Orthogonal Range Counting in The Cache Oblivious Model

M.Sc. Thesis by
Morten Laustsen
Spring 2006
Department of Computer Science
University of Aarhus
Denmark

Abstract

There exist several models of how a computer works putting their emphases on varying aspects of the computer. This paper uses the random access model (RAM), external memory model and cache oblivious model to describe three similar algorithms based on the same general idea to support orthogonal range counting queries in $O(\log_2 n)$ time, $O(\log_B N)$ I/Os and $O(\log_B N)$ I/Os respectively all using linear space.

The cache oblivious orthogonal range counting structure presented in this paper has been published in the proceedings of the 21st Annual ACM Symposium on Computational Geometry [5].

The three algorithms has been implemented and are benchmarked running both on internal and external memory. L2 cache misses are also measured. It is found that the algorithm for the RAM model performs best in RAM and the external algorithm performs best when the data needs to be stored on disk and when working with little available RAM forcing the programs to swap memory. The algorithm for the cache oblivious model is found to work well in both circumstances but not better than the model designed for the specific environment. On the L2 cache the cache oblivious model is found to work on par with the external model algorithm.

It is shown that the method used for the layout of the data structures for the cache oblivious model using the van Emde Boas layout has a positive effect on the performance of the algorithm.

Preface

When I started the initial work on this thesis I did not have any specific goal in mind. I had an idea that I would like to concentrate on a topic in the area of computational geometry and external memory algorithms. So I started skimming different articles within this area of interest looking at what had been done and what results had been achieved. I made some tables (Table 1.1 1.2 and 1.3 in the introduction) of my findings comparing different problems with their results in the RAM, external memory and the cache oblivious memory model. I quickly realized that the RAM and the external models were well described and already had good general results. But in the cache oblivious model hardly any results had been made. So in stead of writing about something that has already been made or improving on it I decided to make something original. I chose range counting since it is a relatively simple problem and there are some good simple results available in both the RAM [12] and the external memory model [16] to try to match up against.

Acknowledgement

Thanks to Gerth Stølting Brodal for supervising my work and helping me through out the two years that it has taken me to come to this end. I would also like to thank Sathish Govindarajan for lending me the source code to his implementation of CRB-trees saving me many hours of work. I would further like to thanks Søren Gjellerup Christiansen and Mads Darø Kristensen for proof reading some of the work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The range counting problem is probably one of the simplest geometric problems. Given a set of points and a geometric shape. Count the number of points inside the geometric shape. The easy solution would be to look at each point and check if it lies within the shape. This is also the fastest way to do it if we are only interested in making one query. But if we want to make several queries to a given point set it is worth while doing some initial calculations making data structures with additional information so that we do not have to look at all points for each query.

Range counting is typically used in areas such as geographic information systems (GIS) in short. A GIS handles geographic informations such as topographic information about a landscape or the road net of a country. Here huge amounts of data are normally present which need to be analyzed. An example could be that you had a detailed map of a region and you wanted to know the number of trees inside a square.

The problem when analyzing the data is that all the data takes up lots of space and cannot all be stored in a computers internal memory. It has to be stored elsewhere in some external memory like on a hard drive or on a network while being analyzed. This represents a problem since the access time to a piece of data stored in the external memory is very large compared to if it was stored in the computers internal memory. To get around this problem we have to develop ways to minimize the number of times that we access the external memory and when we do, use the data we get as much as possible.

1.2 Different types of range queries

Range searching problems have been studied for a long time. They are of the general form: Given a set of input objects in a geometric space and a query range. Calculate a given property of the objects which fall within the query range. This involves problems such as *range searching*, *range counting* and *range max* which

	RAM	External	Cache Oblivious
1 dimension			
2 dimensions	q $O(\log n + t)$ u $O(\log n)$ s $O(\frac{n \log n}{\log \log n})$ [21]	q $O(\log_B N + T/B)$ u $O(\frac{\log_B^2 N}{\log_B \log_B N})$ s $O(\frac{N \log_B N}{\log_B \log_B N})$ [6] q $O(\sqrt{N/B} + T/B)$ u $O(\log_B^2 N)$ s $O(n)$ [25]	q $O(\log_B N + T/B)$ s $O(\frac{N \log_2^2 N}{\log_2 \log_2 N})$ [5]
3 dimensions		q $O((\log \log \log_B N) \log_B N+$ $T/B)$ s $O(N/B \log^4 N/B)$ [27]	
d dimensions	$w < 1$ q $O(\frac{\log n}{(\log \log n)^{d-1}} + t)$ u $O(\log^{w+d-2} n)$ s $O(n \log^{w+d-2} n)$ [20] q $O((\frac{\log n}{\log \log n})^{d-2} + t)$ s $O(n(\frac{\log n}{\log \log n})^{d-3})$ [26]		$B = 2^{2^c}$ q $O((N/B)^{1-1/d} + \frac{T}{B})$ s $O(n)$ i/d $O(\log_B^2 N)$ [2]

Table 1.1: Orthogonal range reporting

are described below. Others could be *empty set*, decide if a given query range does not contain any objects, or *point intersection searching*, given a set of geometric objects decide which objects contain the query point.

In the following discussion of different range searching problems it is assumed that the query range is a simple axis aligned cube in the given dimension. Other query ranges could as an example be a planar disk where range counting can be answered in $O(\sqrt{n \log_2 n})$ time using $O(n \log n)$ space [3].

In the tables below the following terminology is used: q it the query cost, s is the size of the structure used to answer the queries, u is the update cost, i is the insert cost and d is the deletion cost. An expression at the beginning is a requirement for the algorithm.

The problem of range reporting is: Given a set of points, report what points lie within the query range. The results achieved can be seen in Table 1.1. The term T is the result size being reported. For the 2 dimensional case the results achieved for all the models are the same.

Range counting is: given a set of points, count the number of points within the query range. The results achieved for this problem can be seen in Table 1.2. The 2 dimensional results are the algorithms described in this thesis with the exception

	RAM	External	Cache Oblivious
1 dimension			
2 dimensions			$q\ O(\log_B N)$ $s\ O(N)$ [5]
3 dimensions			
d dimensions	$d \geq 2$ $q\ O((\frac{\log n}{\log\log n})^{d-1})$ [26] $s\ O(n(\frac{\log n}{\log\log n})^{d-2})$	$d \geq 2$ $q\ O(\log_B^{d-1} N)$ [16] $s\ O(N \log^{d-2} N)$	

Table 1.2: Orthogonal range counting

	RAM	External	Cache Oblivious
1 dimension	$q/u\ O(\log n)$ $s\ O(n)$ [18] d $O(\log n \log\log n)$	$q\ O(\log^{1+e} n)$ [14] $s\ O(n)$	
2 dimensions	$q\ O(\log_B N)$ [12] $u\ O(\log_B N)$	$q\ O(\log_B^2 N)$ [1] $s\ O(N)$	$q\ O(\log_B N)$ $s\ O(\frac{N \log_2^2 N}{\log_2 \log_2 N})$ [5]
3 dimensions			
d dimensions			

Table 1.3: Orthogonal range max

of the RAM model. For that we use the article by Bernard Chazelle [12] which can perform queries in $O(\log_2 n)$ time and takes up $O(n)$ space.

The problem of range max is: given a set of points where each point has an associated weight, find the point with the largest weight. Table 1.3 shows the existing results found. The cache oblivious version is partly based on the algorithm described in Chapter 7, where a slightly modified version of this is used to calculate 3 sided queries.

Many more different types of range searching problems exist. For a survey of some more results we refer to the survey by Agarwal [3]. It mostly covers results in the RAM model.

Chapter 2

Models of Computation

To describe the performance of a program running on a computer different models have been developed. The basic and most commonly used model is the RAM model which only considers the work done by the CPU. This idealized model does not take into account that real computers only have a limited amount of RAM. To deal with this limited amount of RAM the computer swaps some of the data onto some other storage medium like a hard drive and fetches it again when needed. This results in the external memory model. In recent years the cache oblivious model has been developed. This model does not depend upon information of how and where memory is stored as opposed to the external memory model while still being able to handle large amounts of data. This model has the advantage of handling multi level memory hierarchy seamlessly. This is relevant in e.g. virtual machines where we do not necessarily have information about the specifics of the underlying machine.

2.1 Hardware

A computer consists of a number of different connected pieces of hardware. The central part of any computer is the CPU. This is the one in charge of doing all the calculations. It is capable of many different types of calculations. They range from the basic arithmetic functions like $+, -, \times, \div$, boolean functions like NOT, AND, OR to the more complicated like comparison and branching. It also contains functions to control the behaviors of other pieces of hardware.

The CPU is connected to a hierarchy of several levels of memory. The innermost memory is the CPU registers which contain the variables being worked on at any given time. There are however very few of these. Then comes a number of layers of cache each further away from the CPU but also larger in size. They contain the data and program being worked on at the moment. The first level of cache (L1) contains a subset of information of the next level (L2) and so on. Data is then propagated back and forth between the layers to maintain consistency as needed. The innermost layer is in charge of supplying the CPU with the required information.

The next level after the cache levels is the RAM. This is the main place where the data being work on is stored. It is significantly larger than the cache but slower.

The last normal level of memory is the hard drive. This contains all the programs and data being kept on the computer for long term use. It can however also be used to store memory that the computer is working on in case that the RAM gets full. The hard drive has however a major disadvantage since the access time to data is a lot slower than that of all the other levels.

To get some perspective on the access time for the different levels of memory we can look at the following where the hardware access time and a human equivalent is compared:

[1]

Time	Hardware	Time	Human equivalent
0.5ns	CPU register	~ 0.5s	Time to get information from the brain
5ns	L2 Cache	~5s	Time to read some information from a paper on your desk
50ns	RAM	~50s	Time to look up some information in a book
10ms	Hard drive	~ 10000000s or ~ 1/3 years	Time to walk from the east coast to the west coast of USA lookup some information and then walk back

From this perspective we can see that even though 10ms might seem fast it is very slow compared to the rest of the system.

There exist other types of memory like a CD-ROM drive or a network storage array but we will leave them out. There are also other types of hardware not associated to memory like audio and video cards but they are not relevant to this discussion and will therefore not be discussed.

2.1.1 Page faults

The Linux operating system (OS) presents a linear piece of memory for a program to use. Each piece of data has a virtual address in this memory at which the data is stored. It is then up to the OS to figure out how that address is mapped into an actual physical memory address. If the program's memory requirements grow beyond the available RAM the OS starts swapping some of the data out onto the hard drive. The OS marks the address where the data was stored in the physical memory as being free but does not delete the data there. The program is unaware of this swapping as it still sees the data as having the same virtual address.

When at a later stage the program need the data again there are two scenarios. The OS might not have used the physical address where the data was stored previously to store some other data. In this case the OS simply uses the same

[1] 10^7s $\approx$ 115.7 days. If you walk 5.4 kmph for 16 hours a day for 57.5 days you walk 4968km and USA coast to coast along Route 50 is approximately 4944km[9]

physical memory address for the data using the data already there. This is called a minor page fault. In the case that the OS has used the physical memory address to store some other data a free address in the physical memory address is located and the data is retrieved from the hard drive and put into this address. This is called a major page fault. When a major page fault occurs it takes considerable time before the data is available in the RAM because of the slow access times of a hard drive.

2.2 RAM Model

In the random access memory (RAM) model we have a machine with an infinite amount of memory and an arithmetic unit. What the arithmetic unit can do and how the memory can be accessed differs. In [12] variants of the RAM models are described. The memory can either be accessed by use of pointers which can be followed around in memory or memory can be accessed by a calculated memory address. The arithmetic unit can be limited to only having $+$, having to simulate the rest of the arithmetic functions. Or it can have the full range of arithmetic functions $+, -, \times, \div$, shift, boolean operators, comparison and more.

At present time the generally used model to describe algorithms is the RAM model. It has an infinite amount of memory which can be accessed at random in $O(1)$ time. The arithmetic unit or CPU can perform all common arithmetic functions at a unit cost. The cost of an algorithm is calculated in terms of how many calculations are done on the elements. This model is however not realistic when working with large amounts of data since we do not have infinite amounts of main memory in practice.

2.3 External Memory Model

In the external memory model we have a CPU with some limited amount of internal memory connected to some external memory. The data set consists of a total of n elements which take up N space. An element is considered to be a small set of data describing e.g. a set of coordinates specifying a point. The internal memory can contain M elements and can be accessed randomly for free. The external memory can contain an infinite amount of elements and can also be accessed randomly. To do any calculations on an elements it is necessary that it is in the internal memory. If it is not present in the internal memory an input/output (I/O) operation will have to take place in which the needed data are transferred into the internal memory and replaced with some other data. Each I/O operation on the external memory will load B consecutively stored elements, with $1 \leq B \leq M/2$, into the internal memory. The time to make an I/O is considerably longer than accessing the internal memory or doing calculations on the CPU. So the cost of an algorithm is measured in terms of I/Os done. Linear time is considered to be $O(N/B)$ since that is the time it takes to scan through all the elements. Sorting of elements can be done in $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os [4] where we make use of the knowledge of the memory size by filling the main memory and then sorting parts of the problem there and finally merging it all. For searching we have B-trees [8] which can perform searches in $O(\log_B N)$ I/Os.

2.4 Cache Oblivious Model

In this model we describe the algorithm in the RAM model but we analyze it in the external model. That is, we design an algorithm which is unaware of the memory hierarchy but analyze it as running under that hierarchy. This model was developed by Frigo et al. [15]. In the model we have a CPU which has a limited amount of cache of size Z words. There is also some main memory of which we have no information about except that it is sufficiently large. If the cache is full and some new data is needed from the memory the data will replace some data already present in the cache using the optimal off-line replacement strategy[15]. Since programs are designed to work optimally for all sizes of blocks and memory we can assume that it will work optimally for all layers of memory [15]. So there is only the cache and the main memory that we will have to think about. When accessing data it is first checked if the data is in the cache. If it is, it is given to the CPU. Else the data is fetched from the main memory and replaces some data in the cache which is predicted to be used furthest into the future by the optimal off-line replacement strategy.

We have the usual n elements which take up N space. We do not know anything about the main memory except that it is there and has sufficient space. We do not know about B either. Some algorithms however make use of the tall cache assumption that $B^2 < Z$ to achieve some results. When getting data from the memory the presence of the cache makes it possible to do calculations on what was received. If we make sure that we store data that we need sequentially so that the data received contain several useful pieces of information we do not have to perform as many memory transfers. Thus enabling us to perform procedures such as scanning all elements in $O(N/B)$ I/Os without being aware of the size B. With the tall cache assumption we can, as in the external model, again perform sorting in $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os using funnelsort [15]. Searching can be done in $O(\log_B N)$ I/Os with the help of the van Emde Boas layout which is the cache oblivious version of the B-tree, see Section 3.2.3.

Chapter 3

Terminology

In this chapter some of the terminology used through out this paper is presented. In Section 3.1 the terminology associated with geometry is presented and in Section 3.2 some of the general methods used in the cache oblivious model is described.

