Real time image processing : algorithm parallelization on multicore multithread architecture

Ramzi Mahmoudi

UNIVERSITÉ PARIS-EST

PARIS-EST UNIVERSITY
DOCTORAL SCHOOL MSTIC

A thesis submitted in partial fulfillment for the degree of **Doctor of Philosophy**
In Computer Science

Presented by **Ramzi MAHMOUDI**

Advised by Mohamed AKIL

Common parallelization strategy of topological operators on SMP machines

September 2011[1]

Committee in charge:

- **Committee chair:** Gilles Bertrand, Professor at ESIEE Engineering, Paris, FR
- **First reviewer:** Jean Luc GAUDIOT, Professor at the University of California Irvine, USA
- **Second reviewer:** Olivier DEFORGES, Professor at the University of INSA, Rennes, FR
- **First reader :** Antoine MANZANERA, Associated-Professor at ENSTA, France
- **Director :** Mohamed Akil, Professor at ESIEE Engineering Paris, France

[1] Last version : October 10, 2011

ABSTRACT

Topological features of an object are fundamental in image processing. In many applications, including medical imaging, it is important to maintain or control the topology of the image. However the design of such transformations that preserve topology and geometric characteristics of the input image is a complex task, especially in the case of parallel processing.

Parallel processing is applied to accelerate computation by sharing the workload among multiple processors. In terms of algorithm design, parallel computing strategies profits from the natural parallelism (called also partial order of algorithms) present in the algorithm which provides two main resources of parallelism: data and functional parallelism. Concerning architectural design, it is essential to link the spectacular evolution of parallel architectures and the parallel processing. In effect, if parallelization strategies become necessary, it is thanks to the considerable improvements in multiprocessing systems and the rise of multi-core processors. All these reasons make multiprocessing very practical. In the case of SMP machines, immediate sharing of data provides more flexibility in designing such strategies and exploiting data and functional parallelism, notably with the evolution of interconnection system between processors.

In this perspective, we propose a new parallelization strategy, called SD&M (Split Distribute and Merge) strategy that cover a large class of topological operators. SD&M has been developed in order to provide a parallel processing for many topological transformations.

Based on this strategy, we proposed a series of parallel topological algorithm (new or adapted version). In the following we present our main contributions:

(i) A new approach to compute watershed transform based on MSF transform, that is parallel, preserves the topology, does not need prior minima extraction and suited for SMP machines. Proposed algorithm makes use of Jean Cousty streaming approach and it does not require any sorting step, or the use of any hierarchical queue. This contribution came after an intensive study of all existing watershed transform in the discrete case.

(ii) A similar study on thinning transform was conducted. It concerns sixteen parallel thinning algorithms that preserve topology. In addition to performance criteria, we introduce two qualitative criteria, to compare and classify them. New classification criteria are based on the relationship between the medial axis and the obtained homotopic skeleton. After this classification, we tried to get better results through the proposal of a new adapted version of Couprie's filtered thinning algorithm by applying our strategy.

(iii) An enhanced computation method for topological smoothing through combining parallel computation of Euclidean Distance Transform using Meijster algorithm and parallel Thinning–Thickening processes using the adapted version of Couprie's algorithm already mentioned.

KEYWORDS: PARALLELIZATION STRATEGY, PARALLEL PROCESSING, TOPOLOGY, WATERSHED, SKELETON, SMOOTHING, SHARED MEMORY, THREADS COORDINATION

RÉSUMÉ

Les caractéristiques topologiques d'un objet sont fondamentales dans le traitement d'image. Dans plusieurs applications, notamment l'imagerie médicale, il est important de préserver ou de contrôler la topologie de l'image. Cependant la conception de telles transformations qui préservent à la foi la topologie et les caractéristiques géométriques de l'image est une tache complexe, en particulier dans le cas du traitement parallèle.

