Concurrent computation of topological watershed on shared memory parallel machines

Ramzi Mahmoudi, Mohamed Akil, Mohamed Hedi Bedoui

CONCURRENT COMPUTATION OF TOPOLOGICAL WATERSHED ON SHARED MEMORY PARALLEL MACHINES

Ramzi MAHMOUDI[1], Mohamed AKIL[1], Mohamed Hédi BEDOUI[2]

[1]Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Equipe A3SI
ESIEE Paris - Cité Descartes, BP99, 93162 Noisy Le Grand, France

[2]Université Monastir, Laboratoire Technologie et Imagerie Médicale
Faculté de Médecine de Monastir, Rue Ibn Sina - 5019 Monastir, Tunisie

{mahmoudr, akilm}@esiee.fr, medhedi.bedoui@fmm.rnu.tn

Abstract:

The watershed transform is considered as the most appropriate method for image segmentation in the field of mathematical morphology. In the following paper, we present an adapted topological watershed algorithm suited for a rapid and effective implementation on Shared Memory Parallel Machine (SMPM). The introduced algorithm allows a parallel watershed computing while preserving the given topology. No prior minima extraction is needed, nor the use of any sorting step or hierarchical queue. The strategy that guides the parallel watershed computing, labeled SDM-Strategy (equivalent to Split-Distributes and Merge), is also presented. Experimental analyses such as execution time, performance enhancement, cache consumption, efficiency and scalability are also presented and discussed.

Keywords: Watershed transform; Parallel computing; Image Processing; Parallelization strategy; Computing methodologies, Enhancement ;Parallel algorithms, mathematical morphology.

Introduction

The watershed was extensively studied during the 19th century by J.C. Maxwell [1] and C. Jordan [2] among others. One hundred years later, the watershed transform was introduced by S. Beucher and C. Lantuéjoul[3] for image segmentation, and is now used as a fundamental step in many powerful segmentation procedures[4, 5]. Figure 1 gives a very symbolic description of the mentioned approach. In fact, it shows trends that use watershed transform for image processing.

In order to explain the concept of watershed, let us consider a grayscale image as a topographic surface: the gray level of a pixel becomes the elevation of a point, the basins and valleys of the topographic surface correspond to dark areas, whereas the mountains and crest lines correspond to the light areas. If the topographic relief is flooded by water, watersheds will be the divide lines of the domains of attraction of rain falling over the region [6] or sources of water springing from reliefs' peaks. Another synopsis that has shown consistency is when that topographic surface is immersed in a lake with holes pierced in local minima. Catchment basins will fill up with water starting at these local minima, and, at points where water coming from different basins would meet, dams are built. As a result, the topographic surface is partitioned into different basins separated by dams, called watershed lines.

To simulate these approaches, several techniques are deployed. The oldest one starts by finding basins, then watersheds by taking a set of complements. Other techniques use a boundary detection to rebuild watersheds. But the most innovative technique is proposed by our team [7, 8]; it allows to define rigorously the notion of a watershed in a discrete space and to prove important properties that are not guaranteed by most watershed algorithms [9]. This technique consists in lowering the values of the grayscale image - seen as a map - while preserving some topological properties, namely, the number of connected components of each lower cross-section. In this case, the watershed division is the set of points that are not in any regional minimum of the transformed map. It is important to note that our main interest here is for digital images that provide more rigors to define watersheds. Since, there is no unique definition of the path that a drop of water would follow in the discrete case. In general, three large classes of algorithms to compute watershed transform can be figured out. The first one is based on the flooding approach [10], the second is based on the topographical approach [11] and finally a third class is based on the topological approach [7].

An essential difficulty lies in the fact that the watershed transform is not a local concept. The decision whether a pixel belongs to a basin cannot be based on purely local considerations. Some algorithms' results depend also on the order in which pixels are treated during the execution. In the sequential case, this can be resolved by fixing the scanning order, and then a deterministic result is obtained. In a parallel implementation this is no longer true since the outcome depends on the relative time instants at which different processors treat pixels, and this is unpredictable in the case of asynchronous processors. Task becomes even more complicated if one wishes to parallelize algorithms of the topological class. The question is how to preserve the number of connected components of each lower cross-section despite asynchronous nature of the given threads. We must not fail to deal with this problem, which represents the true challenge of this work.

Figure 1: Watershed applications 1979-2015 ***(a)*** Cleavage fractures in steel, ***(b)*** contour of ***(a)*** the obtained truth watershed definition introduced by Beusher et al. [3] in 1979, ***(c)*** Maximum intensity projection of the original human lower limb ***(d)*** Bone tissue removed using mask extended with 3D watershed transform introduced by Straka et al.[12] in 2003, ***(e)*** original dental X-ray image ***(f)*** segment line obtained trough watershed transform introduced by Hui Li et al. [13] in 2012, ***(g)*** Original MRI brain image ***(h)*** Brain tumor extraction using a marker based Watershed algorithm introduced by Benson et al . [14] in 2015.

In this paper, we propose an adapted algorithm to compute a watershed. The introduced algorithm that is parallel preserves the topology of the input image. It does not need any prior minima extraction, and does not require any sorting step nor the use of any hierarchical queue. It is also suited for a rapid and effective implementation on Shared Memory Parallel Machine (SMPM).

We also present a tailored parallelization approach, called SD&M (Split Distribute and Merge) strategy that guides the parallel watershed computing. In fact, the splitting step is applied directly on an input graph when selecting sources. Unlike the conventional technique of division such as pixel division, or block division, the source selection is completely random. The associated stream computing is fully parallel (read mode data accesses). Then the distribution depends only on the available processors. This flexibility in data manipulation allowed us to obtain very good results especially in terms of efficiency without using the 'Basic-NPS' scheduler. Finally, the merging step allows the fusion of streams, two by two, to build the watershed.

To ensure an objective ranking among other watershed transforms, we propose a qualitative and quantitative study of Parallel Topological (PT) watershed. Experimental analyses such as execution time, performance enhancement, cache consumption, efficiency and scalability are also presented and discussed.

This paper is organized as follows: In section 2, we begin by identifying the global parallelization process to be pursued throughout this work. Performance indicators such as efficiency, scalability and portability of the introduced strategy are presented in a formal way. In section 3, the PT-watershed algorithm is put forward. In section 4, an overall assessment of the proposed algorithm is discussed. Finally, we conclude with a summary and a future work in section 5.

Global parallelization process

In this section, we begin by highlighting the real need for a common parallelization strategy of topological operators. After introducing the basic foundation for any successful parallelization, we will focus on Split Distribute and Merge (SD&M) strategy that we will attempt to classify over all existing strategies. Then we will put forward its detailed conception.

Lack of common parallelization strategy for topological operators

Bertrand [15] introduced connectivity numbers for the grayscale image. These numbers locally describe (in a neighborhood of 3*3) the topology of a point. According to this description, any point can be characterized following its topological characteristics. He also introduced some elementary operations able to modify the gray level of a point without modifying the image topology. These elementary operations of point characterization present the fundamental link of large classes of topological operators including, mainly, skeletonization and crest restoring algorithms [16]. This class can also be extended, under some conditions, to homotopic kernel and leveling kernel transformation [17], topological 2D and 3D object smoothing algorithm [18] and topological watershed algorithm [8] which is the focus of this article. All the mentioned algorithms do have also many algorithmic structure similarities. In fact, associated characterization procedures evolve until stability, which induce common recursively between different algorithms. Also the grey level of any point can be lowered or enhanced more than once. Finally, all the mentioned algorithms get a pixels' array as input and output data structure. It is important to mention that, to date, this class has not been efficiently parallelized like other classes as connected filter of morphological operator, as shown in Wilkinson's work [19]. Hence the need of a common parallelization strategy for topological operators that offers an adapted algorithm structure in a design space. The chosen algorithm structure patterns to be used in the design must be suitable for the SMP machines.

Fundamental basis for parallelization

Before defining the stages of parallelization of any sequential problem, it is essential to link the spectacular evolution of the parallel architectures and the parallel processing. In reality, if the parallelization strategies are so valuable, it is thanks to the substantial improvements in the multiprocessing systems and also to the rise of multi-core processors. In terms of feasibility, it will be easier to design architecture with a single fast processor (clock speed over 3 GHz) than another with many slow processors (clock speed around 1.5 GHz) with the same throughput. But during the last five years the clock speed of processors in multi-core architectures has increased by almost two and the associated cache

size has increased tenfold with the addition of a third cache level L3 which ensures an optimal L2 access speed while increasing the total cache. These twin barriers have flipped the equation, making multiprocessing practical even for small systems.

Generally five steps are necessary to move from a sequential algorithm running on single core architecture to a parallel algorithm that runs with a better performance on a multi-core architecture. G. Mattson et al., see [20], present the first four steps for parallel programming: Finding concurrency, algorithm structure, support structure and implementation mechanisms. In the following, we define the entire steps. We also focus on performance evaluation. A particular attention will be lavished to the second and the third phases which present a basis for our strategy.

(Def.1) **- Finding Concurrency Design Space** is the first analysis of the sequential algorithm to determinate the potential concurrency in terms of tasks and groups of tasks, shared data and task-local data.

After analyzing the original problem to identify exploitable concurrency, usually by using the patterns of Finding Concurrency Design Space (***Def. 1***), information about existing concurrent tasks, associated input data and dependencies are figured out. These elements are necessary to move to the Algorithm Structure Design Space (***Def.2***). G. Mattson et al. propose three possible organizations: organization by tasks, organization by data decomposition, and an organization by flow of data. To remain in the conceptual framework of this section, we limit our self to grassroots organization involving tasks and data.

(Def.2) **-** A**lgorithm Design Space** is the set of all possible algorithm designs and algorithm design parameters that represent how the extracted concurrency can be mapped onto elementary preprocessors.

In several cases, the problem can be decomposed to a finite set of tasks. Tasks can be grouped according to several criteria such as nature of the operation to achieve required operands, action-zone or returned results then groups of tasks can be defined. The way that the tasks within their group interact is the major feature of the concurrency. If the final solution is obtained after a single execution of all tasks and the tasks' dependency is null or quasi-null (temporary access to shared variables or messages exchange for synchronization), we can define the parallel task design. If processing is recursive, the problem can be solved by recursively dividing it into sub-problems, solving each sub-problem independently, and then recombining the sub-solutions into a solution to the original problem. This is the well know pattern of divide and conquer. It's important to note that the application of this principle cannot be independent from the type of the algorithm [21]. In other cases, global processing comes down to a continuous updating of a data structure. Thus it is better to think in terms of organizing data. G. Mattson et al. go further in this classification. He distinguishes between two particular cases: if the organization focuses on the distribution of data between elementary processors, then it's a simple data decomposition pattern. However, if the organization is the distribution of data between tasks' groups: it is a data flow decomposition pattern. More details will be given about this pattern in SD&M strategy classification (see section 2.3).

(Def.3) **- Architecture Design Space** describes the set of platform that support the extension of parallel programming. Pisces of information about how instructions are executed and how memory is managed are presented in this design.

Before a moving to coding, it is important to find out the most appropriate architecture to support the parallel algorithm using parallel architecture design space (***Def. 3***). This design presents a standard classification of parallel computer systems [22]. There are four types: SISD, SIMD, MISD and MIMD. The most significant structure encountered in the parallel application [23] is MIMD (Multiple Instruction, Multiple Data). In a MIMD machines the processors can execute different operations using their own data. Parallel processing via the application of MIMD machines promises a high performance, and experience with parallel processing is accumulating rapidly. In [24], Buzbee shows, through different examples, that a rapid progress is being made in the application of MIMD machines and that parallel processing can yield a high performance. We distinguish between two types of MIMD computers: Shared Memory MIMD machines and Distributed Memory MIMD machines. In the case of distributed memory machines, each processor has its own memory but this does not prevent its access to the memories of other processors if necessary.

(Def .4) **- Parallel Implementation mechanisms** are a set of tools used to write parallel programs. They are able to manage threads (or processes). Thread's synchronization and communication must also be guaranteed.

In contrast, in shared memory parallel machines, all processors share the same memory. Although the cost of an inter-processor or memory communication can be high, SMPM design still very efficient. In fact, this cost can be reduced by using the right mechanisms for parallel programming (***Def. 4***) such as *Locality-Aware Page Table* mechanism to enable thread and data mapping [25]. This mechanism detects the locality of memory accesses in the given hardware, and performs the mappings in the software. Other mechanisms allow a better exploitation of the target architecture through the use of threads. The most used tools within this framework are: MPI [26], OpenMP [27] and TBB [28]. Introduced as an open standard, OpenCL is also designed for programming heterogeneous parallel systems. Some

extensions exist [29] to enable the average OpenCL programmer to focus on the algorithm design rather than scheduling and to automatically gain performance without sacrificing programmability. After coding and running programs, it's important to evaluate the efficiency, the scalability and the portability of the code by using performance metrics for parallel programs (***Def. 5***). These concepts will be listed in details in the last part of this section.