3.1 Geometry

3.1.1 Points

Throughout this paper the same set of points P_G will be used for illustrative purposes. This is done to make it possible to illustrate some of the similarities in the three algorithms. The points can bee seen in Figure 3.1. This particular set of points have been chosen so that they make the structures generated by the algorithms show some interesting features.

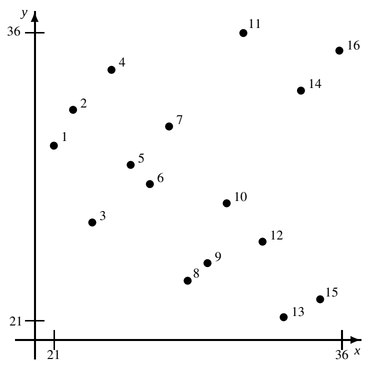

Figure 3.1: The input points P_G

3.1.2 Semigroup

A semigroup is a group of elements $\mathcal{S}$ and a binary operator $+$. For a group $\mathcal{S}$ to be a semigroup the elements have to be associative. For the semigroup to be commutative the operator $+$ has to be commutative. That is that the order at which the elements are applied to the operator does not matter $a + b = b + a$.

3.1.3 Range Counting

A survey of some general range searching algorithms is nicely described in the survey [3] by P. K. Agarwal and J. Erickson. In it the problem of range searching is defined as follow.

"Let $(\mathcal{S}, +)$ be a commutative semigroup. For each point $p \in \mathcal{S}$, we assign a weight $w(p)$. For any subset $\mathcal{S}' \subseteq \mathcal{S}$, let $w(\mathcal{S}') = \sum_{p \subseteq \mathcal{S}'} w(\mathcal{S})$ where addition is taken over the semigroup. For a query range $\gamma \subset \mathcal{R}$, we wish to compute $w(S \cap \gamma)$."

For the problem of range counting we have that $+$ is the normal algebraic addition and that $w(p) = 1$ for all points $p \in \mathcal{S}$. For range reporting we have that the operator $+$ is the union operator $\cup$, the weight of the point is the point it self $w(p) = p$.

3.2 Cache oblivious primitives

3.2.1 Scanning

If we go linearly through all the elements stored consecutively in memory visiting all of them one at the time this is called a scanning. Because of the cache we are able to hold onto small parts of consecutive data which we can scan through without having to make additional memory transfers. Because of this we are able to scan through all the elements in $O(N/B)$ I/Os.

3.2.2 Sorting

We are here talking about comparative sorting. That is given two elements we can compare them with comparative operators, like $<$, to each other and say which one is the smaller. An efficient way of doing this in the cache oblivious model is to use funnel sort [12].

Given an input with n elements. We split it up into $n^{1/3}$ subproblems which we sort recurcivly. We merge the subproblems using a *k-merger*. A *k-merger* works by taking k sorted input streams each holding k^2 elements. The streams are merged recurcivly using $\sqrt{k}$ *k-mergers*. The base case is $k = 2$ producing $k^3 = 8$ sorted elements. When merging we use a buffer to hold the points and we only merge $2k^{3/2}$ elements at the time after which we begin the merging of an other *k-merger* coming back to it when we need more elements from that particular buffer.

This together with the tall cache assumption enables us to perform sorting in the optimal $O(N/B \log_{M/B} N/B)$ I/Os.

3.2.3 van Emde Boas Layout

The van Emde Boas layout [24] is a way to layout a binary tree so that a search can be performed in $O(\log_B N)$ I/Os. It is the cache oblivious version of the B-tree from the external model.

In a normal binary tree the internal nodes of the tree can be laid out in any order one would want. You would then have pointers between the nodes to enable traversal of the tree. This can result in a lot of jumping backward and forward a lot of times while only getting small amounts of data each time when traversing the tree. This is no problem when working in the RAM model since random memory access is free. But when working in the external or cache oblivious model this becomes expensive. However using the van Emde Boas layout enables us to only travel forward in memory while using fewer steps to get the same data. This is done by clustering data that we are likely to use next, close to and forward in memory, making it likely that when you access the memory that you need you will also get the next couple of pieces of data needed.

It works by laying out the nodes of the tree recurcivly. Given a tree C of height h we split the tree in half where the top tree C_0 is of height $\lfloor h/2 \rfloor$. This gives us $s = 2^{\lfloor h/2 \rfloor}$ sub trees C_1 to C_s each of height $\lceil h/2 \rceil$. We then recurse on the top tree first and then on all the subtrees from left to the right. In the case where we have $h = 1$ we store the node next to the previously stored node. See Figure 3.2 for an example of the layout.

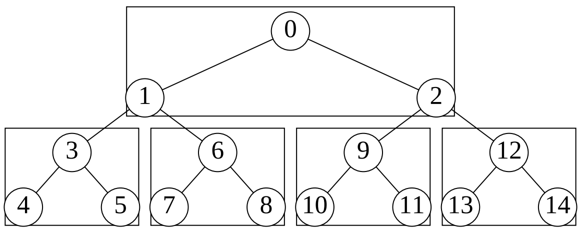

Figure 3.2: The van Emde Boas layout. The node number represent the location in memory of the node.

When accessing a node stored in external memory we get $O(B)$ elements back which form a small subtree of height $O(\log_2 B)$. This can then be traversed without the need for further I/Os. Then the next small tree on the path can be loaded from memory and traversed and so forth. So to traverse a complete tree we need to make $O(\frac{\log_2 n}{\log_2 B}) = O(\log_B n)$ I/Os.

Chapter 4

Orthogonal range counting, general approach

The range counting algorithms described in Chapters 5, 6 and 7 are all based on the same general idea.

4.1 4 sided → 2 sided queries

Given an orthogonal range counting query $Q = [x_1, x_2] \times [y_1, y_2]$ we can split up this query into four separate two sided sub queries:

$$\begin{aligned} Q_1 &=]-\infty, x_2] \times]-\infty, y_2] \\ Q_2 &=]-\infty, x_1[\times]-\infty, y_2] \\ Q_3 &=]-\infty, x_2] \times]-\infty, y_1[\\ Q_4 &=]-\infty, x_2[\times]-\infty, y_1[\end{aligned}$$

The queries are also illustrated in Figure 4.1. This gives us that our query then becomes $Q = Q_1 - Q_2 - Q_3 + Q_4$. Thus reducing the problem of four sided queries into four problems of two sided queries. So all that we have to be concerned about is how to answer a two sided query $q =]-\infty, x_q] \times]-\infty, y_q]$.

4.2 Data structure for 2-sided queries

In the following section we describe a data structure for the 2-sided query $q =]-\infty, x_q] \times]-\infty, y_q]$. We make a search tree X of the points P sorted with respect to the x coordinates. When traversing X to find a value x_q, at a node v we want to be able to know how many points have smaller x values that v_x and has v as an ancestor. Of those we want to find out how many got an y value less than y_q. If we can calculate that then all that we got to do is to traverse X and each time that we reach a node v whose x values is less than x_q we add the calculated sum to the final result. Figure 4.2 shows an example of such a query. To be able to do this we

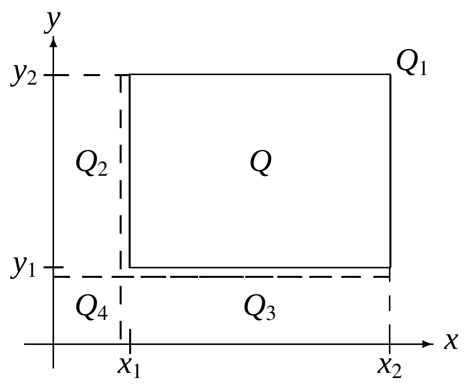

Figure 4.1: The four two sided subqueries needed to make one four sided query

store a list L_v of points at each node v containing all the points which got v as an ancestor. Since each point is stored in one list at each level in the tree this gives us a space requirement of $O(n \log n)$. The list L_v is sorted with respect to the y value. Together with each point we store how many points in the left child got an y value smaller than it self. This rank correspond to what we wanted to calculate above where we wanted to find the number of points with a x values smaller than v_x and y value smaller than y_q at each node.

4.3 Fractional cascading

Searching in the L_v lists after the point with the largest y value smaller than or equal to y_q using a search tree for each L_V would imply an extra $\log n$ time for each L_v giving a total query time of $O(\log^2 n)$.

To eliminate this problem we make use of the idea of fractional cascading [13]. This can be done since the lists of the children of a node v are subsets of the list L_v. We start by having a search tree Y of the points sorted with respect to the y values. This points into the list L_r at the root note v_r of X. We then add pointers from each point p in the list L_v down into the lists of the children of v to the point with the largest y value less than or equal to p_y. This enables us to follow the points with y value less than or equal to y_q at each list without having to search for it except at the root note.

This is the general way that the algorithms in Chapters 5, 6 and 7 works. But they all have their own special way to do certain things reflecting some design choices and the model in which they are designed to work.

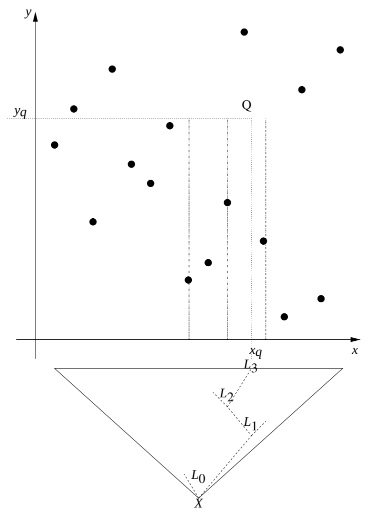

Figure 4.2: A two sided query $q =]-\infty, x_q] \times]-\infty, y_q]$ with the X tree.

Chapter 5

Range Counting in the RAM model

In this section we describe the range counting algorithm by Bernard Chazelle [12] which assumes the RAM model. The reason for choosing this algorithm is that it has linear space cost and queries can be performed in $O(\log_2 n)$ time at the same time as being simple to understand and implement. The CRB tree [16] as described in Section 6 is also based upon this algorithm but it also works well in the external memory model. There exist other algorithms like [26] which is faster than this algorithm but it is not as simple as the one described here.

This algorithm works in a similar way as the general algorithm described in Chapter 4. But it has an other way to split up the four sided query. If we got a query $Q = [x_1, x_2] \times [y_1, y_2]$ the sub queries does not go towards $-\infty$ on the x-axis but rather goes towards the least common ancestor of x_1 and x_2 in the search tree X. To be able to do this it is required that a point p in the lists L_v keeps track of how many points from both the left and right child are small than or equal to p_y.

5.1 Data structure

The data structure consists of four arrays X, Y, B, C each of size n. The arrays X and Y stores the x and y coordinates respectively sorted with respect to the x coordinate while keeping them paired up. The B and C arrays stores bit vectors which has to do with the sorting of the points, bit counting and some intermediate values.

Let $\lambda = \lceil \log_2 n \rceil$ and $\mu = 2(1 + \lfloor \log_2 \lambda \rfloor)$. Each index i in B_i consists of two parts Λ_i and M_i in order to save space. Λ_i is of size λ bits and M_i is of size μ bits together forming B_i. The Λ part represents how a binary mergesort of the y coordinates would have taken place. A position in Λ refers to the position from the beginning of the list upto the point of interest. Λ is basically an array of all the lists L_v for the nodes v of X. The binary mergesort works by merging the children of the nodes of the tree representing the points P. This merging is illustrated in Figure 5.2. The merging starts at the leaves of the tree X going from left to right and moving upwards. Given two lists L_0 and L_1 of sorted points coming from the left and right child of a node in the tree. We then pick the smallest point from the bottom of the lists of either L_0 or L_1 and put it into a new list L_v, where v is the parent of the two lists. If the point chosen was from L_0 or L_1 we put a 0 or 1 respectively into the next position in the L list. This is done until both lists are empty. This continues all the way to the root of the tree. M_i is the number of 1's in the binary representation of the index i. This is used to be able to look up the number of set bits in a variable without having to calculate it first.

C also consists of two parts at each index i. The first part λ_{C_i} of size λ bits is equal to 2^i and is only defined for $i \in [0, \lambda - 1]$ the rest is left black. The second part M_{C_i} of size μ is equal to the total number of 1's in the bit representation of Λ from Λ_0 to Λ_i. This is used to be able to look up the total number of points which has come from a right child up till Λ_i.

To make it possible to fit the two parts into one variable it is required that the word size w is equal to

$$w \geq \lambda + \mu = \lceil \log_2 n \rceil + 2(1 + \lfloor \log_2 \lambda \rfloor) = \lceil \log_2 n \rceil + \lceil 2 * \log_2 \log_2 n \rceil + 2 \qquad (5.1)$$

To give an example of how the data structure works Figure 5.1 illustrates the data structures generated given the input points P and Figure 5.2 shows how the values for Λ_i were created.

P={(24,34),(29,24),(31,36),(30,27),(27,31),(26,28),(32,25),(23,26),(33,21),(36,35),
(21,30),(22,32),(34,33),(35,22),(25,29),(28,23)}

$\lambda = 4, \mu = 6$

X={21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36}

Y={30,32,26,34,29,28,31,23,24,27,36,25,21,33,22,35}

B={0101 000000,1010 000001,0110 000001,0101 000010,1001 000001,1001 000010,
0101 000010,0101 000011,1011 000001,0100 000010,1100 000010,0110 000011,
1101 000010,1010 000011,0000 000011,1011 000100}

C={0001 000010,0010 000100,0100 000110,1000 001000,0000 001010,0000 001100,
0000 001110,0000 010000,0000 010011,0000 010100,0000 010110,0000 011000,
0000 011011,0000 011101,0000 011101,0000 100000}

Figure 5.1: An example of the data structure generated given the input points P_g

5.2 Algorithm

A query $Q = [x_1, x_2] \times [y_1, y_2]$ is split up into 4 sub queries:

Q_1 $[x_1, cut[\times [-\infty, y_2]$

Q_2 $[x_1, cut[\times [-\infty, y_1[$

Q_3 $[cut, x_2] \times [-\infty, y_2]$

Q_4 $[cut, x_2] \times [-\infty, y_1[$

```
                       {(21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36)}
                                        1101101000001011
                     {(23,26,28,29,30,31,32,34),(21,22,24,25,27,33,35,36)}
                              10110100                     11000110
                 {(26,30,32,34),(23,28,29,31),(24,25,27,36),(21,22,33,35)}
                      1001            1001            0101            0101
           {(30,32),(26,34),(28,29),(23,31),(24,27),(25,36),(21,33),(22,35)}
              01        01        10        10        01        10        01        01
  {(30),(32),(26),(34),(29),(28),(31),(23),(24),(27),(36),(25),(21),(33),(22),(35)}
```

Figure 5.2: The merge sort of the points P_G and the resulting values for Λ

This gives that a query is $Q = Q_1 + Q_3 - Q_2 - Q_4$. Here *cut* is the lowest common ancestor in the X tree which spans x_1 and x_2. In the following we will consider a query q to be one of the four described sub queries and x_q and y_q to be the input corner coordinates and *cut* to have been given. *cut* can initially be calculated by following a search for x_1 and x_2 in X and *cut* will be where the searches separate.

