Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Performance analysis and optimization of modern applications on Chip Multiprocessor Architectures

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΑΤΑΛΙΑ ΧΕΡΙΓΚ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής

Αθήνα, Μάρτιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Performance analysis and optimization of modern applications on Chip Multiprocessor Architectures

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΝΑΤΑΛΙΑ ΧΕΡΙΓΚ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Μαρτίου 2014.

Νεκτάριος Κοζύρης Καθηγητής	Νικόλαος Παπασπύρου Αναπληρωτής Καθηγητής	Αριστείδης Παγουρτζής Επίκουρος Καθηγητής

Αθήνα, Μάρτιος 2014

Ναταλία Χέριγκ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο σχεδιασμός συστημάτων παράλληλης επεξεργασίας είναι σημαντικό να γίνεται με τέτοιο τρόπο ώστε να μπορεί να καλύψει τις απαιτήσεις διαφορετικών προγραμμάτων που πρόκειται να εκτελεστούν στο μέλλον. Κάθε σύγχρονο πρόγραμμα έχει διαφορετικές ιδιαιτερότητες που πρέπει να ληφθούν υπόψιν ώστε η παράλληλη εκτέλεσή του να είναι αποδοτική. Στην παρούσα εργασία επεξεργαστήκαμε το PARSEC 3.0 benchmark, το οποίο αποτελείται από διαφορετικά σύγχρονα παράλληλα προγράμματα. Ο στόχος της παρούσας διπλωματικής είναι η μελέτη ενός υποσυνόλου αυτών, ως προς την αξία τους στο σύγχρονο κόσμο και ως προς τους λόγους για τους οποίους η παράλληλη εκτέλεσή τους είναι αποδοτική ή όχι. Δόθηκε βάση στον τρόπο με τον οποίο κάθε εφαρμογή σχεδιάστηκε χρησιμοποιώντας παράλληλο προγραμματισμό όπως επίσης και στις διαφορετικές δομές δεδομένων που αξιοποιήθηκαν. Επιπλέον, μελετήθηκαν ξεχωριστά οι αιτίες που αποτελούν εμπόδιο για την γρήγορη εκτέλεση κάθε προγράμματος. Δοκιμάστηκαν βελτιστοποιήσεις και τροποποιήσεις των προγραμμάτων και των δεδομένων που χρησιμοποιούν, με στόχο την ακριβέστερη συλλογή συμπερασμάτων. Τέλος, μελετήθηκε ο χρονοπρογραμματισμός όλων των εφαρμογών στο ίδιο περιβάλλον. Το συγκεκριμένο μέρος παρουσιάζει ενδιαφέρον, καθώς η συμπεριφορά ενός προγράμματος μπορεί να είναι διαφορετική όταν αυτό εκτελείται απομονωμένα ή σε συνδιασμό με άλλα ταυτόχρονα.

Λέξεις κλειδιά

Παράλληλος προγραμματισμός, χρονοπρογραμματισμός, PARSEC, νήματα, δομές δεδομένων

Abstract

Designing a chip multiprocessor architecture can be very challenging, because it must satisfy the requirements of many different applications that will be executed in the future. Each of the modern programs has different needs in order to be executed efficiently in parallel and have good scaling. In the current project we use the PARSEC 3.0 benchmark suite, which contains many applications from several modern domains. The thesis aims to analyze a subset of these programs extensively and focus on their behavior and problems when running in parallel. We emphasize on the parallelization approaches that are used and in the data structures that are chosen for each program as well. Moreover, we execute measurements in the laboratory and as part of this effort we are changing and in some cases optimizing the implementation of the programs. Last but not least, we are scheduling all the programs of PARSEC to run in parallel at the same time and extract interesting observations. Since the different cores in this architecture are not completely isolated with each other, the behavior of each program depends a lot on the other programs that are being executed simultaneously.

Key words

Parallel computing, Chip multiprocessors, PARSEC, benchmark, threads, data structures

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή Νεκτάριο Κοζύρη για την ευκαιρία που μου έδωσε να εκπονήσω τη διπλωματική μου στο συγκεκριμένο εργαστήριο.

Επίσης, θα ήθελα ιδιαιτέρως να ευχαριστήσω τον Μεταδιδακτορικό Ερευνητή Γεώργιο Γκούμα για την ευκαιρία που μου έδωσε να συνεργαστούμε στα δεδομένα χρονικά περιθώρια που παρουσίαζαν ιδιαιτερότητες. Χάρη στην πολύτιμη καθοδήγησή του, τη στήριξη, τις συμβουλές, τις γνώσεις, την οργάνωση, το κουράγιο και την έμπνευση που μου μετέδωσε κατά τη διάρκεια της εκπόνησης η ολοκλήρωση αυτής της εργασίας έγινε πραγματικότητα. Είναι μεγάλη μου τύχη να μπορώ να έρχομαι σε επαφή με ανθρώπους που θαυμάζω και με επιστημονικά πεδία στα οποία μπορώ να αφοσιωθώ.

Επιπλέον, θα ήθελα να ευχαριστήσω τη μητέρα μου και την οικογένειά μου για την υπομονή και την στήριξη στις διαφορετικές επιλογές μου όλα αυτά τα χρόνια. Τους φίλους μου (που ευτυχώς είναι πολλοί και έτσι δεν τους αναφέρω ονομαστικά) για τις γνώσεις και τις στιγμές που μοιραστήκαμε, για την έμπνευση και για τη συνεισφορά τους να κάνουν το ταξίδι μας μοναδικό και χωρίς τέλος. Τέλος το Μάνο για όλα τα προηγούμενα, για τη βοήθεια και την υπομονή του αυτά τα χρόνια.

Ναταλία Χέριγκ,

Αθήνα, 28η Μαρτίου 2014

Contents

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Contents . . . 11

List of Figures . . . 13

1. Introduction . . . 15
1.1 Benchmarking . . . 15
1.2 Outline of the thesis . . . 15
1.3 Overview of the experimental environment . . . 16
1.4 Factors for scalability limitation . . . 17

2. Applications with good scaling . . . 19
2.1 FREQMINE . . . 19
2.1.1 Overview . . . 19
2.1.2 Description . . . 19
2.1.3 Parallelization approach . . . 22
2.1.4 Measurements and performance analysis . . . 23
2.2 BLACKSCHOLES . . . 25
2.2.1 Overview . . . 25
2.2.2 Description . . . 25
2.2.3 Parallelization approach . . . 25
2.2.4 Data structures . . . 26
2.2.5 Measurements and performance analysis . . . 26

3. Applications with medium achievable speedup . . . 29
3.1 STREAMCLUSTER . . . 29
3.1.1 Overview . . . 29
3.1.2 Description . . . 29
3.1.3 Parallelization approach . . . 30
3.1.4 Data structures . . . 31
3.1.5 Measurements and performance analysis . . . 32
3.2 BODYTRACK . . . 33
3.2.1 Overview . . . 33
3.2.2 Description . . . 33
3.2.3 Parallelization approach . . . 34
3.2.4 Data structures . . . 34
3.2.5 Measurements and performance analysis . . . 36
3.3 FLUIDANIMATE . . . 43

3.3.1 Overview . . . 43
3.3.2 Description . . . 43
3.3.3 Parallelization approach . . . 44
3.3.4 Data structures . . . 45
3.3.5 Measurements and performance analysis . . . 48

4. Co-scheduling PARSEC applications . . . 51
4.1 Introduction . . . 51
4.2 Execution of PARSEC pairs applications: Observations and measurements . . . 51

Bibliography . . . 55

List of Figures

1.1 Dunnington architecture . . . 16

2.1 MapFile structure . . . 21
2.2 FP-tree . . . 21
2.3 Default native input dataset . . . 24
2.4 Customized input dataset . . . 24
2.5 Customized input dataset . . . 24
2.6 Blackscholes Thread execution . . . 26
2.7 Array of OptionData converted to multiple arrays . . . 27
2.8 Native input dataset . . . 28

3.1 Cluster Task - ClusterClassXTask = { CenterTableCountTask, FIxCenterTask, LowerCostTask, CenterCloseTask, SaveMoneyTask } . . . 31
3.2 Streamcluster scaling and time per step . . . 32
3.3 Facility location scaling and time . . . 32
3.4 TBB pipeline . . . 34
3.5 Tracking Model operator . . . 35
3.6 Particle Filter operator . . . 36
3.7 Bodytrack scaling and time per step . . . 36
3.8 Time percentage spent in each step (including the serial parts) . . . 37
3.9 Grayscale to RGB Data . . . 40
3.10 Comparisson of Initial and New Implementations Total Run Time . . . 42
3.11 Step 4 - Calc Weights, Time percentage for Parallel and Serial parts . . . 42
3.12 Scaling of Step 2 (Gaussian Blur), Row and Column Parallel parts . . . 43
3.13 Generic Template Classes . . . 45
3.14 Tree of Thread and Task allocation . . . 45
3.15 Cells and Particles representation . . . 47
3.16 Rebuild step - Using New and Old structure implementations . . . 47
3.17 Fluidanimate time per step . . . 48
3.18 Scaling per step . . . 48
3.19 Time percentage spent on locks . . . 49
3.20 Fluidanimate Run without locks . . . 49

4.1 Time (sec) of Parsec application Pairs . . . 52
4.2 Scheduling of PARSEC applications . . . 53

Chapter 1

Introduction

1.1 Benchmarking

Benchmarking in computer science is the method of running one or more computer programs or operations in order to analyze a computer system. The same set of applications is executed in systems that have different software or hardware characteristics, such as various processor clock frequencies or compilers. The different chip architectures, for instance, can be compared based on the performance results of the executions. Moreover, having a set of workloads from several domains, which require different hardware characteristics in order to achieve satisfying performance, the hardware designer can get information about the behavior of the new architecture in state of the art examples. Of course, designing and implementing programs of a benchmark is a challenging task. Since these applications are part of a test suite and not of a real industry software, the designer of the benchmark has to ensure that they are representative compared to the actual likely workloads. According to the one benchmarking approach, known as "Represenatative Program Selection" [Bien11], the programs of the suite are a subset of all the available ones in the application space. Regarding the second approach, described as "Diverse Range of Characteristics"[Bien11], the suite design involves a bottom – up implementation of the programs, which aim to combine from scratch all the different combinations of characteristics that have to be tested.

1.2 Outline of the thesis

The Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite is a modern benchmark suite composed for multi threaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors[Bien11], [PARS14b]. The suite consists of 13 different programs from multiple application domains, such as computer vision, financial analysis, data mining and media processing. These workloads have different parallel models, machine requirements and runtime behaviors. For example, some of them rely a lot on lock and others on barrier synchronization. Each program has six input datasets provided: test, simdev, simsmall, simmedium, simlarge and native, but only the simlarge and native ones are representative of the real life inputs.

Current project is using some of the applications included in this suite and analyzes them further. Firstly, for each of them we present an overview of the application, focusing mainly on the domain area in which it is part of. We continue by analyzing the algorithm implemented in the program and describing the parallelization approach.

All the programs have been implemented in C/C++ and have been parallelized with at least one of the following APIs: OpenMP[Open14], Threading Building Blocks[Inte14] or POSIX Threads[Blai14]. For each application we choose one of the available implementations and outline the details of it.

Moreover, we analyze the data structures in order to understand better the connection between the algorithm and the actual parallel implementation and enable the detailed analysis of the experimental results. In some cases, we suggest alternative data structures in order to achieve better results at the parallel execution. Lastly, we present the outcome of the measurements that are taken when executing these applications. Most of the applications are split in several steps and we analyze each of them separately as well. This part mainly involves the visual representation of different interesting parameters, such as the achievable speedup and the time spent in each step. For each of them we analyze further the results, present alternative implementations and executions that we tried out and compare them to the initial ones. The applications are divided in two main groups: The one includes those that the achievable speedup is almost the ideal one and the other includes those with worse scaling.

1.3 Overview of the experimental environment

All the experiments of the current project were executed in the "Dunnington" platform, which is represented in figure 1.1. Dunnington is a Chip multiprocessor machine with 24 cores (Intel® Xeon X7460® @ 2.66GHz) in total, grouped in four sockets. Each socket has one CPU and all of them share one main memory of 30GB (28811766 KB). Regarding the caches, each core has a Level 1 cache of 32KB and shares with one more core a Level 2 cache of 3MB. Additionally, all the cores of one socket share a Level 3 cache of 16MB. The four sockets communicate with each other through a front-side bus (FSB). All the programs are part of the version 3.0 of PARSEC benchmark[PARS14a]

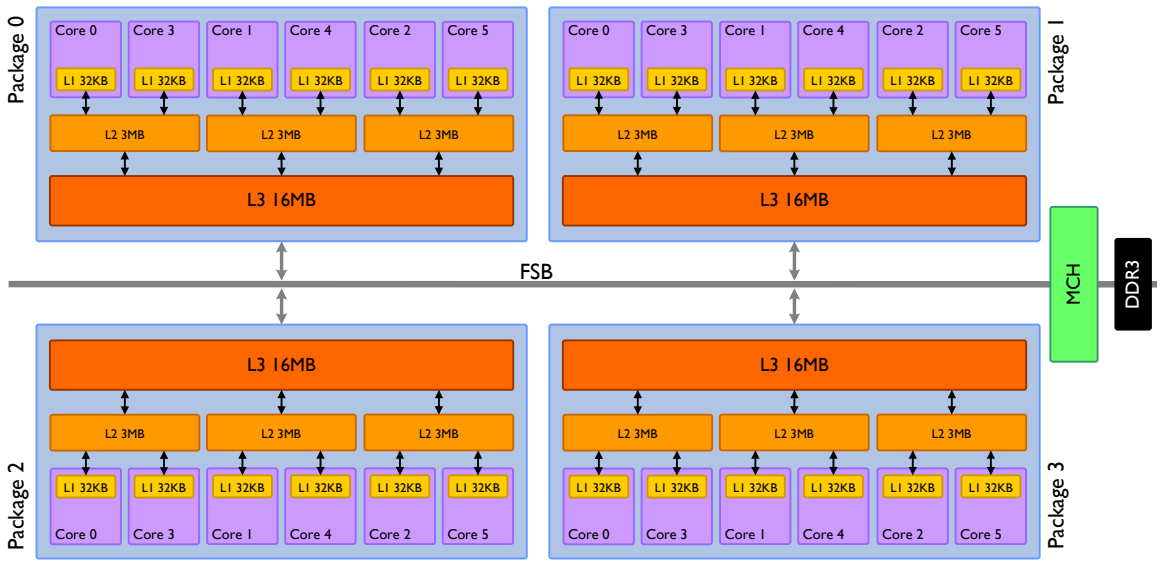

Figure 1.1: Dunnington architecture

and have been compiled with gcc version 4.4.7 (Debian 4.4.7-2). Moreover, all the measurements of the current project use the native input included in the PARSEC suite or some customizations of it which are further defined in the corresponding chapters. We start the experimental execution by creating only 1 thread and repeat it until we have 24 threads. In some cases, we try to get information when the total available threads are more than the number of Dunnington's cores and extract the related information.