(Def.5) - **Performance metrics of parallel programs** are a set of measurements that quantifies the parallel code such as efficiency, scalability and portability.

Classification of SD&M Strategy

As mentioned in section (2.1), the chosen algorithm structure patterns to be used in the design must be suitable for SMP machines. In reality, although the cost of communication (Memory-processor and inter-processors) is high enough, shared memory architectures meet our needs for different reasons: **(i)** These architectures have the advantage of allowing immediate sharing of data which is very helpful in the conception of any parallelization strategy **(ii)** They are from a non-dedicated architecture using a standard component that is economically reliable **(iii)** They also offer some flexibility in use for many application areas, particularly image processing.

Figure 2: Parallelization strategy approach - ***(a)*** Decision tree of algorithm structure design space ***(b)*** SD&M strategy design

In practice the most effective parallel algorithm design might make use of multiple algorithm structures thus the proposed strategy is a combination of the divide and conquer patterns and event-based coordination patterns (*fig*. 2.a). Hence the name that we have assigned is the SD&M (Split Distribute and Merge) strategy. Not to be confused with the famous approach of mixed-parallelism (combining data-parallelism and task-parallelism [30]), it is important to mention that our strategy **(i)** represents the last stitch in the decomposition chain of algorithm design patterns and it provides a fine-grained description of topological operators' parallelization while the mixed-parallelism strategy provides a coarse-grained description without specifying the target algorithm. **(ii)** It only covers the case of recursive algorithms, while mixed-parallelization strategy is effective only in the linear case. **(iii)** It is especially designed for the shared memory architecture with a uniform access.

SDM Strategy Conception

A parallelization strategy did not aim to optimize a single metric such as speedup. Other than improved performance in terms of execution time, a good strategy has to provide a balance between efficiency, scalability, and portability to dissolve all conflicts that exist between these three forces.

Actually, any strategy is facing two major barriers. First, the conflict between efficiency and portability: making a program efficient almost requires that the code takes into account the characteristics of the specific system on which it is intended to run, which limits portability. A design uses special features of a particular programming environment (as multi-thread environment) may lead to an efficient program for that particular environment, but it might be unusable for a different platform. Second, the conflict between scalability and portability: Improving the scalability is based on a good distribution of work over a finite number of processors for a better exploitation of the ***P*** processors' potential. This distribution limits the portability of the program since the number of processor is increased.

(Def. 6) - **Divide and Conquer pattern** is based on a multi-branched recursion. It solves the problem by recursively dividing it into sub-problems. After solving each sub-problem independently, it recombines the sub-solutions into a solution to the original problem.

The relative importance of these diverse metrics will vary according to the nature of the problem at hand. In our case we are dealing with a class of topological operators with common features, as it is shown in (section 2.1). Shared memory parallel architectures turned out to be the best suited for our needs (section 2.3). Therefore, Split Distribute and Merge strategy, that we propose, combine two patterns (see *fig*. 2.b): Divide and Conquer patterns (***Def. 6***) and Event-Based Coordination (***Def. 7***).

(Def. 7) - **Event-Based coordination** is used when dealing with an irregular, dynamic or unpredictable data flow.

The splitting phase

The Divide and Conquer pattern is applied first by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly. Splitting the original problem takes into account, in addition to the original algorithm's characteristics (mainly topology preservation), the mechanisms by which data are generated, stored, transmitted over networks (processor-processor or memory-processor), and passed between different stages of computation.

(Def. 8) - **Scalability** $\psi(P,P')$ is a property which exhibits a performance linearly and proportionally to the number of processors employed.

This first stage of division will primarily affect the rate of scalability (***Def. 8***) of our program. To mount it, we propose the following formalization. Since speedup is the most commonly used metrics for parallel programming, it seems to be a natural choice to begin with. So we assume that every program is made up of two parts, sequential and parallel, to establish the following definitions:

t_s	Processing time of the serial part of a program using one processor.
$t_p(P)$	Processing time of the parallel part of a program using P processors with $P \geq 1$.
$t_T(P) = t_s + t_p(P)$	Total processing time of the serial and parallel part of the program using P processors with $P \geq 1$.
$\alpha(P) = \dfrac{t_s}{t_s + t_p(P)}$	Scaled percentage of the serial part of the program using P processors with $P \geq 1$.
$\beta(P) = (1-\alpha(P)) = \dfrac{t_p(P)}{t_s + t_p(P)}$	Scaled percentage of the parallel part of the program using P processors with $P \geq 1$.

Now we can formalize the fixed-size speedup, which fixes the problem size and emphasizes how fast a problem can be solved. First, the theoretical approach speedup can be seen as the ratio of a quantity of works by a period of time:

(Def. 9) - *Speedup* = $\dfrac{Work}{Time} = \dfrac{W}{T}$

A second formal definition can be given by applying Amdahl's law [31] so the speedup can be defined by the ratio of the total processing time of the serial and the parallel part of the program using one processor by the total processing time of the same parts using ***P*** processors.

$$Speedup = \frac{t_T(1)}{t_T(P)} = \left[\frac{t_s + t_p(1)}{t_s + \dfrac{t_p(1)}{P}} \right]$$

This formula can be written differently using a non-scaled percentage $\beta(1)$ previously defined:

(Def. 10) - *Speedup* = $\left[\dfrac{1}{\beta(1) + \dfrac{1-\beta(1)}{P}} \right]$

An alternative formulation referred to as Gustafson's law [32] exists. This formulation calibrates the serial percentage according to the total parallel time using ***P*** processors.

$$t_T(P) = \alpha(P) + (1-\alpha(P))$$
$$t_T(1) = \alpha(P) + P(1-\alpha(P))$$

Thus, we can define the speedup as follows:

(Def. 11) - $Speedup = \left[\dfrac{t_T(1)}{t_T(P)} \right] = \left[\alpha(P) + P(1-\alpha(P))\right] = \left[P - (P-1)\alpha(P)\right]$

To show the work partition influence on the scalability rate $\psi(P,P')$, suppose that an algorithm runs on a first architecture using P processors with η_P efficiency. The shared amount of work is W_P. The same program runs on a second architecture using P' processor with $\eta_{P'}$ efficient. Shared amount of work is $W_{P'}$.
We recall that the efficiency is considered as the ratio of speedup by the number of processors (More details about efficiency will be given in the next section). Ideally, an algorithm should be effective on a wide range of numbers of processing elements, from two up to a decade. So if η_P and $\eta_{P'}$ represent the optimal efficiency rate, we can draw the following equation using ***(Def. 9)***:

$$\text{If } \eta_P = \eta_{P'} \Leftrightarrow \frac{Speedup(P)}{P} = \frac{Speedup(P')}{P'}$$
$$\Leftrightarrow \frac{W_P}{P * t_T(P)} = \frac{W_{P'}}{P' * t_T(P')} \Leftrightarrow \frac{t_T(P)}{t_T(P')} = \frac{W_P * P'}{W_{P'} * P} = \psi(P,P')$$

Thus, it follows that the only parameter that provides a linear performance in proportion to the number of processors ***(Def. 8)*** is the ratio $\left[\frac{W_P}{W_{P'}}\right]$, hence the importance of the splitting step. Unfortunately such an impact cannot be shown simply by applying Gustafson approach ***(Def. 11)***.

Scalability will be expressed only in terms of the number of processors.

$$\text{If } \eta_P = \eta_{P'} \Leftrightarrow \frac{P-(P-1)\alpha(P)}{P} = \frac{P'-(P'-1)\alpha(P')}{P'}$$
$$\Leftrightarrow P'\left[P-(P-1)\alpha(P)\right] = P\left[P'-(P'-1)\alpha(P')\right]$$
$$\Leftrightarrow P'*P-\left[P'(P-1)\alpha(P)\right] = P*P'-\left[P(P'-1)\alpha(P')\right]$$
$$\Leftrightarrow \left[P'(P-1)\alpha(P)\right] = \left[P(P'-1)\alpha(P')\right]$$
$$\Leftrightarrow \left[P'(P-1)\frac{t_s}{t_s+t_p(P)}\right] = \left[P(P'-1)\frac{t_s}{t_s+t_p(P')}\right]$$
$$\Leftrightarrow \left[\frac{P'*(P-1)*t_s}{t_s+t_p(P)}\right] = \left[\frac{P*(P'-1)*t_s}{t_s+t_p(P')}\right]$$
$$\Leftrightarrow \left[\frac{(P'*P-P')*t_s}{t_s+t_p(P)}\right] = \left[\frac{(P*P'-P)*t_s}{t_s+t_p(P')}\right]$$
$$\Leftrightarrow \left[\frac{P'*(P-1)}{t_s+t_p(P)}\right] = \left[\frac{P*(P'-1)}{t_s+t_p(P')}\right]$$
$$\Leftrightarrow \left[\frac{P'*(P-1)}{t_T(P)}\right] = \left[\frac{P*(P'-1)}{t_T(P')}\right]$$
$$\Leftrightarrow \left[\frac{t_T(P)}{t_T(P')}\right] = \left[\frac{P'*(P-1)}{P*(P'-1)}\right] == \psi(P,P')$$

Distribution phase

We attach a great importance to work distribution because it is a fundamental step to ensure a perfect exploitation of multi-cores architecture's potential. We'll start by briefly recalling some basic notions of distribution techniques then we will introduce our minimal synchronization approach that is particularly suitable for topological recursive algorithms where a simple point characterization is necessary. Our approach is general and applicable to shared memory parallel machines.

The main challenge when performing parallel operations on a simple point characterization is the dynamic nature of work distribution. Since the workload is not known a priori, assigning work units to different cores in advance is impossible. In the literature, there are two main approaches for multi-core work distribution: the first one, called work queues approach, consists on using a shared work queue in main memory and controls the access to it via primitive synchronization. The second approach is work stealing. In this case, every core has a separate work queue which is still accessible to other processors. Cores can steal work units from others' queues whenever their own queue is empty. However, all these techniques do not currently work well on new architectures as Xeon for many reasons. Primarily, work-stealing has also been known to be cache-unfriendly for some applications due to the randomized stealing [33]. For tasks that share the same memory footprints, the randomized locality of the oblivious work-stealing schedulers does nothing to ensure the scheduling of these tasks on workers that share a cache.

(Def. 12) **- Efficiency** is the cost of what is actually produced or performed with what can be achieved with the same consumption of resources (processor frequency, memory size, surface, etc.). It is an important factor in the determination of productivity.

This significantly limits, not only scalability, but also efficiency (***Def. 12***) for some memory-bandwidth bounded applications on machines that have separate caches.

$$\textit{Efficiency} = \eta_P = \left[\frac{Speedup}{P}\right] = \left[\frac{P-(P-1)\alpha(P)}{P}\right] \text{ according to (}\textbf{\textit{Def. 11}}\text{)}$$

$$= \left[\frac{W}{Pt_T}\right] \text{ according to (}\textbf{\textit{Def. 10}}\text{)}$$

It is also important to mention that using memory fence operations, consistency can be enforced, but with a relatively high overhead. Even if memory consistency was not a problem, busy waiting such as by spinning on a lock variable is relatively inefficient on architecture with high memory latency and multi-threaded hardware execution can also lead to priority inversion and prevent other threads on the same core from performing a useful work.

Figure 3: Auto-supplying task system design

The main idea of our approach is an auto-supplying task system. Keeping the local queues filled will be our major goal (see *fig*. 3). Local threads should never have to be broadcast over processors because of an empty queue. They should always find something in their local queues due to an auto-supplying task system that allows an automatic check of different queues then a permanent redistribution of tasks to maintain a certain balance between all processors.

Let's consider a distributed job list where no jobs are duplicated anywhere so each processor local list is unique and exclusive and jobs can be moved between processor only before they go into the "executing" status. Despite that each processor balances itself by requesting or stealing work units from others queues whenever its own queue is empty, we design an auto-supplying system using a shared work queue located in the principal memory that supplies the processor shortage of work. The system maintains a minimum level of work in its queue by importing extra work from other processors queues. This is the automatic load balancing feature that keeps all processors busy. It is important to note that the system does not need to predict in advance which machine must have the most work. Thus there is no burdening; a single system will automatically decide when to move tasks from one queue to another.

(Def.13) **- Portability** is a property which assures that parallel programs are both code portable and performance portable to various parallel machines.

