shortest paths algorithm of Meyer and Zeh and that of the cache-oblivious BFS algorithm of Brodal *et al.*; in particular, our algorithm matches the cost of cache-oblivious BFS if the edge lengths come from a range of constant size.

Problem	Cache-Aware	Cache-Oblivious
BFS	Mehlhorn and Meyer [46] $O\left(\sqrt{\frac{nm}{B}} + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$	Brodal *et al.* [27] $O\left(\sqrt{\frac{nm}{B}} + \frac{m \log n}{B} + \mathrm{MST}_{\mathrm{CO}}(n,m)\right)$
SSSP	Meyer and Zeh [48] $O\left(\sqrt{\frac{nm}{B}} \log W + \mathrm{MST}_{\mathrm{EM}}(n,m)\right)$	Allulli *et al.* [9] $O\left(\sqrt{\frac{nm}{B}} \log W + \frac{m \log n}{B} + \mathrm{MST}_{\mathrm{CO}}(n,m)\right)$

Table 5.1: I/O complexity of algorithms based on the clustering paradigm

5.1 Algorithm outline

Our algorithm, at a high level, is almost identical to the single-source shortest paths algorithm of Meyer and Zeh, which we call MZ-SSSP, described in Section 4.2.2. The priority queue of Section 4.2.2.1 stores tentative distances of unvisited (and of visited, but spuriously updated) vertices. A hot pool $\mathcal{H}$ with $r = \lfloor \log W \rfloor + 1$ levels $\mathcal{H}_1, \ldots, \mathcal{H}_r$, henceforth called *rows*, contains the adjacency lists of vertices. The row in which each adjacency list is stored is determined according to Invariant 4.6. Invariant 4.6 uses the definitions of edge category and of component respectively given in Definition 4.3 and 4.4.

The main difference of our algorithm from MZ-SSSP is that we load the adjacency list of the whole graph in $\mathcal{H}$ at the beginning; according to the invariant, adjacency lists are stored in $\mathcal{H}_r$. Remember that MZ-SSSP makes a decisive use of the fact that components and clusters loaded in level $\mathcal{H}_i$ of the hot pool have diameter less than $2^i\mu$. In our cache-oblivious algorithm, we cannot determine μ, which depends on B: we guess μ similarly to cache-aware BFS algorithm by Brodal *et al.*, which we refer to as CO-BFS.

The purpose of the hot pool $\mathcal{H}$ and of its constituent elements, such as the rows $\mathcal{H}_1, \ldots, \mathcal{H}_r$, is to store adjacency lists. For the sake of simplicity of exposition, if $\mathcal{V} \subseteq V$ is a set of vertices we henceforth say that $\mathcal{H}$ (or one of its constituent elements) *contains* $\mathcal{V}$ to signify that it contains the adjacency lists of all the vertices of $\mathcal{V}$.

We divide every row $\mathcal{H}_i$ of $\mathcal{H}$ in a hierarchy of *buckets* $\mathcal{H}_{i,0}, \ldots, \mathcal{H}_{i,s}$, $s = \lceil \log n \rceil + 2$: bucket $\mathcal{H}_{i,j}$ stores parts of i-components with diameter bounded by 2^{i+j-1} (see Figure 5.1). When an i-component C is loaded into row $\mathcal{H}_i$ (this happens when C is moved from $\mathcal{H}_{i+1}$ to $\mathcal{H}_i$ because one vertex of C is moved from $\mathcal{B}_{i+1}$ to $\mathcal{B}_i$, as in MZ-SSSP), C is stored in a bucket $\mathcal{H}_{i,j}$ adequate to its diameter.

We define a *nested group partition* of the vertex set V in which every i-component is divided into groups, every group into subgroups, and so on. We will the formalize properties of the nested group partition later; for now we present an intuition of how it works:

- The diameter in G of an (i,j)-group $\mathcal{G}$ is bounded by 2^{i+j-1}; thus $\mathcal{G}$ can be stored in bucket $\mathcal{H}_{i,j}$.

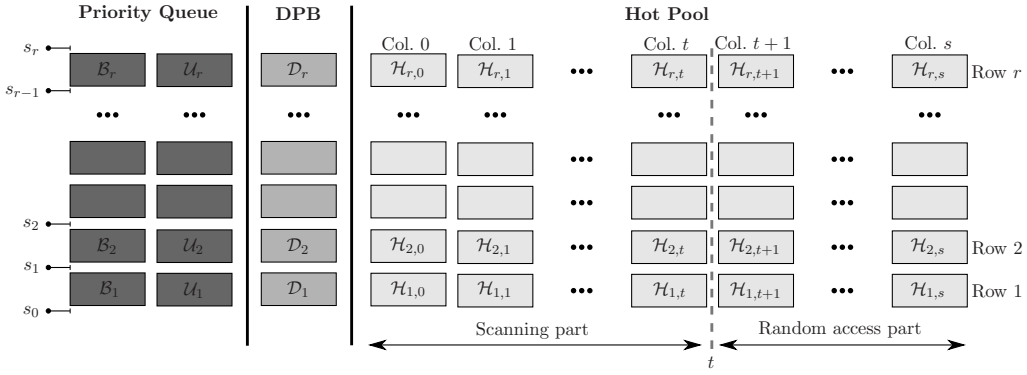

Figure 5.1: The priority queue, the down-propagation buffer (DPB) and the hot pool hierarchy.

- Every i-component is an (i, j)-group for some j; it is stored in bucket $\mathcal{H}_{i,j}$ when it enters row $\mathcal{H}_i$.
- Every (i, j)-group is either an $(i-1)$-component or is the union of disjoint $(i, j-1)$-groups. Groups of the "right" diameter play the same role as clusters in MZ-SSSP. Like clusters, they do not break $(i-1)$-components (see property K3).
- The number of groups is limited. More precisely, for $j \geq 1$ the number of $(\cdot, j)$-groups (i.e. the number of (i, j)-groups for any i) is $\mathrm{O}\big(n/2^j\big)$.

When we need to move an $(i-1)$-component C from row $\mathcal{H}_i$ to row $\mathcal{H}_{i-1}$, we repeatedly extract the (i, j)-group $\mathcal{G}$ containing C from $\mathcal{H}_{i,j}$: if $\mathcal{G} = C$, we move $\mathcal{G}$ to row $\mathcal{H}_{i-1}$; otherwise we split $\mathcal{G}$ into its constituent $(i, j-1)$-groups, which we store in $\mathcal{H}_{i,j-1}$; we iterate the process until we find C.

Since all adjacency lists are stored in $\mathcal{H}$ from the beginning, Invariant 4.6 enforces that the adjacency list of a vertex v is found in $\mathcal{H}_1$ when v is visited. Thus our algorithm does not need a second priority queue to delete from Q vertices inserted by spurious updates: for every such vertex v, $A(v)$ is simply not found in $\mathcal{H}_1$ when v is "spuriously" extracted from Q, and v is not processed a second time.

The idea behind the analysis of the algorithm is the following: There is a t such that $(\cdot, j)$-groups with $j > t$ are few enough ($\mathrm{O}\big(n/2^t\big)$ in total) so that the algorithm affords spending one memory transfer to individually access each of them, when they are requested; whilst $(\cdot, j)$-groups with $j \leq t$ are small enough, in terms of diameter, so that they do not linger in the hot pool for too many distance steps: the algorithm affords to scan every such (i, j)-group every time $\mathcal{H}_i$ is inspected. In other words, the "left" part of the hot

pool, i.e. buckets $\mathcal{H}_{i,0}, \ldots, \mathcal{H}_{i,t}$ for every i, plays the same role of the hot pool in MZ-SSSP, holding adjacency lists that are close enough to being visited; whilst the "right" part of the hierarchy $\mathcal{H}_{i,t+1}, \ldots, \mathcal{H}_{i,s}$ for every i plays the role of a repository for large groups, which is touched only when a group is needed.

The algorithmic challenge is to layout and dynamically maintain the hot pool structures, including an adequate set of indexes, so that in every distance step buckets are individually accessed in an ordered way, with one pass over the arrays where they are stored; so that the access pattern can be interpreted as a scan over (i, j)-groups with $j \leq t$, and as a bunch of random accesses to all the other groups.

Section 5.2 discusses the group partition required by our algorithm. Section 5.3 then discusses how to use this partition to efficiently maintain the hot pool structure.

5.2 Group partition

5.2.1 Properties of the nested group partition

We now formally define the properties of the partition used by our algorithm, and we provide a procedure to build it cache-efficiently.

A *nested group partition* is a set of (i, j)-groups, $1 \leq i \leq r = \lfloor \log W \rfloor + 1$ and $0 \leq j \leq s = \lceil \log n \rceil + 2$, with the following properties; where we refer to an (i, j)-group as a $(\cdot, j)$-group or $(i, \cdot)$-group if we do not want to specify i or j.

(P1) Every $(i, \cdot)$-group is completely contained in an i-component.

(P2) Every i-component is an $(i + 1, j)$-group and an (i, j')-group, for some j and $j' \leq j + 3$. The i-component then contains no $(i + 1, j'')$-groups with $j'' < j$ and no (i, j'')-group with $j'' > j'$.

(P3) For every pair (i, j) and any two distinct (i, j)-groups $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_1 \cap \mathcal{G}_2 = \emptyset$.

(P4) Every (i, j)-group is either equal to an i-component or completely contained in an $(i, j + 1)$-group.

(P5) Every (i, j)-group has diameter less than 2^{i+j-1}.

(P6) For every $j > 0$, there are $\mathrm{O}\big(n/2^j\big)$ $(\cdot, j)$-groups.

We say that an (i, j)-group is *trivial* if it consists of a single $(i - 1)$-component or $(i, j - 1)$-group; otherwise, it is *non-trivial*. Now observe that

the nesting relationship defines a tree-like hierarchy of the groups whose root is the whole graph G and whose leaves are the vertices of G (which are 0-components). Trivial groups are unary nodes in the tree; if we remove them, we are left with a tree with n leaves and with every internal node of arity at least 2. Thus, there are $O(n)$ non-trivial groups in the partition.

We number the vertices in G in the order they are visited by a postorder traversal of the group tree. This implies that the vertices in every (i, j)-group are numbered consecutively.

The *representation* of the partition consists of the concatenation of the adjacency lists $A(v)$ of all vertices, sorted by the identifiers of vertices v; the adjacency list of vertex v is preceded by descriptors of all non-trivial groups in the partition that have v as their minimum vertex. These descriptors are sorted by decreasing numbers of vertices contained in their corresponding groups. Each descriptor describing a non-trivial (i, j)-group stores the pair (i, j), the number of edges in the adjacency lists of all vertices in its group, plus the number of smaller non-trivial groups nested in it. For every (i, j)-group, the corresponding *edge group* is the subsequence of the representation that starts with the group's descriptor and ends with the adjacency list of its last vertex. Since we store only descriptors of non-trivial groups, the total number of descriptors is $O(n)$, so that the total length of this representation is still $O(m)$.

5.2.2 The partitioning procedure

We compute the cluster partition using an extension of the clustering procedure used in MZ-SSSP (see Figure 5.2. We start by computing a minimum spanning tree T of G. This takes $O(\mathrm{MST}_{\mathrm{CO}}(n, m))$ memory transfers. Observe that an i-component of T is a spanning tree of an i-component of G, and the distance between two vertices in T is an upper bound on their distance in G. Hence, it suffices to compute a nested group partition of T, except that, in the final representation as a concatenation of adjacency lists, we have to store the edges in G, not only in T, incident to every vertex.

We number the vertices in T in the order in which they are visited for the first time by an Euler tour $\mathcal{E}$ of T. Computing tour $\mathcal{E}$ takes $O(\mathrm{sort}(n))$ memory transfers (see Section 3.6). The first phase of our clustering procedure produces $O(n)$ group descriptors of the form (i, j, k, p) and $O(n)$ component descriptors of the form (i, k, p). For both, group and component descriptors, k is a unique identifier of the group or component, and p is the identifier of the parent of the group or component in the nesting of groups and components that follows from the definition of a nested group partition. For a component, i is its category. For a group, the first two elements of its descriptor indicate that the group is an (i, j)-group. Note that, since vertices are 0-components,

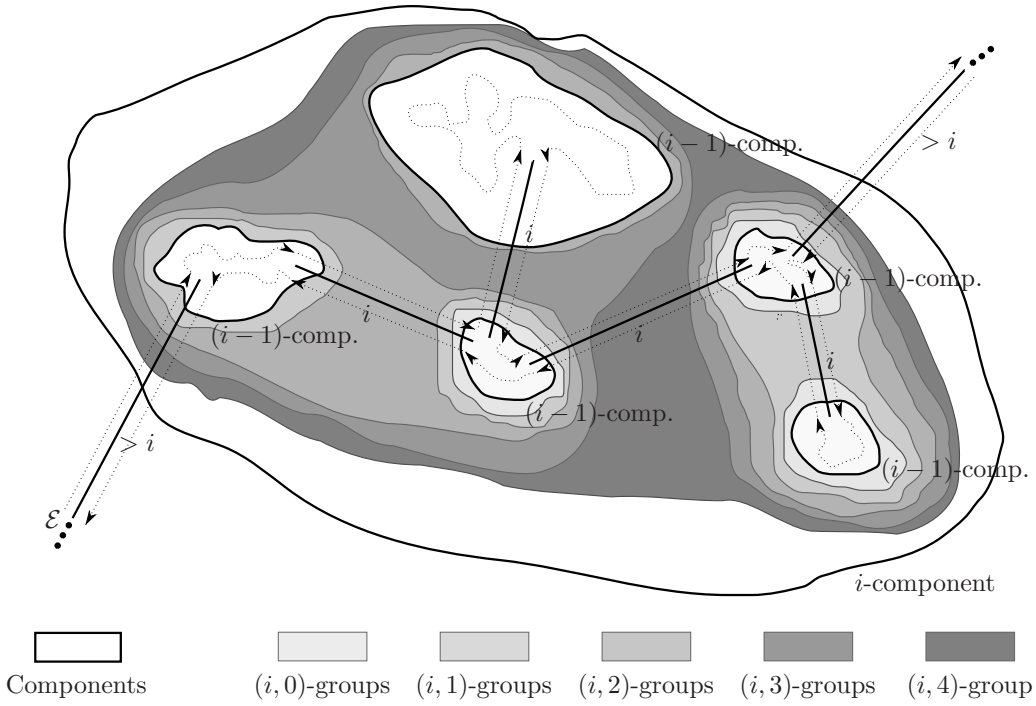

Figure 5.2: Partitioning an i-component into $(i,\cdot)$-groups. Groups are formed during the second scan of the i-th round of the first phase of the partitioning procedure. Even if $(i,\cdot)$-groups are formed scanning $\mathcal{E}$ once, think of forming them with $s+1$ traversals of $\mathcal{E}$, for $j = 0, 1, \ldots, s$: during traversal j, $(i, j-1)$-groups and sufficiently small $(i-1)$-components encountered for the first time are put together to form an (i,j)-group of diameter less than 2^{i+j-1}. An upper bound to the diameter of the group under construction is maintained: when it exceeds 2^{i+j-1}, a new (i,j)-group is created.

they are represented by component descriptors. For every vertex v, we choose the identifier of its 0-component to be v. This makes the second phase easier. The second phase then renumbers the vertices according to their membership in groups and components and constructs the final layout of the edge groups and group identifiers.