1.4 Factors for scalability limitation

There are many reasons for having a multi threaded application with medium or bad achievable speedup in a specific execution environment, or even in different ones. Each of the programs that are further analyzed in the current project are affected by at least one of these problems.

- Serial parts of an algorithm

  There is always an overhead in a parallel program introduced by its serial parts that cannot be parallelized. If the serial algorithms need a lot of time to be executed or are repeated in many places of the application they will impact the performance of it. According to the Amdahl's law, if the best serial time achieved by a program is T_s and f is the portion of the program that cannot be parallelized, the parallel time needed given p processors is described in equation 1.1. Equation 1.2 describes the achievable speedup when using p threads[NTUA14].

$$T_p = fT_s + \frac{(1-f)T_s}{p} \tag{1.1}$$

$$S = \frac{T_s}{T_p} = \frac{1}{f + \frac{1-f}{p}} \tag{1.2}$$

- Load Imbalance

  All the available threads must have ideally equal workload to execute. However, this is not occurring in some programs. Some algorithms can depend a lot on the input and the implementation of the program cannot be generic enough to ensure that the load distribution will be always equal among the threads. In other cases, there has to be a master thread executing more work than the rest of the threads.

- Synchronization

  In some cases the different threads have to exchange data, access global memory elements at the same time or perform an action after a previous one has been already executed by all the active threads. These programs have a high communication intensity and their performance is reduced due to the barrier or lock synchronization that they are using. Frequent I/O operations are also introducing many synchronization points in the application.

- Contention for memory

  When a thread has to access often the main memory, which is shared among the available cores, a bottleneck is introduced. The communication between the CPU and the memory is happening through the memory bus which may have the data of other threads at the same time. Programs must be designed in a way to optimize the cache utilization of each core in order to reduce the communication need with the main memory. For example, the variables that a single thread is reading and writing must be stored in continuous places in the memory so that they can be copied to the cache with a single access.

- Thread management time

  The time spent just on creating the threads can be a drawback when having to parallelize an algorithm which does not require a lot of time in total to be executed. As a result, in some cases it is preferable not to parallelize a program or use less cores than the available ones.

Chapter 2

Applications with good scaling

2.1 FREQMINE

2.1.1 Overview

The freqmine application is part of the domain of data mining. It is based on the Frequent Pattern Tree (FP-tree) data structure and its Frequent Pattern Growth (FP-Growth) algorithm [Han00]. It is one of the methods used for frequent itemset mining (FIM) [Grah03] as part of the association rule mining (ARM) process. Given a transactional database the ARM process aims to extract interesting rules from it based on the combination and frequency that the items of the database have. Freqmine aims to give a solution to the full extent of the problem of data mining across different domain areas. Protein sequences, market data, log analysis [Bien11], machine learning, social media, music classification into genres are just a few areas that use data mining algorithms to achieve better results. Freqmine used the example of data mining in web html documents. Given different web sites for news from several countries, freqmine algorithm can be applied and extract the braking news that are worldwide interesting.

2.1.2 Description

The input database consists of the different transactions, which are in our case the different web sites (by excluding the html tags and the very common words, such as “a”, “and”, “the” etc). Each word of the web site is represented by one item in the corresponding transaction. For example, an article about the new mobile phone would lead in having probably following items in a transaction: “camera”, “network”, “CPU”, “Bluetooth”. Additionally, the transaction would include items that may appear only in the specific article and are not very generic: “WAV ringtones”, “H.263 player”, “MIDP emulator”.

Freqmine is divided into three steps. In the first step, the program reads this database and constructs the FP-tree with the most frequent items. An FP-tree is a compact data structure that contains all the information needed. In the next steps the program traverses and manipulates the FP-tree according to the FP-growth algorithm in order to perform the data mining and to extract the desired information about every item.

For example, an instance of a database that was generated after parsing 7 simple web sites could look like the one in table 2.1, in which items “a”, “b” could represent the words “Camera” and “GPS”. If the minimum item frequency is set to 2, all the items that appear less than this number in all the transactions will be excluded from further analysis. Item “m” in table 2.1 is part of a single transaction and hence not important for extracting any information out of it.

- Step 1: Parse database, Build FP-tree header

TID	Items
1	{b,a,c}
2	{a,h,e,b,g,f}
3	{d,b,a,e}
4	{g,b,i}
5	{m,d,f,a}
6	{g,c,a}
7	{c,h,q}

Table 2.1: Database

The main task of this step is to parse the input database, discard the items that appear in the database fewer times than the desired minimum item frequency and store permanently in decreasing order those that are part of enough transactions. Each item is represented by an integer. In order to store all of them we use the MapFile structure, which consists of several MapFileNode data structures as it is depicted in figure 2.1. Each node contains an integer array with all the item data and depending on the size of a transaction it can store the information of one or more transactions. The MapFile structure can be traversed either as a linked list or as an array of MapFileNodes. This structure is used heavily by the application and it is very efficient for distributing the workload equally among the threads. Afterwards we construct the arrays that will have the information about each item's number of occurrences and ranking.

The overview of the information that this step produces can be represented in the tables 2.2 and 2.3.

1. Frequency array

Item	Frequency
a	5
b	4
c	3
d	2
e	2
f	2
g	2
h	2

Table 2.2: Frequency array

2. Database with the sorted items that remained after the filtering

TID	Items
1	{a,b,c}
2	{a,b,e,f,g,h}
3	{a,b,d,e}
4	{b,g}
5	{a,d,f}
6	{a,c}
7	{c,h}

Table 2.3: Database

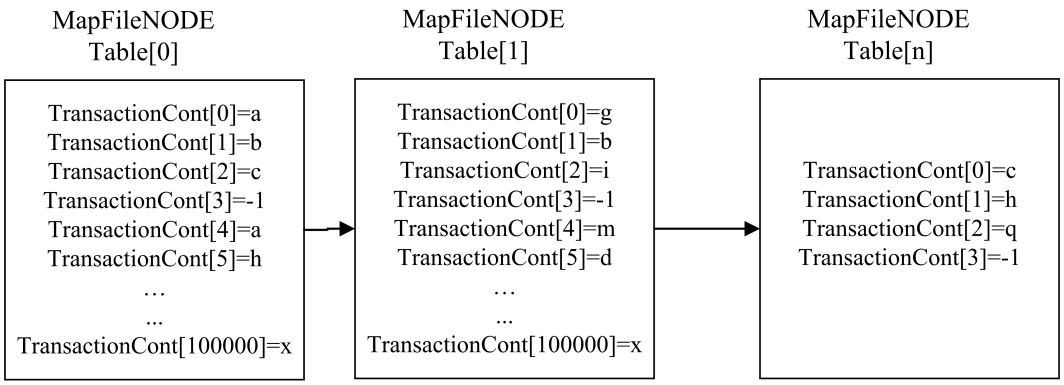

Figure 2.1: MapFile structure

- Step 2: Construct prefix tree

  The aim of this step is to construct the FP-tree by parsing the MapFile data structure that was constructed in the previous step. Each transaction of the database is represented by a path in the FP-tree, where a node contains the number of an item. Transactions that have the same prefix (same items in the beginning) will share the initial path in the tree. Each node maintains a counter that gives information about how many transactions are related to the sub-path that starts from the root until that node. Moreover, the FP-tree contains several right sibling pointers that connect nodes with the same items with each other.

  We distinguish the 2^{16} itemsets that consist of the 16 most frequent 'hot' items. Another MapFile structure is used in order to store the information about the hot and non-hot items of each transaction in a way that will enable better workload distribution. The main goal is that the threads have an equal number of non-hot items to manipulate, instead of an equal number of transactions. The number of different items with low and medium frequency that are included in a database can be really huge and all of them will allocate at least one node in the FP-tree. Another data structure called Fnode is used to create the FP-tree. Each node in the FP-tree contains the item number, the counter that was mentioned above, a pointer to the next node of the path and a right sibling pointer that will potentially point to another node of a different path that has the same item. The final FP-tree will have the 2^{16} hot itemsets as roots, which can be accessed directly, and each of them contains all the Fnodes needed.

  Figure 2.2 represents a simplified version of the FP-tree and does not take into account the hot itemsets, since the database of the example is too small. The counters of each node have the

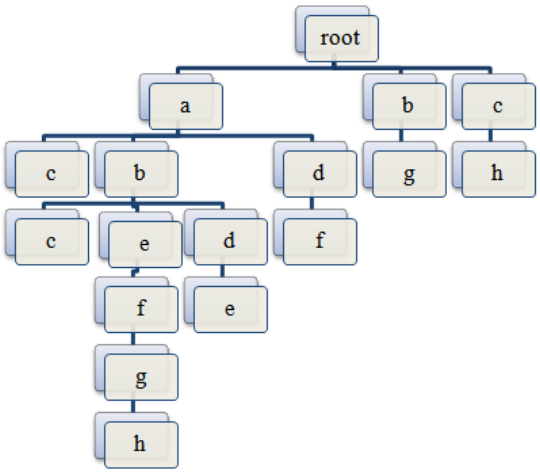

Figure 2.2: FP-tree

following values

Node	Counter
A	5
b (of a-b path)	3
c (of a-c path)	1
d (of a-d path)	1
c (of a-b-c path)	1
e (of a-b-e path)	1
d (of a-b-d path)	1
f (of a-d-f path)	1

Table 2.4: Node Counters

- Step 3: Mine data

  The last step aims to perform the data mining for each item on the FP-tree using the FP-growth algorithm. Firstly, we have to define the conditional FP-tree for an item x: it is the subtree of the initial FP-tree which contains only the transactions that contain the item x without the item x itself. From the remaining nodes we keep those whose counter has a value above a specific threshold. The conditional FP-tree will contain all the frequent itemsets that end with x. In the current algorithm, we have to find, for each item, all frequent itemsets that end with that item. We distribute the different items among the threads and for each item we perform the data mining. We start by creating the conditional FP-tree for the current item x and then recursively create the conditional FP-trees for the itemsets that contain the x until we have a tree just with a single path.

  The frequent itemsets of the example are the following ones.

Suffix item	Frequent Itemsets
h	{h}
g	{g},{b,g}
f	{f},{a,f}
e	{e},{a,b,e},{a,e},{b,e}
d	{d},{a,d}
c	{c},{a,c}
b	{b},{a,b}
a	{a}

Table 2.5: Frequent itemsets

2.1.3 Parallelization approach

The current application is a data-parallel program and it has been parallelized with the use of OpenMP. Each step contains several parallel-for regions, for which dynamic scheduling is always chosen. Each region performs one of the following tasks:

- Initialize data. Most of the arrays in the program have one dimension on which the array is split, with one region belonging to each thread, so that data sharing and locks are avoided. Each thread has to initialize the part of these arrays that it possesses.

- Load and store data. This involves the different tasks that were described in the above steps. In some loops the threads will manipulate equal number of nodes (MapFileNode) and in other similar number of items. The program is designed in a way to distribute the workload as well as possible.

2.1.4 Measurements and performance analysis

We have taken measurements using the default parameters of the native input [Lucc04]. The highest speedup was achieved with 24 threads and it was 15,2175.

The main observation is that freqmine is input-dependent. As mentioned above, each thread will edit transactions that have a similar total number of non-hot items. However, one transaction cannot be split among different threads. This means that if we have a database where the size of the transactions varies a lot, the achievable speedup can also vary a lot. Moreover, the creation and maintenance of the right sibling pointers is also input-dependent. In one scenario, we may only have a few different items that are used in most of the transactions, meaning that there will be many right sibling pointers and each conditional FP-tree will have many different paths. In another scenario, we can have many items appearing relatively rarely in the transactions, meaning that there will be just a few pointers and small conditional FP-trees during the mining. Both scenarios can be related to real databases. The FP-tree can compress the data a lot or it may need more space than the actual input database if the path sharing percentage (common itemsets between transactions) is very low. For these reasons, we manipulated the input dataset to experiment the different behaviors.

The steps 1 and 2 do not achieve a very good speedup. This is understandable, since most of the time is spent for I/O operations in order to read the input file, and for array initializations. However, both steps together are executed in only 6,43% of the total running time, so we do not consider their mediocre scaling as a problem.

We present the measurements that were taken using the default input dataset and parameters and manipulated versions of them.

Figure	**Frequency Threshold**	**Input Database File**	**#Items (*10^6)**	**#Different Items**	**#Transactions (*10^3)**	**Step 3 Speedup (24 Threads)**	**Input size (MB)**
2.3	11000	webdocs_250k	21	829	250	15,2175	210
2.4	10000	webdocs_250k	22	923	250	16,2363	210
2.4	11000	webdocs_138k (webdocs_250k wtih transactions that have at least 200 items)	9,8	501	65	11,032	142
2.5	80000	webdocs_420k (initial webdocs with items that have value below 1000)	94	441	1692	15,86	410
2.5	60000	webdocs_420k	100,5	535	1692	16,7337	410

Table 2.6: Database

As we can see there are changes in the speedup every time we change the input, but the overall scaling is very good. The main reasons for that are the following:

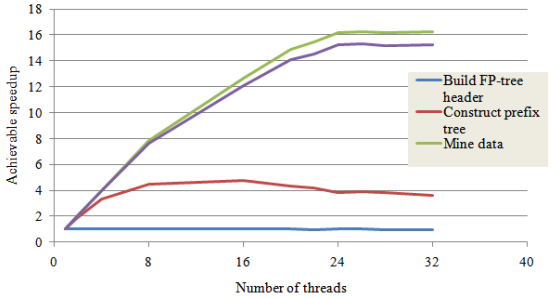

Figure (2.3.1) Scaling per step

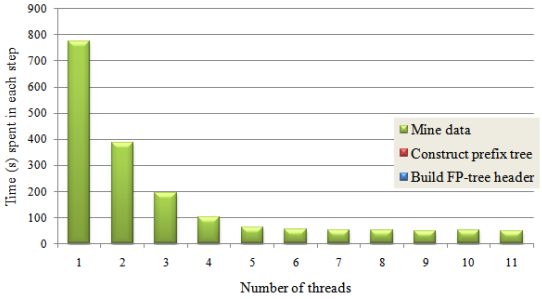

Figure (2.3.2) Time (sec) per step

Figure 2.3: Default native input dataset

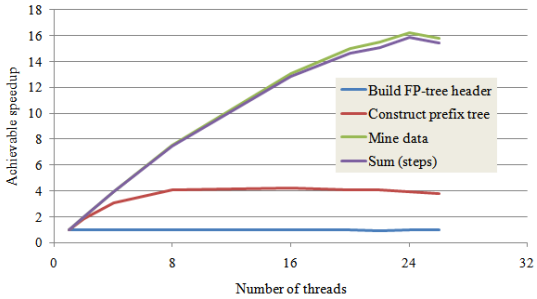

Figure (2.4.1) Scaling per step

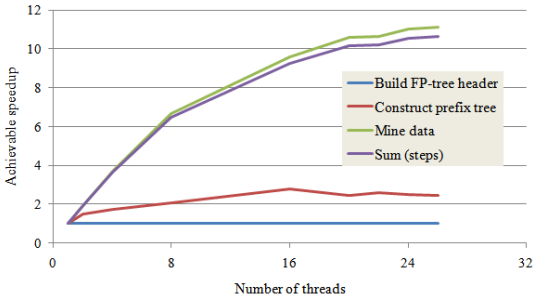

Figure (2.4.2) Scaling per step

Figure 2.4: Customized input dataset

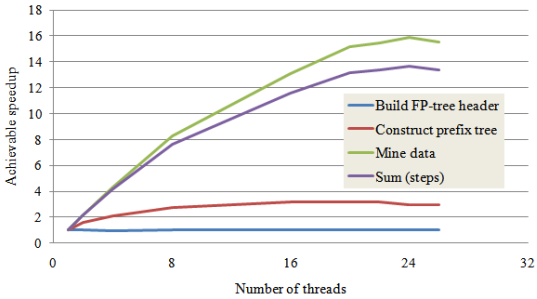

Figure (2.5.1) Scaling per step

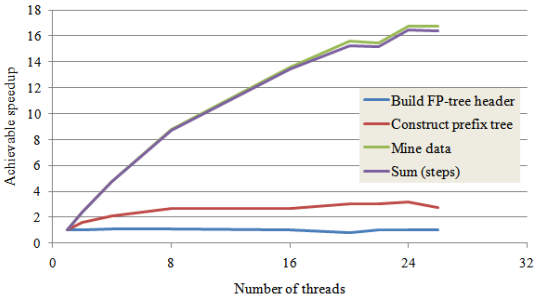

Figure (2.5.2) Scaling per step

Figure 2.5: Customized input dataset

- Good workload distribution among threads and each of them stores data in its own memory elements.