Le principal objectif du traitement parallèle est d'accélérer le calcul en partagent la charge de travail à réaliser entre plusieurs processeurs. Si on approche cet objectif sous l'angle de la conception algorithmique, les stratégies du calcul parallèle exploite l'ordre partiel des algorithmes, désigné également par le parallélisme naturel qui présent dans l'algorithme et qui fournit deux principales sources de parallélisme : le parallélisme de données et le parallélisme fonctionnelle.

De point de vue conception architectural, il est essentiel de lier l'évolution spectaculaire des architectures parallèles et le traitement parallèle. En effet, si les stratégies de parallèlisation sont devenues nécessaire, c'est grâce à des améliorations considérables dans les systèmes de multitraitement ainsi que la montée des architectures multi-core. Toutes ces raisons font du calcule parallèle une approche très efficace. Dans le cas des machines à mémoire partagé, il existe un autre avantage à savoir le partage immédiat des données qui offre plus de souplesse, notamment avec l'évolution du système d'interconnexion entre processeurs, dans la conception de ces stratégies et l'exploitation du parallélisme de données et le parallélisme fonctionnel.

Dans cette perspective, nous proposons une nouvelle stratégie de parallèlisation, baptisé SD&M (Split, Distribute and Merge) stratégie qui couvrent une large classe d'opérateurs topologiques. SD&M a été développée afin de fournir un traitement parallèle de tout opérateur basée sur la transformation topologique. Basé sur cette stratégie, nous avons proposé une série d'algorithmes topologiques parallèle (nouvelle version ou version adapté). Nos principales contributions sont :

(i) Une nouvelle approche pour calculer la ligne de partage des eaux basée sur 'MSF transform'. L'algorithme proposé est parallèle, préserve la topologie, n'a pas besoin d'extraction préalable de minima et adaptée pour les machines parallèle à mémoire partagée. Il utilise la même approche de calcule de flux proposé par Jean Cousty et il ne nécessite aucune étape de tri, ni l'utilisation d'une file d'attente hiérarchique. Cette contribution a été précédé par une étude intensive des algorithmes de calcule de la ligne de partage des eaux dans le cas discret.

(ii) Une étude similaire sur les algorithmes d'amincissement a été menée. Elle concerne seize algorithmes d'amincissement qui préservent la topologie. En sus des critères de performance, nous somme basé sur deux critères qualitative pour les comparer et les classés. Après cette classification, nous avons essayé d'obtenir de meilleurs résultats grâce avec une version adaptée de l'algorithme d'amincissement proposé par Michel Couprie.

(iii) Une méthode de calcul amélioré pour le lissage topologique grâce à la combinaison du calcul parallèle de la distance euclidienne (en utilisant l'algorithme Meijster) et l'amincissement/ épaississement parallèle (en utilisant la version adaptée de l'algorithme de Couprie déjà mentionné).

MOTS CLES: STRATÉGIE DE PARALLELISATION, TRAITEMENT PARALLELE, LA TOPOLOGIE, DES BASSINS VERSANTS, SKELETON, LISSAGE, LA MEMOIRE PARTAGEE, FILS DE COORDINATION

ACKNOWLEDGEMENTS / REMERCIEMENTS

En premier lieu, je tiens à remercier particulièrement et à témoigner toute ma reconnaissance à professeur **Mohamed AKIL**, qui m'a accompagner toutes ces années. Il a su dirigé mes travaux de recherche tout en me laissant suffisamment d'autonomie. Je le remercie pour son beau sens d'écoute et sa patience face à mon atonie lors des moments difficiles de rédactions et de révision des articles. Merci professeur AKIL de m'avoir enseigné la science et m'avoir appris à aimer la recherche.

Je remercie également professeur **Jean-Luc GAUDIOT** et professeur **Olivier DEFORGES** qui m'ont fait l'honneur d'être rapporteur de mon manuscrit de thèse. J'adresse mes plus sincères remerciements à professeur **Antoine MANZANERAT** qui a accepté de consacrer du temps à la relecture de ce manuscrit. Mes profonds remerciements s'adressent également à professeur **Gilles BERTRAND** qui m'a fait l'honneur d'accepter la présidence du jury.