Despite the centralized aspect, our design does not depend on the number of processors, or the minimum load of processors which makes our approach more generic and our parallelization strategy more portable. In fact portability is increasingly cited as a desirable goal in parallelization strategy conception.

$$\textit{Portability: } \sigma_P^k(b \to t) = \frac{S_P^t}{S_P^b} * 100\% = \frac{\left[P-(P-1)\alpha(P)\right]^{tS}}{\left[P-(P-1)\alpha(P)\right]^{bS}} * 100\%$$

Despite the disagreement about the exact meaning of "Portability", we can consider (***Def. 13***) as an accurate definition. According to James D. Mooney [34] the primary goal of portability is to facilitate the activity of moving an application from an environment in which it currently operates to a new or target environment. This activity has two major

aspects: **(i)** transportation - physical movement of the program's instructions and data to the new environment and **(ii)** adaptation - modification of the information as necessary to work satisfactorily in the new environment.

We skip "Adaptation" which involves higher level modifications that might be necessary to adjust the program to work with aspects of the new environment that are intentionally or unavoidably different from the old one. We focus on the physical transportation which includes the use of compatible media or communication channels between processors, and interpreting and translating file formats, character codes, data representations and processor designs. Standard languages and portable compilers bridge the gap between programs and the variety of the CPU interfaces that exists in target environments.

However, many of these mechanisms still define only a part of the environment interface that many applications need. Elements such as file structures, device control, memory management, asynchronous event handling, or the user interface are not adequately defined by most language standards or library specifications. When requirements for communication, concurrency, or timing constraints exist, conventional languages are clearly insufficient.

The Merging phase

The key problem of each parallelization is the merging obtained results. Normally this phase is performed at the end of the process when all results are returned by all threads which usually mean that only one output variable is declared and shared between all threads. But as we mentioned in section 2.2, we are dealing with a dynamic evolution so we can plan the following: After two threads are finished, they directly merge and a new thread is created. This implies the creation of some shared FIFO queue containing all inserted neighbors by both two parent threads. Only one shared data structure will contain pixels lowered by all threads. In the merging threads, there is no hierarchical order; the only criterion is finish time. It is also important to mention that only newly created threads can modify the created FIFO queue and one neighbor cannot be inserted twice. It is a precaution in order to minimize the consumed cache.

Construction of parallel topological watershed

In this section, we start by introducing some basic definitions of the stream notion which is crucial to the flooding paradigm. Then, we introduce in details our parallel topological watershed. An illustration of a parallel computation process is given. Execution time and cache consumption are performed and analyzed. Efficiency and scalability are also presented and discussed.

Basic notions and definitions

Based on the Cousty's approach [35], we define and illustrate the stream notion. For the sake of simplicity, we restrict ourselves to the minimal set of notions that will be useful for our purpose.

We denote by V an edge-weighted graph. Let $L \subset V$. We say that L is a stream if, for any two points x and y of L, there exists, in L, either a path from x to y or from y to x, with the steepest descent for F.

Now, let's consider a stream L, we say that $x \in L$ is a top of L if the altitude of x is greater than, or equal to the altitude of any $y \in L$. If the altitude of x is less than the altitude of any y, then x is considered as a bottom of L.

Let's consider two disjoint streams L_1 and L_2, with $L_1 \cap L_2 = \phi$. Let L be the union of both streams with $L = L_1 \cup L_2$.We say that L_1 is under L_2, written $L_1 \, \mathrm{p} \, L_2$, if there exist a top x of L_1 and a bottom y of L_2, and there is a path L from y to x, with the steepest decent for F. If $L_1 \, \mathrm{p} \, L_2$ then L is considered as a stream. If there is no stream under L, L is considered as an $\mathrm{p}-stream$. Now, any stream L which contains $\mathrm{p}-streams$ is itself an $\mathrm{p}-streams$.

Figure 4: Stream notion illustration - following Cousty's approach.

A Basic illustration of stream notion is given by ***(fig. 4): (a)*** the red superimposed graphs are the minima of corresponding functions. Let us consider G and F as associated graph and depicted function, ***(b)*** the sets $L = \{a,b,e,i\}$ and $\{j,m,n\}$ are two examples of streams, ***(c)*** the set $L = \{i,j,k\}$ is not a stream since there is no path in L, between i and k, with a steepest descent for F.

Note that the sets $\{a,b\}$ and $\{b\}$ are respectively the set of bottoms and tops of L. Here the sets L is under the stream $\{j,m,n\}$ and thus $\{a,b,e,i,j,m,n\}$ is also a stream. There is no stream under $\{a,b,e,i\}$ and $\{a,b,e,i,j,m,n\}$. They are considered as two $\mathrm{p}-streams$ and they contain the set $\{a,b\}$ which is the vertex set of minimum of F.

Streams extracted by Cousty function are $\mathrm{p}-streams$. In the following we recall the link that exist between $\mathrm{p}-streams$ and minima. Let L be a stream. If L is $\mathrm{p}-stream$ then L contains the vertex set of minimum of F and for any $y \in V \setminus L$ adjacent to a bottom x of L, $F(\{x,y\}) > F(x)$. Actually, if L is an $\mathrm{p}-streams$, then the set of all bottoms $\{b_1, b_2, ..., b_n\} \in L$ constitutes the vertex set of a minimum of F. A subset L of V is considered as the vertex set of a minimum of F if and only if it is an $\mathrm{p}-streams$ minimal for the inclusion relationship. We will now move on to define the flow family notions. Actually the vertices of a graph can be arranged in the following manner with the aim of partitioning the vertex set of G from $\mathrm{p}-streams$ of F. Let $\zeta = \{L_1, ..., L_n\}$ be a set of n $\mathrm{p}-streams$. ζ is said to be a flow family if $\cup\{\zeta_i \mid i \in \{1,...,n\}\} = V$ and for any two families L_1 and L_2 in ζ, if $L_1 \cap L_2 \neq \phi$, then $L_1 \cap L_2$ is the vertex set of a minimum of F.

Figure 5: Watershed computing principal - ***(a)*** Input image ***(b)*** Associated weighted graph ***(c)*** Output watershed

Now, we can more formally define the watershed-cut. Let L be a flow family. Let us denote by $M_1, \ldots, M_n$ the minima of F. Let ψ be the map from V to $\{1,...,n\}$ which associates to each vertex x of V, the label i such that M_i is the unique minimum of F included in an $\mathrm{p}-stream$ of L which contains x; we say that ψ is a flow mapping of F. In that case, the set $S = \{\{x,y\} \in E \mid \psi(x) \neq \psi(y)\}$ can be considered as a flow cut for F. As a result, the set $S \subset E$ is considered as a watershed of F if and only if S is a flow cut for F.

In order to compute a watershed, we will go through this relationship established by Jean Cousty, and we propose a new one, that is based on the parallel extraction of streams. That is able to produce a flow-cut hence a watershed. A general illustration is given by ***(fig.5)***.

Parallel stream computing

For that purpose, following an algorithm that will assign, in a parallel way, a label to each point of the graph. Actually, from each non-labeled point x, a stream L composed of non-labeled points and whose top is x is computed. It is important to mention that streams computing at this level are completely independent then streams can be completely computed in parallel, see ***(fig. 6)***. For N point (x_1, x_2… x_n), their associated flows are simultaneously extracted: (L_1, L_2… L_n).

Figure 6: Flow computation - ***(a)*** Partition of input image ***(b)*** Parallel stream computation

Each flow 'L_i' is composed of a point not yet labeled and whose source is x_i. The stream function proposed by Cousty, pleaded in line 5 ***(Alg. 1)***, is launched N times. It allows the extraction of $L_{i \in \{1,2,3...n\}}$. Intuitively, it explores the path of a greatest slope, by mixing iterations first in-depth and width of the approaches. The main invariants of this function are the set 'L', for each iteration, a stream (flow) and the set L' (line 2 - stream function) include all wells of L not yet explored.

Algorithm 1 : Parallel Topological Watershed [Mahmoudi et al.]

Input : (V, E, F) : Edge-weighted graphs;
Output : Ψ : Flow partition of F
foreach x ∈ V do Ψ(x) ← No-Label ; // **No data dependency - FULL PARALLELISM**
nb_labs ← 0 ; // **Global shared attributed label**
i ←0 ; treated stream
foreach (x ∈ V) such as (Ψ(x) = = No-Label) do // **launch N process in parallel**
  [L_i,lab_i] ← **Stream** (V, E, F, Ψ,x_i) ; // **to get associated stream for each x_i**
nb-fusion = i ;
while (nb_fusion != 1)
  for (j =0 ; j <= nb_fusion ; j+=2) do // **launch (nb_fusion/2) process at once**
    if ($L_j \cap L_{j+1}$) = ∅ then **s-labeling** ([L_j,lab_j] , nb_labs) ;
                s-labeling ([L_{j+1},lab_{j+1}] , nb_labs) ;
    else **f-labeling** ([L_j,lab_j] , [L_{j+1},lab_{j+1}] , nb_labs) ;
nb-fusion = nb-fusion / 2 ;

The stream function ***(Alg. 2)*** halts at line 16 when all bottoms of L have been explored or, at line 8, if a point z already labeled is found. In the former case, the returned set L is an $\mathrm{p}-stream$. In the latter case, the label lab of z is also returned and there exists a bottom y of L such that $\langle y, z \rangle$ is a path with the steepest descent.

Algorithm 2: Stream function [Cousty et al.]

Input : (V,E,F) : Edge-weighted graphs; Ψ : a label of V; x : point of V;
Output: [L, lab] : L is a flow obtained from x (source of L) ; lab is the associated label to an Θflux included in L or (–1).
L ←{x}
L'←{x} // the set of sources not yet explored of L
While there exists (y ∈ L') **do**
  L' ← L' \ {y};
  breadth_first ← TRUE ;
  While (breadth_first) **and** (∃ {y,z} ∈ E / z ∉L and F({y,z})= F(y)) **do**
    If (Ψ(z) # No_label) **then**
        return [L,Ψ(z)] // exist an Θflow L already labeled
    Else if ($F^-(z) < F^-(y)$) **then**
        L ← L ∪ {z}; // z is the only well of L
        L' ← {z}; // switch the in-depth exploration first
        breadth_first ← FALSE
      Else
        L ← L ∪ {z}; // therefore z is a well of L
        L' ← L' ∪ {z}; // continue exploration in width first
return [L,-1]

Thus, there is an $\mathrm{p}-stream$ L_1, under L, included in the set of all vertices labeled lab. Remark that, in stream function, the use of breadth-first iterations is required to ensure that produced set L is always an $\mathrm{p}-stream$. Otherwise, if only depth-first iterations were used, the stream could be stuck on plateaus (connected sub-graphs of G with constant altitude) since some bottoms of L would never be explored.

At the end of flow the function executing, a family $\mathcal{L}$, of ***N*** streams (***L_1***, ***L_2***… ***L_n***) whose elements must be labeled is generated. The initial procedure [in the iterative case] is to assign a new label (***nb_labs***) to each '***L_i***' element if the latter is a $\mathrm{p}-stream$. If it is not the case, the old returned label ***lab***, of the $\mathrm{p}-stream$ '***H_i***', included in '***L_i***', is assigned to the different elements of '***L_i***'. Now if we want to launch this procedure in a parallel manner, ***N/2*** flows can be treated at once.

The procedure, in the parallel case, is based on the idea of labeling and merging two obtained flows at once. If two flows (to merge) '***L_i***' and '***L_{i+1}***' contain no common summit, ($L_i \cap L_{i+1}$)=∅, meaning there are no common wells

between the two sources 'x_i' and 'x_{i+1}', in this case the merging is simple, for each flow 'L_i' and 'L_{i+1}', see ***(fig 7.a).*** Note that ***s-labeling*** function ***(Algo. 3)*** launches only the initial procedure [used previously in the iterative case].

Figure 7: Stream merging - ***(a)*** Merging streams without common wells ***(b)*** Merging streams with common wells

If the two flows (to merge) 'L_i' and 'L_{i+1}' contain a common summit, $(L_i \cap L_{i+1}) \neq \varnothing$, meaning there are common wells between the two sources 'x_i' and 'x_{i+1}', see ***(fig 7.b)***, in this case, merging is more complicated. We developed ***f-labeling*** procedure ***(Algo. 4)*** that is able to make a fusion in the following special cases: ***(i)*** 'L_i' and 'L_{i+1}'are two $p-streams$, ***(ii)*** 'L_i' and 'L_{i+1}'are two $streams$ including two $p-streams$, ***(iii)*** 'L_i' is an $p-stream$ and 'L_{i+1}' is a $stream$ including an $p-stream$.

Algorithm 3 : Function s-labeling [Mahmoudi et al.]