5.2.2.1 Computing group descriptors

The first phase consists of r rounds. The i'th round computes all $(i, \cdot)$-groups and their parents, as well as the parents of all $(i-1)$-components. More precisely, every $(i-1)$-component is a trivial (i, j)-group for some j; for each (i, j)-group $\mathcal{G}$, we compute the smallest (i, j')-group that properly contains $\mathcal{G}$ or, if there is no such group, the i-component that is equal to $\mathcal{G}$.

Each round scans the Euler tour $\mathcal{E}$ twice. Before the first round, we label every 0-component, that is, every vertex v with the pair $(0, 0, v)$, where the first 0 is the category of the component, the second 0 is the weight of the edges in this component, and v is the identifier of this component. In general, the i'th round needs to know the weight $w(C)$ and identifier of every $(i-1)$-component C, where the weight of a component is the total number of edges in T that belong to this component. This information is required to be stored with the first vertex in C visited by $\mathcal{E}$.

The same information is computed by the first scan in the i'th round for all i-components. This information is used in the i'th round itself and in the $(i+1)$'st round. During the first scan, $\mathcal{E}$ is scanned backwards, and the appropriate $(i, w(C), k)$ triples are stored with the first vertex in each component C, where k is the identifier of C. The second scan in the i'th round scans $\mathcal{E}$ forward and computes the descriptors of all $(i, \cdot)$-groups, as well as their parents.

The first scan is implemented using two stacks S_e and S_c. The former stores edges that have been traversed in one direction, but not in the other; this stack is used to decide when we see an i-component for the first time. The latter stores quadruples $(i, w(C), k, t)$, for all i-components C that have not been traversed completely yet, where t is a Boolean flag that is true if the i-component consists of a single $(i-1)$-component. The value $w(C)$ is updated as new edges in component C are discovered.

Initially, we push one pair $(i, 0, k, \textbf{true})$ onto stack S_c, where k is a new identifier; stack S_e is initially empty. When traversing an edge vw, we do the following:

- If the category of edge vw is at most i, we add $\ell(vw)$ to the $w(C)$ value of the top entry on S_c. In addition, if the category is exactly i, we set the t-value of the top entry on S_c to **false**.

- If the category of vw is greater than i, we inspect the top edge on S_e and distinguish two cases:
  - If the top edge on S_e is wv, we remove it from S_e and remove the top entry $(i, w(C), k, t)$ from S_c. If $t =$ **true**, we store the triple $(i, w(C'), k')$ with v, where $(i-1, w(C'), k')$ is the descriptor of the $(i-1)$-component containing v, which is already stored with v. We do this because the $(i-1)$-component and the i-component containing v are identical. If $t =$ **false**, we store $(i, w(C)/2, k)$ with the current copy of vertex v. We divide $w(C)$ by two because so far we have counted every edge in C twice.
  - If the top edge on S_e is not equal to wv, we push vw onto S_e and a new quadruple $(i, 0, k, \textbf{true})$ onto S_c, where k is a new identifier.

Finally, once the last edge has been traversed, we remove the last entry $(i, w(C), k, t)$ from S_c. If $t =$ **true**, we store $(i, w(C'), k')$ with the first vertex in $\mathcal{E}$, where $(i-1, w(C'), k')$ is the descriptor of the $(i-1)$-component containing this vertex, which is already stored with this vertex. If $t =$ **false**, we store $(i, w(C)/2, k)$ with the first vertex in $\mathcal{E}$.

Intuitively, whenever we see an edge of category greater than i for the first time, we enter a new i-component and need to start computing its weight, postponing the computation of the weight of the component we are leaving to be resumed later. When we see an edge for the second time (that is, when its opposite edge is on top of S_e), we have finished traversing one component; then we store its weight with its first vertex, which is the endpoint of the edge through which we are leaving the component, and we resume the weight computation of the previous component by making its weight the top of S_c again.

This first scan involves $O(n)$ stack operations and scans $\mathcal{E}$ once. Hence, the first scan takes $O(n/B)$ memory transfers.

The second scan scans $\mathcal{E}$ forward and computes the descriptors of all $(i, \cdot)$-groups, as well as their parents. It maintains the following information: There is a set of $\lceil \log n \rceil + 3$ triples (d_j, t_j, k_j), for $0 \le j \le s = \lceil \log n \rceil + 2$. Triple (d_j, t_j, k_j) indicates that the current (i, j)-group under construction has diameter at most d_j and identifier k_j; t_j is a Boolean flag that is true if the group is trivial. We also store the identifiers K_{i-1} and K_i of the current $(i-1)$- and i-components. In addition, we have a stack S_e as in the first scan and a stack S_q to hold triples (d_j, t_j, k_j) and identifiers K_{i-1} and K_i for i-components we haven't finished scanning yet. These stacks are initially empty.

Initially, we set $d_j = w(C)$, $t_j =$ **true**, and choose each k_j to be a new unique identifier, for all $0 \le j \le s$, where $(i-1, w(C), k)$ is the descriptor of the $(i-1)$-component containing the first vertex of $\mathcal{E}$. We initialize K_{i-1} and

K_i to be the identifiers of the $(i-1)$- and i-components containing the first vertex of $\mathcal{E}$. (Note that this vertex stores all this information.) For every edge vw, we now do the following:

- If edge vw has category less than i, we add its length to every value d_j, $0 \leq j \leq s$.
- If edge vw has category i, we distinguish two cases:
  - If the edge on the top of stack S_e is wv, we remove it from the top of the stack and add $\ell(vw)$ to every value d_j, $0 \leq j \leq s$.
  - If the edge on the top of stack S_e is not equal wv, we push vw on the stack. We add $\ell(vw) + w(C)$ to every d_j value, for $0 \leq j \leq s$, where $w(C)$ is the weight of the $(i-1)$-component containing w, which is stored with w. Then we identify the highest value j' such that $d_{j'} \geq 2^{i+j'-1}$. We set $t_{j'+1} = \textbf{false}$. We then output group descriptors (i, j, k_j, p), for all $0 \leq j \leq j'$ with $t_j = \textbf{false}$. For each descriptor we output, the identifier of the next descriptor we output is its parent p. For the last descriptor, its parent is $k_{j'+1}$. In addition, we output the component descriptor $(i-1, K_{i-1}, k_{j''})$, where $j'' \leq j'+1$ is the smallest j'' such that $t_{j''} = \textbf{false}$, that is, the parent of the last $(i-1)$-component is the (i, j'')-group that contains it.

    Once this is done, we set $d_j = w(C)$, $t_j = \textbf{true}$, and assign a new identifier k_j, for all $0 \leq j \leq j'$. Then we update K_{i-1} to be the identifier of the $(i-1)$-component containing w.
- If edge vw has category greater than i, we again distinguish two cases:
  - If the edge on the top of stack S_e is not wv, we push all triples (d_j, t_j, k_j), as well as identifiers K_{i-1} and K_i on stack S_q. Then we set $d_j = w(C)$, $t_j = \textbf{true}$, and choose k_j to be a new identifier, for all $0 \leq j \leq s$, where $w(C)$ is the weight of the $(i-1)$-component containing w, which is stored with w. We update K_{i-1} and K_i to be the identifiers of the $(i-1)$- and i-components containing w.
  - If the edge on the top of stack S_e is wv, we pop this edge from S_e. Then we distinguish two cases:
    - If $K_{i-1} = K_i$, the i-component we are currently closing is in fact an $(i-1)$-component. In this case, we do not output anything.

* If $K_{i-1} \neq K_i$, there is at least one non-trivial group in the i-component, and we output group descriptors (i, j, p), for all $0 \leq j \leq s$ such that $t_j = \textbf{false}$. Again, we set the parent of each group descriptor we output to be the identifier of the next group descriptor we output. For the last (i, j)-group we output, we set the parent to be K_i. We also output a component descriptor $(i - 1, K_{i-1}, k_{j''})$, where j'' is the minimum index such that $t_{j''} = \textbf{false}$.

Once this is done, we restore the triples and identifiers K_{i-1} and K_i for the previous i-component from the stack S_q.

Once the second scan ends, we perform the same steps as in the second case for edges of category greater than i, except that we do not restore any tuples from S_q. This is necessary to finish forming groups in the i-component containing the last visited vertex.

This second scan incurs $O(\log n/B)$ memory transfers per edge in $\mathcal{E}$ because it has to scan the list of triples (d_j, t_j, k_j) for every edge in $\mathcal{E}$. Thus, the total cost of this second scan is $O((n/B) \log n)$ in total, which dominates the cost of the first scan. Hence, the total cost of each round is $O((n/B) \log n)$. Since there are $\lfloor \log W \rfloor + 1$ rounds, the total cost of this first phase of the clustering algorithm is $O((n/B) \log n \log W)$. In order to reduce this cost to $O((n/B)(\log n + \log W))$, we make the following modifications:

Instead of adding the weight of every edge of category less than i to all values d_j immediately, we keep a total length l of all edges of category less than i traversed since the last time an edge of category at least i was traversed. When traversing the next edge of category at least i, we add l to all d_j-values and reset l to zero. Thus, every edge of category less than i costs only $O(1/B)$ memory transfers amortized.

Similarly, when traversing an edge of category greater than i, we postpone the work to be done for this edge till we see the next edge of category i. More precisely, for every forward edge, that is, every edge that does not find its opposite edge on S_e, we do not initialize triples (d_j, t_j, k_j) as above, but only store a single pair (d, t), which can be used instead of any triple (d_j, t_j, k_j) because the (i, j)-groups are all identical at this point. If we never see a category-i edge in the same i-component, we do not have to do any more work when we traverse the corresponding backward edge because the i-component is in fact an $(i - 1)$-component. If we do see a category-i edge in this i-component, we initialize all triples (d_j, t_j, k_j) to (d, t, k_j) at this point, where k_j is a new identifier, and then proceed as above. The cost of $O(\log n/B)$ memory transfers incurred by the backward edge through which we leave this component can then be charged to the last category-i edge in the component. In summary, the i'th round costs $O(1/B)$ memory transfers per edge of category other

than i and $O(\log n/B)$ memory transfers per category-i edge. Thus, every edge incurs $O(\log n/B)$ memory transfers in exactly one round and $O(1/B)$ in any other round. This leads to a cost of $O((\log n + \log W)/B)$ per edge, $O((n/B)(\log n + \log W))$ in total.

5.2.2.2 Computing the final representation

Computing the final representation is now fairly straightforward. We perform a depth-first traversal of the tree defined by the group and component descriptors and their parent pointers. We number the vertices of G in the order in which the descriptors of their corresponding 0-components are visited by this traversal. The traversal also allows us to assign to every group descriptor the range of vertex numbers in this group, that is, the range of numbers assigned to all 0-components that are descendants of the group descriptor in the tree. This can all be achieved in $O(\text{sort}(n))$ memory transfers using standard tree processing techniques such as Euler tours and list ranking (see Sections 3.5.2 and 3.6) because there are in total only $O(n)$ descriptors involved in this step. To see this, observe that we output group and component descriptors only for non-trivial groups and components, that is, groups and components that are composed of smaller groups or components. Thus, the total number of group descriptors is at most twice the number of vertices in G, that is, at most $2n$; the same holds for component descriptors.

Once the vertices and group descriptors have been labeled like this, we produce a list of group descriptors and vertices, sorted so that the vertices are sorted by increasing numbers and every vertex v is preceded by the descriptors of all groups that have v as their minimum vertex. We break ties between group descriptors with the same minimum vertex by sorting these groups by decreasing number of vertices they contain. This is a simple sorting step, that is, costs $O(\text{sort}(n))$ memory transfers. Let the resulting list be L.

Next we sort and scan L and the list of all adjacency lists to replace every vertex v in L with its adjacency list. Then we scan the resulting list L' to store with every group descriptor the total size of its edge group, that is, the number of edges and descriptors of smaller groups contained in this group. This takes another $O(\text{sort}(m))$ memory transfers, and the total cost of the second phase is $O(\text{sort}(m))$.

We consider an (i, j)-group $\mathcal{G}$ to be part of the partition if it satisfies one of the following three conditions:

- Group $\mathcal{G}$ is non-trivial,
- Group $\mathcal{G}$ is identical to an $(i-1)$-component that is an $(i-1, 0)$-group or a non-trivial $(i-k, j+k)$-group, for some $k > 0$, or

- Group $\mathcal{G}$ is identical to an (i, j')-group with $j' < j$ that satisfies one of the previous two conditions, and it is properly contained in a non-trivial (i, j'')-group with $j'' > j$.

Lemma 5.2 *A nested group partition can be computed in* $O(\mathrm{MST}_{\mathrm{CO}}(n, m) + n(\log n + \log W)/B)$ *memory transfers.*

Proof. The above discussion has already established the claimed complexity bound. We have to prove that the procedure we have described does indeed compute a nested group partition. Properties (P1)–(P4) are easily verified. We prove Properties (P5) and (P6).

To see the diameter bound, observe that every (i, j)-group is composed of $(i-1)$-components. Whenever we are about to add a new $(i-1)$-component to an (i, j)-group by traversing a category-i edge, the clustering procedure checks whether this would increase the diameter of the (i, j)-group to at least 2^{i+j-1}. If this is the case, we start a new (i, j)-group. Thus, we never create any (i, j)-group of diameter at least 2^{i+j-1}, and Property (P5) holds.

Now let us bound the number of $(\cdot, j)$-groups, for $j > 0$. For every (i, j)-group, Phase 1 of our clustering procedure computes an upper bound d_j on its diameter, which we call the *Euler diameter* of the group. We split (i, j)-groups into three groups:

- A type-I (i, j)-group is one that satisfies one of the first two conditions stated before the lemma.
- A type-II (i, j)-group is one that is not a type-I group and is contained in a non-trivial $(i, j+1)$-group $\mathcal{G}$ and which is not the last group contained in $\mathcal{G}$ that is produced in round i of Phase 1 of the construction procedure.
- A type-III group is one that is neither of type I or type II.