Je remercie très vivement touts les membres du laboratoire A3SI, chercheurs, techniciens, et étudiants en thèse pour leur accueil chaleureux. Un grand merci à Thierry, Laurent, Eric, Christophe, Eva, Christine, Jean-Claude, Michel, Denis, Rostom, Jean, Dalila, John, Imran, Oussama… Sachez que je garderai un très bon souvenir de nos discussions au tour du café, du repas, ou du barbecue …

Mes remerciements s'adressent plus spécialement à ceux, invisibles, qui m'ont soutenu et cru en moi : Merci à mes parents, mon frère et mes sœurs pour leur amour inconditionnel, leur soutien et tout ce qu'ils m'ont apporté mais dont ils n'ont qu'une toute petite idée. Merci à celle qui m'a enseigné que l'amour ne s'apprend pas dans les livres et qui m'a offert le plus beau cadeau du monde, mes deux filles Nersrine et Lina : Un merci très spécial à ma femme Marina qui m'a accompagné durant la préparation de cette thèse. Je lui suis très reconnaissant pour sa patience et ses encouragements dans mes moments de doute.

PUBLICATIONS

Some of the work explained in this thesis was already presented in the following publications:

International journals:

- Mahmoudi, R. and Akil, M., "*Enhanced Computation Method of Topological Smoothing on SMP Machines*" EURASIP JOURNAL – Real Time Image processing on Multi-Cores FPGA-based Platforms. (22 p.) [Published]

- Mahmoudi, R. and Akil, M., "*The watershed transform*", IJIP JOURNAL - International Journal of Image Processing, Volume 5, Issue 4, 2011. (21 p.) [Published]

International conferences:

- Mahmoudi, R., and Akil, M., "*Image segmentation based upon topological operators: real-time implementation case study*", IS&T/SPIE Electronic Imaging conf., Paper 7244-1, Volume 7244, 2008, San Jose, California, USA (12 pages) [Published]

- Mahmoudi, R., Akil, M., and Matas, P., "*Parallel Image Thinning through Topological Operators on Shared Memory Parallel Machines*", 43rd Asilomar Conf. on Signals, Systems, and Computers., Paper 1327,2009, Pasific Grove, California, USA (13 pages) [Published]

- Mahmoudi, R. and Akil, M., "*Real Time Topological Image Smoothing On Shared Memory Machines*", IS&T/SPIE Electronic Imaging conf., Paper 7871-9, Volume 7247, 2011, San Francisco, California, USA (12 pages) [Published]

- Mahmoudi, R., and Akil, M., "*Lissage homotopique parallèle sur des architectures multicoeurs à mémoire partagée*", AMINA conf. and Workshop, Paper 60, 2010, Monastir, TN (9 pages) [INVITED]

- Mahmoudi, R. and Akil, M., "*Thinning Algorithms Classification*", CIAE Conf and Workshop, Paper 8, 2011, Casablanca, Ma (4 pages) [INVITED]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

LIST OF ALGORITHMS

INTRODUCTION

1.1 Context and motivations

Imaging applications including medical imaging (MRI[2] scan, PET[3] scan, SPECT[4] scan) 2D/3D (even 4D in some cases) implement different steps: acquisition (raw format), pre-processing, processing, analysis, interpretation and display of results. The growing needs for such applications, variety and complexity of their algorithms, new requirements in terms of performance and quality, require the development of new methods, supports and packaged software implementation of these application on new multi-processor based-architecture[5] that offer more computing power. Such software design flow must include all processing steps from high level specification (algorithm specification) to the low level specification (optimized implementation based on parallel code distributed between different processors of the platform) with respect of real-time constraints such that throughput and latency.