We start by defining some functions to be used by each query. There is a function *One(pos)* which calculates the number of set bits in Λ up till bit number *pos*. This is done by looking at M_{C_i} for $i = pos/\lambda$ which contain the number of ones in Λ_i up till B_{i-1}. We then mask out the bits after *pos* in the λ part of B_i resulting in a value z. To avoid having to count the number of set bits in z we look this up in M_i for $i = z$ which contains the number of set bits in z. Since all we do in this function is arithmetic on $O(1)$ different values in RAM this function runs in $O(1)$ time.

The second function *Newpos* is used to calculate the position of a bit when going from a node v to its child u. That is we calculate the pointers from a list L_v to its child lists. It takes four arguments:

- *dir* indicates if we should go to the left or right child.
- *block* the position of the first bit in Λ_i for the node v.
- *pos* the position of the bit of interest in Λ_i.
- *width* the number of leaves of the subtree rooted at node v.

The function works in two different ways depending on *dir*. If we go to the left we return $pos - n + One(block) - One(pos)$. That is we go from the point of interest and down one level in the tree. We then need to align the position up by subtracting the points coming from the right child of v. If we go to the right we return $block - n + width/2 + One(pos) - One(block)$. That is we go down the tree one level but from where the node v started ending at the beginning of the left child. We then go to the right child and finally aligning it up. This function also only do some arithmetic operations and call a function twice that works in $O(1)$ time so it also runs in $O(1)$ time.

A third function *Cum* calculates the sum of the positions in Λ we have visited when taken a child going away from *cut* in X. It takes four arguments:

- *cut* is the same as *cut* mentioned above.
- *init* is the position in Λ of the point with y value at most q_y at the root node.
- *path* is the path down X to locate q_x. Stored using one bit for each level where 0 is left and 1 is right.
- *dir* is the direction away from *cut*. 0 is left and 1 is right.

We use some variables. *pos* is the position in Λ that we are currently at. Initialized to *init*. *block* is the beginning of the list L_v in Λ. It is initialized to be the position starting position of L_r. *cur* is the size of L_v For each level l in X we do the following. If the bit for the current level in *path* is the same as *dir* we add the result of *Newpos(1-dir,block,pos,cur)* to the result returned. *pos* is updated to be the position in Λ of the child as indicated by *path* in the level below of the point with at most an y value of q_y. *cur* is halved and *block* is set to point to the beginning of the list of the child node in Λ. This function calls *Newpos* one or two times at each level of X for a total of $\log_2 n$ levels giving a total running time of $O(\log_2 n)$.

To make a sub query we first find the paths P_x and P_y down X and Y for the query values x_q and y_q. For each index i in P_{x_i} and P_{y_i} we store the path as a bit where 0 represents a left turn and 1 represents a right turn. The number the bits of P_x and P_y gives are basically the rank of the point, sorted with respect to x or y respectfully, with coordinates at most that of the query. *Cum* is then called on each of the four queries subtracting the two lower queries to get the results.

Cum does not actually give the number of points in a query but rater a sum of positions visited in Λ. To explain this we have to take a look at a level l in X. For a top and a bottom query on the same side the contribution to *Cum* of the position pos_t and pos_b at a level l when subtracted is actually the number of points in the interval we are interested in. Not the sum of positions. Fortunately the order at which we add and subtract numbers does not matter.

So the total time for running the four queries are $O(4 \cdot \log_2 n) = O(\log_2 n)$.

5.3 Implementation

The Pascal code for the algorithm is written in the article [12]. So what had to be done to implement it was just to rewrite it in C++. This code with optimizations can be found in Appendix C. Some optimizations were made to speed up the code. The main speed up done is in the construction algorithm on the code meant to count the number of set bits in a variable used to calculate M_i. The code in the paper runs in time $O(number\ of\ bits\ in\ index)$ which basically adds an extra $O(\log_2 n)$ to the construction time because each index in M_i contains $\log_2 n$ bits. This can be done more efficient as described in Appendix B making that part run in constant time.

The implementation requires that the query lies within the bounding box of the input points. Otherwise the result might be wrong. To get around this a point can be placed at plus and minus infinity or just simply checking if the query falls within the bounding box and if not returning some kind of error. Another limitation to this implementation is that it is required that the number of input points is a power of two. This is simply solved by placing the remaining required points at plus infinity.

Since the B and C array consists of two parts the use of 32 bit integers limits the input size to $n \leq 2^{21}$. This is because at this point, see Equation 5.1, the needed word size is

$$w = \lambda + \mu = \lceil \log_2 2^{21} \rceil + 2(1 + \lfloor \log_2 \lceil \log_2 2^{21} \rceil \rfloor = 21 + 2(1 + \lfloor \log_2 21 \rfloor = 31 \quad (5.2)$$

So to be able to handle larger input sizes it is necessary to use 64 bit integers. This unfortunately, doubles the memory size required. This could of cause be reduced slightly by splitting up the array into its two different parts. Then it would only be necessary with a 32 bit and a 16 bit array saving 16 bit. A better solution is however available as described in Section 5.4.

5.4 I/O analysis

This algorithm is designed to work well in the RAM model without consideration to the other models. In this section we analyze the number of I/Os required to handle queries in the cache oblivious model.

The algorithm starts by finding the path down the binary tree of X and Y. This is done in $O(\log_2 N)$ I/Os. One I/O for each level in the tree. The only function in the main loop which accesses the data structures is the function *One*. This function makes a memory accesses at three different locations each time it is called. This gives us $O(1)$ I/Os. When going down through the tree we call *One* on *pos* and *block* at each level in the tree resulting in a lookup in both B and C in the corresponding position of *pos*. The difference in *pos* between one level and the next is approximately n bits. This is because for a given point as indicated by *pos* and the same point in the level below almost all the other points are also present in Λ. This gives that we can not use what we have already read from memory at the next level. This results in that we have to spend $O(1)$ I/Os at each level for a total of $O(\log_2 N)$ I/Os for each query.

Implementation wise the code could be made more efficient with respect to I/Os. To calculate the path down the X and Y tree a structure like the van Emde Boas layout [24] could be used. This would give $O(\log_B n)$ I/Os instead of the $O(\log_2 n)$. In the function *One* the table lookup in M_i of the number of set bits in a number could be computed when needed saving an I/O at the cost of some few calculations. This will also save some time in the construction part since we do not have to precompute all of M_i. The λ_{C_i} part can also be left out since that is just

2^i. This makes it possible to merge Λ from array B and M_{C_i} from array C into one array saving one array or the extra space in the arrays could be used to increasing the possible input size to 2^{32} without increasing the existing size.

The algorithm will still run in $O(\log_2 N)$ I/Os but should perform better in benchmarks. Hence λ_{C_i} and M_i is removed in the implementation used when running the benchmark tests in Chapter 10 to make it slightly faster and to be able to use more than 2^{21} points without having to increase the memory size needed per point.

So it can be seen that this algorithm does not perform as well in the external memory model where we have a structure like the CRB-tree [16] described in Section 6 which answers queries in $O(\log_B N)$ memory transfers.

Chapter 6

Range Counting in the External Memory Model

This section describes the external memory range counting algorithm for CRB-trees by [16] Agarwal et al. It is based on the range counting algorithm by Chazelle [12] described in Chapter 5 which is designed to work in the RAM model. Based on the assumption that the word size is $\lceil \log_2 n \rceil$ it uses $O(n)$ space and performs queries in $O(\log_B n)$ I/Os. This is the fastest known algorithm for range count in the external memory model[1]. The structure can be constructed in $O(n \log_B n)$ I/Os.

This algorithm also differs from the way the general algorithm described in Chapter 4 split up the query. For a query $Q = [x_1, x_2] \times [y_1, y_2]$ we, as in Chapter 5, count towards the lowest common ancestor of x_1 and x_2 in the search tree X. We also take advantage of the knowledge of B and therefore use trees with nodes of degree B.

6.1 Data structure

The algorithm uses 3 structures. A B-tree [8] Y of the points P sorted with respect to the y coordinates[2]. X is also a B-tree of the points P but sorted with respect to the x coordinates[3]. X also contains a secondary structure Σ_v at each internal node v of X, see Figure 6.1. Σ_v consists of two arrays. A child index array CI_v and a prefix count array PC_v. CI_v is the basic list L_v of which child a given point comes from and PC_v keeps track of how many points from each of the different children of v we have passed at specific intervals along PC_v.

Let v_0 to v_{B-1} be the children of v. CI_v's size is equal to the number of leaves of v. The points are placed in CI_v sorted by increasing y value. Each index in CI_v contains a number from 0 to $B-1$ of which subtree the point is a leaf in. Each index is stored using $\log_2 B$ bits. CI_v is split in chunks of size $\mu = B \log_B N$ points so that

[1]As of 26/4/06

[2]In the article [16] this structure is called Ψ but is renamed to keep the notation consistent.

[3]Originally called T.

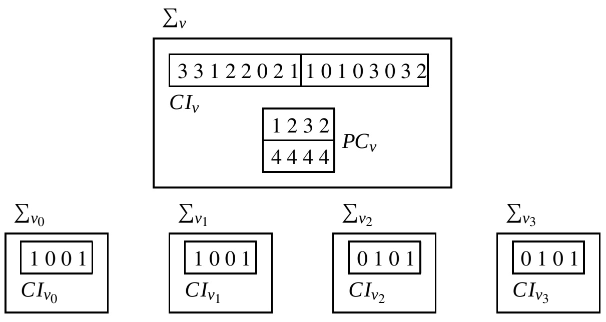

Figure 6.1: The secondary data structure Σ_v on X given the input points P_G

they each take up B space. For each chunk i the number of points originating from each of the B different subtrees including those points from previous chunks are stored in PC_v. This is done so that $PC_v[i, j]$ contains the sum of points in subtree j contained in the chunks up to and including chunk i.

6.2 Algorithm

A query $Q = [x_1, x_2] \times [y_1, y_2]$ is computed by traversing T from the root down to the leaves containing x_1 and x_2. Along the path the count is calculated by use of the secondary structure Σ_v of the nodes visited. Let v be the node of the nearest common ancestor which contains x_1 and x_2 at the root. Let P_v be the points contained in the subtrees rooted at v and N_v be the number of points in P_v. Let v_λ and v_ρ be the children of v which contains x_1 and x_2 respectively. The query is then split up into two parts. The number of points at the leafs of the children inbetween v_λ and v_ρ can be calculated using Σ_v. The remaining parts can be done by a recursive call on v_λ and v_ρ. To calculate the count for child j we need to maintain two values α_v and β_v. They are respectively the rank of the point just above y_1 and just below y_2 of the points in P_v Let $\varphi(j, r)$, for $0 \leq j \leq B - 1$ and $1 \leq r \leq N_v$, be the number of points with rank at most r that belong to P_{v_j}. Then the count of points in between y_1 and y_2 of a child v_j can simply be calculated as $\varphi(j, \beta_v) - \varphi(j, \alpha_v)$. α_v and β_v can initially be found by a search in Ψ. Maintaining them is simple since α_{v_j} is the rank of first point in P_v belonging to P_{v_j} with y coordinate at least y_1 or more precisely $\alpha_{v_j} = \varphi(j, \alpha_v)$. The same also count for$\beta_{v_j} = \varphi(j, \beta_v)$. So the calculations needed at each level is to calculate $\varphi(v_\lambda, \alpha_v)$, $\varphi(v_\rho, \beta_v)$ and $\varphi(v_j, \alpha_v)$ and $\varphi(v_j, \beta_v)$ for $\lambda \leq j \leq \rho$. This can fortunately be done by 4 I/Os. Suppose $r = \mu a + c$ for $a \geq 0$ and $0 \leq c \leq \mu$. Then

$$\begin{aligned}\varphi(j, r) &= |\{k \mid k \leq r \text{ and } CI_v[k] = j\}| \\ &= PC_v[a, j] + |\{k \mid \mu a < k \leq r \text{ and } CI_v[k] = j\}|.\end{aligned}$$

Let d and e be the chunks in CI_v just before where α_v and β_v points respectively. Then in one I/O each we can calculate the following $PC_v[d, 1], ..., PC_v[d, B]$, $PC_v[e, 1], ..., PC_v[e, B]$, $CI_v[\mu d + 1], ..., CI_v[\alpha_v]$ and $CI_v[\mu e + 1], ..., CI_v[\lambda_v]$ giving us the results we need.

6.3 Implementation

An implementation[4] of the algorithm has been done by one of the authors of the article [16] Sathish Govindarajan. In our experiments we used this code with some few changes. Some more benchmark related code was added and changed the header files to reflect the use of TPIE for Linux instead of BSD. TPIE is a software environment (written in C++) that helps facilitates the implementation of external memory algorithms. For more information about TPIE see www.cs.duke.edu/TPIE/.

[4]The implementation uses some specific calls to TPIE which are not present in newer versions of TPIE. So TPIE version 082902 should be used. TPIE version 082902 only works with gcc 3.3 or earlier and not on 64 bit computers.

Chapter 7

Range Counting in the Cache Oblivious Model

In this section a range counting algorithm in the cache oblivious model is presented. This algorithm is also presented in [5]. The data structure is inspired by a combination of [19] to make the $\mathcal{L}$ lists and the van Emde Boas layout [24] to make it cache-oblivious. The use of bit compression comes from the reading of articles [12] and [16] describing the algorithms in Chapter 5 and 6.

7.1 Data structure

The data structures used by this algorithm consists of three arrays. Two arrays containing points which make up two separate search tree for searching in the points with respect to either the x or y coordinates. The points in them are arranged according to the van Emde Boas layout in order to make it possible to make efficient searches in the cache oblivious model. It is build using implicit pointers in order to save space at the cost of CPU time using the technique described in [11]. In stead of following pointers around the tree we calculate the position in the van Emde boas layout of a point by the recursive equation

$$Pos[d] = Pos[D[d]] + T[d] + (i\ AND\ T[d]) \cdot B[d] \tag{7.1}$$

where $D[d]$ is the dept of the root of the corresponding top tree, $T[d]$ is size of the corresponding top tree, i is the position in a BFS of the tree and $B[d]$ is the size of the this bottom tree. $i\ AND\ T[d]$ is the bitwise and of the two values. At the root $Pos[1] = 1$, $d = 1$ and $i = 1$. $D[d]$, $T[d]$ and $B[d]$ can be precomputed given the height of the tree. The third array contains the list L_v of the points at each node v.