Let $n_{i,j}$ be the number of (i, j)-groups, and let $n_j = \sum_i n_{i,j}$ be the number of $(\cdot, j)$-groups. We use $n'_{i,j}$ and n'_j to count the numbers of (i, j)-groups and $(\cdot, j)$-groups of types I and II; $n''_{i,j}$ and n''_j denote the numbers of (i, j)-groups and $(\cdot, j)$-groups of type III. We use E_i to denote the set of category-i edges in T. We include every edge three times in E_i: the first copy accounts for the edge's contribution to the weight of an i-component; the other two copies account for the two occurrences of the edge in the Euler tour. Thus, $\sum_i |E_i| = 3n - 3$. We use $\ell(E_i)$ to denote the total length of the edges in E_i.

First we observe that every (i, j)-group $\mathcal{G}$ of type I has Euler diameter at least 2^{i+j-2}. Indeed, it contains at least two $(i, j-1)$-groups $\mathcal{G}_1$ and $\mathcal{G}_2$. Group $\mathcal{G}_2$ is produced only when adding the first $(i-1)$-component in $\mathcal{G}_2$ to $\mathcal{G}_1$ would increase the Euler diameter of $\mathcal{G}_1$ beyond $2^{i+(j-1)-1}$. Therefore, if we charge

the edges in $\mathcal{G}_1$ and $\mathcal{G}_2$ and the edges in $\mathcal{E}$ between $\mathcal{G}_1$ and $\mathcal{G}_2$ for the creation of $\mathcal{G}$, we charge edges of total length at least 2^{i+j-2} for the creation of $\mathcal{G}$. It is easy to see that every edge in E_i is charged at most once for the creation of a type-I group.

For every (i,j)-group $\mathcal{G}$ of type II, we charge edges of total length at least 2^{i+j-1}: Namely, whenever this group $\mathcal{G}$ is created, adding the first $(i-1)$-component contained in the next (i,j)-group $\mathcal{G}'$ contained in the same $(i,j+1)$-group as $\mathcal{G}$ would increase its Euler diameter beyond 2^{i+j-1}. We charge the edges in $\mathcal{G}$, the edges in the Euler tour between $\mathcal{G}$ and the first edge in $\mathcal{G}'$, and the edges in $\mathcal{G}'$ for the creation of $\mathcal{G}$. Hence, every edge is charged at most twice for the creation of an (i,j)-group of type-II.

Thus, the total number of (i,j)-groups of types I and II is

$$\begin{aligned} n'_{i,j} &\le \sum_{i'\le i} \frac{\ell(E_{i'})}{2^{i+j-2}} + \sum_{i'\le i} \frac{2\ell(E_{i'})}{2^{i+j-1}} \\ &= \sum_{i'\le i} \frac{\ell(E_{i'})}{2^{i+j-3}} \end{aligned}$$

because only edges of category at most i contribute to the diameters of $(i,\cdot)$-groups. Then

$$\begin{aligned} n'_j &= \sum_i n'_{i,j} \\ &\le \sum_i \sum_{i'\le i} \frac{\ell(E_{i'})}{2^{i+j-3}} \\ &= 2^{3-j} \sum_i 2^{-i} \sum_{i'\le i} \ell(E_{i'}) \\ &= 2^{3-j} \sum_{i'} \ell(E_{i'}) \sum_{i\ge i'} 2^{-i} \\ &= 2^{3-j} \sum_{i'} 2^{-i'} \ell(E_{i'}) \sum_{i\ge 0} 2^{-i} \\ &< 2^{4-j} \sum_{i'} |E_{i'}| \\ &< \frac{48n}{2^j}. \end{aligned}$$

The second-last inequality holds because every edge in $E_{i'}$ has length less than $2^{i'}$, that is, $\ell(E_{i'}) < 2^{i'}|E_{i'}|$.

As for type-III (i,j)-groups, we observe that each $(i,j+1)$-group contains at most one of them. Hence, $n''_{i,j} \le n_{i,j+1}$ and $n''_j \le n_{j+1}$. This implies that

$n_j < 96n/2^j$. Indeed, for $j = \lceil \log n \rceil + 2$, there are no type-III groups. Hence, $n_j = n'_j < 48n/2^j$. For $j < \lceil \log n \rceil + 2$, we have

$$\begin{aligned} n_j &= n'_j + n''_j \\ &\leq n'_j + n_{j+1} \\ &< \frac{48n}{2^j} + \frac{96n}{2^{j+1}} \\ &= \frac{96n}{2^j}. \end{aligned}$$

□

Lemma 5.3 *A nested group partition of an undirected graph with n vertices and m edges can be computed in $\mathrm{O}(\mathrm{MST}_{\mathrm{CO}}(n,m) + (n/B)(\log n + \log W))$ memory transfers.*

5.3 Hot pools

The remainder of this section is dedicated to describing the hot pool structure. The hot pool consists of a two-dimensional array of *buckets* $\mathcal{H}_{i,j}$, $1 \leq i \leq r = \lfloor \log W \rfloor + 1$ and $0 \leq j \leq s = \lceil \log n \rceil + 2$; bucket $\mathcal{H}_{i,j}$ stores (i,j)-groups. Initially, the whole graph is viewed as an (r, j^*)-group, for some j^*, which is stored in $\mathcal{H}_{r,j^*}$. We refer to buckets $\mathcal{H}_{i,0}, \ldots, \mathcal{H}_{i,s}$ as *row* i and to buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{r,j}$ as *column* j.

In order to maintain Invariant 4.6, we have to support the following *down-propagation operation* (see Figure 5.3): When moving a list L of vertices from bucket $\mathcal{B}_i$ to bucket $\mathcal{B}_{i-1}$ in the priority queue, we have to move all $(i-1)$-components containing vertices in L from row i to row $i-1$. A special case of this is retrieving (from row 1) the adjacency lists of the vertices returned by a BatchedDeleteMin() operation.

Conceptually, we do the following for every vertex $v \in L$. First we locate the (i,j)-group $\mathcal{G}$ in the i'th row that contains v. If $\mathcal{G}$ is the $(i-1)$-component containing v, we move it to row $i-1$: $\mathcal{G}$ is also an $(i-1, j')$-group, for some $j' \leq j+3$; so we insert it into $\mathcal{H}_{i-1,j'}$. If $\mathcal{G}$ is not an $(i-1)$-component, we split $\mathcal{G}$ into $(i, j-1)$-groups, which we insert into $\mathcal{H}_{i,j-1}$. We keep doing this until there are no more groups in row i containing vertices in L.

To implement this procedure efficiently, we need an appropriate layout of the hot pool buckets in memory, as well as a number of indexing structures to locate groups in the hot pool. The hot pool is represented by *column structures* $\mathcal{C}_0, \ldots, \mathcal{C}_s$, each storing the buckets of a column. Intuitively, upon every BatchedDeleteMin() call the algorithm accesses a prefix of every column

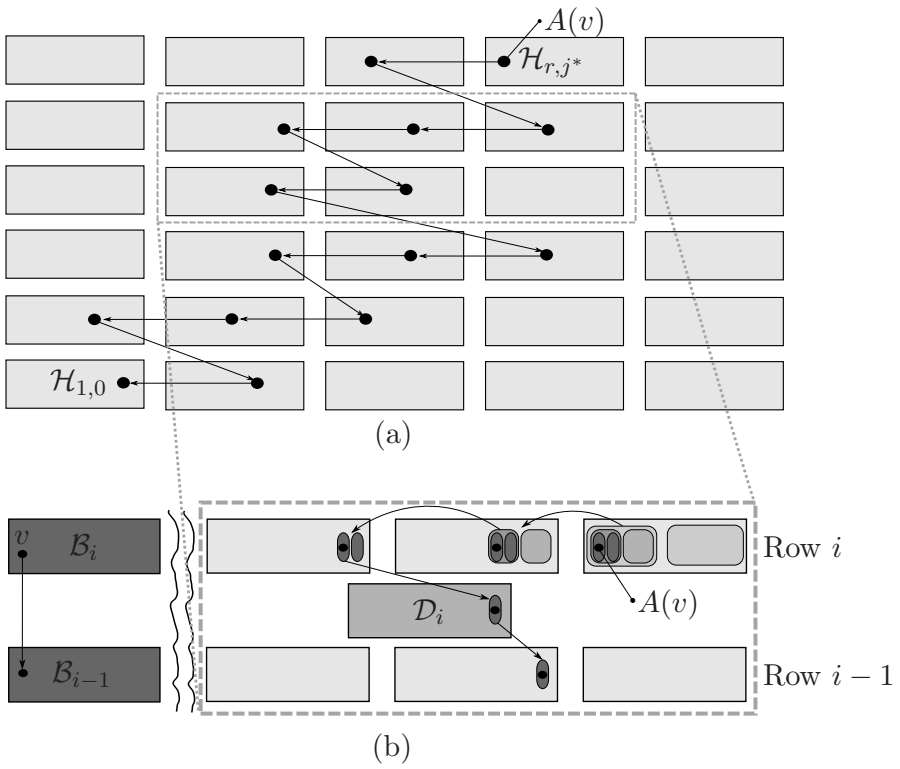

Figure 5.3: (a) Movement of an adjacency list $A(v)$ in the hot pool hierarchy. At the beginning, $A(v)$ is in the only (r, j^*)-group in bucket $\mathcal{H}_{r,j^*}$. Down-propagation operations cause $A(v)$ to move left, down, left and down, or one down and at most three right. At the end, $A(v)$ is found in $\mathcal{H}_{1,0}$. (b) Details of a down-propagation operation: when v moves from $\mathcal{B}_i$ to $\mathcal{B}_{i-1}$, the group containing $A(v)$ is repeatedly extracted from $\mathcal{H}_{i,j}$ and its $(i, j-1)$-subgroups are stored in $\mathcal{H}_{i,j-1}$, until the $(i-1)$-component containing $A(v)$ is found and inserted in the down-propagation buffer. When it leaves the down-propagation buffer, the component is stored in the appropriate bucket at a row $i' < i$.

j, performing one pass over buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$. For a $t = \mathrm{O}(\log B)$ that will be chosen later, a pass over prefix $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$, $j \leq t$, can be interpreted as a scan over that prefix of column j. Column structures allow our algorithm to pass over such prefixes with a cost bounded by the cost of scanning their elements. The tall cache assumption implies that $\Theta(t) = \mathrm{O}(\log B) = \mathrm{O}(B^{\varepsilon})$ blocks can be maintained in cache across BatchedDeleteMin() calls, thus allowing to begin scanning a column structure without performing memory transfers due to random disk accesses. Column structures are discussed in detail in Section 5.3.2

For every row, we have a *row index* $\mathcal{I}_i$, which represents the groups stored in row i. In particular, a group $\mathcal{G}$ in $\mathcal{H}_{i,j}$ containing vertices with identifiers between a and b and being the k'th group ever inserted into the hot pool is represented by a group descriptor (i, j, a, b, k) in $\mathcal{I}_i$. The descriptors in $\mathcal{I}_i$ are sorted by their a-components. Since the intervals of vertex identifiers in the groups in row i are disjoint, $\mathcal{I}_i$ is thus a sorted list of disjoint vertex intervals.

The moving of groups between rows is achieved using a *down-propagation buffer* $\mathcal{D}$, which is a concatenation of buckets $\mathcal{D}_1, \ldots, \mathcal{D}_r$; bucket $\mathcal{D}_i$ stores $(i-1)$-components that are moving to row $i-1$ soon. In particular, each such component contains a vertex in $\mathcal{B}_i$.

The down-propagation of $(i-1)$-components containing vertices in L is now implemented as follows: First we locate all groups in row i that contain vertices in L. Since the vertices in L are given sorted by their identifiers, a single scan of L and $\mathcal{I}_i$ suffices to retrieve the list P of descriptors of all groups in row i that contain vertices in L; we remove these descriptors from $\mathcal{I}_i$. We sort the descriptors in P by their j- and k-components. For each set of descriptors with the same j-component, we retrieve the corresponding groups from the appropriate column structure $\mathcal{C}_j$ and append them to an array $\mathcal{L}$. For each such group, we add a descriptor (i, j, a, b, p) to an index $\mathcal{J}$, where p is a pointer to the location of the group in $\mathcal{L}$.

We now iteratively perform the following procedure until L is empty: We discard all vertices in L that are not contained in any group in $\mathcal{L}$. For every (i, j)-group $\mathcal{G}$ in $\mathcal{L}$, if it does not contain a vertex in L, we insert it into bucket $\mathcal{H}_{i,j}$ and add a corresponding group descriptor to an index $\mathcal{I}'_i$ to be merged with $\mathcal{I}_i$ later. If $\mathcal{G}$ contains a vertex in L and it is an $(i-1)$-component, we identify the vertex in $\mathcal{G}$ with minimum priority (which must belong to L), store its priority with $\mathcal{G}$, and then insert $\mathcal{G}$ into $\mathcal{D}_i$. If $\mathcal{G}$ is not an $(i-1)$-component, we split it into its constituent $(i, j-1)$-groups,[1] which we add to a list $\mathcal{L}'$; for these groups, we add a group descriptor as those in $\mathcal{J}$ to an index $\mathcal{J}'$. Once all groups in $\mathcal{L}$ have been processed, $\mathcal{L}'$ and $\mathcal{J}'$ replace $\mathcal{L}$ and $\mathcal{J}$.

[1] Note that these groups may be trivial. In this case, we still do not move them to columns to the left of column $j-1$ unless they contain vertices in L.

Each iteration of this loop can be implemented by sorting and scanning indexes $\mathcal{J}$ and $\mathcal{J}'$ and by scanning arrays L, $\mathcal{L}$, and $\mathcal{L}'$.

The actual down-propagation is now implemented by inspecting $\mathcal{D}_i$. A group $\mathcal{G}$ remains in $\mathcal{D}_i$ if its priority is at least p_{i-1}. If its priority is less than p_{i-1}, it is moved to row $i-1$. If $\mathcal{G}$ is an $(i-2)$-component, we simply insert it into $\mathcal{D}_{i-1}$. Otherwise, it is a non-trivial $(i-1,j)$-group, for some j. In this case, we insert $\mathcal{G}$ into bucket $\mathcal{H}_{i-1,j}$ and add a corresponding index entry to an index $\mathcal{I}'_{i-1}$. Once we have processed all groups in $\mathcal{D}_i$, we sort the group descriptors in $\mathcal{I}'_{i-1}$ and $\mathcal{I}'_i$ by their a-components and merge them into $\mathcal{I}_{i-1}$ and $\mathcal{I}_i$, respectively.