- Not much time is spent in lock contention.

- Most data that one thread is manipulating in every parallel loop is stored in continuous places in the memory which leads to efficient use of the cache.

2.2 BLACKSCHOLES

2.2.1 Overview

The Black-Scholes application is part of the financial analysis domain. The Black-Scholes model is a well-known mathematical model, used for determiming the fair price of European put or call options [Blac73]. The price is calculated based on the risky and the riskless assets of the market. These parameters are mainly the value of the given stock, the risk-free interest rate, the option's strike price and volatility and the time to the option's expiry. The core implementation of the model is based on the Black-Scholes partial differential equation, which describes the price of the option over the time [Inve14], [Kari03].

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{2.1}$$

2.2.2 Description

Current benchmark implements the numerical computation of the Black-Scholes formula for multiple options and the calculated price either refers to a put or a call option. By using the Cumulative Normal Distribution Function the program calculates the option prices according to the equations 2.2, 2.3.

$$P_{call} = sptprice^* \cdot CNDF(d_1) - strike^* \cdot e^{-rate^* \cdot time} \cdot CNDF(d_2) \tag{2.2}$$

$$P_{put} = strike^* \cdot e^{-rate^* \cdot time}(1 - CNDF(d_2)) - sptprice^* \cdot (1 - CNDF(d_1)) \tag{2.3}$$

$$d_1 = \frac{(rate + \frac{volatility^2}{2})^* \cdot time + \log(\frac{sptprice}{strike})}{volatility \cdot \sqrt{time}} \tag{2.4}$$

$$d_2 = d_1 - volatility \cdot \sqrt{time} \tag{2.5}$$

2.2.3 Parallelization approach

The application can be executed with the use of Pthreads, TBB threading model or OpenMP. In the current project we emphasized in the OpenMP implementation [Open14].

For a given input the program has to compute numOption different prices. The parameters and result of each price do not have any dependency with the rest and this makes the division of the problem across the available threads very easy. The design of the parallelization is following the data-parallel model. Given nThreads, each of them has to execute the equation numOptions/nThreads different times and store the output in numOptions/nThreads different elements of a shared array.

The above computation is repeated a constant number of times, which is defined as 100 in the current program.

In order to enable the parallel execution of the Black-Scholes, the `#pragma omp prarallel` for work – sharing construct is used. Since OpenMP is a shared memory programming model, the output array for storing the pricing results is defined as a private variable in order to avoid conflicts during the repeatable stores.

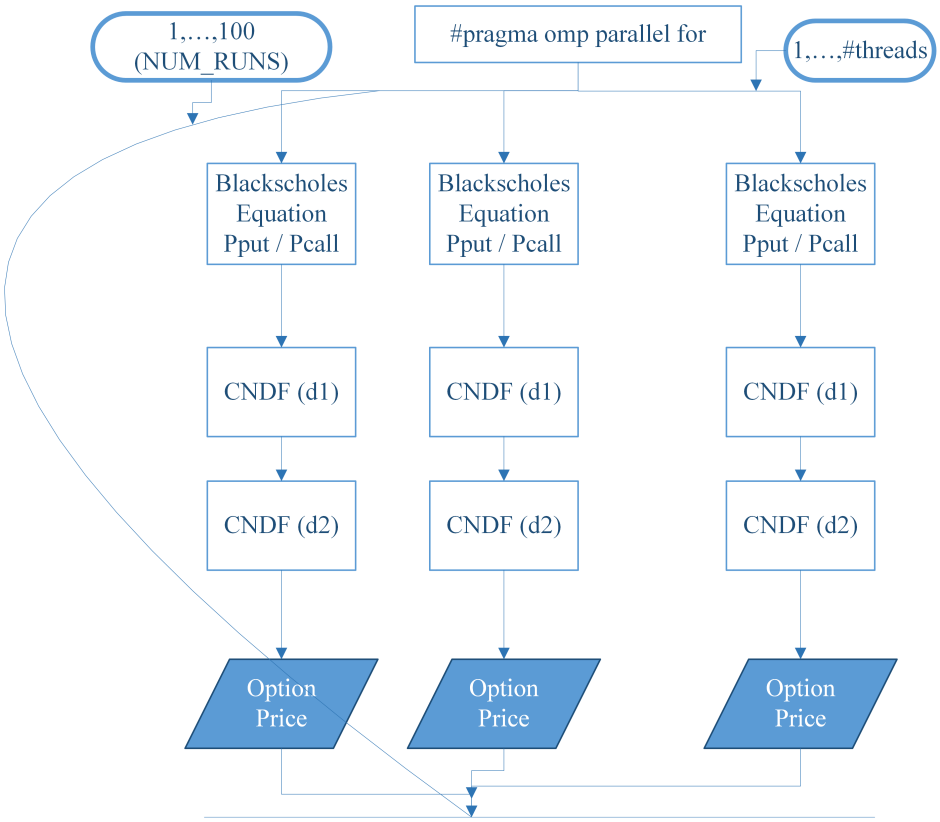

Figure 2.6: Blackscholes Thread execution

2.2.4 Data structures

Current implementation uses arrays in order to store the values of the input file. For each option there are 9 values that need to be read and stored in a specific struct, defined as OptionData. However, for the current computation not all of them are needed to be used by the threads. After storing all the values in an array with numOptions different OptionData structures, the ones that will be used repeatedly are copied to separate arrays. The final arrays are allocated in continuous places in the memory and all the elements are aligned with the cache line size.

Given a cache line size of 64 bytes, each thread will need to copy the values to its cache for each float array every 16 iterations.

2.2.5 Measurements and performance analysis

We executed the blackscholes application in the Dunnington, using from 1 up to 32 threads. For all of them we used the native input, which consists of 10,000,000 different options and the file size of it is only 632 MB.

The application has an ideal achievable speedup of 23,73 when 24 threads are being used. The main reasons for this behavior are the following ones

- The workload is small and requires small amount of memory. Moreover, as mentioned above, the memory needs to be accessed every 16 iterations of the Black-Scholes computation. For each price computation one thread needs only to read from 5 global float variables, one integer variable and store the result in one float element. As a result, the off-chip bandwidth requirements are very limited and they do not impact the performance.

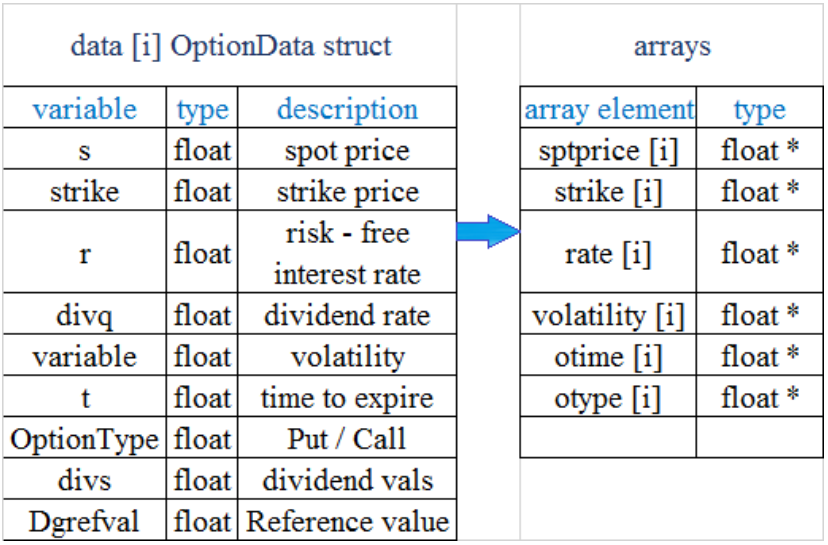

Figure 2.7: Array of OptionData converted to multiple arrays

- There is not any dependency between the data of the different threads. This makes the implementation of the program to be free of locks and multiple synchronization points during the execution.
- All the values that need to be written by a single thread are stored in continuous places in the memory. It is noteworthy that Black-Scholes has the lowest miss rate across all the PARSEC applications.

As it is obvious from figure 2.6 the computations for all the options are repeated 100 times. When monitoring the results, we took separate measurements for the time needed for a single iteration of the options computation (which would be the same as if the NUM_RUNS variable was defined as 1). The main difference in the results is as expected the time spent per execution.

Moreover, when running blackscholes with more than 24 threads the speedup is getting immediately worse (16,7 for 26 threads) since the Dunnington has 24 cores. This is happening due to the fact that the threads are not equally split among the available cores. One core of the Dunnington has to execute the code of multiple threads, perform context switching for each thread change which involves saving the status before each replacement. When having less threads than cores, the OS scheduler is not doing complicated context switching and the thread migration becomes rare. Additionally to that, it has been observed in other runs [Pusu11] that the main thread of blackscholes is taking up to the 85% of the execution time. This leads to have a small workload for the worker threads when increasing too much the number of threads and distributing the input in very small chunks. Last but not least, blackscholes spends a lot of time waiting for the FPU since the computation of the equations is intense. This means, that in a machine which has FPUs of lower performance there would be a bottleneck and the achievable speedup would be worse.

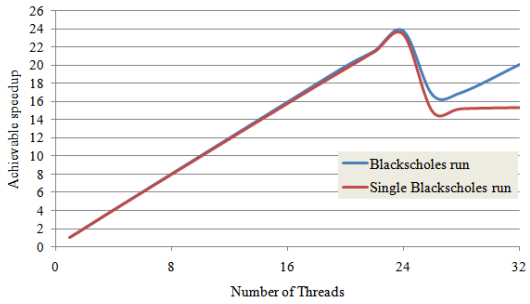

Figure (2.8.1) Scaling

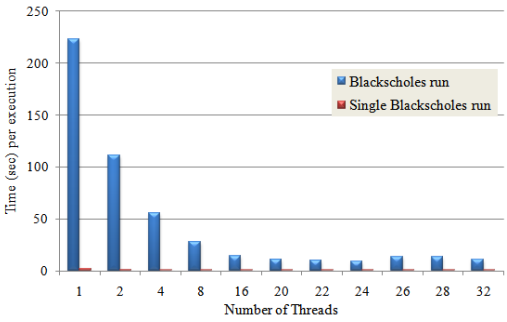

Figure (2.8.2) Time (sec) per execution

Figure 2.8: Native input dataset

Chapter 3

Applications with medium achievable speedup

3.1 STREAMCLUSTER

3.1.1 Overview

The streamcluster application belongs to the domain of data mining. As the name implies, the goal of the program is to group a set of objects into different clusters. Objects that share a common pattern or trait will be part of the same group and once the clustering for all the objects of a problem is finalized the desired information can be extracted. Clustering is a modern problem that is used in many fields. Network intrusion detection, pattern recognition, data mining [Bien11], medical imaging and search result grouping in the web are just a few of them. For example, in the area of the image processing a problem that can take advantage of clustering is the one of image segmentation [Wiki14], in which multiple 3D digital medical images have to be simplified in order to extract patterns across the patients and make better diagnosis. In this problem, the input consists of a stream of pixels. The output image contains a specific number of segments and each of them has many similar pixels in terms of color and position. Ten neighboring pixels of similar color for instance will be grouped in the same cluster and the resulting image will give one unique color to all of them. The main difficulty of the algorithm is to make the proper choice of the centers/ pixels that will represent a specific segment and to group all the objects into the right cluster.

3.1.2 Description

There are many different algorithms to implement the clustering and we are focusing on analyzing the one that was chosen in the PARSEC benchmark. Given two integers k_{min}, k_{max} and a stream of multidimensional integer points we aim to distinguish k elements $k_{min} \leq k \leq k_{max}$ out of this stream and assign them as centers. Each of the remaining points will be related to one center, which will be the most similar to it. For the example with the images, it would make sense to choose a big integer k for medical images of the brain and a smaller one for pictures with different sea scenes that are more simplified.

In the current algorithm the sum of squared distances (SSQ) between every point and the center of its cluster determines the quality of clustering [Ocal02]. A very well-known algorithm is k-means, which is based on the NP-hard optimized solution. The current application is an implementation of another algorithm that according to the references achieves even better results than k-means.

The main parts of the application are the following:

- Parsing a stream of integer numbers $x_1, \ldots, x_n$. The number n is equal to the capacity of one chunk.

- Computation of the sum $|x_i - x_1|^2$, which is the distance between all the dimensions of two points x_i and x_1 (i:2,…,n). The point x_1 is defined as the initial center of the algorithm.

- Shuffling of the points, center selection, which is described as facility opening, and assignment of non-center points to a cluster. This step is repeated until the proper number of centers is created and the total cost is not decreasing further. The extended algorithms that solve this part are the LSEARCH and the FL (Facility Location). If the desired number of centers equals with the total available points of the chuck (n) then all of them will be immediately assigned as centers and each cluster will have only one member.

- Copying of the center's data in the memory, in order to have this information for future incoming streams.