7.2 Algorithm

Given a set S of N points in the plane we do as follow. The N points are stored at the leaves of a tree $\mathcal{T}$ with respect to their x coordinate in increasing order from left to right. At each node v of $\mathcal{T}$ a list L_v is associated with it containing all the points of the leaves rooted at v. L_v is sorted with respect to the y coordinate. With each point p_i in L_v we store three additional things. Two pointers $\mathit{left}(p_i)$ and $\mathit{right}(p_i)$ to the topmost points p_l and p_r in $L_{\mathit{left}(v)}$ and $L_{\mathit{right}(v)}$ respectively, which has a y value of at most $y(p_i)$. And a number $\mathit{leftsum}(p_i)$ whose value is the number of points in $L_{\mathit{left}(v)}$ whose y value is at most $y(p_i)$. At the root v_r we have a binary search tree over L_r. The tree is sorted with respect to descending y. This representation takes up $O(N\log(N))$ space since each element is present at each level in $\mathcal{T}$. In Figure 7.1 the structure of $\mathcal{T}$ is shown.

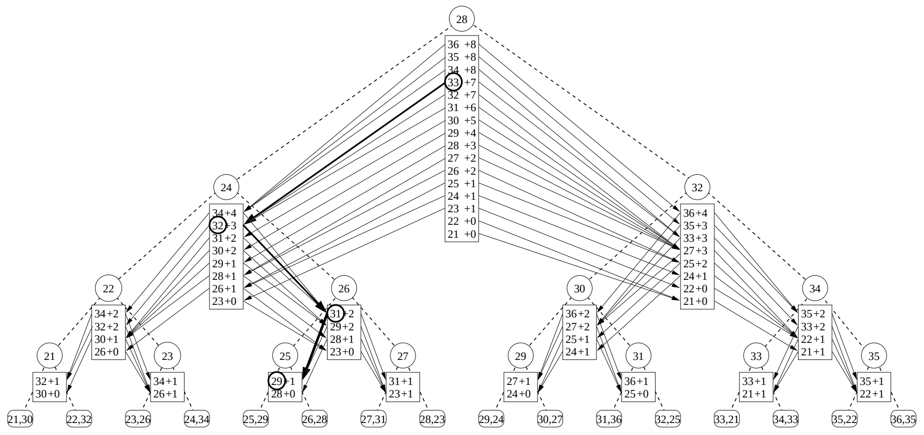

Figure 7.1: The structure of $\mathcal{T}$ without any memory layout of the points P_G

To make a query $Q = (-\infty, x_r]\times(-\infty, y_r]$ we first perform a search in the binary search tree over L_r to find the topmost point p_i in L_r whose y value is at most y_r. We then make a top down traversal of $\mathcal{T}$ down towards the right most point whose x value is at most x_r. At each node v in $\mathcal{T}$ visited during the traversal we follow the $\mathit{left}(p_i)$ or $\mathit{right}(p_i)$ pointer in L_v from the point p_i down to p_l or p_r in $L_{\mathit{left}(v)}$ or $L_{\mathit{right}(v)}$ depending on if we go to the left or right at v. Whenever we go down to the right child we also add the value $\mathit{leftsum}(p_i)$ to the result. If the point reached at the bottom of $\mathcal{T}$ is within Q then we add one more point to the result.

To show that this works we will have to prove that if p_i is the topmost point in L_v with y value at most y_r then $\mathit{left}(p_i)$ and $\mathit{right}(p_i)$ are the topmost points in $L_{\mathit{left}(v)}$ or $L_{\mathit{right}(v)}$ with y value at most y_r. This is true since $L_{\mathit{left}_v} \subseteq L_v$ and $L_{\mathit{right}_v} \subseteq L_v$ for all nodes v.

It can easily be seen that the running time of a query is $O(\log N)$, since we use $O(\log N)$ time to search L_r and $O(1)$ time for each of the $\log N$ levels in $\mathcal{T}$.

To achieve a cache oblivious algorithm we have to consider two different ideas. The first idea is a recursively defined layout based on the van Emde Boas layout for the L_v lists. The second idea is to ensure locality of reference during a search by adding redundant information. This is done by adding *dummy points* to each L_v list to get new lists $\overline{L}_v \supseteq L_v$. We will further ensure that $\overline{L}_r = L_r$ and $\overline{L}_v \subseteq \overline{L}_{\mathit{parent}(v)}$.

The pointers, at each point p_i in $\overline{L}_v$, *left*(p_i) and *right*(p_i) still points down to p_l and p_r in $\overline{L}_{left(v)}$ and $\overline{L}_{right(v)}$ respectively also if it is a dummy point. For the value *leftsum*(p_i) we do not consider the dummy points in $\overline{L}_{left(v)}$. Since $\overline{L}_v \subseteq \overline{L}_{parent(v)}$ and $\overline{L}_v \supseteq L_v$ this does not change the correctness or the time complexity of the algorithm described above.

7.2.1 Memory layout

The cache-oblivious structure consists of three structures X, Y and $\mathcal{L}$. The structures X and Y are binary search trees based on the van Emde Boas layout of the base tree $\mathcal{T}$ and L_r respectively. This insures that the query time for those structures are $O(\log_B N)$ memory transfers. $\mathcal{L}$ is a recursive layout of the $\overline{L}_v$ lists inspired by the van Emde Boas layout.

In the following we will let α denote a constant whose value is 1 at the moment. But when going on to reduce the size to linear space in Section 7.2.3 the value will be $\alpha = \lceil \log_2 N \rceil$[1]. We define the recursive layout using a triple $< C, I, p >$. C is a subtree of $\mathcal{T}$ of height h rooted at node v, I is an y-interval and p is a dummy point to be included in all lists L_u for nodes $u \in C$. We require that p is the lowest point in I and that we have that $|L_u \cap I| \le \alpha 2^h$ where h is the height of C. For the root we have that $C = \mathcal{T}$, I is the interval $(-\infty, \infty)$ and p is the smallest point in S with respect to the y value.

The layout of $< C, I, p >$ is based on the van Emde Boas layout. The subtree C of height h rooted at node v is split up into $s = 2^{\lfloor h/2 \rfloor} + 1$ new subtrees $C_0 ... C_s$. C_0 is the same subtree as C but only the top $\lfloor h/2 \rfloor$ levels. $C_1 ... C_s$ are new subtrees rooted at the leaves of C_0 going from the left to the right. These are of height $\lceil h/2 \rceil$. The layouts of $C_0 ... C_s$ are stored consecutively in memory. The layout of each C_i is also stored recurcivly , see top Figure 7.2, until we have a tree of height 1. For each C_i rooted at v_i we split I into $n_i = |L_{v_i} \cap I| / (\alpha 2^{h/2})$ new intervals $I_i^1 ... I_i^{n_i}$ of size $|I_i^1 \cap L_{v_i}| \le \alpha 2^{h_i}$ for the first and $|I_i^j \cap L_{v_i}| = \alpha 2^{h_i}$ for $1 < j \le n_i$. The recursive layout for each C_i is then $< C_i, I_i^1, p >$ for the first and $< C_i, I_i^j, p^j >$ for $1 < j < n_i$ where p is the original point p from $< C, I, p >$ and p^j is the lowest point in $I_i^j \cap L_{v_i}$.

The space required for X and Y are $O(N)$ [24] and for $\mathcal{L}$, not including the dummy points, the space required is $O(N \log_2 N)$ since each point p is present in the $\overline{L}_v$ lists for each of the $\log_2 N$ ancestors of the leaf containing p. What remains to be shown is that the number of dummy points introduced is $O(N \log_2 N)$.

Lemma 1 *The total number of dummy points introduced in the recursive layout is* $O(N + (N \log_2 N)/\alpha)$.

Proof. For the initial case in the outermost recursion we add the same dummy point p, the smallest point in I, to each node of $\mathcal{T}$ for a total of N times. For a recursive layout of $< C, I, p >$ where we split C in $C_0 ... C_s$ subtrees and C_i has n_i

[1]In the article [5] it is $\lfloor \log_2 N \rfloor$ but it should be $\lceil \log_2 N \rceil$ since the height of the tree would be $\lceil \log_2 N \rceil$

recursive layouts we introduce $n - 1 \leq |L_{v_i} \cap I|/(\alpha 2^{h_i})$ new dummy points into $\overline{L}_u$ for each node $u \in C_i$. We charge them to the points in $L_{v_i} \cap I$ so that we charge each point $O((|C_i| \cdot |L_{v_i} \cap I|)/(\alpha 2^{h_i}) = O(1/\alpha)$ dummy points, where $|C_i| \leq 2^h_i$. For any given point p it can be seen that it will only appear in the lists $\overline{L}_u$ of the nodes u which are ancestors to the leaf storing p. This gives that there are a total of $O(\sum_{i=0}^{\log_2 \log_2 n} 2^i) = O(\log_2 n)$ recursive layouts that a point p is present in. So for each point p we charge it for $O((\log N)/\alpha)$ dummy points or for a total of $O((N \log N)/\alpha)$ dummy points. □

It can be seen that a similar proof can be made that the number of base cases in the recursive layouts, number of lists L_i, is $O(N + (N \log_2 N)/\alpha)$. Every time we have made a dummy point we have also made a list.

Lemma 2 *The total number of base cases introduced in the recursive layout is* $O(N + (N \log_2 N)/\alpha)$.

To get the query performance we first observe that during a search in a layout $< C, I, p >$ we will stay inside the nodes of C when using the *left*(p) and *right*(p) pointers. This is because all nodes in C contains dummy points with $min(I) = y(p)$. The second observation is to note that the size of $< C, I, p >$ is $O(|C|(1 + |L_v \cap I))$ where v is the root of C. This is because p and each point in $L_v \cap I$ can at most be added once to the lists L_u for $u \in C$. This then gives that if C is of height h the size of $< C, I, p >$ is $O(\alpha 2^{2h})$. So if C is of height $\frac{1}{2} \log_2 B$ and α is as previously assumed 1 the size is $O(1)$ blocks. So we can search through a layout $< C, I, p >$ with C of height $\frac{1}{2} \log_2 B$ in $O(1)$ memory transfers. So if the complete layout consists of smaller layouts of heights between $\frac{1}{4} \log_2 B$ and $\frac{1}{2} \log_2 B$ each fitting into $O(1)$ blocks we can traverse the complete layout in $O(\frac{\log_2 N}{(\log_2 B)/4}) = O(\log_B N)$ memory transfers.

Theorem 1 *There exists a cache-oblivious data structure for storing N points in the plane using* $O(N \log_2 N)$ *space, such that a four-sided range counting query can be answered in* $O(\log_B N)$ *memory transfers.*

7.2.2 An alternative layout

There exists a slightly different way to layout the memory than the one described in [5]. This layout is inspired by some of the early works while making [5] and can still be seen in Figure 8 of the paper.

Given the triple $< C, I, p >$ we first split the interval I into $n = |L_v \cap I|/(\alpha 2^h)$ new intervals $I_1 ... I_n$. The subtree C is again split up into $s = 2^{\lfloor h/2 \rfloor} + 1$ new subtrees $C_0 ... C_s$ with C_0 being of height$\lfloor h/2 \rfloor$ and the remain subtrees $C_1 ... C_s$ roted at the leaves of C_0 being of height $\lceil h/2 \rceil$. Each interval I_j is spread out over all the subtrees C_i for $0 \leq i \leq s$. The smallest point p_j in $I_j \cap L_v$ is added as a dummy point in each of the subtrees C_i. For a difference between the two different layouts see Figure 7.2 and for performance differences see Chapter 10.

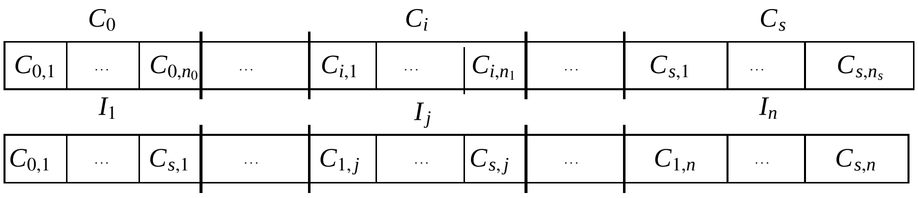
C_0
C_i
C_s
$C_{0,1}$ ··· C_{0,n_0} ··· $C_{i,1}$ ··· C_{i,n_1} ··· $C_{s,1}$ ··· C_{s,n_s}
I_1
I_j
I_n
$C_{0,1}$ ··· $C_{s,1}$ ··· $C_{1,j}$ ··· $C_{s,j}$ ··· $C_{1,n}$ ··· $C_{s,n}$

Figure 7.2: The normal memory layout on top and the alternate memory layout at the bottom.

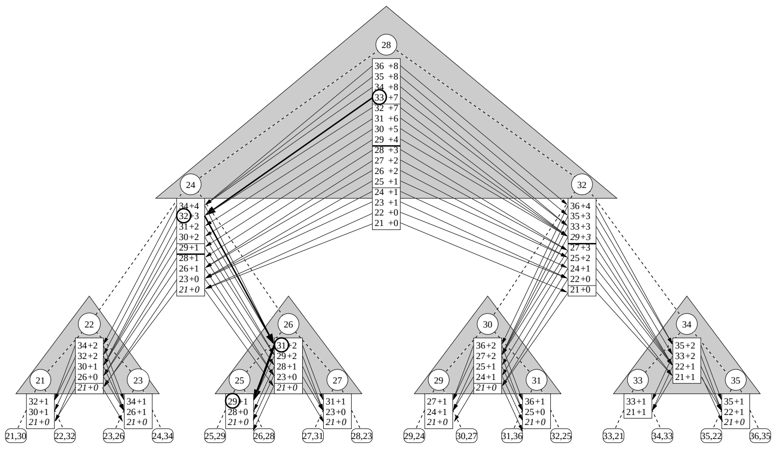

Figure 7.3: The normal memory layout of $\mathcal{L}$ with $\alpha = 2$ to better illustrate the layout of the points P_G.

7.2.3 Linear space

In this section we will continue to work on the memory layout of 7.2.1 to make a structure of size $O(N)$ memory words using bit compression. We will assume that the word size $W \geq \log_2 N$ and that we can perform shifts, addition, multiplication and boolean operations in $O(1)$ time.

In this section we will let $\alpha = \lceil \log_2 N \rceil$. This gives us that by Lemma 2 we now only introduce $O(N)$ dummy points and that the $O(N \log_2 N)$ points in the layout of the L_v lists now consists of $O(N)$ chunks which in the base case are of size $O(\log_2 N)$.