Next we analyze the cost of manipulating lists L, $\mathcal{L}$, and $\mathcal{L}'$, as well as indexes $\mathcal{I}_i$, $\mathcal{I}'_i$, $\mathcal{J}$, and $\mathcal{J}'$. The cost of manipulating the down-propagation buffer is analyzed in Section 5.3.1, which also proves the correctness of our criterion for moving groups from $\mathcal{D}_i$ to row $i-1$. The cost of manipulating the column structures is analyzed in Section 5.3.2.

The following two lemmas will be used throughout the analysis of the hot pool.

Lemma 5.4 *Every edge is involved in at most* $\mathrm{O}(\log n + \log W)$ *insertions into buckets or arrays* $\mathcal{L}$ *and* $\mathcal{L}'$*. After it enters a bucket* $\mathcal{H}_{i,j}$*, it remains in the hot pool for* $\mathrm{O}\big(2^{i+j}\big)$ *distance steps, where a distance step is the increase of the minimum priority of the vertices in the priority queue by one.*

Proof. Consider an edge e. Whenever edge e is inserted into array $\mathcal{L}$ or $\mathcal{L}'$ as part of an (i,j)-group, let us think of e as inserted into bucket $\mathcal{H}_{i,j}$. Then the number of insertions of e into buckets is easily seen to be as stated because the hot pool has $\lceil \log n \rceil + 2$ columns and $\lfloor \log W \rfloor + 1$ rows, and every edge, when it moves, moves left, down, left and down, or one down and at most three right, where we view the rows as numbered bottom to top and the columns as numbered left to right (see Figure 5.3).

An edge stored in $\mathcal{H}_{i,j}$ belongs to an (i,j)-group $\mathcal{G}$. This group is stored in bucket $\mathcal{H}_{i,j}$ because the $(i,j+1)$-group $\mathcal{G}'$ containing $\mathcal{G}$ contains a visited vertex or a vertex that is visited within the next 2^i distance steps. Since the diameter of $\mathcal{G}'$ is less than 2^{i+j}, this implies that every vertex in $\mathcal{G}$ is visited within the next $\mathrm{O}\big(2^{i+j}\big)$ distance steps, which means that every edge in $\mathcal{G}$ disappears from the hot pool by this time. □

Lemma 5.5 *The number of* $(\cdot,0)$*-groups inserted into column 0 is* $\mathrm{O}(n)$.

Proof. First note that there are only $\mathrm{O}(n)$ unique $(\cdot,0)$-groups, as any two such groups are either disjoint or properly nested. Each such group is inserted into at most one bucket $\mathcal{H}_{i,0}$. To see this, observe that every

$(i,0)$-group is an $(i-1)$-component. Hence, when an $(i,0)$-group $\mathcal{G}$ leaves bucket $\mathcal{H}_{i,0}$ it is inserted into $\mathcal{D}$; $\mathcal{G}$ does not leave $\mathcal{D}$ until it disappears completely from the hot pool structure or reaches a row i' where it is a non-trivial (i',j)-group with $j>0$. □

Lemma 5.6 *Manipulating lists L, $\mathcal{L}$, $\mathcal{L}'$ and indexes $\mathcal{I}_i$, $\mathcal{I}'_i$, $\mathcal{J}$, $\mathcal{J}'$ costs $\mathrm{O}((m/B)(\log n+\log W))$ memory transfers.*

Proof. First observe that lists L, $\mathcal{L}$, and $\mathcal{L}'$ and indexes $\mathcal{I}_i$ are never sorted, only scanned. The only indexes we sort are $\mathcal{J}$, $\mathcal{J}'$, and $\mathcal{I}'_i$.

For every (i,j)-group $\mathcal{G}$ with $j>0$, its corresponding group descriptor is contained in $\mathcal{J}$ during only one iteration of a down-propagation step because either $\mathcal{G}$ contains a vertex in L—then it is split into its constituent $(i,j-1)$-groups or moved to $\mathcal{D}$, that is, its group descriptor disappears from $\mathcal{J}$—or it contains no vertex in L—then its group descriptor is inserted into $\mathcal{I}_i$, that is, again disappears from $\mathcal{J}$. Since every group is split into its constituent groups only once, every group is inserted into $\mathcal{J}'$ only once. By Property (P6) and Lemma 5.5, this implies that the total sorting and scanning cost of indexes $\mathcal{J}$ and $\mathcal{J}'$ is $\mathrm{O}(\mathrm{sort}(n))$.

The group descriptor of every group inserted in a bucket gets inserted into $\mathcal{I}'_i$ and $\mathcal{I}'_{i-1}$ at most once; again by Property (P6) and Lemma 5.5 they are $\mathrm{O}(n)$ in total. Hence, the sorting and scanning cost of indexes $\mathcal{I}'_i$ and $\mathcal{I}'_{i-1}$ is also $\mathrm{O}(\mathrm{sort}(n))$.

The scanning cost of index $\mathcal{I}_i$ is $\mathrm{O}((n/B)\log n)$. To see this, observe that every entry in $\mathcal{I}_i$ representing an (i,j)-group remains in $\mathcal{I}_i$ for at most $\mathrm{O}\big(2^{i+j}\big)$ distance steps, by Lemma 5.4. During this time, index $\mathcal{I}_i$ is inspected at most $4+\mathrm{O}\big(2^{i+j}/2^i\big)=\mathrm{O}\big(2^j\big)$ times, by Lemma 4.5. Hence, the scanning cost per $(\cdot,j)$-group is $\mathrm{O}\big(2^j/B\big)$. By Property (P6) and Lemma 5.5, only $\mathrm{O}\big(n/2^j\big)$ $(\cdot,j)$-groups are ever stored in column j. Hence, the scanning cost of index entries representing $(\cdot,j)$-groups is $\mathrm{O}(n/B)$. Summing this over all columns gives the claimed bound.

Next the scanning cost of list L: Consider a given down-propagation step from row i to row $i-1$. A vertex v stays in L one iteration longer than its adjacency list stays in $\mathcal{L}$. For the sake of this analysis, we say that v's adjacency list moves left when the (i,j)-group containing v's adjacency list is split into $(i,j-1)$-components. This is consistent with our concept of moving edges between hot-pool buckets because, if we were to store edges in buckets rather than temporarily storing them in $\mathcal{L}$ until their final location has been determined, then an (i,j)-group would be stored in $\mathcal{H}_{i,j}$, and the $(i,j-1)$-groups into which it is split would be stored in $\mathcal{H}_{i,j-1}$.

With this terminology, a vertex in L is scanned $2+k$ times, where k is

the number of left-moves performed by v's adjacency list during this down-propagation step. Indeed, each time L is scanned, the current group containing v's adjacency list is split into smaller groups, effecting a left-move of v's adjacency list, until the adjacency list moves one level down. Then the vertex is scanned one more time as part of L to detect that its adjacency list has disappeared from $\mathcal{L}$. We charge the k scans to the left-moves of v's adjacency list and the two remaining scans to the downward move of v's adjacency list. Hence, by Lemma 5.4, every vertex is scanned as part of L only $O(\log n + \log W)$ times, and the total scanning cost of list L is $O((n/B)(\log n + \log W))$.

The scanning cost of lists $\mathcal{L}$ and $\mathcal{L}'$ is now easily bounded in a similar manner. During a down-propagation step, every edge remains in $\mathcal{L}$ and $\mathcal{L}'$ for $1 + k$ iterations, where k is the number of left moves it performs. We charge the k scans to the number of left moves and the one extra scan to the downward move of the edge. Hence, the total cost of scanning lists $\mathcal{L}$ and $\mathcal{L}'$ is $O((m/B)(\log n + \log W))$ memory transfers.

Summing the costs of manipulating the different lists and indexes, we obtain the bound in the thesis. □

5.3.1 The down-propagation buffer

First we prove that our method for deciding when to move a group $\mathcal{G}$ in bucket $\mathcal{D}_i$ to a lower row is correct: precisely, that the priority stored with $\mathcal{G}$ is the minimum priority p of all vertices in $\mathcal{G}$, which implies that a vertex in $\mathcal{G}$ moves to row $i - 1$ exactly when $p < p_{i-1}$.

Lemma 5.7 *The minimum priority stored with each group in the down-propagation buffer $\mathcal{D}$ is correct at all times.*

Proof. When a group $\mathcal{G}$ moves into bucket $\mathcal{D}_i$ from a bucket $\mathcal{H}_{i,j}$, we label $\mathcal{G}$ correctly with the minimum priority of any vertex in $\mathcal{G}$ because this vertex must belong to L.

Let p be the priority assigned to $\mathcal{G}$ at this point. We claim that no vertex in $\mathcal{G}$ can *ever* have a priority less than p, which implies that p remains the minimum priority of the vertices in $\mathcal{G}$ throughout $\mathcal{G}$'s life span. Let p^* be the minimum priority in L (and thus in $\mathcal{B}_i$) at the time when p is assigned to $\mathcal{G}$. If a subsequent relaxation of an edge uv changes the priority of a vertex $v \in \mathcal{G}$, then $d(u) \geq p^*$. If $u \in \mathcal{G}$, then $d(u) \geq p$, and $d(v) = d(u) + \ell(uv) \geq p$. So consider the case that $u \notin \mathcal{G}$. Since group $\mathcal{G}$ is an $(i-1)$-component, edge uv must have category at least i, that is, length at least 2^{i-1}. Thus, $d(u) + \ell(uv) \geq p^* + 2^{i-1}$. By the choice of priorities p_{i-1} and p_i, however, we have $p - p^* < p_i - p_{i-1} \leq 2^{i-2}$, that is, $p < d(u) + \ell(uv) = d(v)$. □

The implementation of the down-propagation buffer is fairly straightforward: We maintain $\mathcal{D}$ as the concatenation of buckets $\mathcal{D}_1, \ldots, \mathcal{D}_i$. Each such bucket is a collection of groups, in no particular order. Since $\mathcal{D}_i \neq \emptyset$ only if $\mathcal{B}_i \neq \emptyset$ and we inspect rows only up to the first non-empty bucket $\mathcal{B}_i$, we are always interested in only the first non-empty bucket in $\mathcal{D}$, if any. Hence, manipulating the current bucket $\mathcal{D}_i$ always reduces to scanning an appropriate prefix of $\mathcal{D}$.

Lemma 5.8 *The cost of manipulating the down-propagation buffer is* $\mathrm{O}((m/B)\log W)$.

Proof. Every group $\mathcal{G}$ in a bucket $\mathcal{D}_i$ contains a vertex v in $\mathcal{B}_i$. The next inspection of row i moves v to row $i-1$. Hence, by Lemma 5.7, the priority of $\mathcal{G}$ is less than p_{i-1}, and group $\mathcal{G}$ moves to row $i-1$. Thus, every edge is inspected at most once as part of a bucket $\mathcal{D}_i$. Summing this scanning cost over all $\lfloor \log W \rfloor + 1$ buckets in $\mathcal{D}$ gives the bound claimed in the lemma. □

5.3.2 The column structure

Each column of the hot pool is represented as a column structure consisting of an array $\mathcal{C}_j$ equipped with a number of indexes to support group insertions and extractions. This array consists of $r = \lfloor \log W \rfloor + 1$ chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,r}$, arranged in this order (see Figure 5.4). Chunk $\mathcal{C}_{j,i}$ is associated with bucket $\mathcal{H}_{i,j}$ and stores only elements from buckets $\mathcal{H}_{i',j}$ with $i' \geq i$. Initially, all chunks have size 0 and are empty.

Every memory location in a chunk $\mathcal{C}_{j,i}$ can be in three different states: An *occupied* location stores an element of some bucket $\mathcal{H}_{i',j}$. A *clean* location does not store anything and never has since the time of the creation of $\mathcal{C}_{j,i}$. A location that is neither occupied nor clean is *unoccupied*.

We keep track of the chunks using a *chunk index*, which is an array of size r. The i'th entry stores the following information about chunk $\mathcal{C}_{j,i}$: the physical address $p_{\mathcal{C}_{j,i}}$ of $\mathcal{C}_{j,i}$ in $\mathcal{C}_j$, the size $s_{\mathcal{C}_{j,i}}$ of $\mathcal{C}_{j,i}$, the number $c_{\mathcal{C}_{j,i}}$ of clean memory locations in $\mathcal{C}_{j,i}$ (which are at the beginning of $\mathcal{C}_{j,i}$), and the number $|\mathcal{H}_{i,j}|$ of elements in bucket $\mathcal{H}_{i,j}$.

We keep track of groups stored in the chunks using a *group index*. Each index entry is a triple (i,k,p), which signifies that the corresponding group belongs to bucket $\mathcal{H}_{i,j}$, is the k'th group ever inserted into the hot pool, and is currently stored at position p in $\mathcal{C}_j$. The entries in the group index are sorted by increasing i-components and by decreasing k-components.

When inserting groups into column j in a down-propagation step, these groups are inserted into an *insertion buffer* $\mathcal{Z}_j$. At the end of each down-propagation step, we incorporate the groups in $\mathcal{Z}_j$ into $\mathcal{C}_j$.

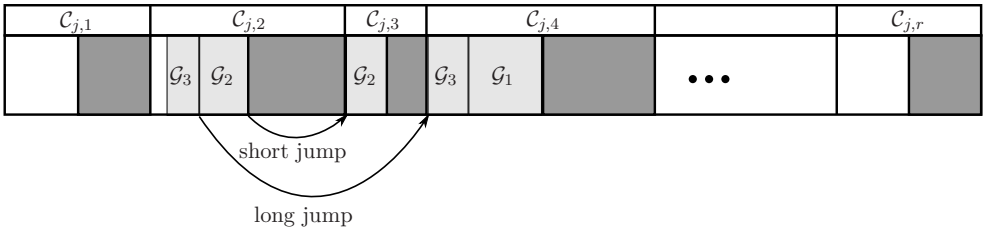

Figure 5.4: The column structure $\mathcal{C}_j$. Clean locations are blank; occupied and unoccupied locations are shadowed. Dark shadow indicates occupied locations at the time of last rebuilding. Light shadow indicates groups inserted after last rebuilding. $\mathcal{G}_1$ and $\mathcal{G}_3$ are $(4, j)$-groups, while $\mathcal{G}_2$ is a $(3, j)$-group. Groups are inserted in the following order: $\mathcal{G}_1$, $\mathcal{G}_2$, $\mathcal{G}_3$. Group $\mathcal{G}_3$ is divided in two segments; accessing them causes a long jump, because chunk $\mathcal{C}_{j,3}$ is non-empty. If $j > t$, the cost of the long jump is charged to the cost of accessing $\mathcal{G}_2$, the first group inserted into $\mathcal{C}_{j,3}$ after the last time $\mathcal{C}_{j,3}$ was rebuilt.