Discrete topology offers a range of essential tools in image processing thus applications in medical imaging calls several algorithms based on topological transformation (these algorithms have the specificity to process over input image while preserving its topology which allows keeping some important information intact). But the increasing size of processed data, due to the improved of capture devices resolution, and constraints in terms of processing time, make the development of such standard application very complex. Indeed, computing power required for these applications currently uses parallel architectures as new computer machines (for reasons of cost and availability). Only parallel processing provides a cost effective solution for this required power by increasing the number of processors and adding an efficient communication system between them which makes coding more complicated: exploring, in optimal way, intra/inter processors parallelism and work distribution among different threads. The availability, on SMP[6] machines, of multi-core processor / RISC-based cores (where the work-load can be shared between different processors) makes it necessary to study and develop appropriate and effective parallelization strategies of such image processing algorithms for this type of machine. However, there is no parallelization strategy common to a set of algorithms based on topological operators to efficiently implement these algorithms on parallel machines. This strategy unifies the optimized implementation of parallel algorithms on these specific architectures via parallelization, work-load distribution and effective management of memory hierarchy. We therefore propose to study and formalize such parallelization strategy and define suitability metrics to assess performance on this type of architecture.

[2] Magnetic Resonance Imaging
[3] Positron Emission Tomography
[4] Single Photon Emission Computed Tomography
[5] Architecture interconnecting many processors: parallel processors (multi-core processor), parallel specific processors (Graphic Processor Unit).
[6] Shared Memory Parallel machines

1.2 Contributions

Research presented in this manuscript has been done in the Gaspard-Monge computer science research laboratory (LIGM) of Paris-Est University, ESIEE A3SI team, CNRS-UMLV-ESIEE (UMR 8046).

We frame our work in the field of algorithms based on topological transformation in order to study their parallelization on shared memory parallel machines (SMP machines).

Topological features of an object are fundamental in image processing. In many applications, including medical imaging, it is important to maintain or control the topology of the image. However the design of such transformations that preserve topology and geometric characteristics of the input image is a complex task, especially in the case of parallel processing. Here, the main goal of parallel processing is to accelerate computation by sharing the workload among multiple processors. In terms of algorithm design, parallel computing strategies profits from the partial order of algorithms, called also the natural parallelism present in the algorithm which provides two main resources of parallelism: data and functional parallelism. Now, from a viewpoint architectural design, it is essential to link the spectacular evolution of parallel architectures and the parallel processing. In effect, if parallelization strategies become necessary, it is thanks to the considerable improvements in multiprocessing systems and the rise of multi-core processors. And during the last decade, clock speed of processors in multi-core architectures has increased by almost two and associated cache size has increased tenfold with the addition of a third cache level L3 which ensures optimal L2 access speed while increasing the total cache. All these reasons make multiprocessing very practical. In the case of SMP machines, it adds another advantage that is the immediate sharing of data which provides more flexibility, notably with the evolution of interconnection system between processors, in designing such strategies and exploiting data and functional parallelism.

In this perspective, we propose a new parallelization strategy, called SD&M (Split Distribute and Merge) strategy that cover a large class of topological operators. SD&M has been developed in order to provide a parallel processing of any operator based on topological transformation. In practice the most effective parallel algorithm design might make use of multiple algorithm structures thus proposed strategy is a combination of the divide and conquer patterns and event-based coordination patterns hence the name that we have assigned. Not to be confused with the mixed-parallelism approach (combining data-parallelism and task-parallelism), it is important to mention that proposed strategy (1) represents the last stitch in the decomposition chain of algorithm design patterns and it provides a fine-grained description of topological operators parallelization while mixed-parallelism strategy provides a coarse-grained description without specifying target algorithm. (2) It covers only the case of recursive algorithms, while mixed-parallelization strategy is effective only in the linear case. (3) It is especially designed for shared memory architecture with uniform access.

Although the cost of communication (Memory-processor and inter-processors) is high enough, shared memory architectures meet our needs for different reasons: (a) These architectures have the advantage of allowing immediate sharing of data with is very helpful in the conception of any parallelization strategy (b) They are non-dedicated architecture using standard component (processor, memory, buses ...) so economically reliable (c) They also offer some flexibility of use in many application areas, particular image processing.