3.1.3 Parallelization approach

The application can be executed with the use of either Pthreads or the TBB threading model. We have analyzed the TBB implementation. The parallel parts of the code can be categorized in three main steps and we proceed with analyzing each of them further. Assume that we have n different points in a chunk, x_i ($0 \leq i \leq n$) represents a multidimensional random point within this chunk and nproc is the available number of threads. The `tbb::blocked_range` template class [Docu14a] is used in all the parallel loops to distribute the points among the nproc available threads. Specifically, the range is always (x_0, x_n) ,where x_0 is the first point stored in the chunk. The grainsize is defined as n/nproc.

- Step 1: Hiz Reduction

  This step computes the sum of equation 3.1 by using the reduce function.

$$\sum_{k=0}^{n-1} |x_k - x_0|^2 \cdot x.weight_k \tag{3.1}$$

  The body is implemented in the HizReduction struct so that each thread computes the sum of its subrange.

- Step 2: Pspeedy method

  This method contains two parallel loops. The first one aims to create one center, point x_0, and assign all the other points to that. For this purpose the parallel for template function is used.

$$x.cost_k = |x_k - x_0|^2 \cdot x.weight_k \tag{3.2}$$

$$x.assign_k = 0 \tag{3.3}$$

  The goal of the second one is to assign to a new center, x_i, all points that have a better weighted distance for x_i than for their previously assigned center. The final outcome is the new total cost of the points, which is a quality metric for the new center that was opened. Reduction is used since we need to both perform the cost updates in parallel and join all the costs in a common result variable.

$$x.cost_k = min\Big\{|x_k - x_i|^2 \cdot x.weigth_k, x.cost_k\Big\} \tag{3.4}$$

$$\sum_{k=0}^{n-1} x.cost_k \tag{3.5}$$

- Step 3: Facility Location – Pgain method

  This step contains 5 parallel parts that follow the same parallel pattern and compute the cost that will be saved by opening a new center. Assume that the actual code that a thread will execute is implemented in the class ClusterClassX that derives from the `tbb::task` class. For every such class there is another one, ClusterClassXTask, that also derives from `tbb::task`. The ClusterClassXTask creates a `tbb::task_list` of nproc-1 ClusterClassX children that will be executed by nproc-1 worker threads and one ClusterClassX root object that will spawn the list. The pgain method is using the `spawn_and_wait` method to start the execution of this chain.

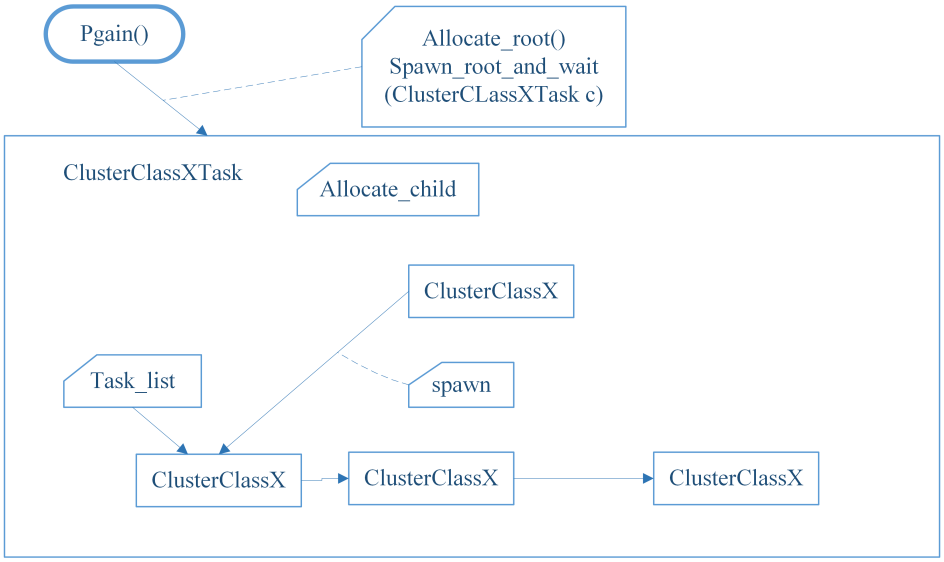

Figure 3.1: Cluster Task - ClusterClassXTask = { CenterTableCountTask, FIxCenterTask, LowerCost-Task, CenterCloseTask, SaveMoneyTask }

3.1.4 Data structures

The main information that needs to be stored in this application is the data of the points of each incoming stream. For this reason we create the structs “Point” and “Points”, which contains a pointer to the first element of a “Point” array. In this way all the data of one point and all the points of a chunk

Variable	Type	Description
num	long	size of Point array
dim	int	dimensions of one point
p	Point *	pointer to first element of Point array

Table 3.1: Points (struct)

Variable	Type	Description
Weight	float	weight of current point
Coord	float *	array with #dim coordinates
Assign	long	id of center to which current point is assigned to
Cost	float	cost of current point

Table 3.2: Point (struct)

are stored continuously in the memory.

3.1.5 Measurements and performance analysis

We took measurements in Dunnington using from 1 up to 32 threads and distinguished the results for the three parallel parts that were mentioned above. Figure 3.2.1 presents the achievable speedup and figure 3.2.2 the time spent on each of these parts. The whole clustering algorithm is being called once per each incoming stream, and for the native execution of the application we have 5 streams. As a result, if we wish to calculate the time spent per stream, we have to divide all the results given in the figure 3.2.2 with 5. Regarding the Hiz Reduction part, the total time spent in it is 0,078 for all the

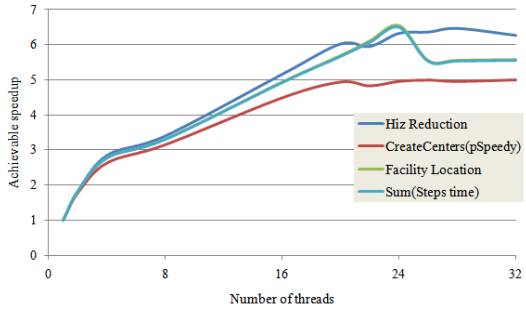

Figure (3.2.1) Scaling per step

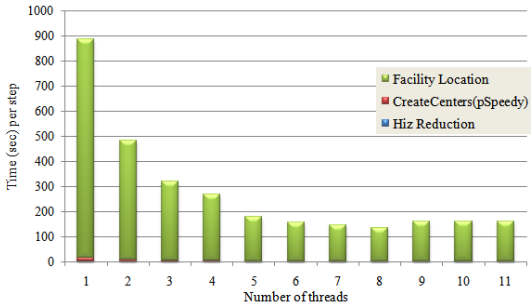

Figure (3.2.2) Time (sec) per step

Figure 3.2: Streamcluster scaling and time per step

streams and 0,0156 per stream. The first reason for the mediocre scaling is the fact that the time needed for thread creation cannot be skipped. The second reason is that the computation of one distance is being executed serially for all the dimensions. In the native input each point has 128 dimensions. Of course this implies that for another input with points of bigger dimension we will introduce a bottleneck, but in this case, the distance calculation could be parallelized. As far as the second part is concerned, apart from the overhead that the thread creation introduces, we have two synchronization points in it. Moreover, pspeedy method is called repeatedly until the creation of an adequate number of centers (k_{min}) is achieved. This means that the time spent per single parallel execution should be divided further and that more synchronization points are introduced. The pspeedy method also depends on random generation of some numbers that will influence the speed of successful center creation.

It is obvious that the most crucial part for the overall performance of the program is the last one. We analyzed further the 5 parallel parts of this step. If we wish to get the actual time spent on each loop we have to divide the results with 8771, since this is how many times the pgain method is called during the execution of the native input. The least time – approximately 0,0002 sec per single iteration - is

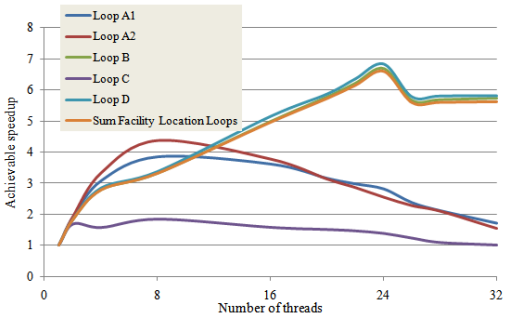

Figure (3.3.1) Scaling of Facility Location parallel steps

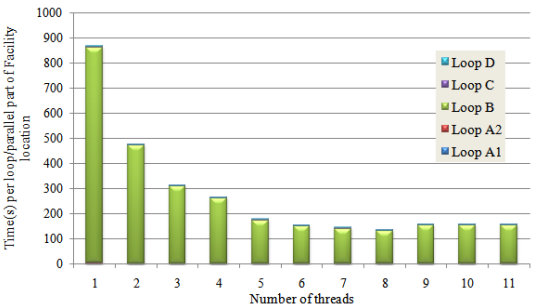

Figure (3.3.2) Time (sec) of each parallel step of Facility Location

Figure 3.3: Facility location scaling and time

spent on loops A1, A2 and C which can explain the bad scaling that these steps achieve.

As it is obvious from the above diagram, the most critical loop regarding time is the B. The loading of several variables of an integer array ("center_table") is one of this part's operations. The variables that each thread is accessing are not stored continuously in memory and as a result this loop is memory bound. The indexes of the array elements that each thread will need to load cannot be specified from the beginning, since they depend on the incoming data and the computation performed until this step. We verified this observation by creating a mockup class that would perform all the computations of loop B but not the crucial one. The mockup code was implemented in a way that the omission of this clause would not have any influence on the rest of the loop. The new loop took similar time to be executed as the other loops of this step.

Besides the analysis of the specific native input that was used in the PARSEC benchmark, we made several observations that are worth mentioning. Streamcluster depends on the input parameters and especially on the dimensions of each point, the size of each data stream and on the number of desired centers to be opened. The more dimensions one point has, the more computational time is added during the execution of the program. Similar overhead is introduced if the size of a chunk is increased. The increase of the number of centers that are needed has an impact on the size of some arrays and the number of several iterations in Pspeedy method and Facility Location parts. Moreover, the random number generation and the shuffling of the points in several places in the program determine how many times some loops will be executed. As a result, even if we run the program with the same parameters, the run time can vary due to this dependency.

3.2 BODYTRACK

3.2.1 Overview

The bodytrack belongs to the domain of computer vision [Bala05]. Given a sequence of images from different cameras that depict a human body in different poses, the program aims to track this 3D body as accurately as possible. The more image sequences and the more cameras that capture the same movement from different angles we have, the more complicated the problem becomes. The body detection problem is very important for the areas of video surveillance, character animation, computer interfaces [Bien11] and within the medical domain which may use body or face recognition for improving the sign language software. Given t frames and c cameras, the input consists of $t \cdot c$ different images and the $t \cdot c$ related foreground images of the body. An annealed particle filter is employed on the input in order to produce the t output images, where the human body is marked with 10 conic cylinders per camera shot.

3.2.2 Description

The bodytrack implementation in PARSEC consists of different steps which are executed for every frame. The parts (iii) to (vi) are repeated m times, where m is an input parameter that represents the number of annealing layers.

(i) Gradient based edge detection on the input image in order to get the direction and magnitude of maximum intensity change at each pixel.

(ii) Edge smoothing by applying a Gaussian filter per row and per column on the image that was computed in the previous part.

(iii) Cumulative distribution function calculation of the particles' weights. Each layer uses N particles. Each of them has a weight π and a multi-variate model configuration X for describing the location of the body's joint angles.

(iv) Monte Carlo resampling given the calculated cdf vector.

(v) Generate a new particles' vector by using the values of the previous layer. The results from steps (iii) and (iv) are used to decide which values will be selected, and then normally distributed random noise is added to them.

(vi) Likelihood and weight calculation for each particle.

3.2.3 Parallelization approach

The current application has been implemented in all the threading models that are supported by PARSEC (Pthreads, TBB and OpenMP), and we have analyzed the TBB version. Although the data-parallel model is used in all parallel parts of the application, the program executes the two-stage `tbb::pipeline` class [Docu14b]. The main motivation for that is to simplify the synchronization and communication between specific parts and between the frames. Moreover, the abstraction that the pipeline model introduces simplifies the understanding and maintenance of the code, since basic methods like the barriers are logically defined [Reed11]. The current pipeline consists of two filters:

Figure 3.4: TBB pipeline

TrackingModel and ParticleFilter. The `operator()` method of these classes contains the code that will be executed for each filter. It takes as an input a token from the previous step and passes it to the next filter. A specific token can be passed from one step of the pipeline flow to another only if the current filter has been processed completely. This programming model enables multiple tokens to run in parallel, which means that at a given point of time some threads will execute code of tokens that belong to different filters. However, the current implementation does not take advantage of this feature and has a restriction of a maximum of one token at any given time. Each thread that takes a token has to keep it as deep in the pipeline as possible in order to prevent context switching and to use the cache more efficiently. The TrackingModel filter contains the first two parts of the program (steps (i), (ii)) and the ParticleFilter the rest of them.

- TrackingModel filter (Figure 3.5): This filter consists of one `tbb::parallel_for` loop which distributes the data across the threads based on the different input cameras. Afterwards, this loop contains three more nested `parallel_for` loops that implement steps (i), (ii). These follow a similar pattern for the data partition and always use the `tbb::auto_partitioner`.
- ParticleFilter (Figure 3.6): This group takes as input the token that was passed from the TrackingModel filter and executes some serial and parallel methods. Steps (v) and (vi) are implemented with the use of `tbb::parallel_for` and the blocked range is defined as (0, number_of_particles, grain_size). The grain_size in the current implementation is defined as 32, meaning that only if we have more than 32 particles will the program take advantage of the existence of multiple threads.

3.2.4 Data structures

Many classes are defined and implemented in the bodytrack application in order to store mainly the data of the images, the body's geometry and pose and the images' projections and measurements.