5.3.2.1 Operations

To insert an (i, j)-group $\mathcal{G}$ into $\mathcal{C}_j$, we append $\mathcal{G}$ to $\mathcal{Z}_j$. We increment the insertion count, store this count with $\mathcal{G}$, and return a group descriptor (a, b, i, j, k) for insertion into the row index $\mathcal{I}_i$, where k is the current insertion count and $[a, b]$ is the range of vertex identifiers in $\mathcal{G}$. We call this an InsertGroup($\mathcal{G}$) operation.

At the end of each down-propagation step from row i to row $i - 1$, we flush each non-empty insertion buffer $\mathcal{Z}_j$, inserting the groups in $\mathcal{Z}_j$ into $\mathcal{C}_j$. We call this a FlushGroups() operation. Note that buffer $\mathcal{Z}_j$ can hold only groups to be inserted into $\mathcal{H}_{i-1,j}$ or $\mathcal{H}_{i,j}$ at this point. The FlushGroups() operation scans $\mathcal{Z}_j$ twice. During the first scan, we insert the groups to be inserted into $\mathcal{H}_{i,j}$. The second scan inserts the remaining groups into $\mathcal{H}_{i-1,j}$. Consider the first scan, the second one being similar. To insert groups into $\mathcal{H}_{i,j}$, we increase $|\mathcal{H}_{i,j}|$ by the total size of these groups and scan the chunk index to find the chunk $\mathcal{C}_{j,x}$ with maximal index $x \leq i$ that has clean positions. Let h be the number of edges to be inserted into $\mathcal{H}_{i,j}$. If $h \leq c_{\mathcal{C}_{j,x}}$, we insert these edges into positions $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - h$ through $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - 1$ and then decrease $c_{\mathcal{C}_{j,x}}$ by h. If $h > c_{\mathcal{C}_{j,x}}$, we insert $c_{\mathcal{C}_{j,x}}$ elements into positions $p_{\mathcal{C}_{j,x}}$ through $p_{\mathcal{C}_{j,x}} + c_{\mathcal{C}_{j,x}} - 1$, set $c_{\mathcal{C}_{j,x}} = 0$, find the next lower chunk $\mathcal{C}_{j,x'}$ that has clean positions and repeat the whole procedure for the remaining $h - c_{\mathcal{C}_{j,x}}$ elements in the group. As a result, a group may be distributed over more than one chunk. We refer to the part of a group stored in a chunk as a *group segment*. For every group, we form a linked list of its segments by storing

with every group segment a pointer to the next segment of the same group, by increasing address.

Finally, we need to update the group index. When we insert a new group $\mathcal{G}$ with number k into $\mathcal{H}_{i,j}$, let p be the position of the first segment of $\mathcal{G}$ in $\mathcal{C}_j$. Then we add a triple (i,k,p) to a list A. Once the processing of all insertions into $\mathcal{H}_{i,j}$ is done, we merge the contents of list A into the group index.

When inserting groups of in total h elements into $\mathcal{H}_{i,j}$, it is possible that $\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} < h$, that is, there is not enough room for inserting the groups into $\mathcal{C}_j$. In this case, we temporarily grow the first chunk $\mathcal{C}_{j,1}$ and increase $c_{\mathcal{C}_{j,1}}$ accordingly to obtain $\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} = h$. Then we perform the insertion as above, leaving $\sum_{x=1}^{i} c_{\mathcal{C}_{j,x}} = 0$. This is followed by rebuilding a prefix of chunks $\mathcal{C}_{j,1},\ldots,\mathcal{C}_{j,i'}$ with $i' \geq i$, as described below.

At the beginning of each down-propagation step from row i to row $i-1$, we extract from $\mathcal{C}_j$ some of the groups in bucket $\mathcal{H}_{i,j}$. We call this an ExtractGroups$(\cdot)$ operation, which is given the descriptors of all groups to be extracted, sorted by decreasing k-values. This operation scans the list of group descriptors and the group index to translate every tuple (a,b,i,j,k) into a tuple (a,b,i,j,p), where p is the physical position of the requested group. Then we scan this pointer list, retrieve each group indexed by a tuple in this list, and add its elements to a list L. In particular, for each group, its corresponding group index entry points to its first segment. We retrieve this segment and then follow pointers between the segments to traverse the list of group segments and thereby collect all the elements in the group. Note that every ExtractGroups$(\cdot)$ operation retrieves groups from exactly one bucket $\mathcal{H}_{i,j}$, in the order in which they are stored in $\mathcal{C}_j$. So no sorting is required. Finally, we decrease $|\mathcal{H}_{i,j}|$ by the total size of the extracted groups, and we return list L.

The removal of the elements in L from $\mathcal{C}_j$ may leave a prefix of $\mathcal{C}_j$ too sparsely populated. Thus, before returning the elements in L, we check whether there exists an index i such that $\sum_{x=1}^{i} |\mathcal{H}_{x,j}| < \frac{1}{4}\sum_{x=1}^{i} s_{\mathcal{C}_{j,x}}$. If so, we rebuild some prefix $\mathcal{C}_{j,1},\ldots,\mathcal{C}_{j,i'}$ with $i' \geq i$.

5.3.2.2 Rebuilding

When a FlushGroups() or ExtractGroups$(\cdot)$ operation triggers the rebuilding of a prefix of $\mathcal{C}_j$, we first identify the set of chunks that need to be rebuilt. We find the maximal index i' such that $\sum_{x=1}^{i'} |\mathcal{H}_{x,j}| < \frac{1}{4}\sum_{x=1}^{i'} s_{\mathcal{C}_{j,x}}$ and $|\mathcal{H}_{i',j}| < \frac{1}{2} s_{\mathcal{C}_{j,i'}}$. (After rebuilding chunk $\mathcal{C}_{j,i'}$, we have $|\mathcal{H}_{i',j}| = \frac{1}{2} s_{\mathcal{C}_{j,i'}}$. Hence, the second condition implies that bucket $\mathcal{H}_{i',j}$ has lost at least one element. Without this condition, Observation 5.11 below would not hold.) If there is no such index, we set $i' = 0$. Next we find the maximal index $i > i'$ satisfying the following two conditions: (i) Bucket $\mathcal{H}_{i,j}$ is non-empty or

$s_{\mathcal{C}_{j,i}} > 0$ and (ii) $c_{\mathcal{C}_{j,x}} = 0$, for all $i' < x \leq i$. If there is no such index i, we choose $i = i'$. Now we rebuild chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$.

Note that, by the choice of index i, the groups in chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ belong to buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$ because $c_{\mathcal{C}_{j,i+1}} > 0$ and any insertion into a bucket $\mathcal{H}_{x,j}$ with $x > i$ would have completely filled $\mathcal{C}_{j,i+1}$ before inserting elements into one of $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$. We perform the following operations for $i' = 1, \ldots, i$: We scan the part of the group index storing pointers to the groups in $\mathcal{H}_{i',j}$, retrieve the corresponding groups from $\mathcal{C}_j$, and append them to a list L. Once we have done this, L stores all groups in buckets $\mathcal{H}_{1,j}, \ldots, \mathcal{H}_{i,j}$, sorted in the same order as their corresponding group index entries; and the elements of each group are stored consecutively, that is, none of these groups is segmented.

We now scan backwards over L, creating chunks $\mathcal{C}_{j,i}, \mathcal{C}_{j,i-1}, \ldots, \mathcal{C}_{j,1}$. For each chunk $\mathcal{C}_{j,x}$, we set $|\mathcal{H}_{x,j}|$ to be the number of elements in $\mathcal{H}_{x,j}$, $c_{\mathcal{C}_{j,x}} = |\mathcal{H}_{x,j}|$ and $s_{\mathcal{C}_{j,x}} = 2|\mathcal{H}_{x,j}|$. Then we allocate $s_{\mathcal{C}_{j,x}}$ memory locations to $\mathcal{C}_{j,x}$ and store the elements in $\mathcal{H}_{x,j}$ in the highest $|\mathcal{H}_{x,j}|$ memory locations of $\mathcal{C}_{j,x}$. Since groups are stored in the same order in L as their corresponding group index entries, and we place the groups in the same order into the new chunks, we can also update the pointers of the corresponding group index entries in a single scan of the corresponding prefix of the group index.

It is easy to prove the following two lemmas. The first one shows that no prefix of $\mathcal{C}_j$ is ever too sparsely populated (except immediately before rebuilding it). The second one establishes that, whenever we rebuild a prefix, we have enough insertions or deletions into or from $\mathcal{C}_j$ that can pay for the rebuilding cost.

Lemma 5.9 *For all $1 \leq i \leq r$, we have $\sum_{x=1}^{i} s_{\mathcal{C}_{j,x}} \leq 4 \sum_{x=1}^{i} |\mathcal{H}_{i,j}|$.*

Lemma 5.10 *For every chunk $\mathcal{C}_{j,x}$, let $u_{\mathcal{C}_{j,x}}$ be the number of elements inserted into or deleted from $\mathcal{C}_{j,x}$ since the last time this chunk was rebuilt. When a prefix $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ is rebuilt, we have $\sum_{x=1}^{i} u_{\mathcal{C}_{j,x}} > \frac{1}{4} \sum_{x=1}^{i} s_{\mathcal{C}_{j,x}}$.*

5.3.2.3 Analysis

It remains to analyze the costs of the different operations. In this analysis, we make use of our tall-cache assumption by assuming that, for some parameter $t \leq \log B$ to be chosen later and all $0 \leq j \leq t$, we can keep the following four blocks of the column structure $\mathcal{C}_j$ in cache: the first blocks of the chunk and group indexes, the first block of array $\mathcal{C}_j$, and the last block of the buffer array $\mathcal{Z}_j$. This requires $\mathrm{O}(B \log B) = \mathrm{O}\big(B^{1+\varepsilon}\big)$ cache space.

First let us analyze the cost of maintaining the chunk indexes. We assume that every chunk index access scans the whole index. Then the cost of scanning chunk indexes is $\mathrm{O}((n/B) \log W)$: the size of each chunk index is

$\lfloor \log W \rfloor + 1$; the chunk index is scanned at most once per FlushGroups(), ExtractGroups($\cdot$), or rebuilding operation, of which there are $O(n)$, by Property (P6) and Lemma 5.5. Chunk index accesses for columns $j > t$ may incur another $O\big(n/2^t\big)$ memory transfers because, when accessing a chunk of a column $j > t$, we may have to load the first block of the chunk index into cache. By Property (P6), this happens at most $O\big(n/2^t\big)$ times. Before bounding the cost of accessing group indexes, we need the following observation.

Observation 5.11 *Between every two accesses to a chunk $\mathcal{C}_{j,i}$ in array $\mathcal{C}_j$, there is at least one access to a bucket $\mathcal{H}_{i',j}$ with $i' \geq i$.*

When accessing a group index, it is to access the entries of (i,j)-groups in one bucket $\mathcal{H}_{i,j}$ or to access the entries of all groups in chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ during a rebuild operation. In both cases, we bound the cost of accessing the group index by scanning the prefix of the group index up to the last group in $\mathcal{C}_{j,i}$. By Observation 5.11 and Lemmas 4.5 and 5.4, every group index entry in the j'th column is scanned $O\big(2^j\big)$ times. By Property (P6) and Lemma 5.5, there are $O\big(n/2^j\big)$ groups in column j. Hence, the scanning cost of group index entries in a single column is $O(n/B)$. Summing over all columns, the scanning cost is $O((n/B)\log n)$. Every access to a group index of a column $j > t$ may cost one extra memory transfer. Since there are $O\big(n/2^t\big)$ accesses to columns $j > t$, this amounts to an extra cost of $O\big(n/2^t\big)$ memory transfers.

The accesses to insertion buffers cost $O\big(n/2^t + (m/B)(\log n + \log W)\big)$: The first term is the cost of loading the last block of buffer $\mathcal{Z}_j$ if $j > t$. The second term is the scanning cost because every edge is scanned $O(1)$ times per insertion into a column, and an edge is inserted into $O(\log n + \log W)$ buckets, by Lemma 5.4.

The final part of the analysis concerns the cost of accessing arrays $\mathcal{C}_j$. We distinguish the two cases $j \leq t$ and $j > t$. For $j \leq t$, we can bound the cost of every access to chunk $\mathcal{C}_{j,i}$, or chunks $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$ if rebuilding is required, by the cost of a *staggered scan* of these chunks, defined as follows: We first scan chunk $\mathcal{C}_{j,1}$, then chunks $\mathcal{C}_{j,1}, \mathcal{C}_{j,2}$ (scanning $\mathcal{C}_{j,1}$ again!), then chunks $\mathcal{C}_{j,1}, \mathcal{C}_{j,2}, \mathcal{C}_{j,3}$, and so on until we finally scan $\mathcal{C}_{j,1}, \ldots, \mathcal{C}_{j,i}$. The following two lemmas analyze the costs of accessing the two different kinds of columns.

Lemma 5.12 *The total cost of manipulating arrays $\mathcal{C}_j$, $0 \leq j \leq t$, is $O\big((m2^t/B)\log W\big)$ memory transfers.*

Proof. By Lemma 5.9, every staggered scan reads at most a constant factor more memory locations than there are elements in the corresponding buckets. Thus, for this analysis we can think of every chunk $\mathcal{C}_{j,i}$ as being the same as bucket $\mathcal{H}_{i,j}$. We prove that every edge e is accessed at most $O\big(2^t \log W\big)$ times. Summing over all edges proves the lemma.

In this proof, we consider edge e to be in row i if it is stored in a bucket $\mathcal{H}_{i,j}$ with $j \leq t$. In particular, we do not consider it to be in row i if it is contained in a bucket $\mathcal{H}_{i,j}$ with $j > t$. This captures that we are interested only in the cost edge e incurs in the first t columns.