Input: (L, lab, nb_labs)
if (lab = -1) **then** // L is $p-stream$
  nb_labs ++ ;
  foreach (y ∈ L) **do** Ψ(y) ← nb_labs ;
else
  foreach (y ∈ L) **do** Ψ(y) ← lab ;
return NULL

The major problem in concurrent merging of multiple flows is summed up in labels' assignment. If two streams share the same well, which label should be given to involved pixels? Our proposed solution is inspired from the flooding paradigm. Indeed, we started by studying all possible cases of merging two water streams gushing from different sources, see (***fig. 8***). Our goal is to identify which stream will be the first to reach the well. This latter will mark the well by its own label. The starting point is the steepest descent approach with the following conditions: ***(i)*** Water flow rate is identical for all sources, ***(ii)*** Flow surface is perfectly smooth and ***(iii)*** Runoff velocity is uniform for each flow. If these conditions are fully met, three factors come into play to determine the flow velocity: the source altitude, distance between source and sink, and finally the slope. In fact, the topographic slope particularly influences the runoff. The inclination of the slope is surely the most important topographical aspect. Normally, its impact is limited on a short slope. It is more visible on a longer slope even if runoff needs a certain distance to reach its maximum velocity. The Mathematic formulation of a flow medium speed can be is given by the Chezy formula: $V_c = c\left(h * s\right)^{1/2}$ that was introduced in 1769. 'C' refers to the roughness coefficient of Chezy. 'S' refers to the slope, and 'h' refers to the altitude of the source.

Figure 8: Merging techniques approach

If we draw the truth table with these three factors (d: distance, s: slope, a: altitude), by varying one parameter each time, we can identify only five possible cases: The two streams have the same altitude, slope and distances that separate sources from the well. In the 2nd case, both flows traverse the same distance but slopes and sources altitudes are different. In the 3rd case, the two streams run down the same slope but they travel different distances since sources altitudes are different. In the 4th case, the altitude is the same for both sources, but traveled distances and slopes are different. Finally, the altitude of the sources, the distances separating them from the well and the slopes of the followed paths are different for the two streams.

Algorithm 4 : Function f-labeling [Mahmoudi et al.]

Input: (L_a, lab_a, L_b, lab_b, nb_labs)
// L_a AND L_b ARE two $p-stream$
if (laba = -1) && (labb = -1) then
nb_labs ++ ;
Attrib_lab(L_a,L_b,nb_lab) ;
// L_a OR L_b INCLUDES an $p-stream$ **already labeled**
else if (($lab_a \neq -1$) && ($lab_b = -1$))
Attrib_lab(L_a,L_b,lab_a) ;
else if (($lab_a = -1$) && ($lab_b \neq -1$))
Attrib_lab(L_a,L_b,lab_b) ;
// L_a AND L_b INCLUDE two $p-stream$ **already labeled**
else if (**AVspeed**(L_a)> **AVspeed**(L_b))
Attrib_lab(L_a,L_b,lab_a) ;
else **Attrib_lab**(L_a,L_b,lab_b) ;
Return NULL
Function Attrib_lab(L_1,L_2,lab) :
foreach ($z \in \{L_1 \cap L_2\}$) do $\Psi(z) \leftarrow$ lab ;
foreach ($x \in L_1$) such as ($\Psi(x)$ = = No-Label) do $\Psi(x) \leftarrow$lab;
foreach ($y \in L_2$) such as ($\Psi(y)$ = = No-Label) do $\Psi(y) \leftarrow$lab;
Return Null

The question now is: does one of these five situations necessarily appear when merging? If we are dealing with two $p-stream$, this problem does not arise because we are forced to generate a new label for the identified wells (line 1, ***Algo. 4***). Also, if one of the two streams includes an $p-stream$, it means there exists one label what is already generated that we can assign to the common wells (line 5, ***Algo. 4***). Finally, if both streams include $p-stream$ then, two labels already exist. In order to decide which one to assign, we compute approximately the flow's average speed using Chezy formula. It is important to mention that the gray level of a pixel represent its altitude. Altitude refers to vertices values in the graph. Slope and distance between sources and wells can be computed through pixels' coordination. According to fixed conditions, roughness coefficient is equal to one.

Results and discussion

The main objective of this session is to present a qualitative and quantitative analysis of the parallel topological (PT) watershed transform introduced in this paper.

We start by including comparisons with previous studies. We will learn from different syntheses presented in Roerdink [36] and Audigier [37] works. We will also recall our analysis [38] of watershed transform (WT) in the discrete case about WT based on flooding, WT based on path-cost minimization, WT based on topology preservation, WT based on local conditions and WT based on a minimum spanning forest.

The following table summarizes comparison criteria between different algorithms. Selected criteria are justified by our objective to identify a ranking for the proposed algorithm.

Table 1 : Comparison between main watersheds' transformation

The starting point is the definition space; we note that IFT-Watershed, MSF-Watershed and PT-Watershed definitions are limited to the discrete space while the other watersheds definitions are spread into a continuous space.

IFT-Watershed, MSF-watershed, LC-Watershed and PT-watershed belong to a region based watershed transform family since pixels are assigned to basins. Flooding-Watershed, TD-Watershed and Topological-Watershed form the line based watershed family since some pixels are labeled as watershed. Only Topological-Watershed defines lines that consistently separate basins while Flooding-Watershed and TD-Watershed merely swing between thick and disconnected watershed lines.

Through definitions, only Flooding-Watershed and TD-Watershed return a unique solution while all other definitions return multiple solutions. Note that a set of solutions returned by the IFT-Watershed can be unified by creating litigious zones when solutions differ [37].

All algorithms, that do not exactly include their definitions, return unique solution but do not preserve the number of connected components of the original input image. Actually, Vincent-Soille, Meyer and Lin's algorithms do not preserve important topological features. Only IFT-Watershed, Topological-Watershed, MSF-Watershed and PT-watershed are correct from this point of view.

Taking into consideration the computing process, only Flooding-Watershed needs pixels' sorting while other transformation will pass this costly step. But this does not preclude associated algorithms to use hierarchical structures when being implemented. Except MSF-watershed and PT-watershed transformations that do not need any hierarchical queue. Vincent and Meyer's algorithms impose also a prior minima computation, which is not the case of the others.

In terms of complexity, we observe that Vincent and Soille algorithm runs in a linear time with respect to the number N of pixels in the image which is processed. In most current situations of image analysis, where the number of possible values for priority function is limited and the number of neighbors of a point is a small constant, Couprie's algorithm runs also in a linear time with O(n+m) complexity. IFT-Watershed, MSF-Watershed and PT-Watershed algorithms run also in linear time. Cousty's algorithm is executed at most O(|E|) times as well as the proposed algorithm.

Trough this analysis, MSF-Watershed and PT-Watershed algorithms hold best characteristics. The fact that the sorting step is not required, the hierarchical queue is not used and minima are not computed and they are top ranked. To go further in this analysis, we turn to a quantitative assessment. A preliminary comparison between IFT-Watershed, Flooding-watershed, MSF-watershed and PT-watershed, in terms of execution time, is given by the following figure. It is important to mention that we are programming with tcl/tk using the Ubuntu operating system.

Figure 9: Global watershed transformation profiling - ***(a)*** Execution time of IFT-Watershed, Flooding-watershed, MSF-watershed and PT-watershed ***(b)*** Execution time of MSF-watershed and PT-watershed ***(c)*** Execution time of PT-watershed

It perfectly shows that PT-watershed transform has the best execution time. Indeed, the performance of the proposed algorithm becomes visible when the image size exceeds 640*640 pixels. These preliminary assessments were made on a single processor. For a deeper assessment to prove the adequacy between our algorithm and the SMP architecture which is main interest of this article, we move to a performance evaluation through the SDM strategy metrics already introduced. We begin by presenting test conditions. Then, the obtained results in terms of execution time and cache use are presented and discussed using variable architectures. Based on these results, we compute efficiency and scalability of our implementation. We enhance our discussion on scalability by computing the amount of work required to reach the average speed. Unfortunately, portability will not be assessed for purely technical reasons.

Table 2: Standard tested images

For profiling, we used a microscopic view of a cross-section of a uranium oxide ceramics (see *fig. 14.a*). To choose the right size, we compared the number of streams' intersections during the merging step for each image. The obtained results (see *table 2*) show that the cut-size (*640*640*) is the most appropriate for profiling. Indeed, for cuts with a less size, number of full intersections (which means that some common wells are detected) is very low compared to the number of empty intersections (which is the ideal case - labeling is done in parallel with new labels). Concerning big size cuts, the total intersection number is very high which may cause much confusion when cache profiling (Determinate instructions' number).

Table 3: Used processors features

Wall-clock execution times for numbers of threads equal to 1, 2, 4, 8, 16 and 32 were determined. We considered a commonly used Intel processor configuration. The number of processors varies from one to eight. The frequency varies between 1,73 GHz and 3,4 GHz as shown in table 3. The ***L1*** caches have at least a 32-byte block size, while the capacity vary between 16 Kbytes and 32 Kbytes, and for the associativity, only eight ways are considered. The ***L2*** caches have at least a 64-byte block size, while capacities varies between 512 Kbytes and 8 Mbytes, and the associativity varies between two and sixteen ways. The minimum value of two timings was taken as the most indicative of algorithms' speed. The results of implementation on the different architectures are shown in the following table.

Table 4: Wall clock (ms)

We note that the execution time drops from an average of 4636 ms with a single thread on one CPU down to 713 ms with 8 threads on 8 CPUs. The speed up was computed using formula Ts/Tp with Ts for 1 CPU = 4360 ms. A remarkable result about speedup is also shown in table 5.

In fact, speed-up increases as we increase the number of threads beyond the number of processors in our machines. In the first implementation, using two CPUs, the speedup at 2 threads is 1.37±0.01. However, for the second implementation, using 8 CPUs, the speedup has increased to 6.11±0.01.

Another common result between different architectures is the stability of execution time on each ***p***-core machine since the code uses n or more threads. For a better illustration, we establish the execution time and the speedup curve (see *fig.10).*

Table 5: Performance improvement

Figure 10: Overlap time - ***(a)*** Execution time ***(b)*** Performance

In the following we present our experimental analysis of caches. As a result of this experiment, *(fig. 11.a)*, we found that two performance regions are clearly evident: In the leftmost region, as long as the cache capacity can effectively serve the growing number of threads. Increasing the number of threads improves the performance, as more processors are utilized. This area is generally identified as cache-efficiency zone. Balanced workloads offer a higher locality and a better exploitation of cache and hence expand the cache efficiency zone to the right and up. An outstanding example is given by table 6 which summarizes the number of instructions, ***L1*** and ***L2*** data misses on four architectures using SMP scheduling policy. We note that the number of instructions increases from an average of ($34x10^6$) instr. on 1 ***CPU*** to ($790x10^6$) instr. on 8 ***CPUs***.

To highlight the cache performance, we compute also the waiting status which refers to the delay experienced by the processor when accessing the external ***L2*** caches each time that information is missing in ***L1***. Since ***L1miss*** is followed either by an ***L2hit*** (success) or ***L2miss***, the waiting status can be computed by following the given formula: the sum of ***L2hit*** and ***L2miss***. We suppose that ***L2*** access time is estimated at 10 cycles (in hit case) and 100 cycles (in miss case). $WS_{cm} = ((D1miss - L2Dmiss)*10) + (L2dmiss*100)$.

Figure 11: Cache profiling - ***(a)*** Number of instructions ***(b)*** L1d Miss ***(c)*** L2d Miss ***(d)*** Evaluation of wait status

To estimate the lost time during the memory access, we simply multiply the waiting status by P4 660 frequency (3.6 GHz) and E5405 frequency (2 GHz). Thus we realize that the estimated lost time on 8CPU is insignificant compared to the lost time on 1CPU. This result is very visible on the ***E8400*** and ***E5335*** architectures. For the ***E5405*** architecture, the result is less visible due to the cache structure: While ***E5405*** is considered as eight CPUs architectural, but physically they are two ***Quads*** on the same chip (***L2*** = 2x4Mb).

Table 6: Cache memory consumption

In the sequel, we turn to efficiency evaluation, using ***Def. 12*** *(with ts = 360ms)*, in order to describe the exploitation degree of each processor in used SMP machines. As introduced in [42], this profiling will highlight the limitations introduced by the parallel watershed implementation on SMP machines. Indeed, the efficiency decreases by **30%** when switching from mono core architecture to a dual core one. Despite a slight increase on quad core architecture, the efficiency is **20%** lower than that measured with the 1CPU. More details are shown in table 7, see also ***fig. 12***.

Table 7: Efficiency values

The causes for losses of efficiency can be explained by the following reasons, partially introduced in [42] as parallel computing delays: ***(i)*** I/0 delays due to the need to distribute parallel data across local PE data stores. ***(ii)*** Communication delays, due to the need for PES to access data which are not located in their local data stores. ***(iii)*** Set-up delays due to the set-up of control and the processing logic and the network for inter-PE communication.