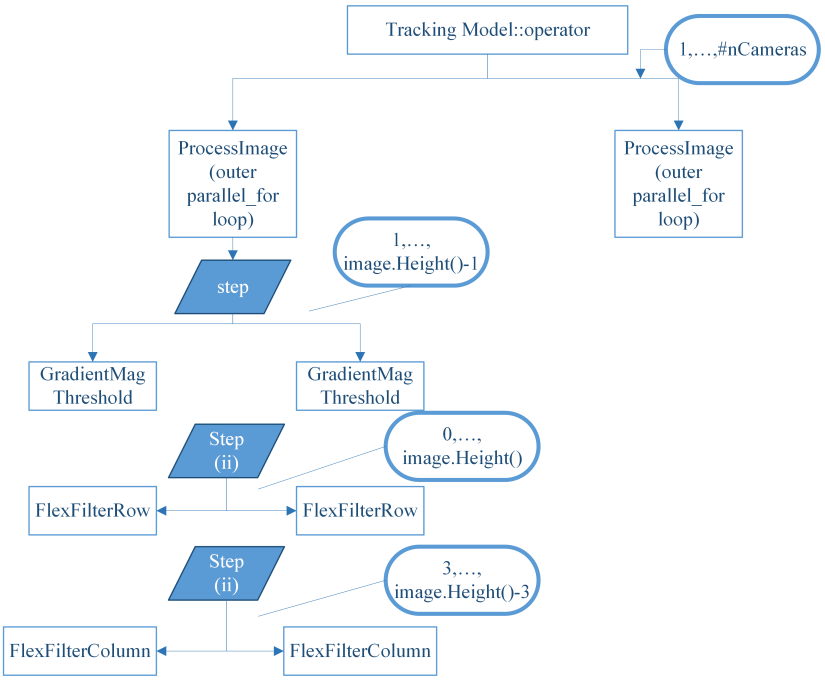

Figure 3.5: Tracking Model operator

The ones that are mostly worth mentioning are the generic template FlexImage<class T, int C> and FlexImageStore< class T, int C > classes (Tables 3.3 and 3.4), which are responsible for constructing and deconstructing the objects that store one image's data, storing and updating the information about the image's size and performing several operations on it if needed. Generic class T defines the type of one pixel's data and int C defines the number of one pixel's channels. In the current implementation one pixel is stored in one unsigned char and has either one channel (grayscale) or three (RGB). The

Variable	Type	Description
mStore	*FlexImageStore	
mData	*unsigned char	pointer to mStore.mData
mBpp	int *	pointer to first element of Point array
mStepBytes	int	size in bytes of one image row
mSize	class int,int	image dimensions: width, height
mStatus	enum	error status codes

Table 3.3: FlexImage<unsigned char,1>

Variable	Type	Description
mCount	int	reference counter
mData	*unsigned char	pointer to pixel's data. Pixels chars are stored per row.
mSize	class int,int	image dimensions: width, height

Table 3.4: FlexImageStore<unsigned char,1>

details provided by the FlexImage structure regarding the size in bytes of one pixel and one image row enables the program to easily distribute the data among the threads. Having this information and the memory address of the first image pixel, it is easy to access a pixel with specific coordinates with one operation.

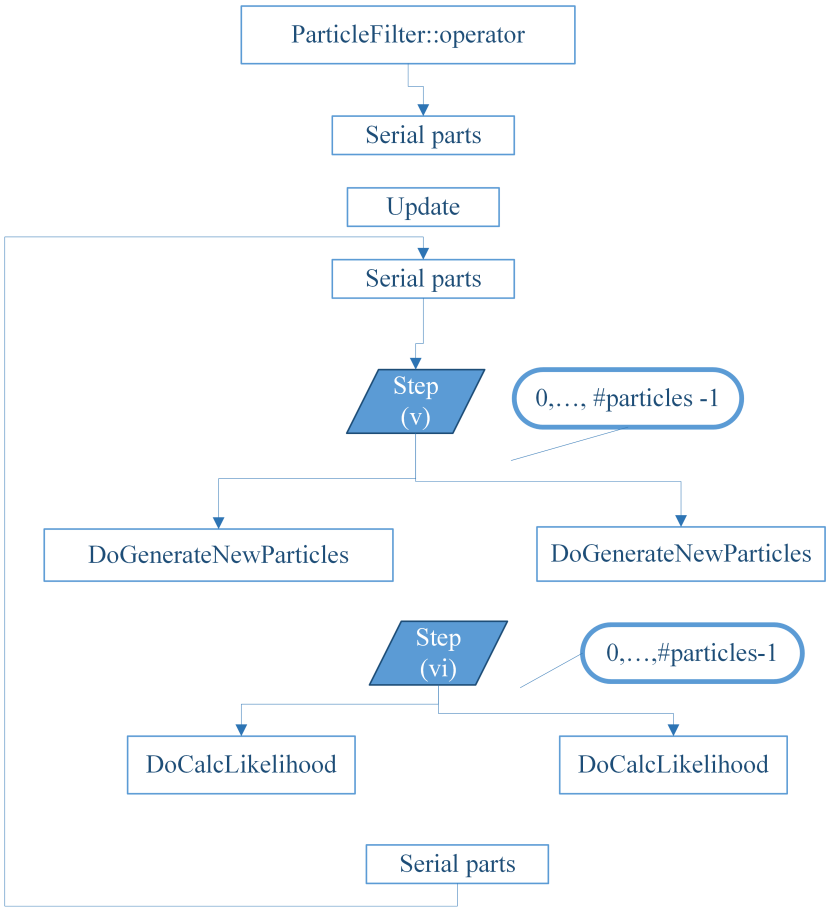

Figure 3.6: Particle Filter operator

3.2.5 Measurements and performance analysis

Firstly, we took measurements for the initial implementation of bodytrack and analyzed the scaling and time spent on each of the four parallel parts (steps (i), (ii), (v) and (vi)). All the results are produced with the use of the native input and the calculated time of all the parts is the sum of all the frames. This input uses 261 frames, so the duration of one single execution of one part equals the current calculated time divided by 261. The size of all input files is approximately 610 MB and the size of all the output images that are produced is 220MB. Based on the Figure 3.7 one of the main issues

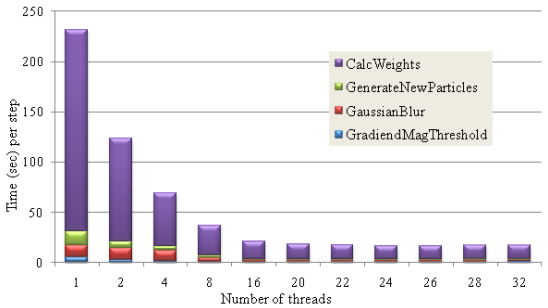

Figure (3.7.1) Scaling per step

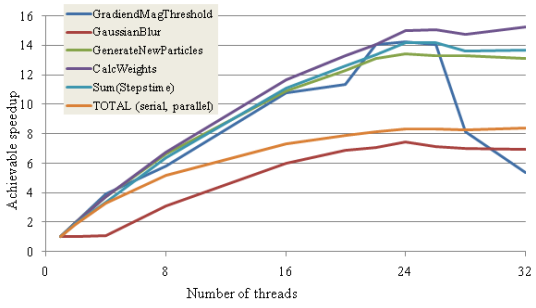

Figure (3.7.2) Time (sec) per step

Figure 3.7: Bodytrack scaling and time per step

for the bad achieved speedup are the serial parts of the programs. For this reason, we extracted the serial algorithms and we present each of them in the following section. Also, we present the following

figure, in which the increasing impact of the serial parts as the number of threads increases is quite obvious.

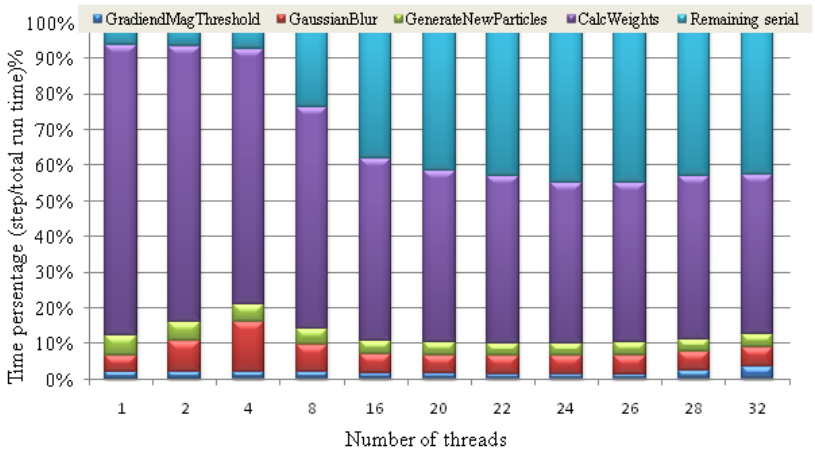

Figure 3.8: Time percentage spent in each step (including the serial parts)

(i) Cumulative Distribution Function calculation

Memory elements

- vector<float> weights : input vector, size: 4000
- vector<float> cdf: destination vector with cumulative distribution values, size: 4000

Function

$$cdf[i] = \begin{cases} weigths[i], & i = 0 \\ cdf[i-1] + weights[i], & 0 < i < 4000 \end{cases} \tag{3.6}$$

All cdf values are normalized.

Information, Comments

Each element depends on the value of its previous element on the list. As a result, this algorithm cannot be parallelized. The algorithm includes 4000 floating point additions, 4000 FP read operations (of weights array) and 4000 FP write operations (of cdf array). We take advantage of the locality of both arrays, since the elements are accessed one after the other. This means that from the beginning of the algorithm, both arrays will be fully loaded in the cache and we do not introduce problems with the memory bus. In bodytrack this algorithm occupied just the 0,08% of the total run time and the 0,17% of the serial time of the application.

(ii) Monte Carlo Resampling

Memory elements

- vector<float> cdf : input vector, size: 4000.
- vector<float> samples: vector with set of sorted random samples.
- vector<float> bins: destination vector which relates cdf with sample list.

Function

$$samples[i] = \begin{cases} RandExp(), & i = 0 \\ samples[i-1] + RandExp(), & 0 < i < 3999 \end{cases} \tag{3.7}$$

Algorithm

```
//Initialize bins.size()=samples.size() && bins[]=0

p=0;
for (int i=0;i<samples.size();i++)
{
    while (cdf[p] < samples[i]) p++;
    bins[p]=bins[p]+1;
}
```

Information, Comments

Each of the sample's elements depends on the previous one of the list. As a result, the computation of this vector cannot be parallelized. A small improvement would be to precompute the RandExp() values in parallel and have them stored in an array random[4000]. Currently, the random generation function is called while computing the values of the samples vector.

The computation of the bins vector is by definition based on a serial algorithm. Starting with the first element of both cdf and samples lists we count the number of initial samples elements that have a value smaller than the current cdf element. We cannot predict the distribution of these values from before and the only way to generate this vector is to start from the beginning of both arrays.

In our current application this algorithm occupied the 1,84% of the total run time and the 3,65% of the serial time of the application.

(iii) Estimated Model Configuration Calculation

Function

$$estimate[i] = \sum_{j=1}^{4000} particles[j][i] \cdot weights[j] \tag{3.8}$$

Information, Comments

Each of the 4000 particles has 31 estimated values for the body's pose. The aim of the current algorithm is to compute a value for each of the 31 body model's joint angles and anchors, based on the separate estimation for these elements that has been made by each particle. The algorithm can be parallelized in both i and j dimensions, but after taking all the required measurements we concluded that the overhead introduced by the thread creation and synchronization leads to worse results. The current execution occupies only the 0,31% of the application's serial time.

(iv) Load/Save .bmp Input/Output Files

Algorithm

```
//Open file for reading/writing
//Read/Store file header
//Load/Write pixel data (count: #width * #height = 640*480)
for (int j=0;j<file.height();j++)
{
        //Allocate line j of img data
        for (int i=0;i<file.width();i++)
        {
                //Read pixel (j,i)
                //Store pixel in img->data(j,i)
```

```
        }
}
//Close file
```

Information, Comments

It is well known, that trying to parallelize and speedup disk I/O is not a task that programmers put into practice. The loading part takes the 3,82% and the writing part the 12,99% of the total running time.

(v) Write Pose.txt Output FIle

Algorithm

```
for (int i=0;i<pose.size();i++)
{
        poseOutputFile << pose[i] << “” ;
}
```

Information, Comments

As mentioned before the I/O operation should be executed serially. The current part that saves the body’s pose information needs only the 0,12% of the total run time. As a result, the total time percentage spent on I/O operations is 16,93% and around 350 ioctl() calls are produced per second which is obviously a main bottleneck of the program [Pusu11]. All the threads that execute I/O operations need to perform often state transitions, mainly between “waiting” and “runnable” states, which leads to the increase of thread migration. When a sleeping thread is again ready to be executed but the core that it was using previously is not available anymore, it may move to another one and the likelihood of cache misses increases.

(vi) Downsample Image by Factor N with Anti-Aliasing

Memory elements

- FlexImage<char,1> srcImage: input image data that will be downsampled (size: 480*640 = 307200 pixels that are stored in 307200 chars).
- FlexImage<char,1> dstImage: destination image that will be the srcImage downsampled by factor 2 (size: (480/2)*(640/2)=76800 pixels that are stored in 76800 chars).

Algorithm

We will present the algorithm for the Gaussian filter per image row. Similar algorithm is implemented for the column filtering.

```
//define float kernel[k] (in current implementation k=3)
n=k/2;
for (int j = 0; j<srcImage.Height(); j++)
{
        char *currentSource = &srcImage[j][n];
        char *currentDest = &dstImage[j][n];
        for (int i=n; i<srcImage.Width()-n; i++)
        {
                int p=0;
                int sum=0;
                for (int a=-n ; a<=n ; a++)
                {
                        sum+=currentSource[a]*kernel[p++];
                }
```

```
                currentDest = sum;
                currentDest++;
                currentSource++;
        }
}
```

Image downsampling by factor 2

```
for (int j = 0; j<srcImage.Height()/2; j++)
{
        char *currentSource = srcImage[j*2][0];
        char *currentDest = dstImage[j][0];

        for (int i=0; i<srcImage.Width()/2; i++)
        {
                currentDest[j][i]=currentSource;
                currentSource = currentSource + 2;
        }
}
```

Information, Comments

Both row and column filtering algorithms can be parallelized in the j dimension. The suggested parallelization is being used as part of another algorithm in the application, and are using an updated version in this step as well. Moreover, we parallelized the downsampling algorithm in the j dimension. The initial version occupied the 19,23% of the total run time but the results of the updated version were very satisfying.

(vii) Convert Grayscale to RGB Image

Figure 3.9: Grayscale to RGB Data

This algorithm is the basic one for converting a grayscale image into an RGB image, by copying the initial pixel data to each of the three new channels. In the current implementation, none of the for loops has been parallelized, so we updated the code accordingly. Moreover, the height or width dimension can also be parallelized, and we preferred to choose the height dimension in order to keep advantage of the way both arrays are stored in the memory.

(viii) Beta Annealing Factor Calculation

Memory elements

- vector<float> weights : input vector, size 4000 (weights[i] is the computed weight of particle[i])
- 2. float alpha_desired : input value for the desired particle survival rate
- float beta_min, beta_max: input values for defining the range of the calculation

Function

Delta alpha calculation

$$DeltaAlpha(beta) = \frac{4000 \cdot B^2}{F}, \tag{3.9}$$

where

$$B = \sum_{i=1}^{4000} e^{beta \cdot weights[i]} \ \wedge \ F = \sum_{i=1}^{4000} e^{2 \cdot beta \cdot weights[i]}$$

Algorithm

```
//Initializations
int i = 0;
float beta_min = 0;
float beta_max = 1000;
float beta = (beta_min + beta_max)/2
float delta_alpha_min = DeltaAplha(beta_min);
float delta_alpha_max = DeltaAlpha(beta_max);
float delta_alpha_beta = DeltaAlpha(beta);

while ( (abs(delta_alpha_beta) > 0.00001) && (i < 100) )
{
        if      ((delta_alpha_min, delta_alpha_beta > 0) ||
                (delta_alpha_min, delta_alpha_beta <= 0))
        {
                beta_min = beta;
                delta_alpha_min = delta_alpha_beta;
        }
        else
        {
                beta_max = beta;
                delta_alpha_max = delta_alpha_beta;
        }
        beta = (beta_min + beta_max)/2
        delta_alpha_beta = DeltaAlpha(beta);
        i++;
}
```

Information, Comments

Every time the beta annealing factor computation method is called, around 20 iterations are executed. In the current implementation, neither the DeltaAlpha function nor the function that handles the whole computation of the factor have any parallel part.