We now consider priorities $q_0 \geq q_1 \geq \cdots \geq q_r$, which are defined as follows: q_0 is the minimum priority of any vertex retrieved by the BATCHED-DELETEMIN() operation that also retrieves edge e. For $i > 0$, we define $q_i = q_{i-1}$ if edge e never enters row i (in the above sense); otherwise, let q_i be the minimum priority of any vertex retrieved by the BATCHEDDELETEMIN() operation that puts e into row i.

Now let $a_{i,i'}$ be the number of accesses to a row $i' \geq i$ while edge e is in row i, let $a^*_{i,i'}$ be the number of accesses to rows i' and above while edge e is in row i, and let $A_{i'}$ be the number of accesses to rows i' and above while edge e is in row i' or below. The total number of accesses to edge e in the first t rows is at most

$$\begin{aligned}\sum_{i=1}^{r}\sum_{i'\geq i}(i'-i+1)a_{i,i'} &= \sum_{i=1}^{r}\sum_{i'\geq i} a^*_{i,i'} \\ &= \sum_{i'=1}^{r}\sum_{i\leq i'} a^*_{i,i'} \\ &= \sum_{i'=1}^{r} A_{i'}.\end{aligned}$$

By Lemma 4.5, we have $A_{i'} \leq 4 + 12(q_0 - q_{i'})/2^{i'}$. By Lemma 5.4, we have $q_0 - q_{i'} = \mathrm{O}\Big(2^{i'+t}\Big)$. Thus, $A_{i'} = \mathrm{O}\big(1 + 2^t\big) = \mathrm{O}\big(2^t\big)$. Inserting into the above summation, we obtain that edge e is scanned at most $\mathrm{O}\big(2^t \log W\big)$ times as part of columns 1 through t. This implies the lemma. □

Lemma 5.13 *The total cost of manipulating arrays $\mathcal{C}_j$, $j > t$, is $\mathrm{O}\big(n/2^t + (m/B)(\log n + \log W)\big)$.*

Proof. The first term in the bound accounts for the cost of having to access the first segment of each group; by Property (P6), there are at most $\mathrm{O}\big(n/2^t\big)$ groups in columns $j > t$.

The second term accounts for the cost of scanning edges while inserting or extracting them; the bound follows because, by Lemma 5.4, every edge is involved in at most $\mathrm{O}(\log n + \log W)$ insertions or extractions. Note that, by Lemma 5.10, we can charge the cost of scanning edges during rebuilding to the inserted or deleted edges that triggered the rebuilding.

What remains is to bound the number of memory transfers incurred by accessing groups that have more than one segment, that is, are distributed over multiple chunks. We call following the pointer from one group segment to the next a *jump*. A jump is *short* if it is from a chunk $\mathcal{C}_{j,x}$ to a chunk $\mathcal{C}_{j,z}$ with $z > x$ and $s_{\mathcal{C}_{j,y}} = 0$ for all $x < y < z$; otherwise, it is *long* (see Figure 5.4).

The cost of every long jump is at most one memory transfer. Consider a group $\mathcal{G}$. For a long jump from a chunk $\mathcal{C}_{j,x}$ to a chunk $\mathcal{C}_{j,z}$, there must be a chunk $\mathcal{C}_{j,y}$, $x < y < z$, with $s_{\mathcal{C}_{j,y}} > 0$. We choose y maximally so. Since the elements of $\mathcal{G}$ were not inserted into $\mathcal{C}_{j,y}$ at the time of $\mathcal{G}$'s insertion, we must have had $c_{\mathcal{C}_{j,y}} = 0$ at that time. However, since $s_{\mathcal{C}_{j,y}} > 0$, we must have had $c_{\mathcal{C}_{j,y}} > 0$ immediately after $\mathcal{C}_{j,y}$ was rebuilt. Hence, there must have been an insertion into $\mathcal{C}_{j,y}$ between the rebuilding of $\mathcal{C}_{j,y}$ and the insertion of $\mathcal{G}$. We charge the first group $\mathcal{G}'$ inserted into $\mathcal{C}_{j,y}$ after the last time $\mathcal{C}_{j,y}$ was rebuilt for the cost of the long jump. Observe that $\mathcal{G}'$ can be charged only once: It can obviously be charged only until the next time $\mathcal{C}_{j,y}$ is rebuilt; and before rebuilding $\mathcal{C}_{j,y}$ again, there can be only one group $\mathcal{G}$ that jumps over chunk $\mathcal{C}_{j,y}$ and charges $\mathcal{G}'$.

The cost of a short jump is bounded by the cost of scanning the whole chunk $\mathcal{C}_{j,x}$ from which it originates. Since there can be at most one jump out of $\mathcal{C}_{j,x}$ before $\mathcal{C}_{j,x}$ is rebuilt again, we can charge this cost to the update operations that cause the next rebuilding of $\mathcal{C}_{j,x}$, increasing the scanning cost of $\mathcal{C}_{j,x}$ incurred while rebuilding chunks by a constant factor. If $\mathcal{C}_{j,x}$ is never rebuilt again after this short jump, we bound the cost of the short jump by one memory transfer and charge any of the groups in $\mathcal{C}_{j,x}$ at the time it was rebuilt last for this jump. This charges every group at most once. □

By summing the bounds in Lemmas 5.6, 5.8, 5.12, and 5.13 and choosing $t = \left\lceil \log \sqrt{nB/(m \log W)} \right\rceil \leq \lceil \log B \rceil$, we obtain the following corollary, which together with Lemmas 4.7, 5.3 and the cost of priority queue operations described in Section 4.2.2.1 implies Theorem 5.1.

Corollary 5.14 *The total cost of hot pool manipulations is* $\mathrm{O}\Big(\sqrt{(nm \log W)/B} + (m/B) \log n\Big)$.

Chapter 6

CO_2: a platform for experimental analysis of cache-oblivious algorithms

The content of this chapter is joint work with Fabrizio d'Amore and Enrico Puddu.

One of the most prominent open questions in the area of cache-oblivious algorithms is their practicability. As we discussed in Chapter 2, in spite of their sophistication which translates into larger CPU costs, cache-oblivious algorithms have the potential for outperforming internal memory algorithms on medium-sized input instances, and for remaining competitive with external memory algorithms on large instances; this is confirmed by a few, significant experimental works (see Section 2.3.3). It is thus desirable a thorough experimental analysis of the performance of complex cache-oblivious algorithms, such as those discussed in this thesis.

Despite cache-oblivious algorithms are algorithms for the RAM model, implementing cache-oblivious algorithms and data structures is not an easy task, because they use peculiar techniques that are not (easily) supported by general-purpose programming languages and environments such as C, C++, or Java. For example, consider the following issues, that arise when implementing a cache-oblivious algorithm:

- **Memory management**. General-purpose memory managers, such as common implementations of the `malloc(·)` function, keep linked lists of the memory areas containing deallocated memory elements. When a new memory element is allocated, the memory manager uses a linked

list to find a large enough free memory area. If the element pointed by the linked list is not in cache, the allocation provokes one cache miss.

- **Usage of pointers to perform elaborated tasks**. In C, C++, and even more in Java, only simple tasks can be performed without using pointers (or references). Allocating an array whose size is not known at compile time, or creating a record-like structure which "contains" an object whose size is not known at compile time, is normally achieved through pointers. Pointers are problematic for cache-oblivious algorithms and data structures, because following a pointer may cause a cache miss. As we discussed in the previous chapters, cache-oblivious algorithms avoid following pointers by building and using complex memory layouts, that are often described by recursive definitions. Unfortunately, these layouts are difficult to obtain using standard C++ only: even if it would be technically possible to allocate a large array of memory words and manage it in a completely custom way, this approach is certainly inconvenient, and stops the programmer from taking advantage of the abstractions of high-level programming languages, such as arrays, structures and objects.

To find a solution to the mentioned issues, and since a library of cache-oblivious algorithms and data structures did not exist, we decided to create CO_2, a C++ platform for engineering cache-oblivious algorithms and data structures. The purpose of our platform is twofold: to facilitate the implementation of algorithms, as generic as possible and usable in real-world applications; and to support the engineering through experimental analysis of such algorithms. We thus defined the following design goals for CO_2:

- The programmer should be able to easily use **abstractions** similar to those offered by high-level programming languages, such as arrays, structures, etc., in order to build and work with complex memory layouts.

- While the programmer should be able to easily ask for and release **memory**, the cost (I/O and CPU cost) of these operations should be predictable and should not alter, to the maximum extent possible, the asymptotic cost of the implemented algorithm.

- **Modularity** is important: implemented algorithms should be as generic as possible, so that they can be used within other algorithms, or in real-world applicative scenarios. Modularity allows to test and compare implementation alternatives; in this light, even the parts of CO_2 related to memory management should be modular.

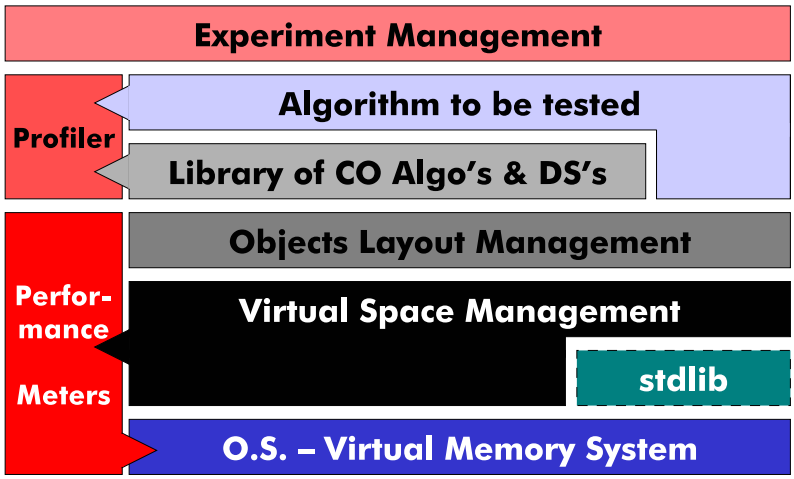

Figure 6.1: Architecture of the platform.

- Experiments should be supported by **versatile performance meters**. It should be possible to measure the actual performance of a cache-oblivious algorithm on a real-world machine. Also the simulation of the execution of an algorithm on a hierarchical memory would be useful, allowing to easily identify critical portions of the algorithm (for example, with respect to page fault rate); and to study the influence of factors such as non-optimal paging strategies, limited associativity, non-tall caches etc. Notice that some experimental work on cache-oblivious algorithms (for example the study of cache-oblivious B-trees in [16]) was conducted on simulated hardware. Furthermore, it should be possible to break down the CPU and I/O cost of an algorithm into the costs of its parts (procedures, sub-algorithms); in particular, the platform should allow to break down both the real and the simulated costs.

- It should be possible to **automate** experiments to a large extent.

In Fig. 6.1 we show a layered view of the architecture of CO_2. A detailed description of the layers is given in the following sections.

6.1 The operating system's virtual memory

Although it is not strictly a layer of CO_2, we describe the virtual memory system on top of which we build our platform. The abstraction offered by hardware and operating system is closely related to the Random Access Machine (or, equivalently, the Ideal Cache) model: each process can see and use a very large virtual memory space (also known as *address space*). The size of

Figure 6.2: The virtual memory space of a process in UNIX.

the virtual memory space is 2^n, where n is the number of bits of a memory address (usually 32 or 64). Physically, the memory space is divided into *pages*; those pages that are actually used are stored in the main memory or the disk. An important divergence from the RAM model is that a process must explicitly tell the operating system which memory pages it is actually using, so that the operating system can avoid to map the entire memory space to disk or to memory.

We consider, for example, the UNIX class of operating systems. Not the whole virtual memory space is available to a process. In Fig 6.2 we sketch the address space of a process. At the beginning, some space is occupied by the executable code of the process (*text*), and by variables and data related to the process and used by the operating system (*init*, *bss*, *data*). At the end of the address space some space is reserved and used by the kernel. The remaining portion of the address space is available to the process. In particular, the portion of space immediately preceding kernel space is devoted to the activation stack, and grows backwards. In order to use a portion of the remaining space, the process must reserve it. Two system calls can be used:

- `brk(void* break)`: reserves the whole address space up to location `break`. Location `break`, called the *system break*, can be dynamically modified by the process.
- `mmap(...)`: reserves an arbitrary memory page in the address space. The `mmap` system call is very flexible; it allows to map a set of memory pages to a portion of any file in the filesystem, and to share memory pages between processes.

6.2 Virtual space management

High-level programming languages provide convenient mechanisms to allocate memory: the program, instead of requesting certain portions of the address

space to the operating system, can ask for memory areas of a certain size. In C and C++ this is achieved through the use of the `malloc(·)` function, either directly or indirectly through the `new` operator. As we said earlier, no guarantee on the I/O cost of a `malloc(·)` call is given; each request may produce a page fault because the function may return the address of a memory page that is not in cache.

In the implementation of a cache-oblivious algorithm this issue must be considered; several approaches are possible:

- Memory can be directly requested to the operating system. This approach grants a complete control over memory allocation, and faithfully adheres to the RAM/Ideal Cache model. But it has several drawbacks: it is inconvenient; it is incomplete, in the sense that it does not make it clear how the address space should be organized by the algorithm; and it is not modular, because it is not clear how several cache-oblivious algorithms that directly manage memory can be combined.

- The algorithm can use the ordinary `malloc(·)` to allocate memory. The influence of random accesses due to memory allocations should be experimentally analyzed. In order to reduce the number of potential random accesses, memory allocations that are related one to another can be aggregated into a single memory allocation. If we take this approach to its extreme, once an algorithm knows the size of its input instance it can preallocate a large enough memory area and then manage it directly, incurring at most one random access for the preallocation. Preallocating memory presents two of the drawbacks of direct memory management, i.e. it does not make it clear how preallocated memory should be organized internally, and it is rather inconvenient. Moreover, it is often difficult to obtain a good upper bound on the total memory used by an algorithm, and one may end up in oversizing the allocated memory too much.

- The ordinary `malloc(·)` function can be replaced with a custom, locality-friendly memory allocator. In order to choose a memory allocator, one must trade off space with I/O cost, since both these figures may be altered, even asymptotically, from the ones obtained with theoretical analysis. We will show examples of custom memory allocators in the following subsections. Even if it is undesirable that the asymptotic I/O cost of an algorithm is altered by the memory allocator it uses, observe that we face an analogous situation when we deal with internal memory algorithms and data structures. In the analysis of such algorithms it is commonly assumed that a memory allocation can be performed in time

O(1); but memory allocators in general do not guarantee this cost (see [43] for an overview on the problem of managing memory).