Figure 12: Efficiency improvement

With a further evaluation, we extend the speedup profiling of parallel watershed computing into a scalability analysis. According to Fruehe theoretic study [43], a very high scalability can be achieved on a multi-core architecture. For instance, a dual-core architecture offers a scalability of roughly ***80%*** for the second processor, depending on the OS, application, compiler, and other factors. That mean that the first processor may deliver ***100%*** of its processing power, but the second processor typically suffers from some overhead from multiprocessing activities. As a result, the two processors do not scale linearly. Thus, a dual-processor system does not achieve a ***200%*** performance increase over a mono-core architecture, but instead it provides approximately ***180%*** of the performance that a single-processor system provides.

In our evaluation framework, we first introduce the average unit speed. This parameter, seen as the ratio between the achieved speedup and the number of processors, will be very useful to determinate the scalability. We can also extend this definition into the maximum average speed which is defined as a ratio of maximum achieved speedup by the number of processors.. $Max(Av_{us}) = \frac{\max(A_{speed})}{P}$ The obtained results are introduced in the following table:

Table 8: Maximum Average Unit Speed

The concerning scalability, see ***Def. 8***, it can be written as $\psi(P,P') = \frac{P' \times W}{P \times W'}$. In this formula, W refers to the amount of work of our algorithm when P processors are employed and W' refers to amount of work of our algorithm when P' processors are employed to maintain the average speed. In an ideal situation, W' is equivalent to $\frac{P' \times W}{P}$ thus $\psi(P,P')$ is equal to 1. Unfortunately, this never happens in a real situation, actually $W' > \frac{P' \times W}{P}$, thus $\psi(P,P') < 1$.

To calculate the different values of efficiency foreach architecture, we must first determine the necessary amount of work W', as shown in ***Table 9***, to reach the Average Speed Unit Av_{us}. Note that the chosen Average Unit Speed is 0,787 on 4 CPUs using 4 Threads (Associated W = 334.020.732).

Table 9: Average speed Unit - using 4 and 8 CPUs

The scalability results for the parallel watershed processing are shown in ***table 10***. Our experiments demonstrate a very good scalability across all tested architectures.

Table 10: Scalability profiling

As the number of threads increases, a linear speedup has been observed (see ***fig. 13***). Also, the speedup improves as the problems' size increases. Note that when number of threads exceeds the number of cores, the total execution time is dramatically reduced. The difference between each efficiency curve with the ideal curve (constant efficiency equal to 1) decreases as the number of threads increases.

Figure 13: Scalability improvement

Conclusion

In this paper, we have presented an adapted algorithm to compute a watershed (see *fig. 14*) that is parallel, preserves the topology of the input image, does not need a prior minima extraction and suited for SMP machines. Considered as major contribution of this work, Parallel Topological (PT)-Watershed does not require any sorting step, or the use of any hierarchical queue.

The second contribution concerns an adequate parallelization approach that guides the parallel watershed computing. In fact, splitting step is applied directly on an input graph when selecting sources. Unlike the conventional technique of division such as pixel division, or block division, the source selection is completely random. The associated stream computing is fully parallel (read mode data accesses). Then the distribution depends only on the available processors. This flexibility in data manipulation allowed us to obtain very good results especially in terms of efficiency without using 'Basic-NPS' scheduler. Finally, the merging step allows two by two streams fusion to build a final watershed. The emphasis is mostly on the fundamental bases for a successful parallelization on a Shared Memory Parallel Machine (SMPM).

Figure 14: PT-watershed illustration - ***(a)*** Cross-section of a uranium oxide ceramics ***(b)*** Output image using PT-Watershed Transform

The third contribution concerns a qualitative and quantitative study of the PT-watershed algorithm in order to guarantee an objective ranking among other watershed transforms. Experimental analyzes such as execution time, performance enhancement, cache consumption, efficiency and scalability are also presented and discussed.

Note that our algorithm cannot be applied directly over a grayscale image. Actually three major steps are needed: the passage from grayscale image to edge-weighted graphs, then the application of the parallel watersheds-cut algorithm on the plot and finally the visualization of the graph in the Khalimsky space [44, 45].

References

1. J. Maxwell, On hills and dales, Philosophical Magazine vol. 4/40 (1870) 421-427.

2. C. Jordan, Nouvelles observations sur les lignes de faite et de thalweg , Comptes Rendus des Séances de l'Académie des Sciences vol. 75 (1872) 1023-1025.

3. S. Beucher, C. Lantuéjoul, Use of watersheds in contour detection, Int. Workshop on Image Processing Real-Time Edge and Motion Detection/ Estimation (1979) 17-21.

4. F. Meyer, S. Beucher, Morphological segmentation, Journal of Visual Communication and Image Representation vol. 1 (1990) 21-46.

5. S. Beucher, F. Meyer, The morphological approach to segmentation: the watershed transformation, In Dougherty ed. Mathematical Morphology in Image Processing Marcel Decker (1993) 433-481.

6. J. Serra, Image Analysis and Mathematical Morphology, Academic Press Inc., Orlando, FL, USA (1982).

7. M. Couprie, G. Bertrand, Topological grayscale watershed transform, SPIE Vision Geometry 5 Vol. 3168 (1997) 136-146.

8. G. Bertrand, On topological watersheds, Journal of Mathematical Imaging and Vision Vol. 22 (2005) 217-230.

9. M. Couprie, L. Najman, G. Bertrand, Quasi-linear algorithms for the topological watershed, Journal of Mathematical Imaging and Vision Vol. 22 (2005) 231-249.

10. P. Soille, Morphological Image Analysis, Springer-Verlag New York, Inc. Secaucus, NJ, USA (1999).

11. F. Meyer, Topographic distance and watershed lines, Signal Processing Journal Special issue on mathematical morphology and its applications to signal processing Vol. 38 (1993) 113-125.

12. M. Straka, A. Cruz, A. Köchl, M. Šrámek, E. Gröller, D. Fleischmann, 3D Watershed Transform Combined with a Probabilistic Atlas for Medical Image Segmentation, Journal of Medical Informatics and Technologies (2003) 1-10.

13. H. Li, G. Sun, H. Sun, W. Liu, Watershed algorithm based on morphology for dental X-ray images segmentation, IEEE 11th International Conference on Signal Processing (2012) 877-880.

14. C.C. Benson, V. L. Lajish, K. Rajamani, Brain tumor extraction from MRI brain images using marker based watershed algorithm, International Conference on Advances in Computing, Communications and Informatics (2015) 318-323.

15. G. Bertrand, J. C. Everat, M. Couprie, Topological approach to image segmentation, SPIE Vision Geometry V, Vol. 2826 (1996) 65-76.

16. M. Couprie, F. N. Bezerra, G. Bertrand, Topological operators for grayscale image processing, Journal of Electronic Imaging Vol. 10 (2001) 1003-1015.

17. G. Bertrand, J. C. Everat, M. Couprie, Image segmentation through operators based on topology, Journal of Electronic Imaging (1997) 395-405.

18. M. Couprie, G. Bertrand, Topology preserving alternating sequential filter for smoothing 2D and 3D objects, Journal of Electronic Imaging Vol. 13 (2004) 720-730.

19. M.H.F. Wilkinson, H. Gao, W.H. Hesselink, J.E. Jonker, A. Meijster, Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines, IEEE Transactions on Pattern Analysis and Machine Intelligence (2008) 1800-1813.

20. T. G. Mattson, B. A. Sanders, B. Massingill, Patterns for parallel programming, Addison-Wesley Professional, Boston, USA (2004).

21. L. Wangqing, S. Mingren, P. Ogunbona, A New Divide and Conquer Algorithm for Graph-based Image and Video Segmentation, IEEE 7th Multimedia Signal Processing Conf. (2005) 1-4.

22. M. J. Quinn, Parallel Computing : Theory and practice, McGraw-Hill Series in Computer Science, New York, USA (1994).

23. Ad. J. Van Der Steen, Overview of recent supercomputers, Report, NCF/HPC Research, The Netherlands (2008).

24. B. L. Buzbee, Applications of MIMD machines, In Computer Physics Communications Vol. 37 (1985) 1-5.

25. E. H.M. Cruz, M. Diener, M. A. Z. Alves, L. L. Pilla, P. O.A. Navaux, LAPT: A locality-aware page table for thread and data mapping, In Parallel Computing Journal Vol. 54 (2016) 59–71.

26. Foster, Designing and Building Parallel Programs, Addison Wesley, 1st Edition, Boston, USA (1995).

27. R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, Parallel Programming in OpenMP, Morgan Kaufmann, 1st Edition, Massachusetts, USA (2000).

28. J. Reinders, Intel Threading Building Blocks, O'Reilly Media Inc. CA, USA (2007).

29. M. Aji, A. Peña, P. Balajic, W. Fengd, MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL, Parallel Computing Journal Vol. 58 (2016) 37–55.

30. S. Ramaswamy, S. Sapatnekar, P. Banerjee, A framework for exploiting task and data parallelism on distributed memory multicomputers, IEEE Transactions on Parallel and Distributed Systems Vol. 8 (1997) 1098-1116.

31. G. M. Amdahl, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, AFIPS Conference Proceedings Vol. 30 (1967) 483–485.

32. J. L. Gustafson, Reevaluating Amdahl's Law, Communications of the ACM 31(5) (1988) 532-533.

33. U. A. Acar, G. E. Blelloch, R. D. Blumofe, The data locality of work stealing, Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures (2000) 1-12.

34. D. Mooney James, Strategies for Supporting Application Portability, in Computer, Vol. 23 (11) (1990) 59-70.

35. J. Cousty, G. Bertrand, L. Najman, M. Couprie, Watershed Cuts: Minimum Spanning Forests and the Drop of Water Principle, IEEE Trans. Pattern Anal. Mach. Intell. (2009) 1362-1374.

36. J.B. T.M. Roerdink, A. Meijster, The watershed transform: Definitions, algorithms and parallelization strategies, Fundamenta Informaticae, Vol. 41 (2001) 187-228.

37. R. Audigier, R. A. Lotufo, Watershed by image foresting transform, tie-zone, and theoretical relationship with other watershed definitions, In Mathematical Morphology and its Applications to Signal and Image Processing (2007) 277–288.

38. R. Mahmoudi, M. Akil, Analyses of the Watershed Transform, In International Journal Of Image Processing (IJIP) Vol. 5 Issue 5 (2011) 521-541.

39. L. Vincent, P. Soille, Watersheds in digital spaces : An efficient algorithm based on immersion simulations, In IEEE Trans. Pattern Analysis and Machine Intelligence Vol. 13 (1991) 583-598.

40. R. A. Lotufo, A. X. Falcão, The Ordered Queue and the Optimality of the Watershed Approaches, 5th International Symposium on Mathematical Morphology (2000) 341-350.

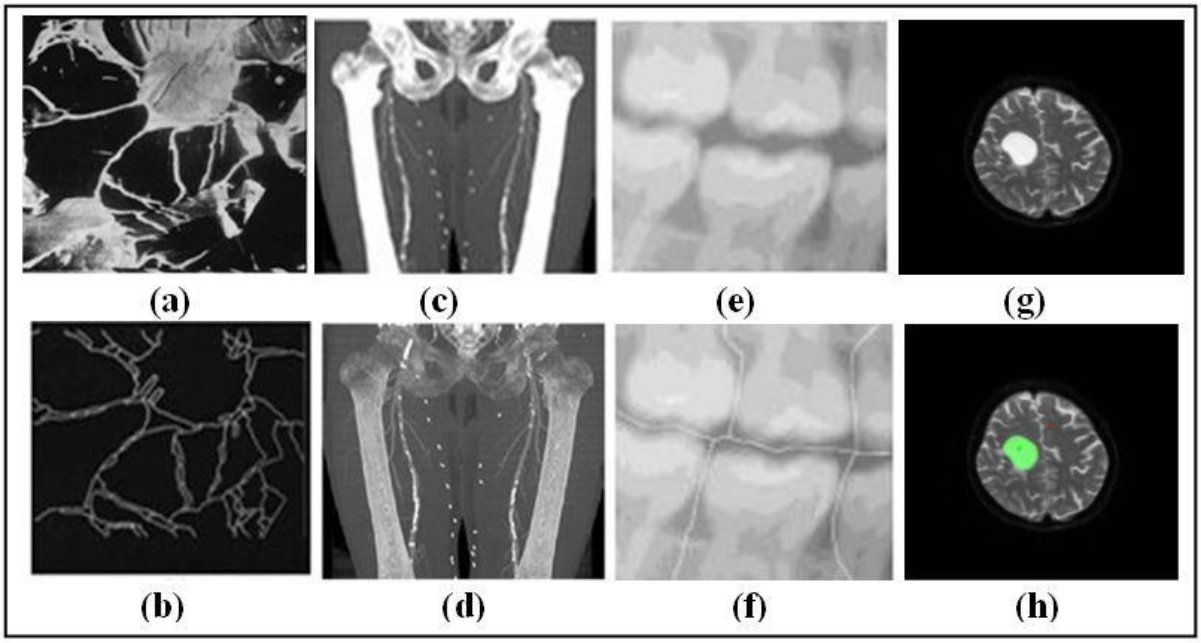
41. Y.C. Lin, Y.P. Tsai, Y.P. Hung, Z.C. Shih, Comparison Between Immersion-Based and Toboggan-Based Watershed Image Segmentation, 15th IEEE Transactions on Image Processing Vol. 3 (2006) 632-640.