The beta annealing algorithm is calling the DeltaAlpha function in order to compute the delta value based on a beta value, which is passed as a parameter. The possible beta values can be easily represented on a binary tree and can be known from the beginning of the execution of the algorithm. It is still unknown, though, which the path that will be chosen each time will be. If the DeltaAlpha value of each node was already stored in the memory, the execution of the beta algorithm would be much faster. The current program would not take benefit of the use of hyper-threading, since the depth of the tree is very short. However, given a more complex scenario in which more nodes of the tree would need to be calculated the logic of hyper-threading could be useful.

After applying all the optimizations that were mentioned above, as figure 3.10.1 depicts, the time spent in the serial time was improved by 26,29% and the total one by 11,95%. Figure 3.10.2 represents the

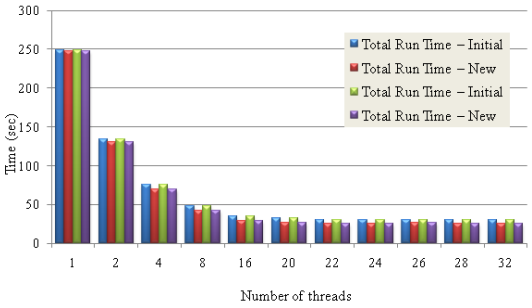

Figure (3.10.1) Optimizations improvement in sec

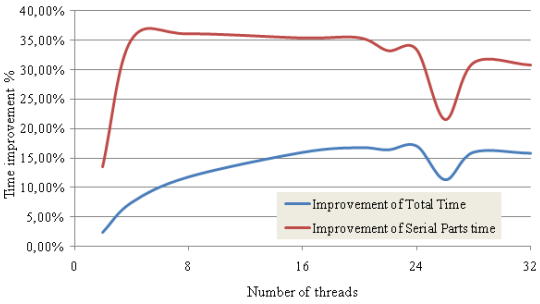

Figure (3.10.2) Optimizations improvement percentage

Figure 3.10: Comparisson of Initial and New Implementations Total Run Time

improved scaling of the bodytrack application. As far as the parallel parts are concerned, the ones that need to be analyzed most are the Gaussian Blur and the Calc Weights. In the Calc Weights, the main reason for the mediocre scaling is that some of the serial parts which were analyzed above introduce a bottleneck in this step of the pipeline. It is suggested that a solution to that would be to use pairs of processors for the execution of each thread, where one would have a higher frequency in order to execute the serial parts of the thread. Moreover, the more layers we use for getting better accuracy on the body detection, the more synchronization points we introduce, since the loop in which the CalcWeights part belongs to is executed more times. This part is also memory intensive and many floating point operations need to be executed, since it has to create the geometric representation of the body model, compute the required scaling and translation of the coordinates and perform the projection in the 2D space. Afterwards, three floating point arrays with information about the 4000 particles will be updated. Regarding the Gaussian Blur step, we present a further analysis of the row and the column

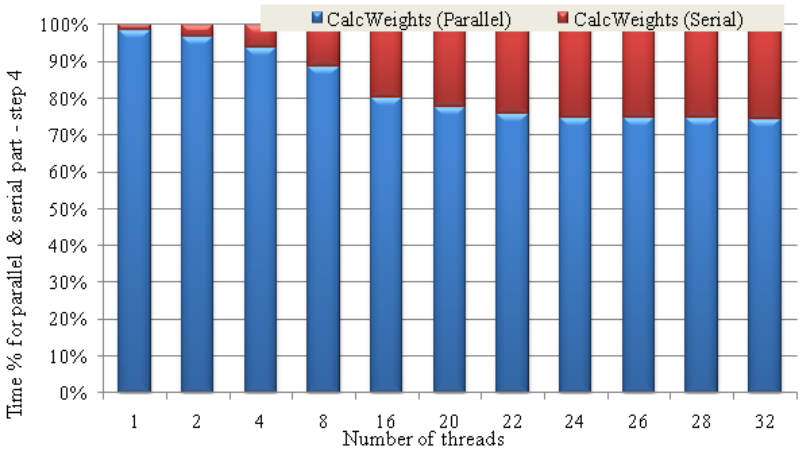

Figure 3.11: Step 4 - Calc Weights, Time percentage for Parallel and Serial parts

filtering scaling in the figure 3.12. The parallel for loop that executes the row bluring is better because the threads take advantage of the data locality. The pixel data is stored per row in the memory and the data is distributed per row. A thread that performs column filtering, instead, will manipulate the pixel of one or more columns, so it will bring to the local cache data from pixels that are not needed. Last but not least, in all parts above, the time spent on lock contention is one more reason which makes the achievable speedup worse. According to already collected data 50% of one thread's time is spent on waiting for user locks and condition variables [Cass09], [Pusu11].

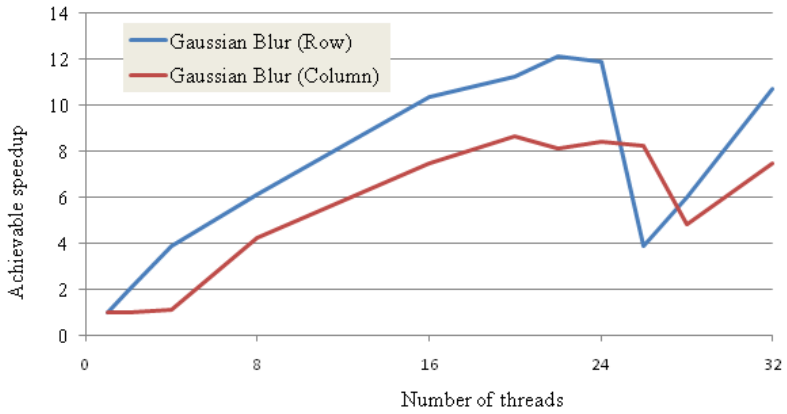

Figure 3.12: Scaling of Step 2 (Gaussian Blur), Row and Column Parallel parts

3.3 FLUIDANIMATE

3.3.1 Overview

The fluidanimate belongs to the domain of computer animation and specifically of real time fluid simulation. The program aims to visualize the fluids efficiently and can be applied in both scientific and game animations. Computer games use extensively fluid animation and the program needs to have both good performance since the data is generated in real time and produce good image quality to satisfy the player's demands. The program implements an extension of the smoothed-particle hydrodynamics (SPH) computational method, which uses a set of particles to represent the fluid. The input contains the values of the initial position coordinates, velocity and viscosity for each particle and any other properties, such as external forces, viscosity flow forces and fluid's material that may exist. The viscosity is a measure of a fluid's resistance to deformation. For example, the honey and the water have completely different viscosity and if a force is applied to these fluids the results after a specific timeframe will be less intense for the honey than for the water.

3.3.2 Description

The fluidanimate as it was customized for the PARSEC benchmark uses only one external force and has predefined the fluid's colors and the scene's geometry. With the use of the initial values, the equations provided by the SPH algorithm and the Navier Stokes equation (eq. 3.10) for incompressible fluids the program calculates the density and acceleration of each particle.

$$\rho(\frac{\partial v}{\partial t} + v \cdot \nabla v) = -\nabla p + \rho g + \mu \nabla^2 v \tag{3.10}$$

Simplified version of Navier-Stokes equation, where v is the velocity, ρ the density, p the presure, g the external force and μ the fluid's viscosity.

The results are used to update the particle's initial values. This number of times that this procedure is executed is defined by the input parameters and since the program is real-time it could be repeated unlimited times. The equations 3.11, 3.12 and 3.13 are used in each iteration for updating each particle's parameters.

$$position_{new} = velocity_{old} \cdot time + acceleration_{new} \cdot time^2 \tag{3.11}$$

$$velocity_{new} = velocity_{old} + acceleration_{new} \cdot time \tag{3.12}$$

$$viscosity_{new} = hv_{old} + \frac{1}{2} acceleration_{new} \cdot time \tag{3.13}$$

Time is a constant value defined currently 0,001.

The whole procedure of calculating the acceleration and the density is broken up into five parallel parts which can be easily maintained in order to execute a more complicated input with more external forces and a different scene. The scene is divided into cells and each of them contains zero or more particles.

- Rebuild spatial index

  Each cell can contain different particles after each iteration, since the particles are changing positions. This step that is implemented by two functions (ClearParticles and RebuildGrid) is storing each particle in the proper cell.

- Compute densities

  Firstly, this kernel calls InitDensitiesAndForces, which initializes the variables acceleration and density on the external force and 0 respectively. Afterwards, the ComputeDensities and ComputeDensities2 functions are updating the density value by using the position coordinates. The closer the particles are with each other, the higher is the density of them. Only the particles that belong in the 27 neighbouring cells of the 3D space from the current one are taken into consideration to perform these computations.

- Compute forces

  This step is implemented by ComputeForces function and calculates the acceleration given the position coordinates, the density and the viscosity values. As mentioned before the collisions between particles are handled only between a limited number of cells.

- Handle collisions with scene geometry

  The function ProcessCollisions is updating the acceleration values by taking into account the scene geometry and all the particle's parameters (position, viscosity, velocity).

- Update position, velocity and viscosity of particles

  The function AdvanceParticles is responsible for updating the position, viscosity and velocity of each particle according to the equations 3.11, 3.12 and 3.13.

3.3.3 Parallelization approach

Fluidanimate is supported in PARSEC by Pthreads and TBB threading models and in our work we focused on the TBB version. All parallel parts are implemented with the use of generic template classes. Firstly, a generic class model is implemented which creates a list of tasks that will be executed by the threads and defines the geometry region that each thread is rensponsible for. Secondly, eight classes implement the steps mentioned above are and are executed in parallel by the available threads [Lee10]. Given num_threads defined available threads, the program creates in the end $num\_threads \cdot 8$ objects of a concrete class T which will execute a specific algorithm of the program and each thread will need to execute the 8 of these. This abstract pattern gives the freedom to implement different classes T which are independent of the way that the threads were created, the exact data distribution and execute them easily in parallel. Moreover, it is easier to follow the logic of the whole program and maintain the thread creation and the implementation of the algorithms separately.

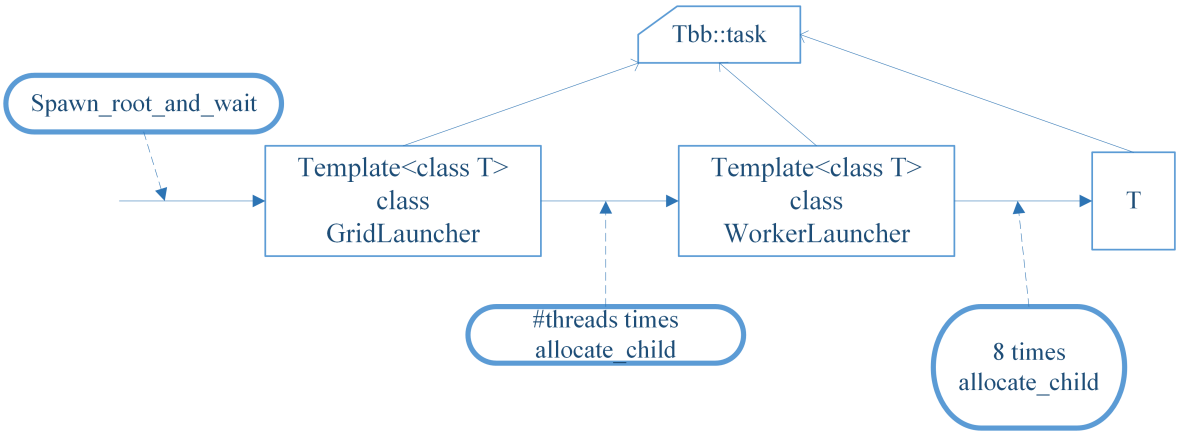

Figure 3.13: Generic Template Classes

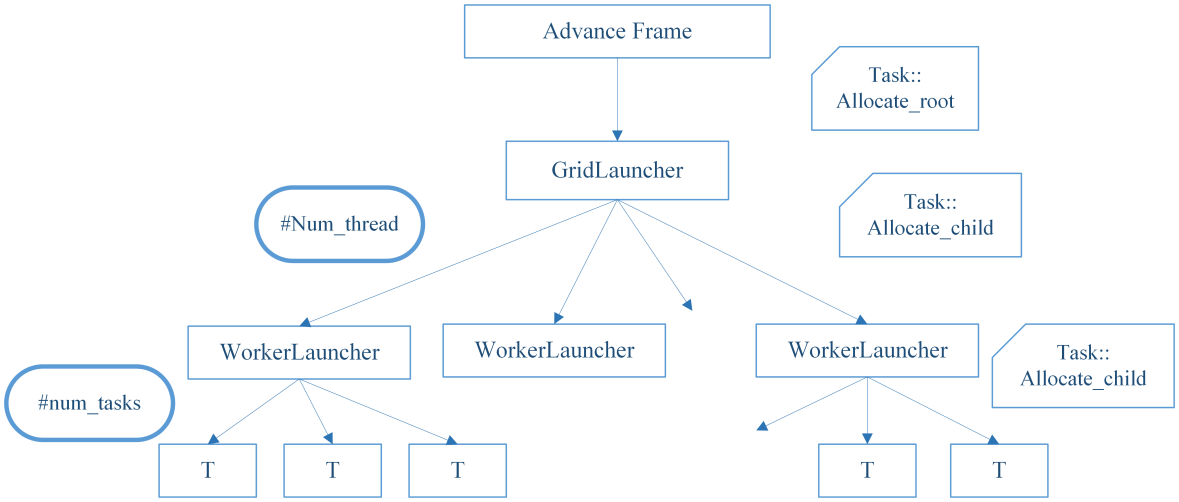

Figure 3.14: Tree of Thread and Task allocation

3.3.4 Data structures

The data structures that were implemented in fluidanimate for storing the pixel's data and keeping track of the geometric cell in which they belong to are very interesting. The challenge was to handle boundary conditions where a cell has at some point of time too many pixels in comparison to its initial state and to the average cell capacity and to have allocated the maximum memory needed before the parallel thread creation and execution started. It is efficient to prevent the program from allocating dynamic memory by multiple threads in parallel since this requires more time than allocating it dynamically from one thread.