Based on this strategy, we proposed a series of parallel topological algorithm (new or adapted version). In the following we present our main contributions:

- A new approach to compute watershed transform based on MSF[7] transform, that is parallel, preserves the topology, does not need prior minima extraction and suited for SMP machines. Proposed algorithm makes use of Jean Cousty streaming approach [1] and it does not require any sorting step, or the use of any hierarchical queue. This contribution came after an intensive study of all existing watershed transform in the discrete case: WT based on flooding, WT based on path-cost minimization, watershed based on topology preservation, WT based on local condition and WT based on minimum spanning forest. This study can be seen as an update of Roerdink research [2]. Actually, this study presents detailed description of each watershed approach, associated processing procedure followed by mathematical foundations and the algorithm of reference. Recent publications based on some approaches are also presented and discussed. Our study concludes with a classification of different algorithms studied according to solution uniqueness, topology preservation, prerequisites minima computing and linearity.

- A similar study on thinning transform was conducted. It concerns five parallel thinning algorithms that preserve topology: Bernard and Manzanera [3], Jang and Chin [4], Eckhardt and Maderlechner [5], Guo and Hall [6], and Hall [7]. Based on the relationship between the medial axis and the obtained homotopic skeleton, we introduce two classification criteria to compare and classify them. After this classification, we tried to get better results through the proposal of a new adapted version of Couprie's filtered thinning algorithm [8] by applying our strategy.

- An enhanced computation method for topological smoothing through combining parallel computation of Euclidean Distance Transform using Meijster algorithm [9] and parallel Thinning–Thickening processes using the adapted version of Couprie's algorithm already mentioned.

[7] Minimum Spanning Forest

1.3 Report organization

This thesis initially aimed to study how to parallelize topological operators on SMP machines. All along the chapters of this work, parallel version (adapted and new) of fundamental algorithms of this class will be introduced. Beyond new parallel algorithms, we will also present obtained results in terms of computation time, efficiency, cache consumption.

The **second chapter** will be about parallelization strategy. The global goal of the project surrounding this work was to design an efficient strategy to parallelize algorithm based on topological transform. To do so, we first start by identifying real needs for such strategy, then we define fundamental basis for any parallelization moving from finding concurrency to performance metrics of parallel programs. Then, we briefly introduce our strategy called SD&M[8] with the aim of classifying it among all existing strategy. Finally, we will conclude by presenting all details of SD&M conception: splitting, merging and merging phases.

The **third chapter** is about watershed transform. Our work starts with a comparative study between five different approaches: we present a review of several definitions of the watershed transform and the associated sequential algorithms, emphasizing the distinction between definition and algorithm specification. The study concludes with a classification of different algorithms according to criteria of recursion, complexity, basins computing and topology preservation. Since we identify the most suited approach to compute parallel watershed, we propose a new algorithm that is parallel, preserves the topology of the input image, does not need prior minima extraction and suited for SMP machines. Contrarily to previously published algorithms, proposed algorithm do not require any sorting step, or the use of any hierarchical queue.

The **fourth chapter** focuses on thinning algorithms in the framework of critical kernel. We start by resuming Couprie's study on verification methods for the topological soundness of thinning algorithms. Based on this first evaluation of 2D topological thinning algorithm, we propose to go further in this study through new quantitative and qualitative criteria. Since we have establish a new classification of thinning operators and being convinced that best performance can be reached, we propose an adapted version of Couprie's thinning algorithm that we call parallel λ – Skeleton algorithm.

In the **fifth chapter**, we move to another aspect of topological processing: smoothing filter. We present a new parallel computation method for topological smoothing through combining parallel computation of Euclidean Distance Transform using Meijster algorithm and parallel Thinning–Thickening processes using an adapted version of Couprie's algorithm.

In the **sixth chapter**, we present a review of the research in the form of a critical summary of presented work restating contributions of the thesis. Future work is also presented, summarizing the next steps to follow into the research of parallelization strategy.