42. H.T. Kumm, R.M. Lea, Parallel Computing Efficiency: Climbing the Learning Curve, 10's Ninth Annual International Conference on Frontiers of Computer Technology (1994) 728-732.

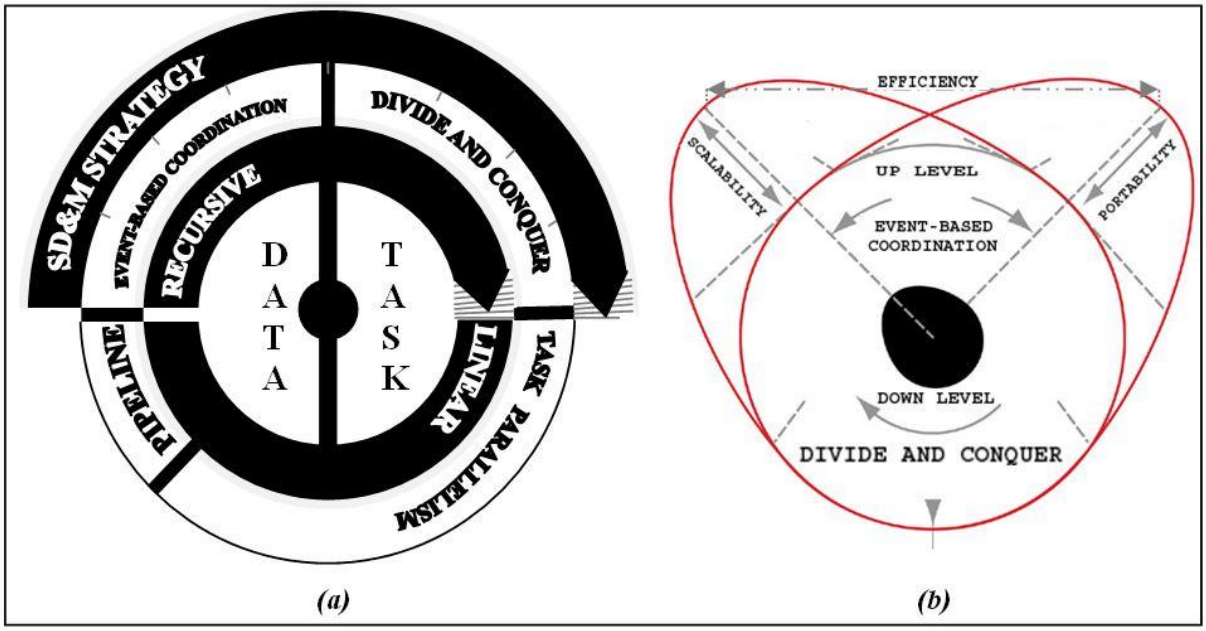
43. J. Fruehe, Planning Considerations for Multicore Processor Technology, Dell Power Solutions (2005).

44. E. Khalimsky, Topological graph theory foundations of design and control in multidimensional discrete systems, IEEE international conference on System, Man, and cybernetics (1994) 1628-1933.

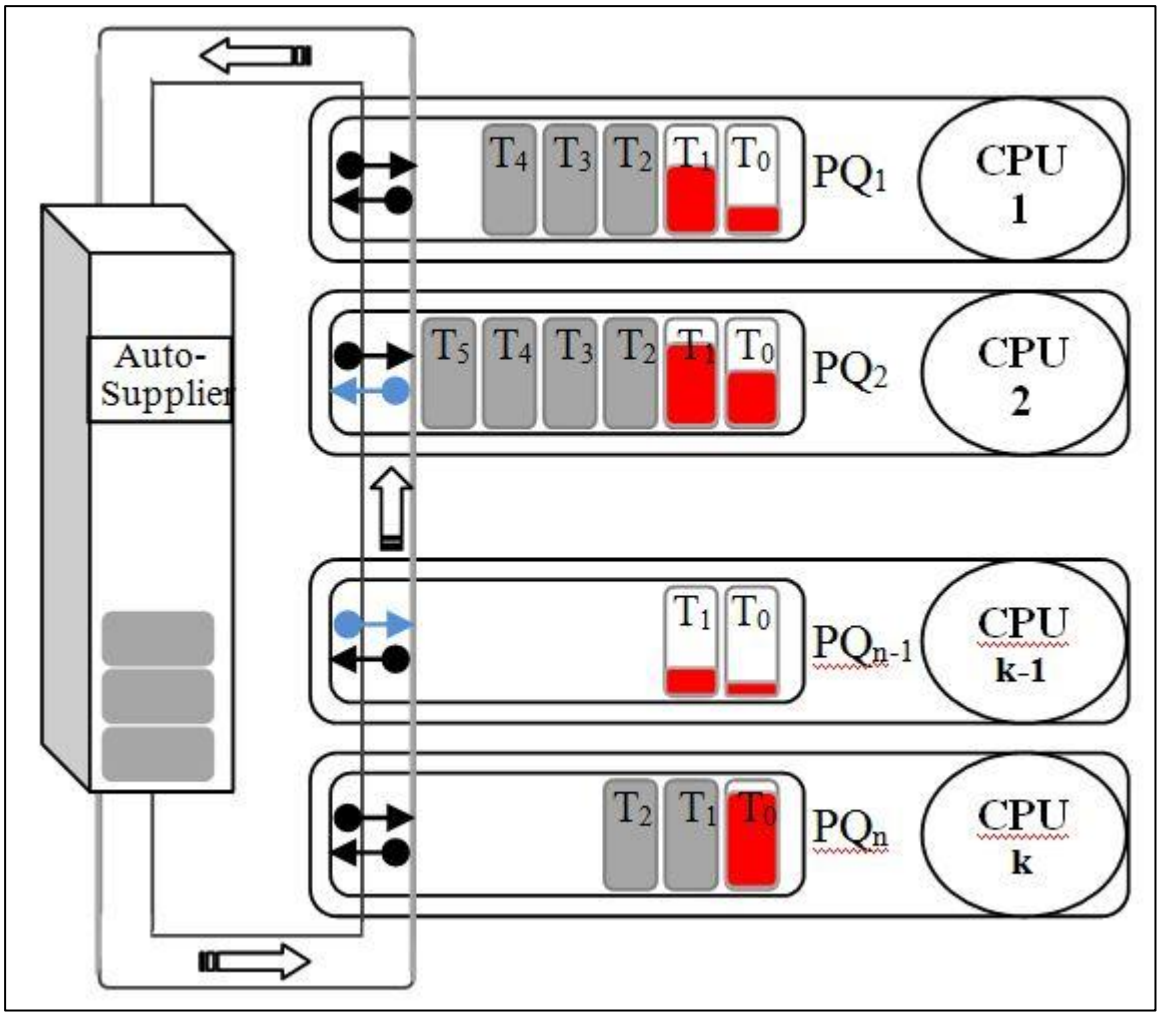
45. E. Khalimsky, Topological structures in computer science, Journal of Appl. Math. Simulation Vol. 1 (1) (1987) 25-40.