Lemma 3 *Each chunk of a list $\overline{L}_v$ contains left and right pointers to points in at most $O(1)$ different chunks.*

Proof. Consider a chunk c of $\overline{L}_v$ with root at node v with a child u. Let I be the y-interval spanning the chunk c and v is a node of a top chunk C_0 and u is the top node of a bottom chunk C_i. The dummy points in c insures that all pointers to $\overline{L}_u$ stays within I. Since u is a top node in C_i all the points in $L_u \cap I$ are partitioned into chunks of size α except the last chunk which is of size at most α plus a possible dummy point. Because the $O(\alpha)$ pointers in L_v all points to consecutive points in

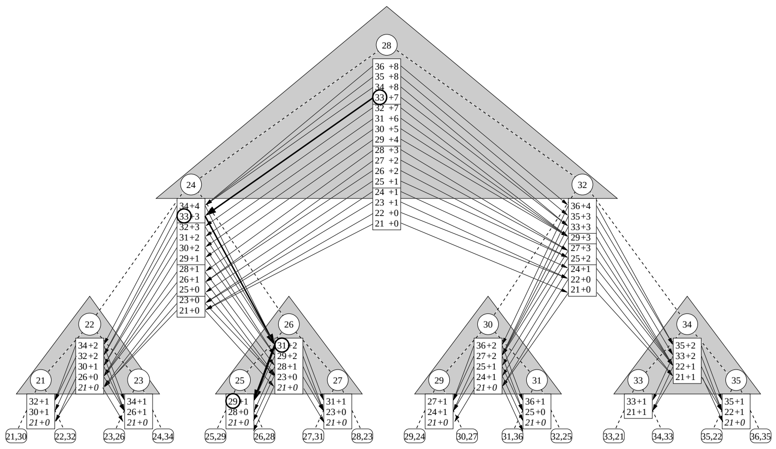

Figure 7.4: The alternate memory layout of $\mathcal{L}$ of the points P_G

L_u it follows that at most $O(1)$ different chunks in L_U can at most be hit from a chunk in L_v. $\square$

What remains to be described is how to make each of the $O(N)$ chunks fit into $O(1)$ space so that we get a total size of $O(N)$ words.

A pointer from a chunk to an other chunk is stored as a pair $< chunck, offset >$ where *chuck* is the chunk being pointed to and *offset* is position of the point being pointed to in its chunk. For each chunk we store three values, $left(p_0)$, $right(p_0)$ and $leftsum(p_0)$ where p_0 is the smallest point in the chunk. For all of the remaining points $p_1, p_2, ...$ in the chunk we store three bits. $\Delta left(i)$, $\Delta right(i)$ and $\Delta leftsum(i)$ where $\Delta left(i) = left(p_i) - left(p_{i-1})$, $\Delta right(i) = right(p_i) - right(p_{i-1})$ and $\Delta leftsum(i) = leftsum(p_i) - leftsum(p_{i-1})$. The values $\Delta left(i)$, $\Delta right(i)$ and $\Delta leftsum(i)$ can be stored in $O(1)$ space since there are at most α points in each chunk each requiring 3 bits for a total of at most 3 words. We also stores explicit pointers for $left(p_i)$ if $left(p_i)$ and $left(p_{i-1})$ points to two different chunks. There are at most $O(1)$ such points as is shown in Lemma 3. The same explicit pointers is done for $right(p_i)$.

If we have a pointer $< c, i >$ to a point p in a chunk c we can calculate $leftsum(p_i)$ as $leftsum(p_o) + \sum_{j=0}^{i} \Delta leftsum(i)$. To calculate $left(p_i)$ we find the highest explicit pointer $left(p_k) =< q, o >$ where $k \leq i$ and we then have that $left(p_i) =< q, o + (\sum_{j=0}^{i} \Delta left(j) - \sum_{j=0}^{k} \Delta left(j)) >$. The same is done to calculate $right(p_i)$.

To calculate $\sum_{j=0}^{i} \Delta leftsum(i)$,$\sum_{j=0}^{i} \Delta left(j)$ and $\sum_{j=0}^{i} \Delta right(j)$ we need to have a function $bitcount(w, i)$ which returns the number of bits equal to 1 in the word w from $1 \leq i \leq |w|$ and is done in $O(1)$ time. *bitcount* can either be calculated by hardware [2] or done with the help of bitmasking, shifting and some arithmetic. For

[2]Processors like the Itanium http://www.nersc.gov/vendor_docs/intel/c_ug/comm1059.htm and the Cray http://ed-thelen.org/comp-hist/vs-cray-res.html supports this in the hardware

64 bit words it can be done like [7] :

```
w -=  (w>>1) & 0x5555555555555555;
w  = ((w>>2) & 0x3333333333333333) + (w & 0x3333333333333333);
w  = ((w>>4) + w) & 0x0f0f0f0f0f0f0f0f;
w *= 0x0101010101010101;
return  w>>56;
```

See also Appendix B for more information on bitcounting.

We have argued that we can compress a chunk into $O(1)$ space and that it can be transversed in $O(1)$ time. To bound how long time it takes to make a complete traversal we note that if we have a chunk $< C, I, P >$ with a root v it satisfies that $|L_v \cap I| \leq \alpha|C|$ which gives us that the total size is $O(|C||L_v \cap I|/\alpha) = O(\alpha|C|^2/\alpha) = O(|C|^2)$ which we have already argued in 7.2.1 gives that we can make a search in $O(\log_B N)$ memory transfers.

Theorem 2 *There exists a cache-oblivious data structure for storing N points in the plane using $O(N)$ space, such that a four sided range counting query can be answered in $O(\log_B N)$ memory transfers.*

7.3 Implementation

In the first part of this section the implementation of the X and Y trees are described. Then in Section 7.3.1 it is described how the basic $O(N \log_2 N)$ data structure is implemented. In Section 7.3.2 it is described how the linear data structure is implemented and how other improvements are made.

The implementation basically consists of 5 different parts. The main part which is managing it all. A class for making Cache-oblivious search trees using the van Emde Boas layout and implicit pointers used for the X and Y trees. Then a class for making the lists with dummy points, left and right pointers and leftsum count. To make those two classes work an implementation of a cache oblivious funnelsort [15] was used which was found on the internet[3] and adapted for use in this implementation. Finally there is a file defining various structures, classes and operators used through out the code.

To make it possible to easily get the data structures again for later use, especially when using small amounts of RAM, all the data structures are stored using mmap [17] and using no pointers only index's into arrays. The reason for this is to make it possible to save this structure for later use without having to generate it all again. This is helpful when testing using small amounts of RAM to get a feel of how the structure performs with respect to memory transfers. Because otherwise the time taken to generate this structure with small amounts of RAM would be considerable. A requirement to make it to be possible to load again is that no pointers to other memory locations are made. This is because it is not likely that

[3]See http://www.diku.dk/forskning/performance-engineering/frederik/ for more information

any given piece of data would be located at the same memory location at a later time. To avoid this all pointers point to an index of an array.

The funnelsort code was basically used as it was. It has been modified slightly so that it can compile with never versions of gcc. The function kSize has been moved to its own separate .cpp and .hpp file to fix a problem with multiple definitions of the function when linking it all together.

The construction of a search tree consists of 3 steps. First sorting the points with respect to either the x or y value. Then a scan through all the points marking the points position in a van Emde Boas layout. This is done by simulating a Dept first traversal of the tree starting at the bottom left. Points are assigned its position only when traveling up to the point in the tree. This would follow the points in its sorted order. The position in the van Emde Boas layout can be calculated from the technique described in [11] getting the position in the tree from the traversal. Finally the points are again sorted but this time with respect to their position in the van Emde Boas layout. The space for the trees are allocated using mmap.

There are two functions used for making querying the trees. The first one *find* is used for finding the point in L_r at which to start from. This is done by a standard binary search. Returning the largest point which is at most as big as the query point. The second function *path* is used for calculating the path down X, returning the result as an array of integers. This is also done by a standard binary search down the tree remembering the path down. If during the path down the actual query point is visited the right child is chosen. The coordinates of the leaf at the end of the path is also returned at the end of the array.

7.3.1 Constructing $\mathcal{L}$

The creation of the structure $\mathcal{L}$ was the hardest one. It works by recursing on the triple $< C, I, p >$ so as to create the layout of the $\overline{L}_v$ lists. For each recursion the input is: the triple $< C, I, p >$, information about $\overline{L}_u$ of the top nodes of the child subtrees of C and some information about where it is in the X tree. It returns the information it has generated about $\overline{L}_v$ where v is the top node of C. It works in two different ways depending upon if the height of C is one or otherwise.

Before the recursion starts a bit of setup has to be done. All the lists are kept in a single array *yp* which is of size $c * n * (\log_2 n)/8$ for some constant c which depend on the type of lists generated and the term $(\log_2 n)/8$ is the word size. The size is an overestimation since no exact result can be calculated because it depends upon how the size of the intervals are distributed which is only known when the actual lists are made. The array is allocated using mmap. There is a structure *treeinfo* which is used to keep track of the progress of the lists at each node in X. The structure is indexed according to the number of the node in a breath first search. This stores where we have reached in *leftsum* and how far down in the lists of $L_{left}(v)$ and $L_{right}(v)$ we are at a node v in the recursion. A list of the leafs is initialized. This list is used to pass to the recursion as the information of what is beneath the complete tree. A sort is then made with respect to y-coordinate in descending order to form the initial *I* over the complete interval spanning all the points.

Then the main recursion starts which consists of two branches. It depends upon if the height of the subtree is 1 or something else. In the case where the height is different from 1 the following pseudo code explains what happens if we do as in Section 7.2.1

```
1 Split the points in I up into the subtrees they come from
2 Add a dummy point where necessary
3 Allocate structure to be filled by bottomtrees and passed to
  the toptree
4 Forall subtrees
5    Forall intervals
6       Call recurcivly on the interval for the subtree
7 Forall intervals of the toptree
8    Call recurcivly on the interval
```

or if we do as described in Section 7.2.2

```
1 Forall intervals
2    Split the points in the interval up into the subtrees they
     come from
3    Add dummy point where necessary
4    Allocate structure to be filled by bottomtrees and passed to
     the toptree
5    Forall subtrees
6       Call recurcivly on the interval for the subtree
7    Call recurcivly on the interval for the toptree
```

In the other case where the height is 1 the following is done:

```
1 Initialize information with respect to where we have reached in
  the child lists
2 Forall(points p in I)
3    insert left(p_i), right(p_i) and leftsum(p_i) into yp
4    If(p==left(p_i)) update left(p_i) to the next and
     decrement leftsum(p_i)
5    If(p==right(p_i)) update right(p_i) to the next
6 Save information of where we reached
```

yp is filled from the end to make the lists be inserted into the correct positions. To make it possible to fill it from the end a global variable is kept. The information of where it has reached for this node in the *X* tree is stored in a global array called *treeinfo* indexed by the nodes position as it is visited in a breadth first search.

To make queries, two additional functions are used. One called *bettermatch* which is used to find the correct point to report in the case that several points have the same coordinates. This should not give rice to additional I/O since the input point is already in memory and hence also the points surrounding it from where the correct point is found. The second function is *count*, the one doing the counting. This is a very simple function. It follows the path down X as indicated by what is returned by *path*. It starts at L_r at the point given by *bettermatch*. From a point p it jumps forward in yp as indicated by either *left*(p) or *right*(p). If we go to the right increments the count by *leftsum*(p). At the bottom of the tree it is checked if the point stored at the end of *path* is inside the query and if so the count is incremented by 1. Finally the count is returned.

The main part is straight forward. First the X tree is made, then $\mathcal{L}$ and finally the Y tree. Then the query is split up into the four subqueries. Each query is then made by first locating the point in L_r from the Y tree. Then finding the *path* down the X tree. And then finally calculating the count by using $\mathcal{L}$.

7.3.2 Compressing $\mathcal{L}$

To compress $\mathcal{L}$ the interval lists are shrunk into one unit. This unit consists of the usual *left*, *right* and *leftsum* but only for the first element. Then there are 2 variables *bleft* and *bright* which are treated as bit vectors in which each of the remaining elements is represented by one bit. A 1 bit if it should go one step further into its child's bitrepresentation and 0 is it should not. There is no *bleftsum* since it is identical to *bleft*. Then there are two more values *offl* and *offr* to indicate an additional offset from where to count from in the child. Finally there are *cutl*, *cutr*, *cutoffsetl* and *cutoffsetr* which are used in the cases where the child interval are split up into 2 subintervals where these variables then points to the second interval. This can happens when an interval points down into an other interval which has been split into two subintervals where the splitting was done at least 2 steps further up into the building recursion of $\mathcal{L}$ from when the intervals were first split.

The construction of is quite simple when we first have the code for the original construction of $\mathcal{L}$. When we make the intervals at height 1 we first initialize all the variables. That is basically just to make it point down to the first *left*(p_i) and *right*(p_i) that it would point down to in the uncompressed version. We then goes through all the points in the interval. When ever we would point to a new lower element in the child lists we just set a 1 bit at the corresponding place for the point in the bit vector for either *bleft* or *bright* depending on weather it was the *left* or *rigth* child. If we go to a new child interval we set the *cut..* variables accordingly to point to this interval as well.

To make the queries we do the same as we have done before. But now we also have to count the preceding bits in the bit vector and add this to the offset to jump into the correct place in the bit vector below as well as subtracting the number of 1 bits in *bleft* from *leftsum*.

It can also be seen that *bright* is equal to the bitwise complement of *bleft* which then makes it possible to skip one of them.

To further reduce the size of $\mathcal{L}$ all the variables can be saved into a bit array where only the relevant number of bits in each variable is saved. To construct what appears as a bit array some functions to save variable length of bits into an array of int's was made. The functions support bit lengths of upto 64 bits[4]. The bits wrap around the int's so as to not waste any space.

Since it is now possible to only store the needed bits of a variable it now makes sense to make the values of variables as small as possible so as to store less bits. We can observe that the index i in the array of where a list is stored is near to where the *left* pointer points to and that the *left* and *right* pointer is even closer to each other. So an obvious thing would be to only store the difference $|i - left|$ and $|left - right|$.

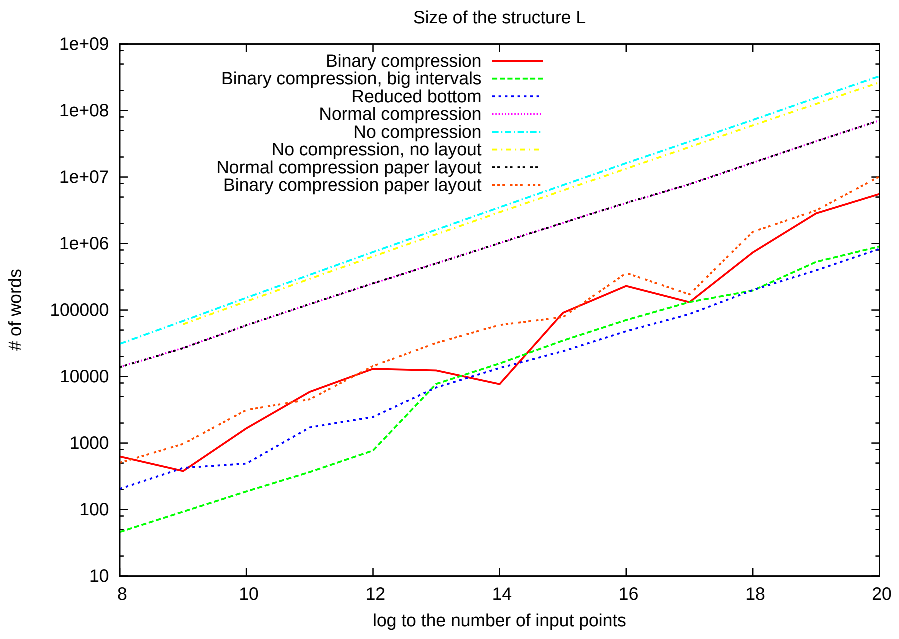

Figure 7.5: The graph shows the various sizes of $\mathcal{T}$ in words of the various implementations made. See Section 10 for more information of the different implementations.