[8] Split Distribute & Merge

PARALLELIZATION STRATEGY

Multiprocessor chips make computing more efficient by exploiting parallelism which is one of the outstanding challenges of modern computer sciences. Parallel processing can be defined as the division of a fixed amount of work among different processors that run simultaneously with a common objective. The primary advantage of such processing comes exactly out of its ability to handle a large volume of tasks (or data) with reasonable latency and cadency. Obtained result should be faster with better efficiency, scalability and portability in comparison with single processor implementation. Improving these metrics doesn't depend only on the parallel programming design. The scope of algorithm design space should be expand by including finding concurrency design space, architecture design space, parallel implementation mechanisms and performance metrics of parallel programs. These steps, listed in chronological order, sets out a strategy for parallelization.

However, it is generally recognized that designing such strategy is not an easy task [10,11,12]. In fact, such design is significantly more complex than sequential programs design on single processor computers. As cited in [13], some example of parallel software design [14,15,16] illustrate encountered difficulty when scientific code has been hand-crafted for specific machines and problems, at immense expense.

The study of parallelism, or how parallel process could be expressed in programming terms, start since the sixties. Dijkstra [17] was among the first to develop an initial proposal for the treatment of parallelism in a programming language by adding new concepts to sequential programming, in order to extend it into a concurrent programming. He introduced new elements such as mutual exclusion, event synchronization and critical section. The most important concept that was introduced by Dijkstra is semaphores. He didn't bring only solutions, He also raise new issues related to parallel processing such as deadlock. Ten years later, things become more formal and definitions more explicit with Hoare's work [18]. Actually, he defines new language for the formal specification of parallel algorithms, known as Communicating Sequential Processes. His work starts by analyzing general basic structures used in programming such as assignment, sequence, repetition, and selection. Then, he introduced new structures for expressing parallelism, communication, and control of non-determinism between processes within a multiprocessor architecture. More details about both approaches can be found in [13]. These different approaches [17, 18, 19] intersect at a common point: they emphasize the close link between strategy and the nature of the application to parallelize. We therefore propose to study and formalize parallelization strategy of image processing operators (topological operators) and define suitability metrics to assess performance on SMP architecture.

This chapter is organized as follow: we begin, in section 2.1, by highlighting the real need for a common parallelization strategy of topological operators. After introducing the basic foundation for any successful parallelization (section 2.2), we will focus on Split Distribute and Merge (SD&M) strategy that we propose by an initial classification over all existing strategies (section 2.3), followed by detailed description of SD&M conception in section 2.4.

2.1 Lack of common parallelization strategy for topological operators

In 1996, Bertrand [20] introduced connectivity numbers for grayscale image. These numbers describe locally (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 gray level of a point without modifying image topology. These elementary operations of point characterization present the fundamental link of large class of topological operators including, mainly, skeletonization and crest restoring algorithms [8]. This class can also be extended, under condition, to homotopic kernel and leveling kernel transformation [21], topological 2D and 3D object smoothing algorithm [22] and topological watershed algorithm [23]. All mentioned algorithms get also many algorithmic structure similarities. In fact associated characterizations procedures evolve until stability with 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 pixel's array as input and output data structure. Expect in special cases where graphs are used. It is important to mention that, to date, this class has not been efficiently parallelized like other classes as connected filter of morphological operator which recently has been parallelized in Wilkinson's work [24]. Parallelization strategy proposed by Sienstra [25] for local operators and point to point operators can also be cited as example. For global operators, an adapted parallelization strategy is given in Meijster work [9]. Hence the need of a common parallelization strategy for topological operators that offers adapted algorithm structure design space. Chosen algorithm structure patterns to be used in the design must be suitable for SMP machines.

2.2 Fundamental basis for parallelization

Before defining parallelization's stages of any sequential problem, it is essential to link the spectacular evolution of parallel architectures and the parallel processing. In reality, if the parallelization strategies are so valuable, it is thanks to substantial improvements in multiprocessing systems and 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 one with many slow processors (clock speed around 1.5 GHz) with the same throughput. But during last years the clock speed of processors in multi-core architectures has increased ,see (***tab. 1***), by almost two and associated cache size has increased tenfold with the addition of a third cache level L3 which ensures optimal L2 access speed while increasing the total cache. These twin barriers have flipped the equation, making multiprocessing very practical and advised even for small applications.