Firstly, the basic data structure for the cell is defined and contains the data for a fixed number of pixels. This number is defined as 8 in the program. Most of the pixel's parameters are stored in a 3D float vector (Vec3) which was implemented to support many functions and operations between different vectors. Ideally, if the fluid did not move and each cell contained the same number of pixels during the execution it would be enough to let each thread allocate a specific number of cell structures and handle only these memory elements. Given num_cells geometric cells in total, the program allocates in the beginning a cell array, "cells", with num_cells elements that are not connected with each other. If more than $particles\_per\_cell = 8$ pixels move into a cell, however, the working thread is able to store the data in a new cell that is already allocated in the memory and link it with the initial one. A cellpool structure is implemented that contains a linked list with cells that are able to store the data of at least $4 \cdot (total\_num\_particles/num\_threads \cdot 8)$ particles. The list does not have any actual pixel values

Variable	Type	Description
p[particles_per_cell]	Vec3	p[i]: 3D position coordinates of pixel i
hv[particles_per_cell]	Vec3	hv[i]: 3D velocity values of pixel i
v[particles_per_cell]	Vec3	v[i]: 3D viscosity values of pixel i
a[particles_per_cell]	Vec3	a[i]: 3D acceleration values of pixel i
density[particles_per_cell]	float	density[i]: density value of pixel i
next	*Cell	pointer to the next cell
padding	char	padding used for cache alignment purposes

Table 3.5: Struct Cell

stored in the beginning of the execution, but works as a pool to grab cell structures when needed. A cellpool array of size $num\_threads \cdot 8$ is allocated and has $num\_threads \cdot 8$ different cell linked lists. Every time a thread needs a new cell it takes one from this array. The linked list that is selected for taking the cell follows the round robin algorithm in order to avoid picking all the cells from one list. The existence of the "next" pointer in each cell makes the cell connections easy to implement and quick to execute. The drawback of this logic is that when a particle changes the geometric cell it has

Variable	Type	Description
cells	Cell *	pointer to the first node of the Cell linked list
alloc	int	number of cells allocated in the cells linked list
datablocks		internal structure for cache alignment purposes

Table 3.6: Cellpool (struct)

to be stored in another cell list as well, since a specific geometric cell is represented by a specific list of the "cells" array. A separate class, RebuildGridMTWorker is implemented to handle the copying of the values to the proper cells. This means, that the program has to copy the values of three 3D floating point vectors every time the particle has to move in another cell.

In order to avoid this very costly part we implemented another data structure model which is depicted in figure 3.15. The main idea of the new version we implemented is only to change pointers instead of moving the actual values in the memory. The vectors that describe one particle will be stored in the same memory addresses during the whole execution, but they will be pointed by different cells according to their position that will probably change one or more times.

Firstly, we defined the "ParticleNode" structure which contains four 3D vectors (position, velocity, viscosity, acceleration) and one float value for the density. Additionally, it has a pointer of type "ParticleNode *" which points to its neighboring particle. The allocation of numParticles number of ParticleNode structures is happening in the beginning of the execution. Secondly, we defined the "Cell" structure which will represent a geometrical cell. Each cell points to a linked list of ParticleNodes and the structure itself just contains a "ParticleNode *" element pointing to the head of the list and an integer defining the size of this list. The allocation of the num_cells different "Cell" structures is occurring in the beginning as well. In this way, each time a particle changes cell we just have to change two pointers: the next pointer of its previous particle or the head pointer of its previous cell and the next pointer of the current particle. The drawback of this implementation is that after some iterations and if the fluid is moving a lot while the time is passing the particles of each cell will not be allocated in continuous places in the memory. This is very costly for the other steps of the program in which each thread is making calculations for one or more cells. It has a bad impact on the cache utilization when we have many available threads to handle the execution, since one thread might need to use between two iterations data from different memory addresses. As a result, it is better to use the new implementation if the program will be executed for a small number of frames but prefer the initial one if we have a huge number of frames.

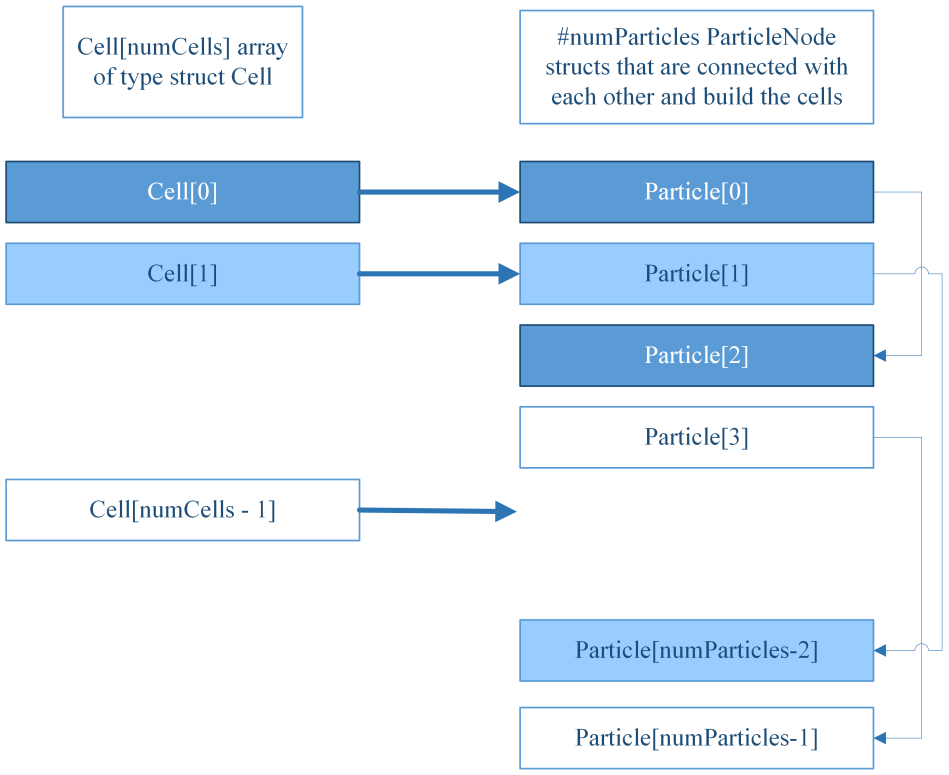

Figure 3.15: Cells and Particles representation

For all the following measurements the programs were executed for 500 frames by using the native input file of 18 MB which requires approximately 290 MB of memory allocation.

In figures 3.16.1 and 3.16.2 it is obvious to see that the new implementation improves the first step of the program. However, this improvement would not be the same for the other stpes. A suggestion

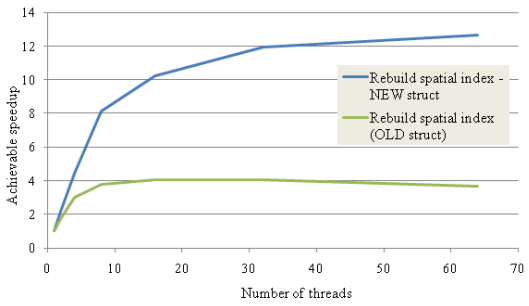

Figure (3.16.1) Scaling

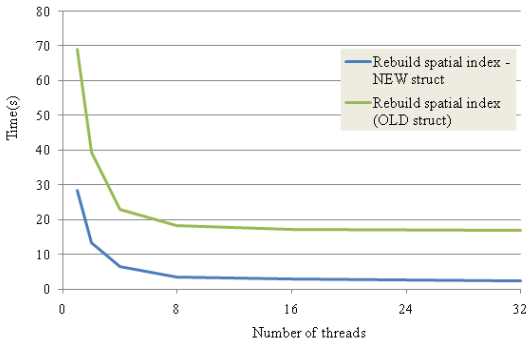

Figure (3.16.2) Time (sec) per step

Figure 3.16: Rebuild step - Using New and Old structure implementations

on that would be to implement a hybrid model, which would follow mainly the logic of the new implementation. In every 16 frames it would iterate through all the particles and perform copying of their data so that they can be stored in continuous memory addresses for each cell. It is worth mentioning, that there have been multiple attempts to optimize the current application. Jim Dempsey implemented fluidanimate using QuickThread Programming [Quic] and different data structures than the ones used in the initial program [Demp10]. His results both in terms of the impact that the different data structures have and of the benefits that QuickThread Programming has were very interesting to investigate further.

3.3.5 Measurements and performance analysis

Initially, we took measurements using the native input that was mentioned before for up to 64 threads. The best scaling was achieved for 32 threads, despite the fact that the machine that was used (Dunnington) has 24 available cores. Fluidanimate can only be executing by using a number of threads that is a power of two, so it was actually not possible to use exactly 24 threads. Figures 3.17 and 3.18 present how the total execution time was splited among the several steps and what was the achievable speedup. Regarding the "Rebuild spatial index" step, the bad scaling is due to the bus contention when

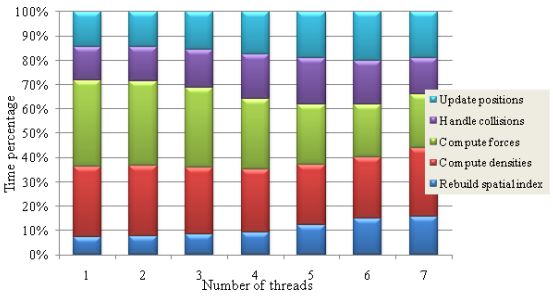

Figure (3.17.1) Time percentage spent in each step

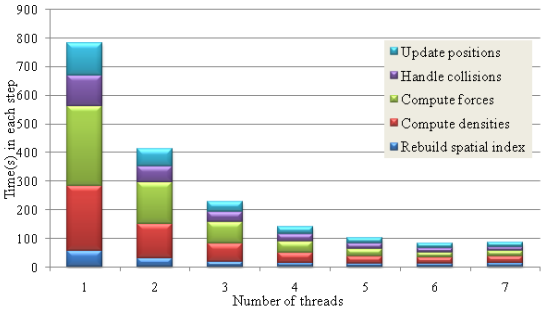

Figure (3.17.2) Time (sec) per step

Figure 3.17: Fluidanimate time per step

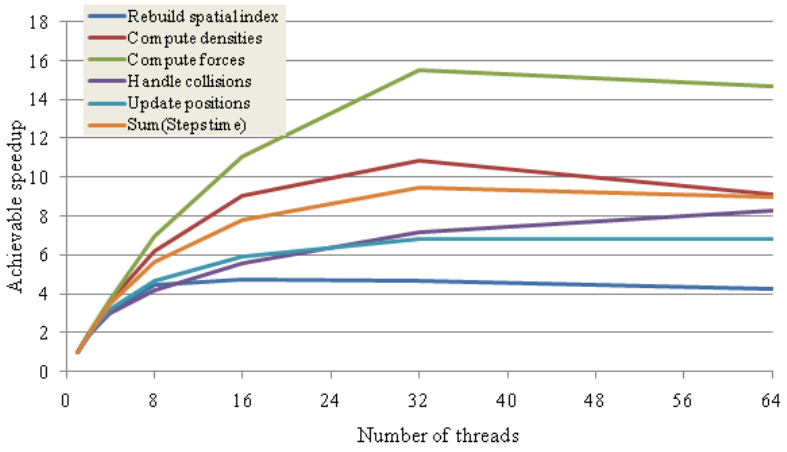

Figure 3.18: Scaling per step

all the particle's value have to be copied in the between the cells. This step was explicitly analyzed in part 3 and alternative solutions were presented. As far as the "Compute densities" and "Compute forces" steps are concerned, the number of locks that is increasing as the number of threads grows is the main reason for not being able to have a better scaling after the 16 threads. Until that point the scaling is almost ideal, but afterwards the improvement is very poor. The context switch rate and the volume of shared data are increasing [Pusu11]. In these steps each particle requires the data of its nine neighboring particles and when the number of threads increases it is more probable that this data will be part of another's threads region and locks have to be used. The last two steps ("Handle collisions" and "Update positions") are impacted by the bus contention. The number of computations that they perform is very low and not costly, but during their execution they need to read the values of all the vectors and update them as well. This increases the off-chip traffic, the read and write misses. Of course, depending on the cache size the performance of these steps can vary. It was also observed that having a larger cache line size is preferable, because the prefetching is valuable in the current data program where one particle's data consist of 13 floating point values.

In order to make more clear the impact of the locks in fluidanimate, we mocked the computations of steps 1, 2 and 3 and skipped the use of locks. Figure 3.19 shows the percentage of time that these steps allocate for the locks. From the other figures (3.20.1 and 3.20.2) that are similar to the above ones we can prove bad impact of the locks on the achievable speedup. Another issue that is worth

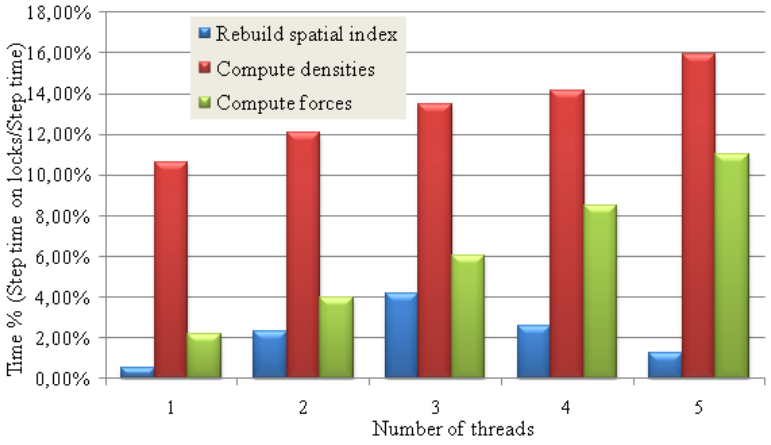

Figure 3.19: Time percentage spent on locks

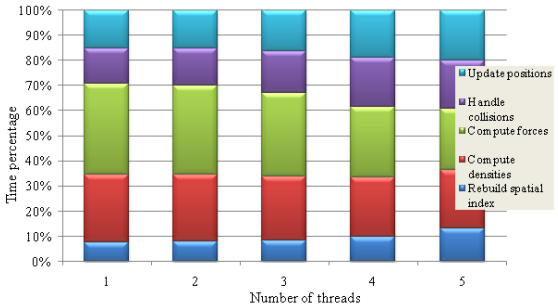

Figure (3.20.1) Time percentage spent in each step

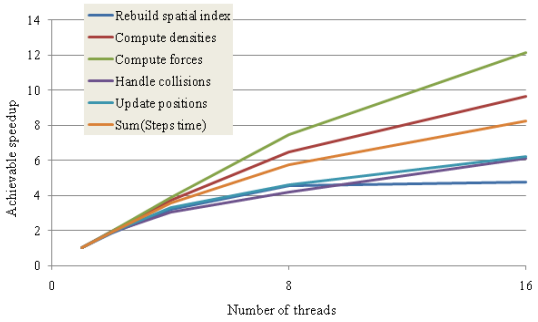

Figure (3.20.2) Scaling per step

Figure 3.20: Fluidanimate Run without locks

mentioning is the fact that fluidanimate depends a lot on the specific input file. Given an extreme case in which most of the particles of the fluid are concentrated in one cell and most of the image is empty the program will crash. One cell will have to store the data of all the particles and all the computations will be handled by a single thread regardless how many are in total available. Another case that is more probable to happen in the reality is that the fluid is moving from the one border of the image to the other one. During the whole execution all the cells will store at some point data and all the threads will be performing computations. However, in a single execution of a frame the snapshot of the image will include all the particles in just a few cells, meaning that only a few threads will be really taking part in the execution.