CO_2 adopts the latter approach, without excluding the possibility to use the ordinary `malloc(·)`. The *virtual space management* layer is in charge of satisfying requests of memory, with a policy decided by the application. It is composed of several memory managers, where a *memory manager* is an object that provides the following functions:

- `long Allocate(size_t size)`: allocates a *memory area* of size `size`, and returns an identifier of the created area;
- `void* GetPointer(long id)`: returns a pointer to the memory area identified by `id`;
- `void Resize(long id, size_t newSize)`: resizes the memory area identified by `id` from its old size `oldSize` to `newSize`, preserving the first $\min\{$`oldSize`, `newSize`$\}$ memory elements;
- `void Deallocate(long id)`: learns that the memory area identified by `id` is no longer used by the application.

To be more precise, two versions of `Allocate(·)` are provided: `AllocateStatic(·)` and `AllocateDynamic(·)`. The former should be used to create a memory area that will not be resized; otherwise, the latter should be called.

Each memory manager is characterized by a memory source (which may be the operating system, the ordinary `malloc(·)`, or another memory manager), and by a policy for managing the memory taken from the memory source. We discuss the policies of the memory managers currently implemented in our platform in the following sections.

From an architectural point of view, a memory manager is an object; an abstract class `MemManager` (with the role of an interface) defines the above mentioned functions; for each memory policy we define a subclass of `MemManager`. Thus a cache-oblivious algorithm for CO_2 may receive a memory manager as a parameter: in this way experimenting with memory management is as simple as passing different objects to a function. Furthermore, it is possible to instance and use more than one memory manager at a time. The reason why this can be useful will be clarified in next sections.

Before describing the memory managers, we introduce some common notation. We call *reserved memory* the memory that a memory manager obtains from its memory source. If a is a memory area (i.e. a portion of memory that the memory manager granted to the application, as the result of an

`Allocate(·)` or a `Resize(·)` invocation), we denote by $s(a)$ its *size*, that is, the size requested with the last `Allocate(·)` or `Resize(·)` invocation concerning a; and by $n(a)$ its *total length*, i.e. the length of the portion of reserved memory that is currently devoted to a. Clearly $n(a) \geq s(a)$ holds.

All of our memory managers managers deal with memory areas of dynamic size with the standard rebuilding technique, that we describe in the following. We fix two parameters $\delta_{\min} < 1$ and $\delta_{\max} > 1$. When a memory area a of size $s(a)$ is allocated with an `AllocateStatic(·)` (resp. `AllocateDynamic(·)`), a portion of memory of length $n(a)$ is assigned to a from the reserved memory, with $n(a) = s(a)$ (resp. $n(a) = s(a) \cdot \delta_{\max}$). When a memory area a of size $s(a)$ and length $n(a)$ is resized to size s', if $s' \notin [\delta_{\min} n(a), n(a)]$ then a is *rebuilt*: a new memory area of length $n'(a) = s'(a) \cdot \delta_{\max}$ from reserved memory is assigned to a, and its first $\min\{s(a), s'(a)\}$ memory elements are copied from the old memory area containing a. Then we set $n(a) \leftarrow n'(a)$ and $s(a) \leftarrow s'(a)$.

It is easy to prove that the cost of scanning a memory area in order to rebuild it can be bounded by the cost of accessing the elements of the rebuilt memory area if memory usage is fair, according to the following definitions.

Definition 6.1 (Fair memory request) *An allocation of a memory area a of size $s(a)$ is said* fair *if $\Omega(s(a))$ memory elements of a are read or written before the next allocation or resizing of a. Resizing a memory area a from size s to size s' is said to be* fair *if $\Omega(|s' - s|)$ memory elements of a are read or written before a is deallocated or resized again.*

Definition 6.2 (Fair memory usage) *An algorithm A* uses memory fairly *if every memory allocation and resizing performed by A is fair.*

We remark that most cache-oblivious algorithms (in particular, all of the algorithms described in this thesis) use memory fairly.

6.2.1 `malloc`-based memory manager

The malloc-based memory manager simply uses the ordinary `malloc(·)` function to get the memory it needs. As we discussed earlier, ordinary `malloc(·)` implementations are not designed for cache-efficiency, and thus they do not provide guarantees on the I/O cost of a memory allocation; however they are carefully engineered to be highly efficient, in practice, for internal memory operations. Notice that several free, commercial and academic `malloc(·)` implementations are available; some of them are optimized for locality (see, for example, [37]).

Upon the allocation of a memory area a, a portion of memory of length $n(a)$ is asked to `malloc(·)`. `Deallocate(·)` invocations translate to calls to `free(·)`. Finally, rebuilding operations involve a `malloc(·)` call to get the new

area of length $n'(a)$, and a call to `free(·)` to deallocate the old area of length $n(a)$.

6.2.2 Non-deallocating memory manager

The non-deallocating memory manager is the simplest conceivable memory manager. It asks for memory to the operating system (or, possibly, to another memory manager): it grows a unique array of reserved memory, that extends from some memory location r_b to some location r_e. Upon each allocation of a memory area a, it extends reserved memory with additional $n(a)$ elements, from location $r_e + 1$ to $r_e + n(a)$; then it sets $r_e \leftarrow r_e + n(a)$. `Deallocate(·)` does nothing, i.e., reserved memory is never reused. Since memory is never reused we take $\delta_{\min} = 0$, i.e. no rebuilding occurs while shrinking a memory area.

The non-deallocating memory manager guarantees that, under fair memory usage, the I/O cost of managing memory can be charged to the cost of accessing it; and thus the asymptotic I/O cost of a cache-oblivious algorithm is not altered by the manager itself. To see this, it is sufficient to assume that the last memory block of the reserved area (i.e. the block containing element with address r_e) is always kept in cache. When we allocate a new memory area a the amortized cost of loading the new block at the end of the reserved memory in cache is bounded by the cost $\mathrm{O}(n(a)/B)$ of scanning the allocated memory area; here B denotes, as usual, the size of a block. The same result holds for resizing a memory area of length $n(a)$ to $s'(a) > n(a)$: the cost of enlarging the reserved area and rebuilding it can be upper bounded by the scanning cost $\mathrm{O}(n'(a)/B) = \mathrm{O}((s'(a) - s(a))/B)$ (because $s'(a) > n(a)$ implies that $s'(a) - s(a) = \Omega(n'(a))$).

The obvious disadvantage of the non-deallocating memory manager is that it does not provide a good guarantee, in general, on the size of reserved memory: if an algorithm continuously allocates and deallocates memory areas, then reserved memory continuously increases in size. Under some circumstances, however, the non-deallocating memory manager offers a good guarantee even on memory utilization. For example, it can be used to simulate the preallocation of a bunch of arrays when it is difficult to compute a tight bound on their size. Consider an algorithm A that uses a set of arrays $\{a_1, \ldots, a_k\}$; consider a run of A, and let $\overline{s}_i$ be the maximum size of array a_i at any time during the run of A. In order to use preallocation, array a_i should be given a size of at least $\overline{s}_i$ at the beginning, for a total memory usage of at least $\Omega\left(\sum_{i=1}^{k} \overline{s}_i\right)$. Alternatively, we can allocate a_i, $i \in \{1, \ldots, n\}$, with initial size 0 on the non-deallocating manager, and increase its size only when we need to. In this way we end up using, for every array a_i, not more than $\sum_{j=0}^{\infty} \overline{s}_i \delta_{\max}/\delta_{\max}^{j} = \mathrm{O}(\overline{s}_i)$ memory space. Thus the non-deallocating memory manager allows us to run

A with space $\mathrm{O}\left(\sum_{i=1}^{k} \overline{s}_i\right)$ without worsening its asymptotic I/O cost, if A's memory usage is fair.

6.2.3 Compacting memory managers

In order to reuse deallocated memory in a cache-friendly manner, we can resort to a *compacting memory manager*. As the non-deallocating memory manager, the reserved memory of a compacting memory manager is a contiguous array extending from a location r_b to a location r_e. Allocating, resizing and deallocating memory areas work as in the non-deallocating memory manager (but with $\delta_{\min} > 0$). Additionally, the compacting memory manager enforces that the total size of reserved memory is proportional to the total size of allocated memory. If after a `Deallocate(·)` or a `Resize(·)` this condition is violated, then the entire structure is *compacted*: *active* (i.e. actively used) memory areas are moved to the beginning of the reserved space, and the reserved space is shrunk. It is easy to see that, under fair memory usage, the I/O cost of scanning the active memory areas during compacting operations can be bounded by the cost of accessing memory elements.

Since memory areas are moved within the reserved space, there is a need of an index $\mathcal{I}$ to find out, given the identifier of a memory area a, where a is actually located. In order to guarantee that both the time and the I/O asymptotic costs of accessing a memory area do not worsen due to index accesses, keys of $\mathcal{I}$ are memory address; in other words, the identifier of a memory area a is the memory address of a location where the actual address of a is stored.

The presence of an index creates two important problems: $\mathcal{I}$ must be updated when compacting memory; and elements of $\mathcal{I}$ should be recycled when they are no longer used. Recycling index elements poses the same problems of recycling memory elements, i.e. an access to a recycled index (due to a new memory allocation) may cause a random I/O; thus we decide not to recycle elements of $\mathcal{I}$. As for the cost of updating $\mathcal{I}$, we consider three techniques for compacting memory:

Backward pointers. We store with each memory element a backward pointer to its index element. In order to compact memory we scan the reserved memory; for every active memory area, we move it and we update its index by following the backward pointer. This approach costs, in the worst case, 1 random I/O per active memory element, to follow the backward pointer.

Forward pointers. In order to compact memory, we scan $\mathcal{I}$ and, for every active index element, we move the pointed memory area and update the index

element itself. Also this approach costs 1 random I/O per active element, in order to follow the forward pointer. There is an additional cost of $O(u/B)$ memory accesses per compacting operation that cannot be amortized, where u is the number of no longer used index elements: in fact unused index elements must be read through in every index scan.

Scanning and sorting. It is a variation of the backward pointers technique. Here, instead of updating backward pointers immediately, we build an array $\mathcal{J}$ of pairs of the form (f_i, b_i), where f_i is the new address of a moved memory area a_i, and b_i is the address of its index element. We sort $\mathcal{J}$ by second component, and perform a parallel scan of $\mathcal{I}$ and $\mathcal{J}$ to update $\mathcal{I}$. Here the non-amortizable cost of compacting is $O(\text{sort}(t) + u/B)$, where t is the number of active memory areas, and u is the number of unused index elements.

6.2.4 Stack-like memory manager

Allocating memory on the activation stack, whenever possible, is a simple technique to avoid to incur random I/O's that cannot be amortized, because it is possible to assume that the block containing the stack top element is always kept in cache; under fair memory usage, the I/O cost of allocating a memory area can be charged to the cost of accessing its elements. Allocating memory on the stack is handy to store local variables and function parameters.

Unfortunately, in ISO C++ it is not possible to allocate on the system stack a memory element whose size is not known at compile time. Our stack-like memory manager implements a stack. It gets the memory it needs from another memory manager, and provides, along with the functions inherited from `MemManager`, two additional functions, namely `Enter()` and `Leave()`. The former is used to create a new *stack frame* on the top of the stack, where newly created memory areas are stored; the latter pops the top frame out from the stack. `Deallocate(·)` does nothing, because stack elements are actually deleted by `Leave()`. `Resize(·)` works as usual, as long as the resized memory element resides in the top stack frame. Otherwise, if the resized element does not reside in the top stack frame and it needs to be rebuilt, then it is temporarily rebuilt on the top stack frame; it is moved back towards its proper frame during subsequent `Leave()` calls, causing additional costs that, in general, cannot be amortized.

6.3 Objects layout management

The *objects layout management* layer allows to easily build complex memory structures at runtime. To this aim, it provides the abstractions of enhanced class and enhanced object. *Enhanced classes* are data types whose instances,

```
stud[4]->age = 21;
s = stud[4];
printf("The student is %d years old", s->age);
```

Figure 6.3: Accessing enhanced objects through redefined C++ operators

the *enhanced objects*, are more flexible than ordinary C++ objects. In particular, the size and (partially) the structure of an enhanced object is not computed statically at compile time, but is determined when the enhanced object is created, and possibly altered afterwards.

We define two kinds of enhanced classes, namely *static* and *dynamic* classes. The size of a static object cannot change once the object is created (but it can be determined at runtime when the object is created); while the size of a dynamic object may vary with time.

We introduce three abstractions to construct composite data types:

- **Enhanced structure.** An *enhanced structure* is an enhanced class composed of a number of named fields, known at compile time. Each field has a type, which may be an ordinary C++ type or an enhanced static class; in the latter case its size will be determined at runtime, and will contribute in computing the total size of the structure.
- **Enhanced array.** An *enhanced array* is an enhanced class providing arrays of enhanced static objects, whose type is known at compile time. The number and the size of such objects is determined at runtime, with the constraint that all the array elements have the same size. An enhanced array can be either static or dynamic.
- **Structarray.** A *structarray* is an enhanced class providing arrays of enhanced static objects, all of which have the same type, but whose size may differ from element to element.

For each enhanced class, an associated *enhanced reference* class allows to access enhanced objects using the usual `operator->` or `operator[]` C++ operators. Fig. 6.3 exemplifies the usage of these operators: in the figure, `s` is an enhanced reference to an object of an enhanced structure `_Student`[1], and `stud` is an enhanced reference to an enhanced array of `_Student`. We remark that the assignment `s = stud[4]` is a reference assignment, not an object assignment.

In order to build enhanced objects, we introduce factory objects. We associate a *factory class* to each enhanced class. Factory objects are responsible

[1]Conventionally, names of enhanced classes begin with an underscore symbol.

```
_Student::Factory sf;
_Array<_Student>::Factory asf(sf, n);
_Professor::Factory pf;
_Array<_Professor>::Factory apf(pf, m);
_University::Factory uf(asf, apf);

Ref<_University> univ = uf.Build(memoryManager);

univ->stud[0]->age = 21;
```

Figure 6.4: Creating an enhanced class using factories.

for organizing the memory layout necessary to construct enhanced objects at runtime.

Let c be an enhanced class, and f_c the associated factory class. In order to create an enhanced object o of class c, a factory object f_o of class f_c is instantiated. f_o knows everything that is necessary to create o, including the factory objects of enhanced objects to be contained in o. Thus f_o is able to (i) determine the size of o, and (ii) create o, if enough memory is given to f_o (either directly or through a memory manager passed to f_o). In Fig. 6.4 we exemplify the creation of an enhanced object of structure `_University` which contains an enhanced static array of `n` students and of `m` professors; `memoryManager` is an arbitrary memory manager.