When looking at the consumption of space for making the structure $\mathcal{L}$ at each level of the tree, see Figure 7.6 looking at the binary compression, it is quite noticeable how much space is used by the levels at the bottom compared to the rest of the levels. This is because the number of points in each of the lists at those levels are small compared to at other levels while it is still necessary to save the same amount of information about pointers and leftsum. To minimize the space consumption of those levels we can when making the first recursion chose to divide the bottom subtrees so as to give them a predefined height. Because then it is possible to precalculate all the pointers and leftsums in the entire bottom part of

[4]On a 32 bit architecture 64 bits is the max length of a variable. But on a 64 bit machine 128 bits variables are possible and the functions can easily be changed to support that.

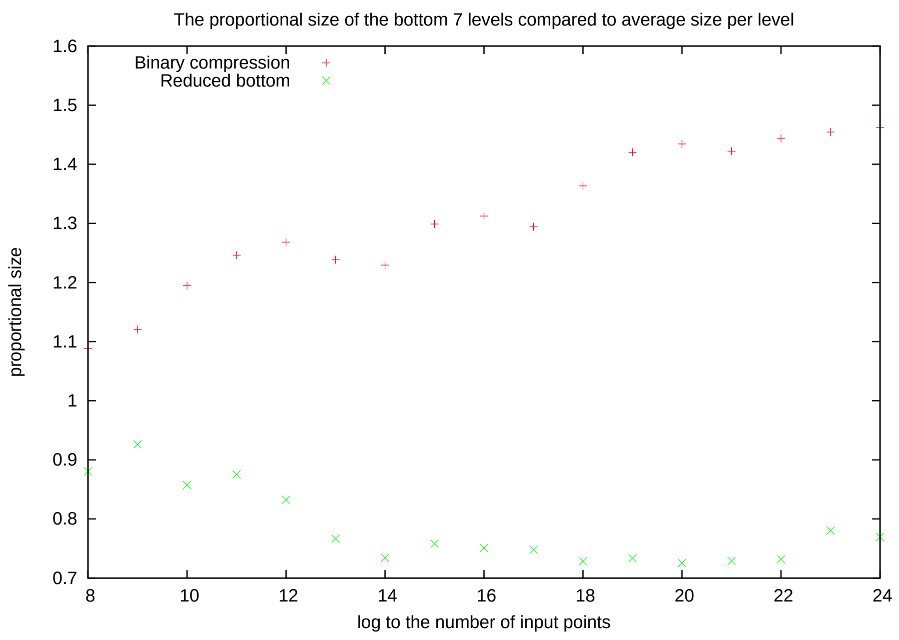

Figure 7.6: The graph shows the proportional size of the bottom 7 levels of $\mathcal{T}$ compared to the fraction of the total number of levels. Binary compression is the normal binary compressed structure as described in Section 7.2.3. Reduced bottom is the structure described in Section 7.3.2

the tree so that the only information necessary to store is *bleft* in each of the list. This is possible because the size of each list then just depends on the height from the bottom. In essence we make a lot of small versions of the structure described in Chapter 5 but storing it using the van Emde boas layout.

Chapter 8

Comparing the three different algorithms

Since the algorithms described in the previous three chapters are based on the same idea as described in Chapter 4 they share some similarities but also have some differences based on the model they are made for. In the following some of those are described.

The search trees for X and Y differs to reflect the different models for which they are developed. Binary search tree for RAM, B-Tree for external and van Emde Boas layout for the cache model.

To achieve the linear size data structure they all use the same technique. Each piece of information about a point is stored using $\log_2(\textit{degree of nodes})$ number of bits. They then group the points together in groups together with some extra information necessary to make meaning of the compressed information. They then make use of the assumption that the word size is $O(\log_2 n)$ bits so that each group of points take up a fixed amount of space.

The real difference in how the algorithms work is in the organization of how the memory of the secondary structure is stored. The RAM model just stores the data consecutively in memory not necessarily storing all information about a specific point in the list at the same location. This results in many apparently random reads in memory when performing queries which is fine when working only in main memory but is very inefficient when working outside main memory. The external model uses the knowledge of the underlying system to make the wide fanout of the trees resulting in shorter paths of length $\lceil \log_B n \rceil$ instead of the usual $\lceil \log_2 n \rceil$ down the trees. This gives fewer lists to visit and hence the reduced number of I/Os. In the cache oblivious model we do not have this information about the system. So here all information on a list is stored together and the possible lists to visit next is stored as close to as possible using the van Emde Boas layout. This then make it so that each time we make an I/O we get some extra information along as well containing the lists that we are going to visit after the one currently at.

As an example of the similarities between the algorithms we can take a look at Figure 5.2, 6.1 and 7.4. The second last line of bits in Figure 5.2 is the same as the bits for $\sum_{v_0}$ ... $\sum_{v_3}$ in Figure 6.1 which are the same as the bits of *bright*, as described in Section 7.3.2, of the lists L_{22}, L_{26}, L_{30} and L_{34} in Figure 7.4

Chapter 9

Experimental setup

In order to make it possible to compare the results all the timing was done in the same way in the three programs. To get the process time used the standard C++ function *clock()* was used. This function has a resolution of 10ms making it necessary to make several queries in a row to get useful results but this does also help giving a better average. It only counts the time that the program spends running on the CPU. This does not include the time spent doing I/Os . To get some sort of idea of how much time is spent including I/O time the function *gettimeofday* is used. This gives the wall clock with a resolution of 1 micro second. Using this time is however not the most reliable since the time other programs have spent doing calculations or doing I/Os is also included in this time. So the number of background processes has to be minimized. In the file */proc/self/stat* the number of major and minor page faults can be found. Major page faults is the number of times that the OS has had to swap memory from the hard drive into the main memory.

To make the test programs run out of main memory fast so that they start swapping the linux kernel was forced to only allocate a total of 41 MB memory. This is done by the kernel boot option $mem = 41m$ where the kernel reserves 9MB of memory leaving 32MB free for programs to use.. The data structures are however generated with max amounts of memory to save preprocessing time and then stored on the hard drive for later use.

The kernel is run at run level 1 in order to minimize the effect that background programs could have on the cache, memory and I/Os.

The machine used to perform the benchmarking was an:
CPU: AMD 64 X2 4200+, 2GHz, 512 KB L2 cache
memory: 2*1024MB PC 3200 Kingston KVR400X64C3AK2/1G
hard drive: 80GB Seagate Barracuda ATA IV
motherboard: Asus A8N-SLI SE
with: Kernel 2.6.17-rc1 64 bit smp, with perfctr[23] 2.6.21 patch applied

Because of an error in the kernel when using an AMD64 processor with smp and 64bit as documented in [22] it was necessary to use a newer kernel like the one used. The error causes the timer to run at twice the speed some of the time.

This makes benchmarking results unreliable as the time results retrieved consists of a mixture of both normal running time and double running time. The error was discovered while making test results for Appendix B. The test results varied a lot from different results but were normally either a minimum value or twice that. To resolve the problem smp support had to be disabled when running benchmarks.

Chapter 10

Benchmarking

In this chapter the benchmarking of the three previously described algorithms are presented. In the first section the results of running the programs where all the data can be contained in the RAM is discussed. In the second section it is shown what happens when the programs run out of memory and has to start swapping memory from the hard drive.

There has only been tested one version of the program for the RAM model and the external model. For the cache oblivious model several different versions has been tested to test various things. The following versions has been tested:

- *Binary compression* using the alternate layout described in Section 7.2.2 using the compression described in Section 7.3.2. This is the main program which uses all the techniques used except for reducing the bottom.
- *Binary compression*, *big interval* as above but where α has been increased. To see the effect of using bigger intervals
- *Reduced bottom* using the alternate layout described in Section 7.2.2 and the compression described in Section 7.3.2 together with the method of collecting the bottom 7 levels into one. The reason for choosing 7 levels is because the bottom trees then contains 128 points which can be stored using two 64 bit integers for $bleft$.
- *Normal compression* using the alternate layout described in Section 7.2.2 and the compression described in Section 7.2.3. To see how the algorithm performs without the use of binary compression
- *No compression* using the alternate layout described in Section 7.2.2 while using no compression techniques. This is to see how the $O(N \log_2 N)$ space algorithm works.
- *No compression*, *no layout* using no layout at all. Basically setting α to infinity. This shows how the basic algorithm performs before the introduction of the memory layout.

- *Normal compression, paper layout* using the layout described in Section 7.2.1 and the compression described in Section 7.2.3. This is to see if there is any difference between the original way of splitting the list up and the one described in [5].

- *Binary compression, paper layout* using the layout described in Section 7.2.1 using the compression described in Section 7.3.2. Same as above.

All the program were tested on input sizes ranging from 2^8 to 2^{24} with the exception of *Reduced bottom* and *Normal compression* which was only tested upto 2^{22} because of memory constraints.

10.1 Running only in RAM

In this section we look at how the programs perform when all the data structures can fit inside the RAM of the computer.

The query time as reported by clock() depending on the input size can be seen in Figure 10.1. This is the processor time excluding time spent doing I/Os. It is easy to see that the external program here takes the longest time. This is probably because of the use of TPIE where it handles its own memory and that it is this overhead that we see here when the input size starts to grow. The RAM program performs best all the way through. When looking at the cache oblivious programs they are initially split up into two groups. There are the ones using no compression and normal compression which make up the best performing group. And then there are the ones using binary compression which has to perform some extra calculations to decompress the data and therefore takes a bit longer. But as the input size grows the two without compression start to head off followed a bit later by the programs using normal compression. So that in the end it is the programs using binary compression that performs the best. To explain this we will have to look at the number of minor page faults shown in Figure 10.2. The graphs look pretty similar with the different graphs starting to grow at the same input sizes. For completnes the time spent for the external program can be seen in Figure 10.3. This is quite a lot more than the rest.

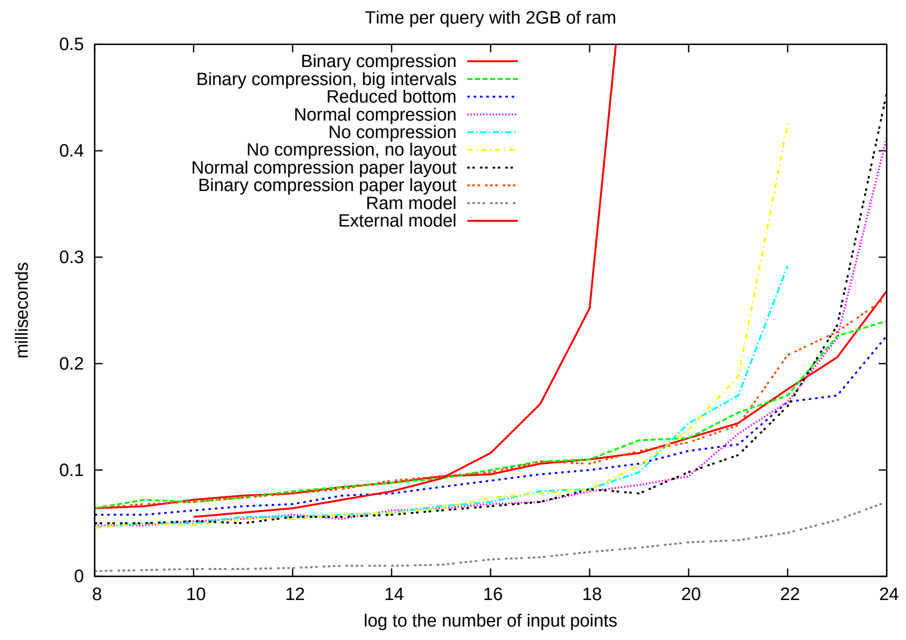

Figure 10.1: Time per query with 2 GB of ram. Zoomed in on the cache oblivious programs.

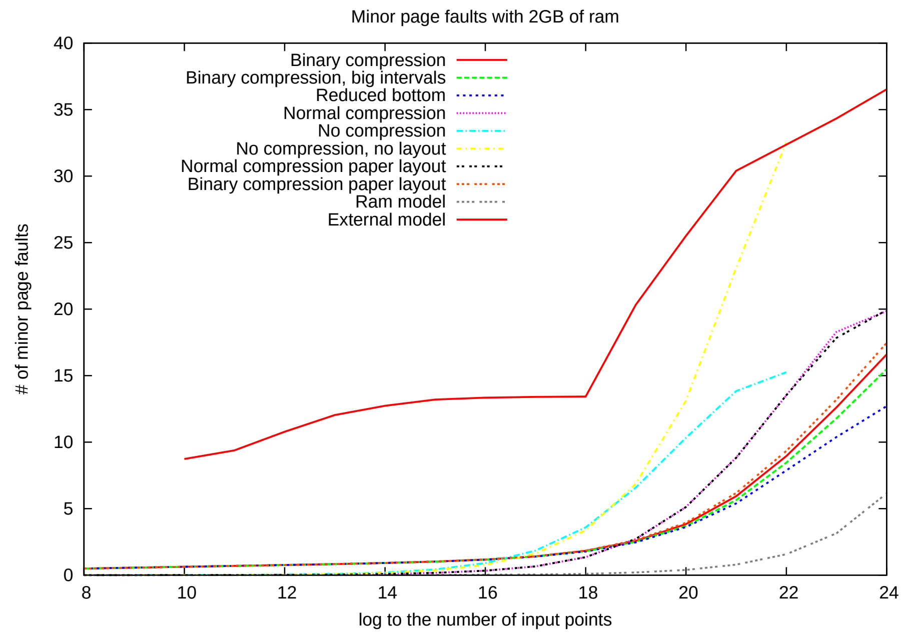

Figure 10.2: Number of minor page faults per query with 2 GB of ram

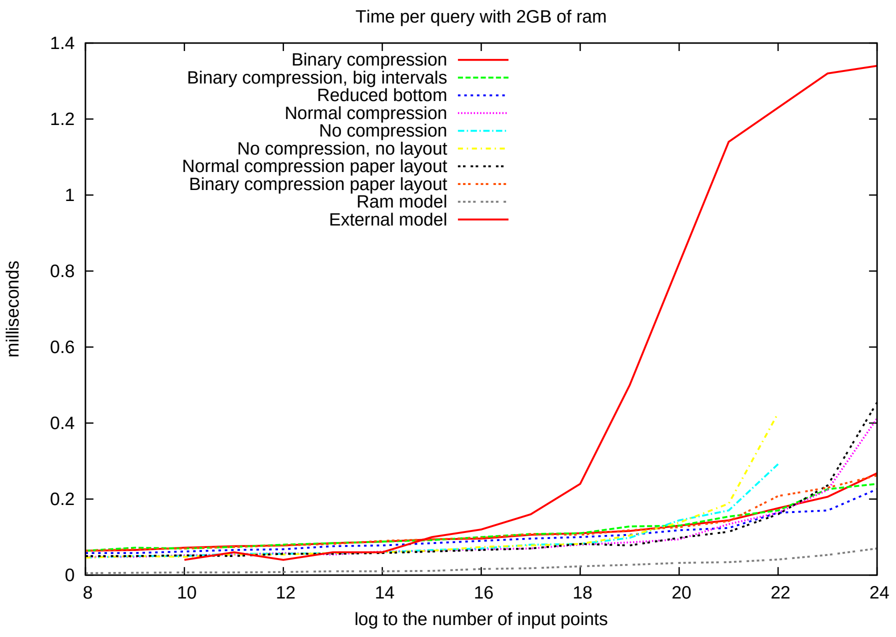

Figure 10.3: Time per query with 2 GB or ram. Showing the full graph.