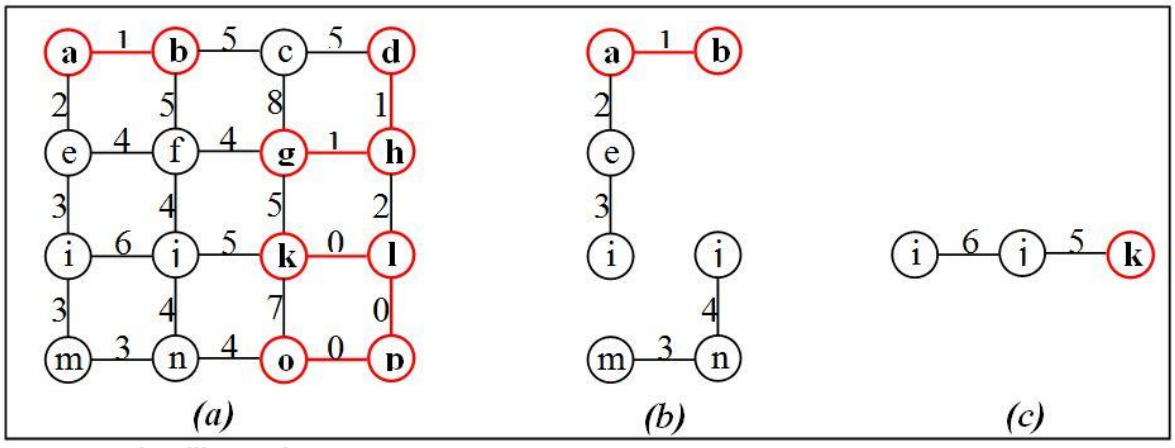
Watershed applications 1979-2015



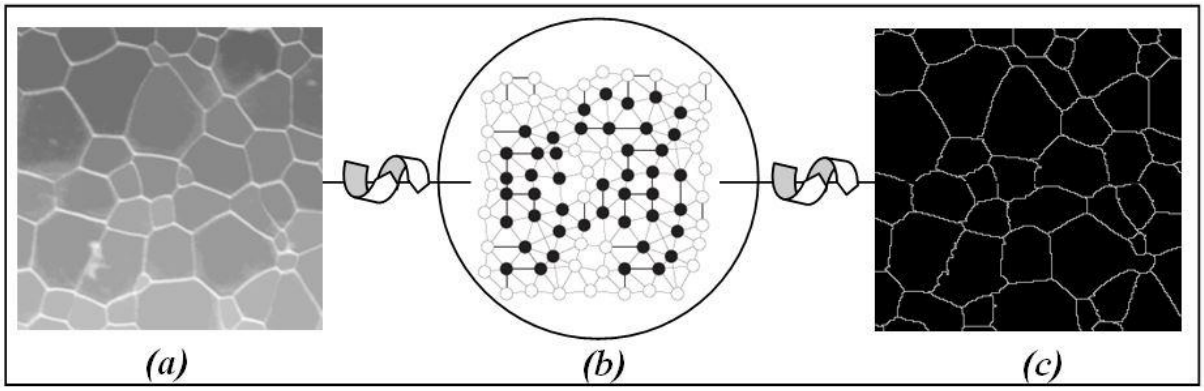
Parallelization strategy approach



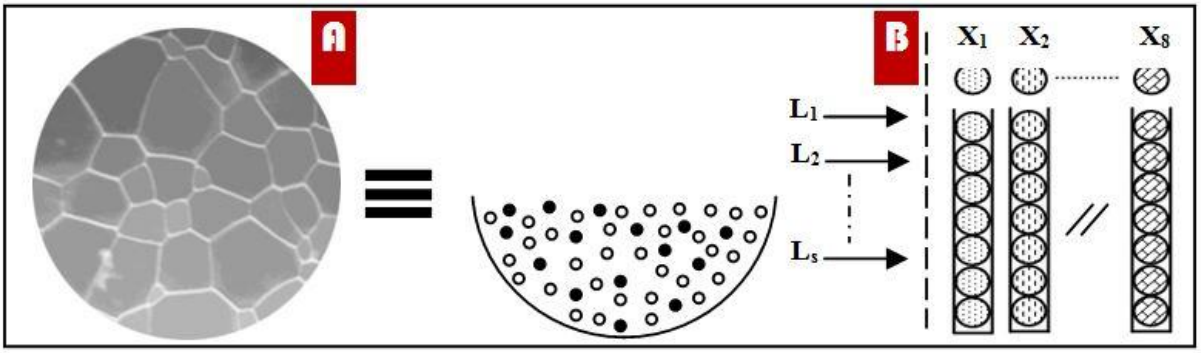
Auto-supplying task system design



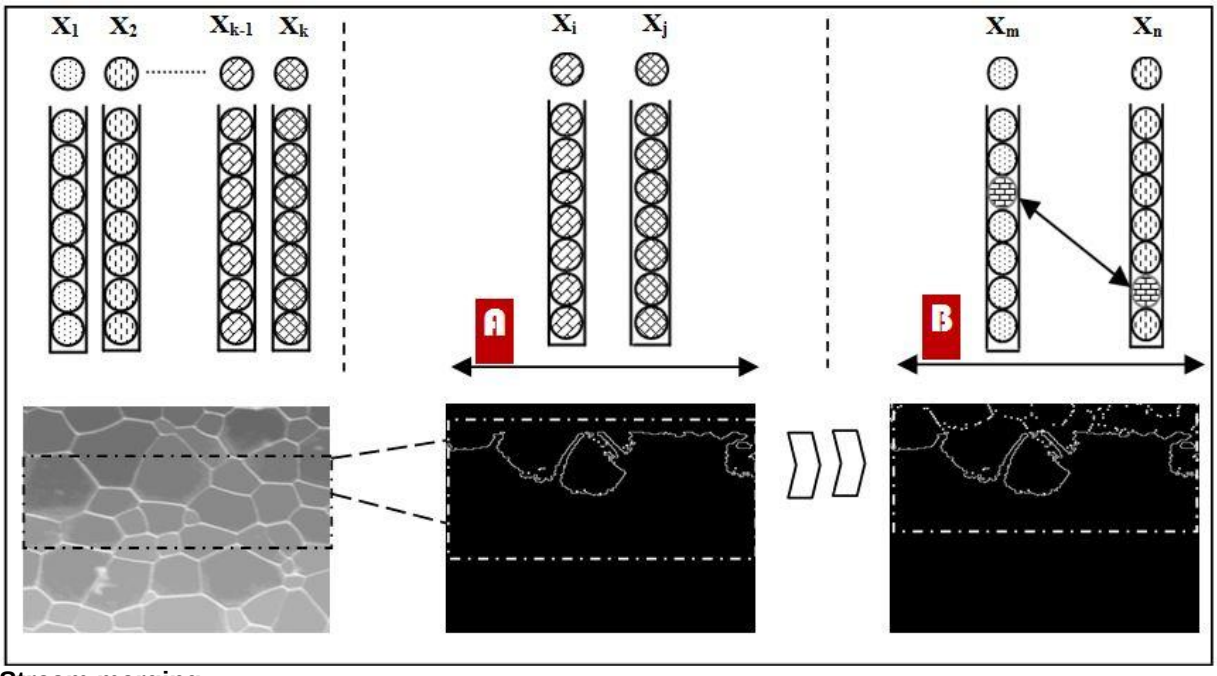
Stream notion illustration



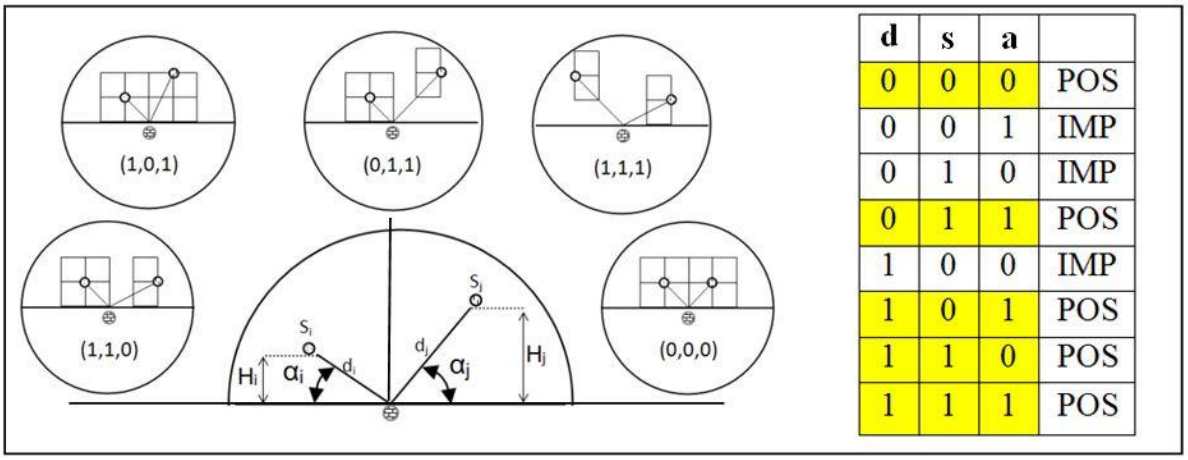
Watershed computing principal



Flow computation

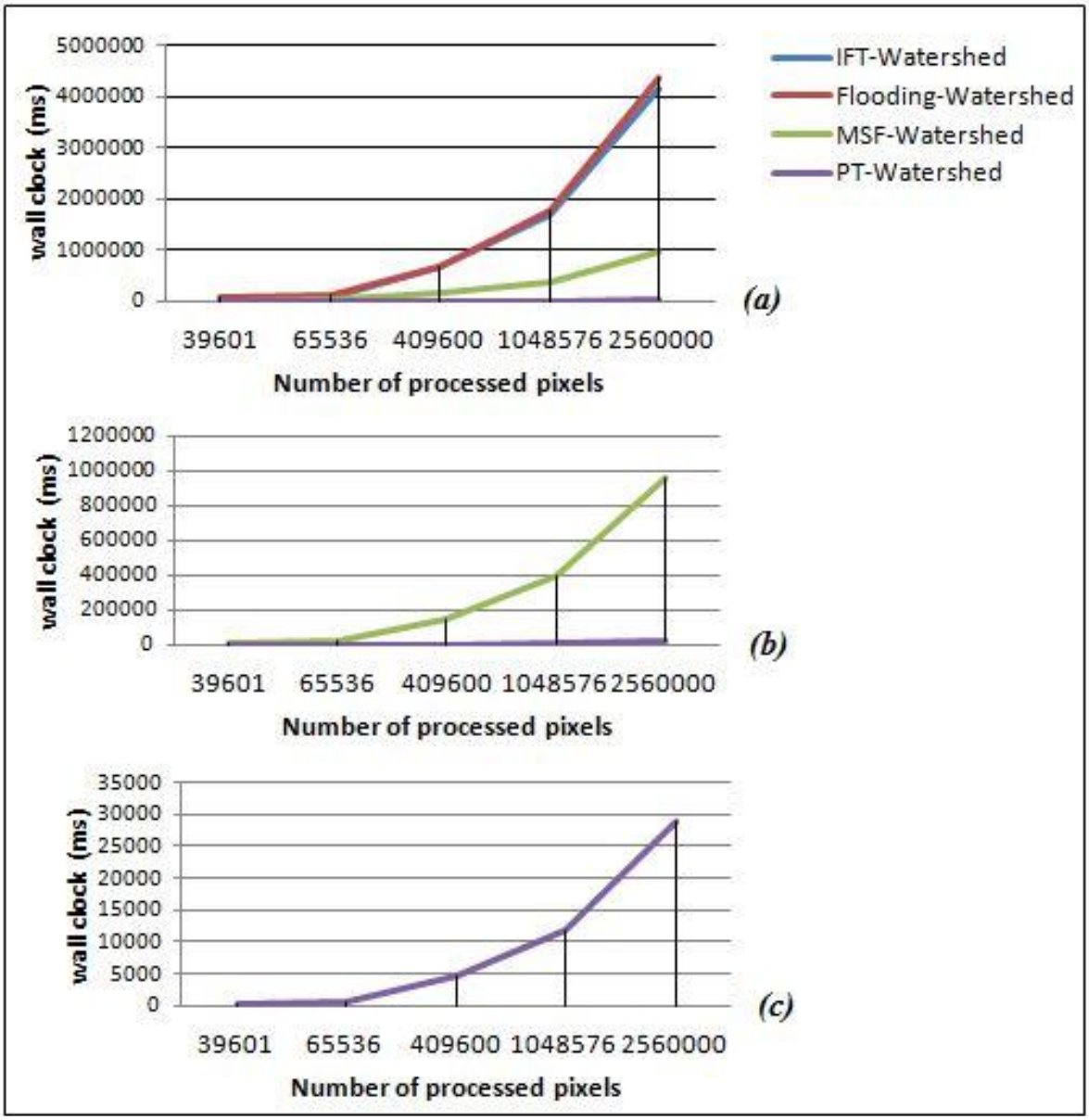


Stream merging

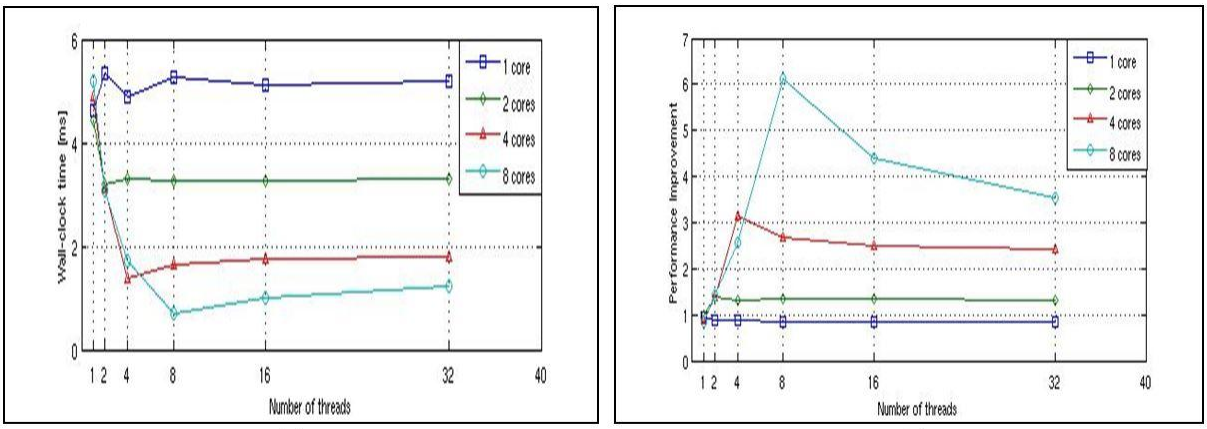


d	s	a	
0	0	0	POS
0	0	1	IMP
0	1	0	IMP
0	1	1	POS
1	0	0	IMP
1	0	1	POS
1	1	0	POS
1	1	1	POS

Merging techniques approach

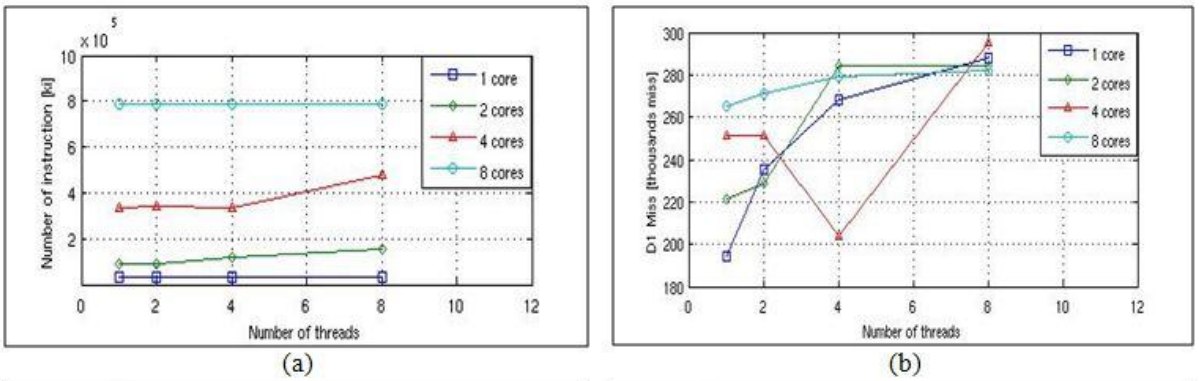


Global watershed transform profiling

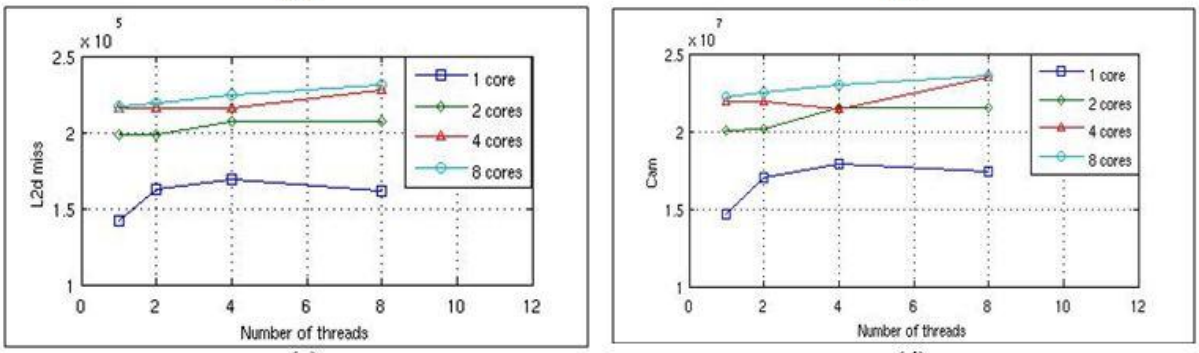


(a) (b)