	Pentium 4 Processor Extreme Edition	**Intel Core 2 Duo Processor E4300**	**Intel Xeon Processor X7560**
Code name	Prescott	Conroe	Nehalem-EX
# of Cores	1	2	8
# of Threads	2	2	16
Clock Speed	3.73 GHz	1.8 GHz	2.266 GHz
Cache	2 MB L2 Cache	2 MB L2 Cache	24 MB L3 Cache
Bus Type	FSB	FSB	QPI
System Bus	1066 MHz	800 MHz	6.4 GT/s

Table 1 : Processors performance evolution

Generally five steps, see (***fig. 1***), are necessary to move from sequential algorithm running on single core architecture to parallel algorithm that runs with better performance on a multi-core architecture. G. Mattson and al., see [26], present the first four steps for parallel programming: Finding concurrency, algorithm structure, support structure and implementation mechanisms. Based on Mattson logic, we define all five steps. We should like to stress the importance that we attach to the second and third phase which represent basis of our strategy as we will show later.

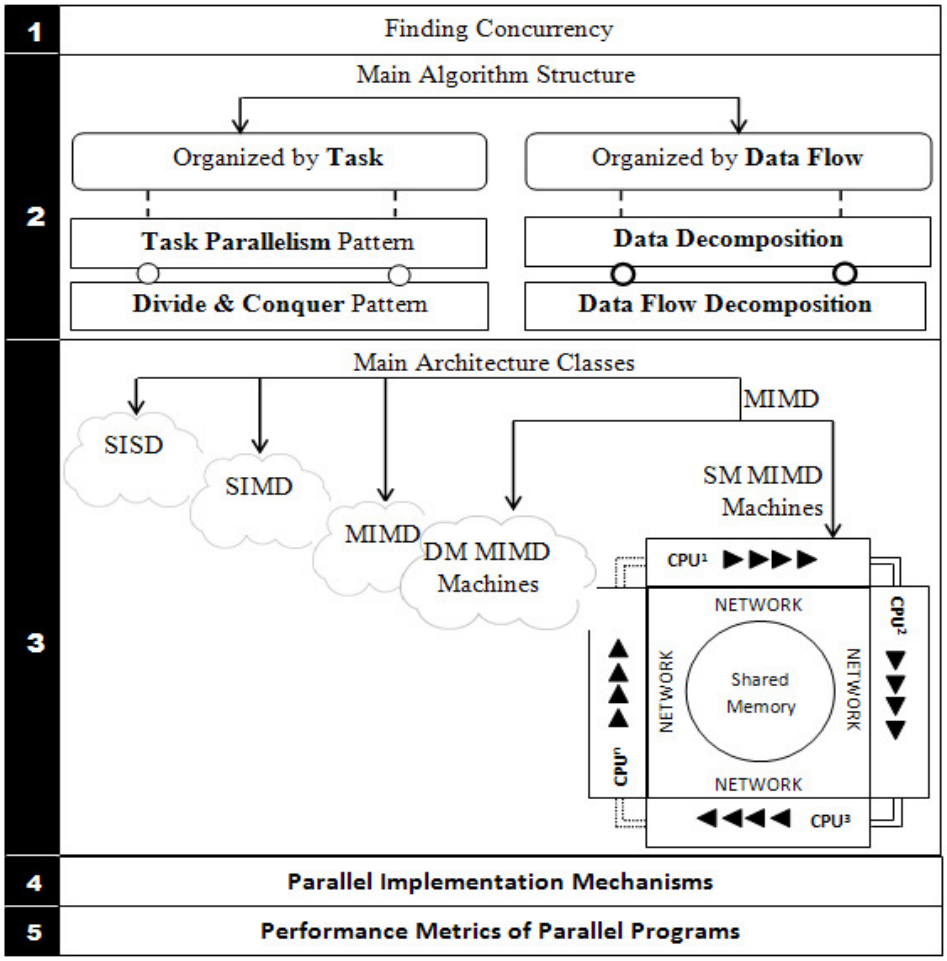


Figure 1 : Overview of parallel computing steps

Definition (1): *Finding Concurrency Design Space*

It's 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 and al. propose three possible organizations: organization by tasks, organization by data decomposition, and organization by flow of data. To remain in the conceptual framework of this section, we provide only the two original organizations: by tasks and by data [27].