10.2 Beyond RAM

In these set of benchmarks the computer is limited to only having 32 MB of memory available to run programs. This is done to force the programs into making major page faults.

In Figure 10.4 the number of major page faults can be seen. When the RAM program begins to run out of memory the number of major page faults starts to climb steeply. The external program performs best in this test. This is also expected as this is what the algorithm is designed for. Of the cache oblivious programs there is one that stands out in particular. This is the one with no layout and no compression. Comparing this to the other one with no compression shows that the use of the memory layout describe in Section 7.2.1 is what makes the algorithm perform well. Figure 10.5 shows a zoom of the results to make it easier to distinguish the different programs. From this it can be seen that the programs falls in groups according to the type of compression used. Or in other words the size of the structure $\mathcal{T}$ which is shown in Figure 7.5. The only one to really fall out a bit is the reduces bottom which performs better in the end than the others using binary compression.

In Figure 10.6 the wall time per query is shown. That is the time including I/O time. We can see that this figure is similar to Figure 10.4. There is however something strange with it. It appears as if for some reason the time taken to perform an I/O in the cache oblivious programs takes longer than on the other two programs.

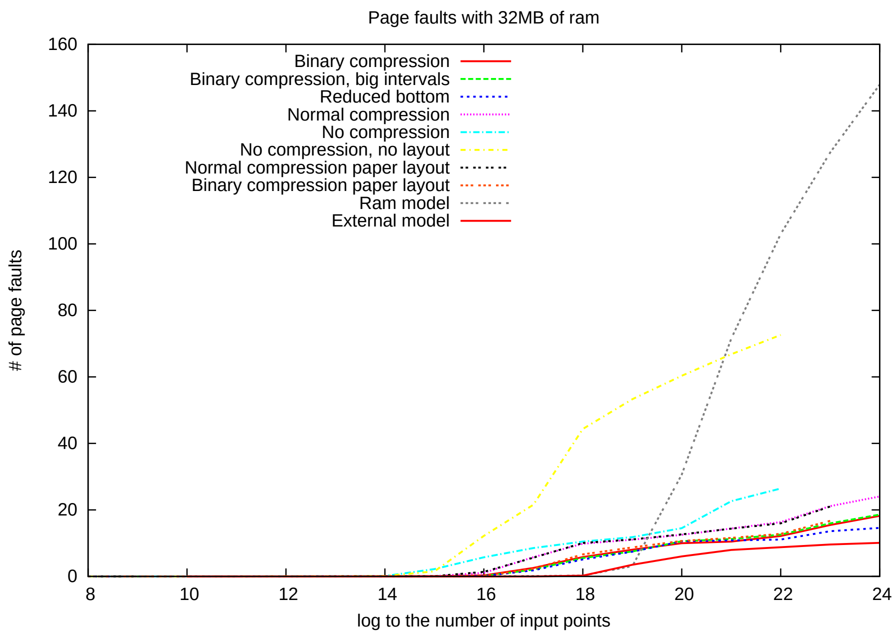

Figure 10.4: Number of major page faults per query with 32 MB of ram

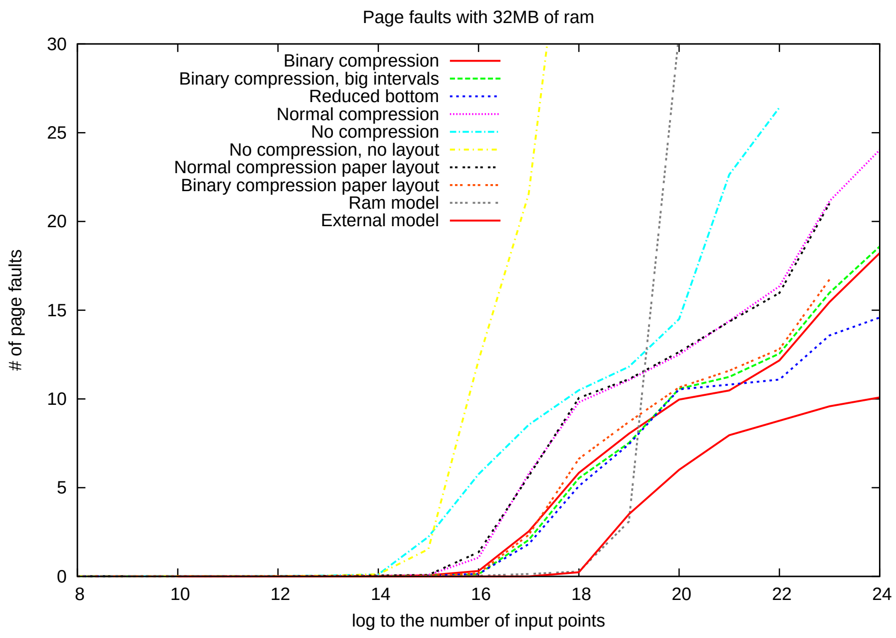

Figure 10.5: Number of major page faults per query with 32 MB of ram. Zoomed in on the cache oblivious programs.

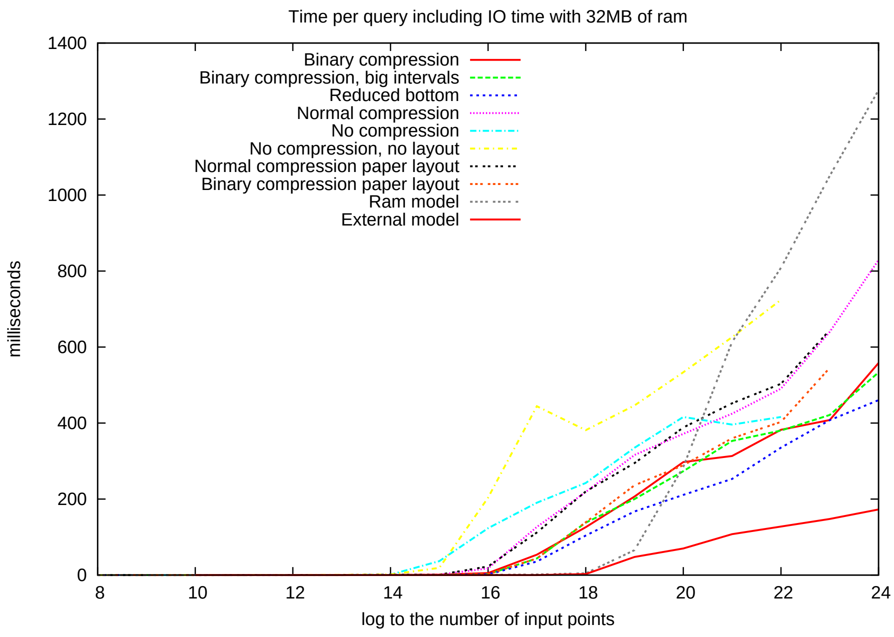

Figure 10.6: Wall time per query with 32 MB or ram.

10.3 L2 cache

In this section we look at the number of cache misses of the programs. The programs were all tested on the default computer and also on a computer with a P4 CPU which got a 1024 KB L2 cache. [htbp]

In Figure 10.7 we can see the number of L2 cache misses. The programs lines up like the results in Section 10.2 with the RAM program with most misses. Then comes the cache oblivious with out compression followed by the ones with normal compression and then binary compression. With the best one being the external program.

It is clear to see when the ram program fill out the cache. This happens when the input size is 2^{15} because there the four arrays of ints take up $4 * 4 * 2^{15} = 524288 bytes = 512 KB$ which is the size of the L2 cache. It also interesting to note how close the bottom version comes to the external program and if the trend from the graph continues would probably beet it.

On the P4 as seen in Figure 10.8 we see the same result. The cache oblivious programs were only tested upto input sizes of 2^{23} because of memory restrictions on 32 bit computers. The cache oblivious programs using binary compression almost perform on par with the external program and in one case even beating it.

It is again easy to see when the RAM program hits the cache limit at an input size of 2^{16} where it takes up $4 * 4 * 2^{16} = 1048576 bytes = 1024 KB$.

When comparing the two Figures 10.7 and 10.8 we can see that the twice as large L2 cache of the P4 does not appear to have the big impact since the results are

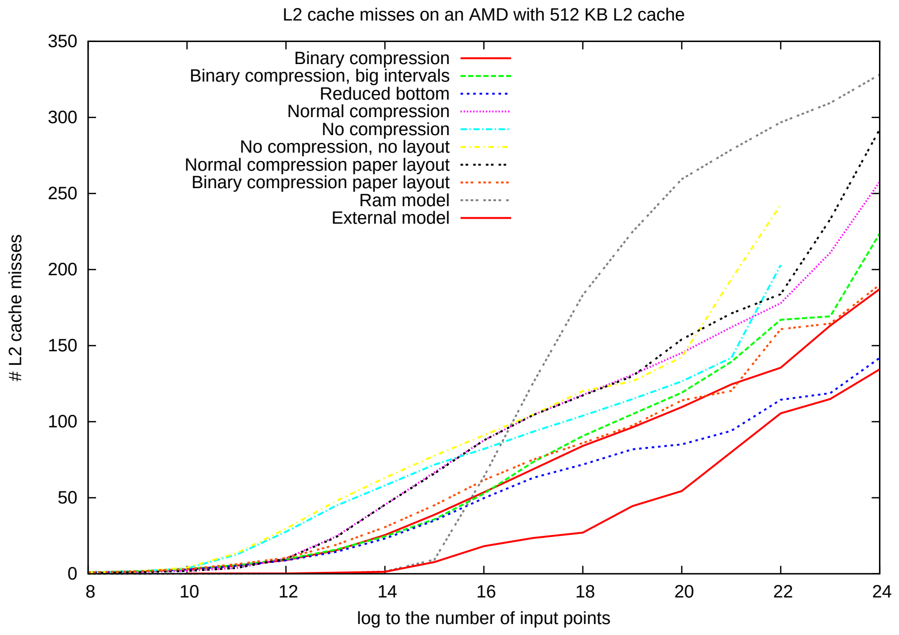

Figure 10.7: Number of L2 cache misses on an AMD X2 with 512 KB L2.

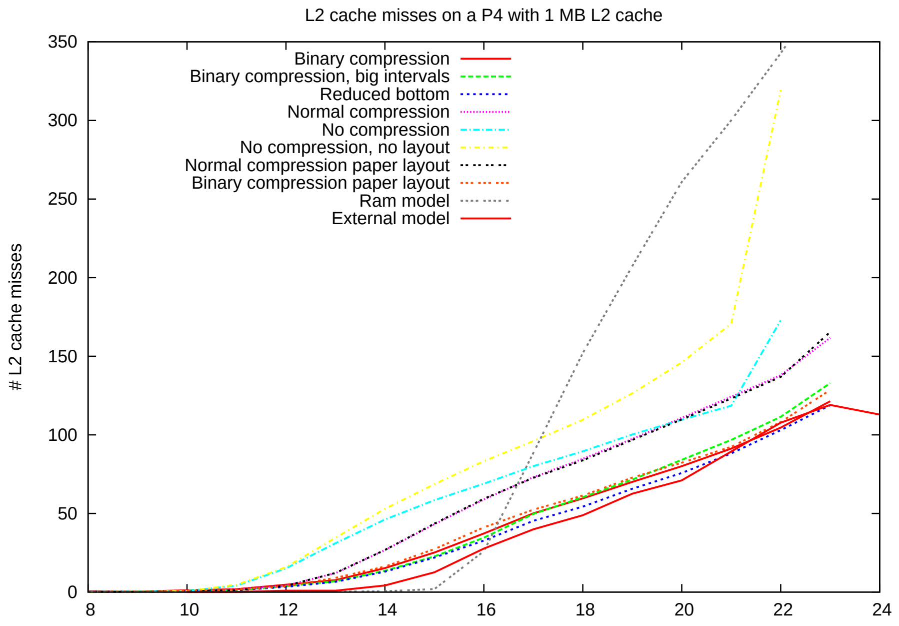

Figure 10.8: Number of L2 cache misses on a P4 with 1024 KB L2.

comparable. But the two non memory optimized programs, the RAM and the no compression no layout seems to be hit harder on the P4 as compared to the AMD.

10.4 Benchmark Conclusions

As expected the RAM program performed well when running on exclucivly on RAM and the external program worked well when working on external storage. The external program is found to also work well on the cache level.

The cache oblivious programs performed well in all cases showing that splitting up large lists of data and laying it out in memory using the van Emde Boas works well on all layers of memory. When looking at memory access it is seen that it is worth while compressing the data to make it fit into as little space as possible. There is not found a significant difference between the two ways of splitting up the lists for the cache oblivious algorithm.

When comparing the graphs of L2 cache misses withe the number of major page faults with 32MB of RAM we see that they have a similar shape. But when performance starts to rise steeply it seems that the major page faults are hit the worst.

Bibliography

[1] P. Agarwal, L. Arge, J. Yang, and K. Yi. I/o-efficient structures for orthogonal range-max and stabbing-max queries.

[2] P. K. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proc. ACM Symposium on Computational Geometry*, pages 237–245, 2003.

[3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, E. Goodman, and R. Pollack, editors, *Discrete and Computational Geometry: Ten Years Later*. Mathematical Society Press, 1997.

[4] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] L. Arge, G. S. Brodal, R. Fagerberg, and M. Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Proc. 21st Annual ACM Symposium on Computational Geometry*, pages 160–169, 2005.

[6] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.

[7] J. Arndt. , http://www.jjj.de/bitwizardry/files/bitcount.h.

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[9] W. Berg. http://www.route50.com/.

[10] C. Beust. , http://www.beust.com/weblog/archives/000160.html.

[11] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[12] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, June 1988.

[13] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[14] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[16] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proc. International Conference on Database Theory*, pages 143–157, 2003.

[17] IEEE. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, http://www.opengroup.org/onlinepubs/009695399/functions/mmap.html.

[18] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proc. ACM Symposium on Theory of Computation*, pages 639–648, 2003.

[19] G. S. Lueker. A data structure for orthogonal range queries. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 28–34, 1978.