Of course, the goal of this application is to be able to observe in detail the behavior of each step regarding the cache utilization, the bus contention, the use of locks and other issues related to the multiprocessor architecture and not to have an excellent program for fluid simulation.

Chapter 4

Co-scheduling PARSEC applications

4.1 Introduction

Part of the current work is to analyze how the different applications of the PARSEC benchmark suite would behave if they had to be executed in parallel by the same computer. The applications have different characteristics with each other regarding their demands for memory, arithmetic operations, cache utilization, memory bus traffic, lock contention and thread migration. As a result, it is interesting to understand how the execution of one application would affect another one if both have to be executed in parallel given the same resources.

Currently, we divide eight of the PARSEC applications into pairs and execute them in the same machine of the laboratory (Dunnington). In our first scenario, we run each pair in parallel and each application is executed by 12 cores in order to make use of all the 24 available cores. The two applications use different caches, so that we can keep them as isolated from each other as possible. In our second scenario, we follow similar approach with the first one, but we let a cache be available for both applications. In other words, the cores that share a cache are allocated by both applications. In our last scenario, we allocate 12 cores out of the 24 available ones and execute each application serially.

4.2 Execution of PARSEC pairs applications: Observations and measurements

Firstly, regardless if the execution of a pair is parallel or serial, we run each of the application only once. As it is expected, the parallel execution is always achieving better results than the serial one. However, depending on the specific combination the achievable speedup differs a lot.

For instance, when bodytrack is running in parallel with x264 the scaling is only 1,22 whereas when bodytrack is being executed together with swaptions or freqmine the final speedup is 1,80 which is almost the ideal one. x264 is part of the media processing domain and is aiming to encode video by performing the required operations in different macroblocks of pixels. Both bodytrack and x264 are memory intensive applications and have a lot of off-chip traffic. Because both are storing their results into the new frames in each iteration, they need to perform many writes to shared data and thus when running together they can conflict on the memory bus. It is worth mentioning, that they do not have conflicts because of the cache sharing, since the time spent when executing them with separate caches is the same with the one achieved when sharing the caches. On the other hand, as it is already analyzed, each thread of freqmine is making a good use of the cache and like that this application does not have conflicts when running together with a memory bound one. Most of the data load and stores of a freqmine thread are occurring internally and there is not often the need to access the memory in order to process the execution. Bodytrack with the exception of x264 and canneal was achieving good speedup with its pairs. The reason is that bodytrack has many serial parts that spend

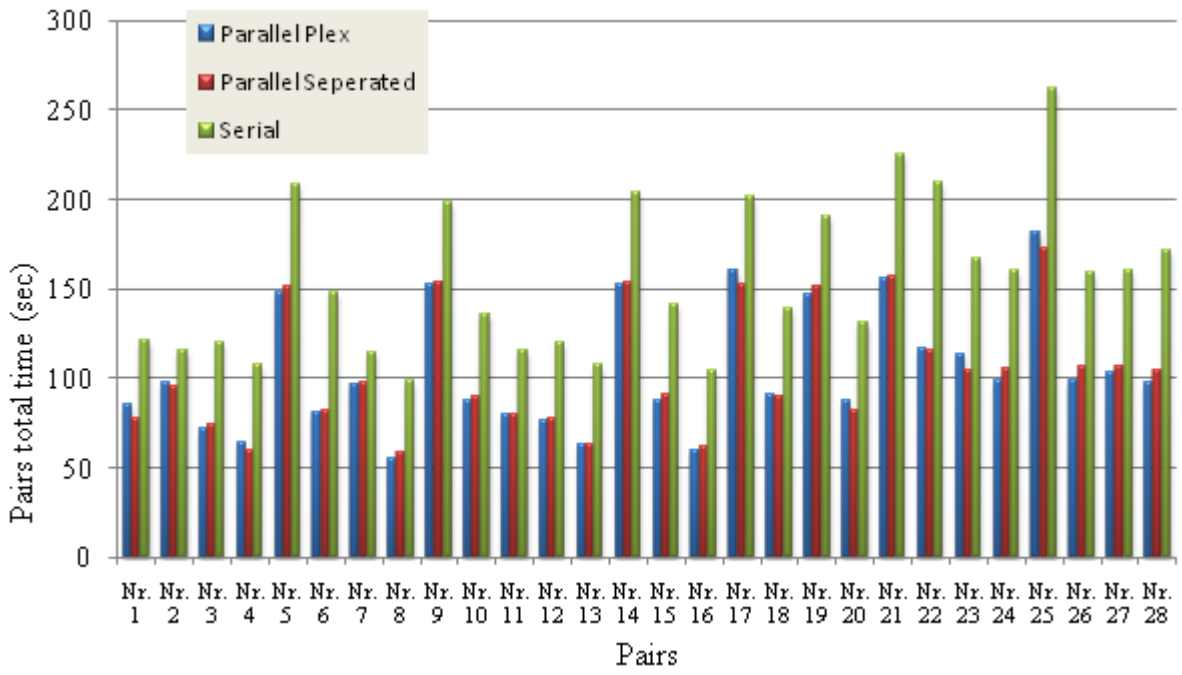

Figure 4.1: Time (sec) of Parsec application Pairs

big amount of the CPUs time, but this is not a barrier for other applications being executed at the same time. Bodytrack is accessing a lot the memory by the end of each frame computation, but not during the computations. Moreover, fluidanimate achieved pretty good execution time when running together with all the other applications. The average improvement introduced by the parallel execution when allocating 16 threads for fluidanimate and 8 for its pair in comparisson to the serial run time was 37%. Fluidanimate is divided in many steps and only two of them are memroy intensive. Most part of the application is spending time on locks or in copying elements between arrays, which is not influencing the behavior of other programs that can run in the same chip. For this reason, fluidanimate guarantees that it can have a good scaling when being scheduled together with any other application. Of course, the best improvement was observed with freqmine. As mentioned before, freqmine does not have any frequent off-chip traffic and this enables fluidanimate execute efficiently the steps that access often the memory. The best results were achieved by the pairs that included freqmine. However, only 30% improvement was introduced by the pair freqmine - canneal. Canneal is an memory intense application with a very big working set and even if it is being exeucted seperately the achievable speedup is very poor. When scheduling both applications to be executed multiple times in parallel, context switching will occur frequently.

For applications that use the cache efficiently this will become a bottleneck. In figure 4.1 we present the total time spent for each pair when executing serially, parallel with separate caches and parallel with shared caches. Moreover, we extract a more detailed graph (Figure 4.2.1) presenting the speedup of the different combinations for the applications that we analyzed above.

Additionally to that, we executed most of the above pairs in the same environment, but we let each of the application run multiple times. The motivation for that is that we do not want to have an application running while the other one is already executed. Based on the serial times, we customized the multiplications so that the total time the one application of the pair needs to run is the same with the total time of the other one in a serial environment. From these measurements figure 4.2.2 depicting the achievable speedup was extracted.

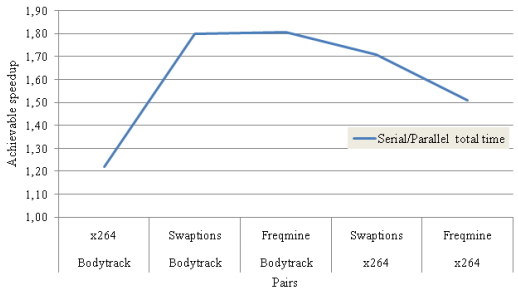

Figure (4.2.1) Scaling of combinations Bodytrack, x264, Freqmine and Swaptions applications

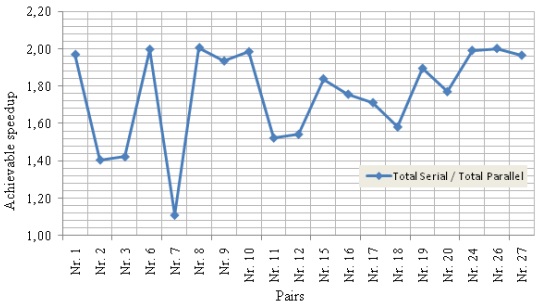

Figure (4.2.2) Scaling of Pairs running multiple times

Figure 4.2: Scheduling of PARSEC applications

Number	Application 1	Application 2
Nr. 1	Bodytrack	Blackscholes
Nr. 2	Bodytrack	x264
Nr. 3	Bodytrack	Ferret
Nr. 4	Bodytrack	Swaptions
Nr. 5	Bodytrack	Canneal
Nr. 6	Bodytrack	Freqmine
Nr. 7	x264	Ferret
Nr. 8	x264	Swaptions
Nr. 9	x264	Canneal
Nr. 10	x264	Freqmine
Nr. 11	x264	Blackscholes
Nr. 12	Blackscholes	Ferret
Nr. 13	Blackscholes	Swaptions
Nr. 14	Blackscholes	Canneal
Nr. 15	Blackscholes	Freqmine
Nr. 16	Ferret	Swaptions
Nr. 17	Ferret	Canneal
Nr. 18	Ferret	Freqmine
Nr. 19	Swaptions	Canneal
Nr. 20	Swaptions	Freqmine
Nr. 21	Canneal	Freqmine
Nr. 22	Fluidanimate	Freqmine
Nr. 23	Bodytrack	Fluidanimate
Nr. 24	x264	Fluidanimate
Nr. 25	Canneal	Fluidanimate
Nr. 26	Swaptions	Fluidanimate
Nr. 27	Ferret	Fluidanimate
Nr. 28	Blackscholes	Fluidanimate

Table 4.1: Points (struct)

Bibliography

[Bala05] Alexandru O Balan, Leonid Sigal and Michael J Black, "A quantitative evaluation of video-based 3D person tracking", in *Visual Surveillance and Performance Evaluation of Tracking and Surveillance, 2005. 2nd Joint IEEE International Workshop on*, pp. 349–356, IEEE, 2005.

[Bien11] Christian Bienia, *Benchmarking Modern Multiprocessors*, Ph.D. thesis, Princeton University, January 2011.

[Blac73] Fischer Black and Myron Scholes, "The pricing of options and corporate liabilities", *The journal of political economy*, pp. 637–654, 1973.

[Blai14] Lawrence Livermore National Laboratory Blaise Barney, "POSIX Threads Programming", Available from https://computing.llnl.gov/tutorials/pthreads/, 2014. [Online; accessed February 23, 2014].

[Cass09] Jimmy Cassis and Mario Flajslik, "CS315A Final Project: PARSEC Beyond 16", 2009.

[Demp10] Jim Dempsey, "Fluid Animate Particle Simulation", Available from http://www.drdobbs.com/parallel/fluid-animate-particle-simulation/228800371, 2010. [Online; accessed February 23, 2014].

[Docu14a] Intel Threading Building Blocks Documentation, "blocked range Template Class", Available from http://www.threadingbuildingblocks.org/docs/help/reference/algorithms/range_concept/blocked_range_cls.htm, 2014. [Online; accessed February 23, 2014].

[Docu14b] Intel Threading Building Blocks Documentation, "pipeline Class", Available from http://www.threadingbuildingblocks.org/docs/help/reference/algorithms/pipeline_cls.htm, 2014. [Online; accessed February 23, 2014].

[Grah03] Gösta Grahne and Jianfei Zhu, "Efficiently using prefix-trees in mining frequent itemsets.", in *FIMI*, vol. 3, pp. 123–132, 2003.

[Han00] Jiawei Han, Jian Pei and Yiwen Yin, "Mining frequent patterns without candidate generation", in *ACM SIGMOD Record*, vol. 29, pp. 1–12, ACM, 2000.

[Inte14] Intel®, "Threading Building Blocks (Intel® TBB)", Available from https://www.threadingbuildingblocks.org/, 2014. [Online; accessed February 23, 2014].

[Inve14] Investopedia, "Black Scholes Model", Available from http://www.investopedia.com/terms/b/blackscholes.asp, 2014. [Online; accessed February 23, 2014].

[Kari03] Takeaki Kariya and Regina Y Liu, "Options, Futures and Other Derivatives", in *Asset Pricing*, pp. 9–26, Springer, 2003.

[Lee10] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran and Nathan Clark, "Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications", in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 270–279, ACM, 2010.

[Lucc04] Claudio Lucchese, Salvatore Orlando, Raffaele Perego and Fabrizio Silvestri, "WebDocs: a real-life huge transactional dataset.", in *FIMI*, 2004.

[NTUA14] CSLAB NTUA, "Parallel Processing Systems Introduction", Available from http://cslab.ece.ntua.gr/courses/pps/files/fall2013/pps-parallel-programming-lec1-Fall2013.pdf, 2014. [Online; accessed February 23, 2014].

[Ocal02] Liadan O'callaghan, Nina Mishra, Sudipto Guha, Adam Meyerson and Rajeev Motwani, "Streaming-data algorithms for high-quality clustering", in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 0685–0685, IEEE Computer Society, 2002.

[Open14] OpenMP, "The OpenMP API Specification For Parallel Programming", Available from http://openmp.org/wp/, 2014. [Online; accessed February 23, 2014].

[PARS14a] PARSEC, "Download PARSEC 3.0", Available from http://parsec.cs.princeton.edu/download.htm#parsec, 2014. [Online; accessed February 23, 2014].

[PARS14b] PARSEC, "Overview PARSEC", Available from http://parsec.cs.princeton.edu/overview.htm, 2014. [Online; accessed February 23, 2014].

[Pusu11] Kishore Kumar Pusukuri, Rajiv Gupta and Laxmi N Bhuyan, "Thread reinforcer: Dynamically determining number of threads via os level monitoring", in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 116–125, IEEE, 2011.

[Quic] LLC QuickThread Programming, "QuickThread Programming, Parallel Programming Toolkit", Available from http://www.quickthreadprogramming.com/. [Online; accessed February 23, 2014].

[Reed11] Eric C Reed, Nicholas Chen and Ralph E Johnson, "Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't", in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pp. 133–138, ACM, 2011.

[Wiki14] Wikipedia, "Image Segmentation", Available from http://en.wikipedia.org/wiki/Image_segmentation, 2014. [Online; accessed February 23, 2014].