Overlap time

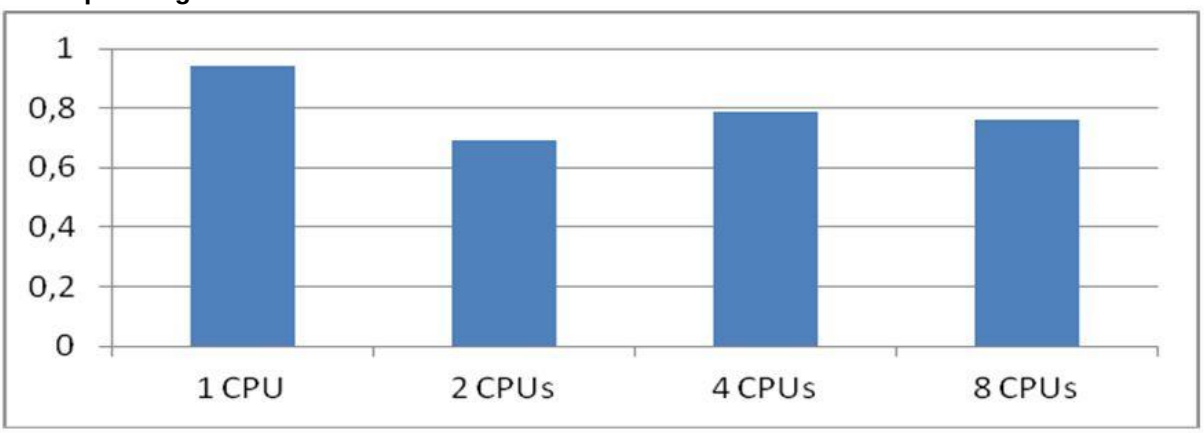


(a) (b)

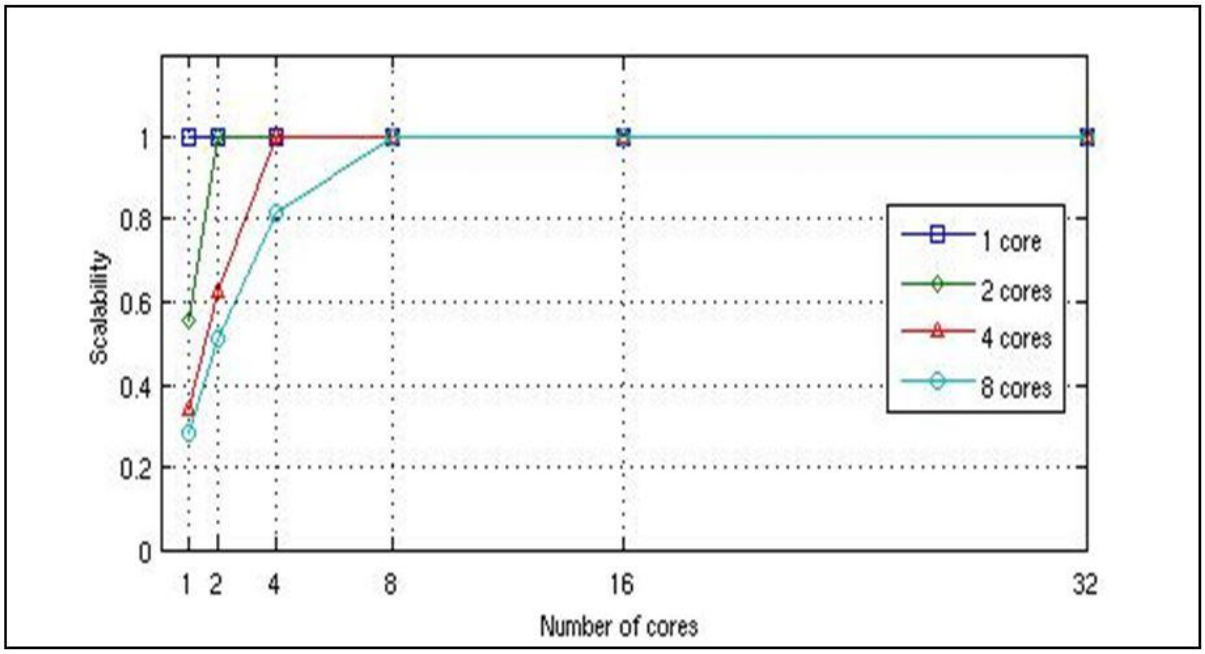


(c) (d)

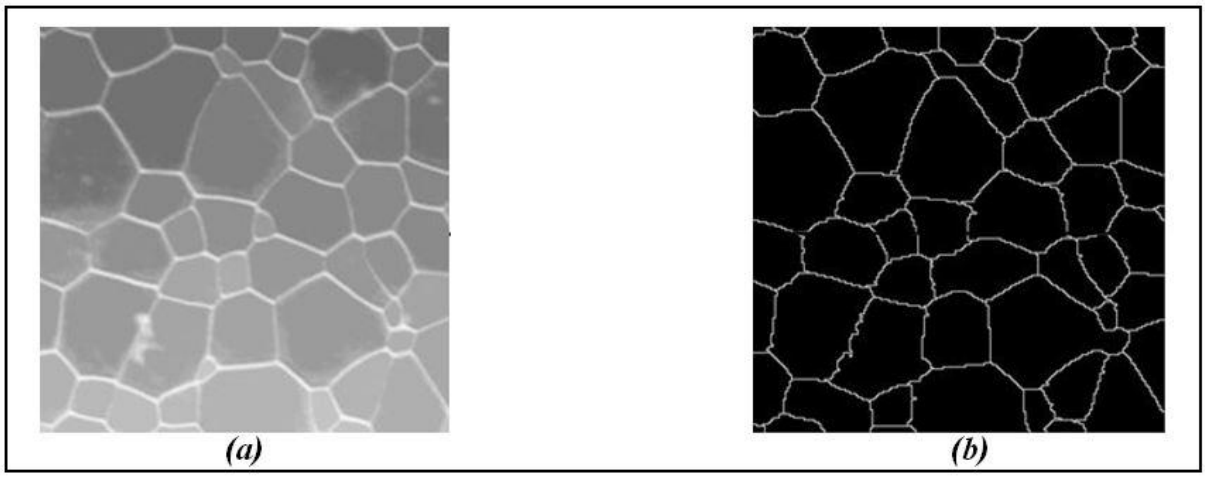
Cache profiling



Efficiency improvement



Scalability improvement



PT-watershed illustration

Watershed based on						
	Path-cost minimization					
Flooding	*TD*	*IFT*	*MSF*	*Local condition*	*Topology*	*PT*
Vincent & Soille [39]	Meyer [11]	Lotufo [40]	Cousty [35]	Lin [41]	Couprie [7]	*Proposed algorithm*
Defined in	Discrete & continuous space	Discrete & continuous space	Discrete space	Discrete space	Discrete & continuous space	Discrete & continuous space	Discrete space
Classified as	Line WT	Line WT	Region WT	Region WT	Region WT	Line WT	Region WT
Gives unique solution	Yes	Yes	No	No	No	No	No
Preserve topology	No	No	Yes	Yes	No	Yes	Yes
Requires a sorting step	Yes	No	No	No	No	No	No
Use hierarchical queue	Yes	Yes	Yes	No	Yes	Yes	No
Minima computing	Yes	Yes	No	No	-	No	No
linearity	Linear	-	linear	Linear	-	Linear*	Linear
Computing	Sequential	Sequential	Sequential	Sequential	Sequential	Sequential	Parallel

Comparison between main watersheds transforms

TESTED IMAGES					
Original size	199*199	256*256	640*640	1024*1024	1600*1600
Original colors	256	256	256	256	256
Number of unique colors	146	149	152	152	152
Disk size	38,7 KB	64.04	400 KB	1,00 MB	2,44 MB
Memory size	40 KB	65.04	401 KB	1,00 MB	2,44 MB
Number of processed pixels	39601	65536	409600	1048576	2560000
Number of intersection	7928	29249	193950	466478	1614014
Empty intersection	7901	28955	192612	463997	1096307
Full intersection	27	294	1338	2481	6139

Standard tested images

		Intel P4-660	Intel Dual C. E8400	Intel C2 Quad E5335	Intel Xeon E5405
Number of processor		1	2	4	2 x 4
SMT		Yes	Yes	Yes	Yes
Frequency		3,60 GHz	3,00 GHz	2,00 GHz	2,00 GHz
L1 Instruction Cache	**Size**	16Kb	32Ko	4 x 32Ko	8 x 32Ko
	Asso.	8-way	8-way	8-way	8-way
	Block size	32byte	64byte	64byte	32byte
L1 Data Cache	**Size**	16Kb	32Ko	4 x 32Ko	32Ko
	Asso.	8-way	8-way	8-way	8-way
	Block size	64byte	64byte	64byte	64byte
L2 Cache	**Size**	2Mb	6 MB	2 x 4Mb	2 x 6Mb
	Asso.	8-way	16-way	8-way	8-way
	Block size	64byte	64byte	64byte	64byte
RAM size		2Gb	2Gb	2Gb	8Gb

Used processors features

	1 CPU	2 CPUs	4 CPUs	8 CPUs
1 Thread	**4638**	4448	4898	5190
2 Threads	5321	**3182**	3114	3092
4 Threads	4898	3303	1384	1709
8 Threads	5253	3253	1639	**713**
16 Threads	5129	3278	1744	990
32 Threads	5190	3303	1794	1235

Wall clock (ms)

	1 CPU	2 CPUs	4 CPUs	8 CPUs
1 Thread	0.94	0.98	0.89	0.84
2 Threads	0.87	1.37	1.4	1.41
4 Threads	0.89	1.32	3.15	2.55
8 Threads	0.83	1.34	2.66	6.11
16 Threads	0.85	1.33	2.5	4.4
32 Threads	0.84	1.32	2.43	3.53

Performance improvement

		Nbr. Instr.	D1 miss	L2d miss	WScm
1CPU	1 Thread	**34.598.772**	194.274	141.738	14,699,160
	2 Threads	34.204.145	235.764	162.775	17,007,390
	4 Threads	34.340.721	268.779	168.539	17,856,300
	8 Threads	34.441.168	288.664	161.337	17,406,970
2CPUs	1 Thread	90.783.715	221.199	198.110	20,041,890
	2 Threads	**90.875.188**	229.857	198.282	20,143,950
	4 Threads	116.984.996	284.697	207.448	21,517,290
	8 Threads	152.704.881	284.753	207.448	21,517,850
4CPUs	1 Thread	334.816.008	251.241	215.443	21,902,280
	2 Threads	339.998.650	251.982	215.582	21,922,200
	4 Threads	**334.020.732**	204.315	215.860	21,470,550
	8 Threads	474.895.119	295.774	228.187	23,494,570
8CPUs	1 Thread	784.648.688	265.884	216.760	22,167,240
	2 Threads	784.745.461	271.859	219.487	22,472,420
	4 Threads	789.432.158	279.951	224.625	23,015,760
	8 Threads	**790.804.849**	282.122	231.142	23,624,000

Cache memory consumption

	Intel P4-660	Intel Dual C. E8400	Intel C2 Quad E5335	Intel Xeon E5405
EFFICIENCY RATE	0.94	0.69	0.79	0.76

Efficiency values

	Intel P4-660	Intel Dual C. E8400	Intel C2 Quad E5335	Intel Xeon E5405
1 Thread	0.94	-	-	-
2 Threads	-	0.685	-	-
4 Threads	-	-	0.787	-
8 Threads	-	-	-	0.763
16 Threads	-	-	-	-
32 Threads	-	-	-	-

Maximum Average Unit Speed

		1 CPU		2 CPUs	
1 Thread	Av_{us}	0.94	0.787	-	
	Work (W')	34.598.772	28.967.270		
2 Threads	Av_{us}	-		0.763	0.787
	Work (W')			790.804.849	815.679.444
		4 CPUs		8 CPUs	
4 Threads	Av_{us}	0.787		-	
	Work (W')	334.020.732			
8 Threads	Average Unit Speed	-		0.763	0.787
	Av_{us}			790.804.849	815.679.444

Average speed Unit - using 4 and 8 CPUs

	1 CPU	2 CPUs	4 CPUs	8 CPUs
1 CPU	1	0.554	0.338	0.284
2 CPUs	-	1	0.626	0.512
4 CPUs	-	-	1	0.819
8 CPUs	-	-	-	1

Scalability profiling