[20] C. W. Mortensen. Fully-dynamic orthogonal range reporting on ram.

[21] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time, 03.

[22] osdl. http://bugme.osdl.org/show_bug.cgi?id=3927.

[23] M. Pettersson. http://user.it.uu.se/ mikpe/linux/perfctr/.

[24] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[25] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[26] Q. Shi, J. F. JaJa, and C. W. Mortensen. Space-efficient and fast algorithms for multidimensional dominance reporting and range counting. Technical report, Institute of Advanced Computer Studies (UMIACS), University of Maryland, October 2003.

[27] D. E. Vengroff and J. S. Vitter. Efficient 3-D range searching in external memory. In *Proc. ACM Symposium on Theory of Computation*, pages 192–201, 1996.

Appendix A

Availability

The source code for the programs used in this paper including this paper can be found at:
http://daimi.au.dk/~mol/spec
TPIE can be found at:
http://www.cs.duke.edu/TPIE/
PAPI can be found at:
http://icl.cs.utk.edu/papi/
More recent versions of perfctr, to support more recent versions of the kernel, can be found at:
http://user.it.uu.se/ mikpe/linux/perfctr/

To compile using PAPI use the makefile Makefile.papi. The makefile should be able to detect if you are using 32 bit or 64 bit CPU's. But remember you can only use the external program compiled to 32bit.

The RAM program can be run by the command:
./count < *height* > <*number of queries*> <*-w file*> <*-k file*> <*-papi*>
where *height* is the height of the tree. *number of queries* is the number of queries to be performed. *-w file* is to use a pregenerate *file* for the data structures. *-k file* is to keep the data structure in a *file* after the programs terminates. *-papi* is to use papi for benchmarking as well, must be the fourth argument.

The cache oblivious versions is run by:
./count <*height*> <*number of queries*> <*-f file*> <*-p x1 y1 x2 y2*> <*-wl filel filet*> <*-wx file*> <*-wy file*> <*-kl filel filet*> <*-kx file*> <*-ky file*> <*-papi*> <*-nocomp*> <*-bincomp*> <*-paper*> <*-paperbin*> <*-fast*> <*-fastlong*> <*-long*>

where *height* is the height of the tree. *number of queries* is the number of queries to performe. *-f file* is to load the points in *file*. *-p x1 y1 x2 y2* is to perform a query $Q = [x1, x2] \times [y1, y2]$. *-wl filel filet* is to use *filel* for the L lists and *filet* for handling multiple points at the same location. *-wx file* is to use *file* for the X tree. *-wy file* is to use *file* for the Y tree. *-kl filel filet* is to keep the L lists in *filel* and to keep the handling multiple points at the same location in file *filet*. *-kx file* is to keep the *X* tree in *file*. *-ky file* is to keep the *Y* tree in *file*. *-papi* is to use papi when benchmarking. *-nocomp*, *-bincomp*, *-paper*, *-paperbin*, *-fast*, *-fastlong* and *-long* is to use the given version of the program as specified in Section 10, can only be used one at the time.

The external program be run by:
./test_compressed_range_tree <*number of queries*> <*file*>
where *number of queries* is the number of queries to performe. *file* is the input file with points, if not specified the program will use the file in ./data/uniform.dat.

To compile the external program it is assumed that TPIE is in the folder ../tpie/. This can be changed in the Makefile.

Appendix B

Bitcounting

Bitcounting or also known as population count is counting the number of set bits 1's in a variable. There are various methods of doing that [7] [10] ranging from checking every bit to see if it is set to using a constant number of bitmasks without any conditionals.

I have tested five different methods of doing bitcounting on 32bit integers.

```
inline int bitcount1(int x){
  x -=  (x>>1) & 0x55555555;
  x  = ((x>>2) & 0x33333333) + (x & 0x33333333);
  x  = ((x>>4) + x) & 0x0f0f0f0f;
  x *= 0x01010101;
  return  x>>24;
}

inline int bitcount2(int x){
  x -= (x>>1) & 0x55555555;
  x = ((x>>2) & 0x33333333) + (x & 0x33333333);
  x = ((x>>4) + x) & 0x0f0f0f0f;
  x = x + (x>>8);
  x = x + (x>>16);
  return x & 0x3f;
}

inline int bitcount3(int cur){
  int counter = 0;
  while (cur != 0) {
    counter++;
    cur = cur & (cur-1);
  }
  return counter;
}
```

```
inline int bitcount4(int cur){
  int counter=0;
  for(int n=0;n<32;n++){
    int i=1<<n;
    counter+=(cur&i)>>n;
  }
  return counter;
}

inline int bitcount5(int cur){
  int counter = 0;
  while(cur>0){
    int tmp=cur;
    cur=cur/2;
    counter+=tmp-2*cur;
  }
  return counter;
}
```

bitcount1 and bitcount2 works by using bitmasks. The first three lines are the same. Here they first count set bits in pairs of 2 then 4 and finally 8 bits. The last lines differ in the way that final results is calculated. bitcount1 uses multiplication to add the 4 sums of 8 bits where bitcount2 adds them together and them mask out the result. They both work in constant time. bitcount3 works by counting the number of times that it can remove the least significant bit in the variable. bitcount3 works in $O(setbits)$. bitcount4 works by checking each bit from 0 to 31 to see if it is 1. It has the advantage for the CPU that it does not require any branching since the loop can be unrolled. Runs in $O(n)$. bitcount5 is the basic method of counting. Half the number each time and check if there is a remainder. This also runs in $O(n)$.

The performance of each method can be seen in Figure B.1. The tests were run on an array of 100 million random integers where the numbers where rounded to the desired size. The compiler had full knowledge of the whole program to optimize the various methods. The program was run on the same machine as described in Section 9. This means that it is possible to performed bitcount in only 10 clock cycles using bitcount1.

To compare this to something i made a program with an array storing the values of bitcount from 0 to 2^{18} which could all fit inside L2 cache. It took an average of 5.8 nanoseconds for each loop compared to 5.4 nanoseconds for bitcount1. This basically means that it is not worth while making table lookup for bitcounting since the processor can calculate it faster than the time it takes to look it up in L2 cache.

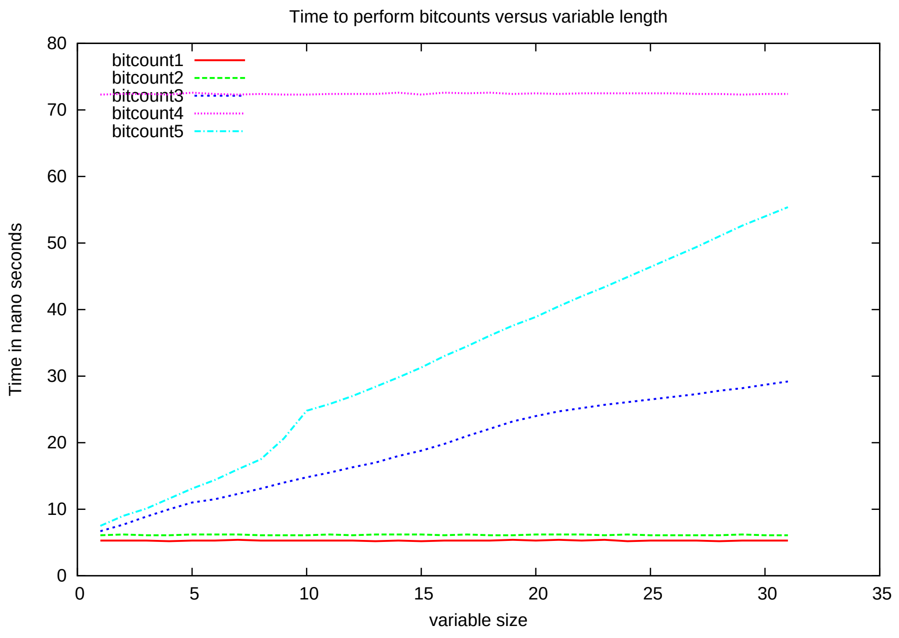

Figure B.1: Time taken to perform various methods of bitcounting.

Appendix C

Source code for range counting in the RAM model

This is the C++ source code for the range counting program for the RAM model. It contains various optimizations compared to the code presented in [12] as explained in Section 5.3 and 5.4.

```
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <papi.h>

#define MAX_INT (int)1<<26
#define RANGE ((int)1<<20)
#define exch(A, B) { int t = A; A = B; B = t; }
#define array(A,B) { int t=B;A=t;}

int n=8;
int lambda=(int)(log((double)n)/log(2.0));
int my=2*(1+(int)floor(log((double)lambda)/log(2.0)));
int M=(int)pow(2.0,my);
int *X;
int *Y;
int *B;
int *C;
int bitcount(int x){
  x -=  (x>>1) & 0x55555555;
  x  = ((x>>2) & 0x33333333) + (x & 0x33333333);
  x  = ((x>>4) + x) & 0x0f0f0f0f;
  x *= 0x01010101;
  return  x>>24;
}

int One(int pos){
  int i=pos/lambda;
  int j=B[i]>>(lambda-pos%lambda);
  int z=bitcount(j);
  if(0<i) z=z+C[i-1];
  return z;
}

int Newpos(int dir, int block, int pos, int width){
  int r;
  if(dir==0)r=pos-n+One(block)-One(pos);
  else r=block-n+One(pos)-One(block)+width/2;
  return r;
}

int Path(int *A, int q){
  int l=0;
  int r=n-1;
  while(l<r){
    int k=(l+r)/2;
    if(q<=A[k])r=k;
    else l=k+1;
  }
  return l;
}

int Cum(int cut, int init, int path, int dir){
  int pos=init;
  int z=0;
  int block=(lambda-1)*n;
  int cur=n;
  while(cur>=2){
    int bit=((2*path)/cur)-(2*(path/cur));
    if((cur<cut) && (bit==dir)){
      z=z+Newpos(1-dir,block,pos,cur);
    }
    pos=Newpos(bit,block,pos,cur);
    cur=cur/2;
    block=block-n+bit*cur;

  }
  return z;
}

void Preprocessing(){
  int fill=0;
  int index=0;
  int step=2;
  int* T=new int[n];
  while(step<=n){
    int l=0;
    while(l<n){
      int r=l+step-1;
      int u=(l+r)/2;
      {for(int k=l;k<=r;k++){
T[k]=Y[k];
      }}
      T[r+1]=MAX_INT;
      int i=l;
      int j=u+1;
      int k=l-1;
      while((i<=u) || (j<=r)){
B[index]=2*B[index];
k=k+1;
if((T[j]<T[i]) || (i>u)){
  Y[k]=T[j];
  j++;
  B[index]=B[index]+1;
  C[index]=C[index]+1;
}else {
  Y[k]=T[i];
  i++;
}
fill++;
if(lambda==fill){
  index++;
  fill=0;
  if(index<n) C[index]=C[index-1];
}
```

```
      }
      l=l+step;
    }
    step=step*2;
  }
  delete[] T;
}

bool LegalQuary(int x1,int x2, int y1, int y2){
  if(x1>X[0] && x2<X[n-1] && y1>Y[0] && y2<Y[n-1])return true;
  return false;
}

void quicksort(int a[],int b[],int l,int r) {
  int i, j, k, p, q; int v;
  if (r <= l) return;
  v = a[r]; i = l-1; j = r; p = l-1; q = r;
  for (;;) {
    while (a[++i]<v);
    while (v<a[--j]) if (j == l) break;
    if (i >= j) break;
    exch(a[i], a[j]);
    exch(b[i], b[j]);
    if (a[i]==v) { p++; exch(a[p], a[i]);exch(b[p], b[i]); }
    if (v==a[j]) { q--; exch(a[q], a[j]);exch(b[q], b[j]);}
  }
  exch(a[i], a[r]);exch(b[i], b[r]); j = i-1; i = i+1;
  for (k = l  ; k < p; k++, j--){exch(a[k], a[j]);exch(b[k], b[j]);}
  for (k = r-1; k > q; k--, i++){exch(a[k], a[i]);exch(b[k], b[i]);}
  quicksort(a, b, l, j);
  quicksort(a, b, i, r);
}

int main(int argc, char **argv) {
  srand(time(NULL));
  bool loaded=false;
  int totalsize=0;
  int numberoftimes=1;
  int x1;
  int x2;
  int y1;
  int y2;
  int n=1<<atoi(argv[1]);
  numberoftimes=atoi(argv[2]);
```

```
lambda=(int)(log((float)n)/log(2.0));
my=2*(1+(int)floor(log((float)lambda)/log(2.0)));
M=(int)pow(2.0,my);
loaded=true;
  totalsize=n*sizeof(int);
X=(int*)malloc(n*sizeof(int));
Y=(int*)malloc(n*sizeof(int));
B=(int*)malloc(n*sizeof(int));
C=(int*)malloc(n*sizeof(int));
X[0]=0;
Y[0]=0;
X[1]=RANGE;
Y[1]=RANGE;
for(int a=2;a<n;a++){
  X[a]=(int)rand()%RANGE;
  Y[a]=(int)rand()%RANGE;
}
x1=(int)rand()%RANGE;
x2=(int)rand()%RANGE;
if(x2<x1)exch(x1,x2);
y1=(int)rand()%RANGE;
y2=(int)rand()%RANGE;
if(y2<y1)exch(y1,y2);
quicksort(X,Y,0,n-1);
clock_t t1,t2;
Preprocessing();
time((time_t *)0);
int i=0;
timeval* wallstart=new timeval();
timeval* wallend=new timeval();
gettimeofday(wallstart, NULL);
t1=clock();
long cum=0;
while(i++<numberoftimes){
  int low=Path(Y,y1)+(lambda-1)*n;
  int high=Path(Y,y2)+(lambda-1)*n;
  int left=Path(X,x1)-1;
  int right=Path(X,x2);
  int cut=n;
  while(((2*left)/cut)==((2*right)/cut)){
    cut=cut/2;
  }
  int z=Cum(cut,high,left,0)+Cum(cut,high,right,1);
  z=z-Cum(cut,low,left,0)-Cum(cut,low,right,1);
```

```
    cum+=z;
    //query MUST be within the bounding box of the points
    x1=((int)rand()%(RANGE-1))+1;
    x2=((int)rand()%(RANGE-1))+1;
    if(x2<x1)exch(x1,x2);
    y1=((int)rand()%(RANGE-1))+1;
    y2=((int)rand()%(RANGE-1))+1;
    if(y2<y1)exch(y1,y2);
  }
  t2=clock();
  gettimeofday(wallend, NULL);
  //necesary because else it cheats when optimising
  printf("%i %lu ",atoi(argv[2]),cum/numberoftimes);
  printf("%.3f ",((t2 - t1)/(CLOCKS_PER_SEC / (double) 1000.0))
    /(double)numberoftimes);
  printf("%.5f ",((wallend->tv_sec - wallstart->tv_sec)*1000+
    (wallend->tv_usec-wallstart->tv_usec)/1000.0)/(double)numberoftimes);
  printf("\n");
  return 0;
}
```

Index