6.4 The library of cache oblivious algorithms and data structures

CO_2 contains a library of several cache oblivious algorithms and data structures, to provide users with a collection of tools to speed-up implementation. When we began to write CO_2, our primal goal was to implement the improved cache-oblivious single-source shortest paths algorithm described in Chapter 5. Thus, it contains most of the algorithms presented Chapter 3, including the median-finding algorithm of [39] (Section 3.2), the priority queue by Arge *et al.* [11] and the batched priority queue by Meyer and Zeh [48] (Section 3.4.1 and 4.2.2.1), the time-forward processing technique [10, 31] (Section 3.4.1.1), the maximal independent set algorithm by Zeh [58] (Section 3.5.1), the list ranking algorithm by Chiang *et al.* [31] (Section 3.5.1), the Euler tour technique (Section 3.5.2), the minimum spanning tree algorithm by Arge *et al.* [11] (Section 3.6), and obviously the single-source shortest paths algorithm by Allulli *et al.* (Chapter 5). Additionally, common operations performed by

cache-oblivious algorithms, such as scanning several arrays in parallel, merging arrays etc., are supported by apposite classes.

6.5 Performance meters and the profiler

In order to analyze both real and simulated performance of algorithms, CO_2 features a modular system of *meters*. Each meter measures a particular cost; it can be started, stopped, and read, reporting the cumulative value of the measured cost.

Measured costs include physical and simulated costs. Examples of physical costs are the CPU time (broken down into user and kernel time), the I/O waiting time, the total time, and the number of page faults performed by the operating system. Examples of simulated costs are the number of page faults incurred in each level of a specified memory hierarchy; in particular, it is possible to specify several parameters of each level of the hierarchy, such as its total size, block size, paging strategy, and associativity. In order to perform simulations, there are "hooks" in the layout management layer used to track memory accesses, performed whenever the algorithm uses the `operator->` or the `operator[]` operator. Since tracking memory accesses is rather costly, it is possible to enable or disable tracking at compile time by defining a macro; when tracking is disabled, access operators are expanded inline and cause a small overhead.

In order to analyze the contribution of each function of the algorithm to the measured costs CO_2 provides a profiler. The *profiler* is fed with a set of meters, and is notified when each function is entered and left (explicit `Enter(·)` and `Leave(·)` calls to the profiler must be inserted in each function; these calls can be excluded at compile time using macros). When enabled, the profiler creates and visits an activation tree T, whose nodes are labeled with names of functions, as follows: upon initialization, the profiler builds up and visits the root r of T; r is labeled with `main`. Now assume that the profiler is visiting a node n of T: as soon as the activation of a function with name f is notified to it, if there exists a child n_f of n labeled with f, then the profiler visits n_f; otherwise, it creates a new child n_f of n, labels it with f and visits n_f.

The profiler stores a set of measures, one for each meter, with every tree node. When a node n is created, all its measures are set to 0. Whenever the profiler begins to visit a node n it reads every meter, obtaining a vector $(m_1, \ldots, m_k)$ of measures; as soon as the visit to n ends, the profiler reads the meters again, computes the difference with $(m_1, \ldots, m_k)$, and adds the difference to the values stored with n.

6.6 Status

All the core functionality of CO_2 has been implemented and works correctly. In particular:

- All of the memory managers described in Section 6.3 have been implemented. The `malloc`-based manager and the stack-like manager are platform agnostic, while the other memory managers have been implemented on UNIX systems, and tested in Linux. In the current implementation of compacting memory managers it is possible to allocate a limited number of memory areas, where the limit can be freely set.
- The layout management layer has been implemented.
- The library includes the single-source shortest paths algorithm of Chapter 5, as well as all of the algorithms and data structures cited in Section 6.4, on which our shortest paths algorithm is based. However, we have not implemented an optimal cache-oblivious sorting algorithm yet: currently sorting is performed through MERGESORT. Our plan is to study whether the implementation of LAZY FUNNELSORT by Brodal *et al.* [28] can be easily ported to our platform, or a novel implementation is necessary.
- Performance analysis has been carried on, until now, using simple performance meters; we are able to measure physical performance, as well as simulated performance on a two-levels fully associative cache managed by LRU. A prototype of the meters-profiler modular subsystem described in Section 6.5 has been implemented, but not yet integrated with CO_2.

The most urgent action in order to be able to perform significant experiments is to improve the space requirement of the implemented algorithms: currently it is excessive, and our implemented algorithms (in particular, the minimum spanning forest algorithm) saturate main memory on much smaller input instances than, for examples, those considered in [8]. In order to use less memory we should improve (i) the memory space taken by *enhanced references*, i.e. references to enhanced objects, and (ii) memory space occupied by some of our enhanced structures. Unfortunately accomplishing some of these actions, in particular the latter, impacts the generality of CO_2 and of some algorithms of CO_2's library.

Bibliography

[1] Abello, J., Buchsbaum, A. L., and Westbrook, J. A functional approach to external graph algorithms. *Algorithmica 32*, 3 (2002), 437–458.

[2] Agarwal, P. K., Arge, L., Danner, A., and Holland-Minkley, B. Cache-oblivious data structures for orthogonal range searching. In *SCG '03: Proceedings of the nineteenth annual Symposium on Computational geometry* (New York, NY, USA, 2003), ACM Press, pp. 237–245.

[3] Aggarwal, A., Alpern, B., Chandra, A., and Snir, M. A model for hierarchical memory. In *Proc. of the nineteenth annual ACM conference on Theory of computing* (New York, NY, USA, 1987), ACM Press, pp. 305–314.

[4] Aggarwal, A., Chandra, A. K., and Snir, M. Hierarchical memory with block transfer. In *Proc. 28th Annual Symposium on Foundations of Computer Science* (1987), IEEE, pp. 204–216.

[5] Aggarwal, A., and Vitter, J. S. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 9 (1988), 1116–1127.

[6] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[7] Ajwani, D., Dementiev, R., and Meyer, U. A computational study of external-memory BFS algorithms. In *Proc. of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006* (2006), ACM Press, pp. 601–610.

[8] Ajwani, D., Meyer, U., and Osipov, V. Improved external memory BFS implementations. In *Proceedings of the ninth Workshop on Algorithm Engineering and Experiments* (2007), pp. 3–12.

[9] Allulli, L., Lichodzijewski, P., and Zeh, N. A faster cache-oblivious shortest-path algorithm for undirected graphs with bounded

edge lengths. In *SODA* (2007), N. Bansal, K. Pruhs, and C. Stein, Eds., SIAM, pp. 910–919.

[10] Arge, L. The buffer tree: A technique for designing batched external data structures. *Algorithmica 37*, 1 (2003), 1–24.

[11] Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the thiry-fourth annual ACM Symposium on Theory of computing* (2002), ACM Press, pp. 268–276.

[12] Arge, L., Brodal, G. S., Fagerberg, R., and Laustsen, M. Cache-oblivious planar orthogonal range searching and counting. In *SCG '05: Proceedings of the twenty-first annual Symposium on Computational geometry* (New York, NY, USA, 2005), ACM Press, pp. 160–169.

[13] Arge, L., Brodal, G. S., and Toma, L. On external-memory mst, sssp and multi-way planar graph separation. *J. Algorithms 53*, 2 (2004), 186–206.

[14] Bayer, R., and McCreight, E. M. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (February 1972), 173–189.

[15] Bender, M. A., Demaine, E. D., and Farach-Colton, M. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* (2000), IEEE Computer Society, p. 399.

[16] Bender, M. A., Duan, Z., Iacono, J., and Wu, J. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete algorithms* (2002), Society for Industrial and Applied Mathematics, pp. 29–38.

[17] Bilardi, G., Ekanadham, K., and Pattnaik, P. Computational power of pipelined memory hierarchies. In *Proceedings of the thirteenth annual ACM Symposium on Parallel algorithms and architectures* (2001), ACM Press, pp. 144–152.

[18] Bilardi, G., Ekanadham, K., and Pattnaik, P. Optimal organizations for pipelined hierarchical memories. In *SPAA '02: Proceedings of the fourteenth annual ACM Symposium on Parallel algorithms and architectures* (New York, NY, USA, 2002), ACM Press, pp. 109–116.

[19] Bilardi, G., Ekanadham, K., and Pattnaik, P. An address dependence model of computation for hierarchical memories with pipelined transfer. In *Proc. 19th International Parallel and Distributed Processing Symposium* (2005), IEEE Computer Society.

[20] Bilardi, G., and Peserico, E. An approach towards an analytical characterization of locality and its portability. *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'01)* (2001).

[21] Bilardi, G., and Preparata, F. P. Horizons of parallel computation. *J. Parallel Distrib. Comput. 27*, 2 (1995), 172–182.

[22] Brodal, G. S., and Fagerberg, R. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (2002), Springer-Verlag, pp. 426–438.

[23] Brodal, G. S., and Fagerberg, R. Funnel heap - a cache oblivious priority queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation* (2002), Springer-Verlag, pp. 219–228.

[24] Brodal, G. S., and Fagerberg, R. On the limits of cache-obliviousness. In *Proceedings of the thirty-fifth annual ACM Symposium on Theory of computing* (2003), ACM Press, pp. 307–315.

[25] Brodal, G. S., and Fagerberg, R. Cache-oblivious string dictionaries. In *Proceedings of the seventeenth annual ACM-SIAM Symposium on Discrete algorithm* (New York, NY, USA, 2006), ACM Press, pp. 581–590.

[26] Brodal, G. S., Fagerberg, R., and Jacob, R. Cache oblivious search trees via binary trees of small height. In *Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete algorithms* (2002), Society for Industrial and Applied Mathematics, pp. 39–48.

[27] Brodal, G. S., Fagerberg, R., Meyer, U., and Zeh, N. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, vol. 3111 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 2004, pp. 480–492.

[28] Brodal, G. S., Fagerberg, R., and Vinther, K. Engineering a cache-oblivious sorting algorithm. In *Proceedings of the sixth Workshop on Algorithm Engineering and Experiments* (2004), pp. 4–17.

[29] Buchsbaum, A. L., Goldwasser, M., Venkatasubramanian, S., and Westbrook, J. R. On external memory graph traversal. In *Proceedings of the eleventh annual ACM-SIAM Symposium on Discrete algorithms* (2000), Society for Industrial and Applied Mathematics, pp. 859–860.

[30] Chazelle, B., and Monier, L. A model of computation for vlsi with related complexity results. *J. ACM 32*, 3 (1985), 573–588.

[31] Chiang, Y.-J., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., and Vitter, J. S. External-memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM Symposium on Discrete algorithms* (1995), Society for Industrial and Applied Mathematics, pp. 139–149.

[32] Chowdhury, R. A., and Ramachandran, V. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the sixteenth annual ACM Symposium on Parallelism in algorithms and architectures* (2004), ACM Press, pp. 245–254.

[33] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[34] Demaine, E. D. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, Lecture Notes in Computer Science. BRICS, University of Aarhus, Denmark, June 27–July 1 2002. To appear.

[35] Dijkstra, E. W. A note on two problems in connection with graphs. *Numerical Mathematics 1* (1959), 269–271.

[36] Doweck, J. *Inside Intel Core Microarchitecture and Smart Memory Access*. Intel, 2006. Available online at `http://download.intel.com/technology/architecture/sma.pdf`.

[37] Feng, Y., and Berger, E. D. A locality-improving dynamic memory allocator. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance* (New York, NY, USA, 2005), ACM Press, pp. 68–77.

[38] Franceschini, G. Proximity mergesort: optimal in-place sorting in the cache-oblivious model. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete algorithms* (Philadelphia, PA, USA, 2004), Society for Industrial and Applied Mathematics, pp. 291–299.

[39] Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), IEEE Computer Society Press, pp. 285–297.

[40] Hennessy, J., and Patterson, D. *Computer Architecture: A Quantitative Approach*, fourth ed. Morgan Kaufmann, 2006.

[41] Kath, R. *The Virtual-Memory Manager in Windows NT*. Microsoft Developer Network, 1992. Available online at `http://msdn2.microsoft.com/en-us/library/ms810616.aspx`.

[42] Katriel, I., and Meyer, U. Elementary graph algorithms in external memory. In Meyer et al. [47], pp. 62–84.

[43] Knuth, D. E. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

[44] Kumar, V., and Schwabe, E. J. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)* (1996), IEEE Computer Society, p. 169.

[45] Maheshwari, A., and Zeh, N. A survey of techniques for designing i/o-efficient algorithms. In Meyer et al. [47], pp. 36–61.

[46] Mehlhorn, K., and Meyer, U. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms* (2002), Springer-Verlag, pp. 723–735.

[47] Meyer, U., Sanders, P., and Sibeyn, J. F., Eds. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]* (2003), vol. 2625 of *Lecture Notes in Computer Science*, Springer.

[48] Meyer, U., and Zeh, N. I/O-efficient undirected shortest paths. In *ESA* (2003), G. D. Battista and U. Zwick, Eds., vol. 2832 of *Lecture Notes in Computer Science*, Springer, pp. 434–445.

[49] Meyer, U., and Zeh, N. I/O-efficient undirected shortest paths with unbounded edge lengths. In *ESA* (2006), Y. Azar and T. Erlebach, Eds., vol. 4168 of *Lecture Notes in Computer Science*, Springer, pp. 540–551.

[50] Munagala, K., and Ranade, A. I/O-complexity of graph algorithms. In *Proceedings of the tenth annual ACM-SIAM Symposium on Discrete algorithms* (1999), Society for Industrial and Applied Mathematics, pp. 687–694.

[51] Pan, S., Cherng, C., Dick, K., and Ladner, R. E. Algorithms to take advantage of hardware prefetching. In *Proceedings of the ninth Workshop on Algorithm Engineering and Experiments* (2007), pp. 91–98.

[52] Prakash, T. K. Performance analysis of intel core 2 duo processor. Master's thesis, Louisiana State University and Agricultural and Mechanical College, 2007.

[53] Prokop, H. Cache-oblivious algorithms, June 1999. Master's thesis, MIT Department of Electrical Engineering and Computer Science.

[54] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Commun. ACM 28*, 2 (1985), 202–208.

[55] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6*, 3 (1977), 80–82.

[56] van Emde Boas, P., Kaas, R., and Zijlstra, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10* (1977), 99–127.

[57] Vitter, J. S. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys 33*, 2 (2001), 209–271.

[58] Zeh, N. *I/O-efficient algorithms for shortest path related problems*. PhD thesis, School of Computer Science, Carleton University, 2002.