Definition (2): *Algorithm Design Space*

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: nature of the operation to achieve required operands, action-zone or returned result then groups of task 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 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 principal cannot be independent from the type of the algorithm [28]. 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 goes 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.

Definition (3): *Architecture Design Space*

It describes the set of platform that support the extension of parallel programming. Information about how instructions are executed and how memory is managed are presented in this design.

Before moving to coding, it is important to find the most appropriate architecture to support the parallel algorithm using parallel architecture design space (***Def.3***). This design presents standard classification of parallel computer systems [29]. According to Flynn classification [30], there are four types: SISD[9], SIMD[10], MISD[11] and MIMD. The most significant structure encountered in the parallel application [31] is MIMD (Multiple Instruction, Multiple Data). In a MIMD machine the processors can execute different operations using their own data. Parallel processing via the application of MIMD machines offers the promise of high performance, and experience with parallel processing is accumulating rapidly. In [32], Buzbee show, through different examples, that rapid progress is being made in the application of MIMD machines and that parallel processing can yield 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.

[9] Single Instruction Single Data
[10] Single Instruction Multiple Data
[11] Multiple Instruction Single Data

Definition (4): *Parallel Implementation mechanisms*

They are set of tools used to write parallel programs. They are able to manage threads. 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 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***). These mechanisms allow better exploitation of target architecture through the use of threads. The most used tools within this framework are: MPI [33], OpenMP [34] and TBB [35]. After coding and running programs, it's important to evaluate efficiency, scalability and portability of the code by using performance metrics for parallel programs (***Def.5***). These concepts will list in detail in the last part of this chapter.

Definition (5): *Performance metrics of parallel programs*

They are a set of measurements that quantify the parallel code such as efficiency, scalability and portability.

2.3 Classification of SD&M strategy

As mentioned in last section, chosen algorithm structure patterns to be used in the design must be suitable for SMP machines. In fact, shared memory parallel machines allow access from any processor to any location through shared memory using an interconnection network (processor-processor and memory-processor). Flexibility of these architectures does not lie in such interconnection network (which is usually predefined by manufacturer), but on shared memory. Actually, programmer perceives a shared, central, and continuous memory. Each memory location or address in unique and identical for any processor of the system. Thus, he take profits of the immediate sharing of data with is very helpful in the conception of any parallelization strategy: The global address space provides a user-friendly programming perspective to memory and data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs. Communication between CPUs can be assured trough shared variables (reading/writing). Network selects appropriate memory block when reading (writing) process from a specific memory address is lunched. Data integrity is guaranteed by synchronization, and coordination mechanizes such as semaphores (Dijkstra), and monitors (Hoare) already introduced in introduction. SMP machines are also non-dedicated architecture using standard component (processor, memory...) so economically reliable. They also offer some flexibility of use in many application areas, particular image processing.

In practice the most effective parallel algorithm design might make use of multiple algorithm structures thus proposed strategy is a combination of the divide and conquer patterns and event-based coordination patterns, see ***(fig. 2)***, hence the name that we have assigned: SD&M (Split Distribute and Merge) strategy. Not to be confused with mixed-parallelism approach (combining data-parallelism and task-parallelism [36]), it is important to mention that our strategy represents the last stitch in the decomposition chain of algorithm design patterns and it provides a fine-grained description of topological operators parallelization while mixed-parallelism strategy

provides a coarse-grained description without specifying target algorithm. It covers only the case of recursive algorithms, while mixed-parallelization strategy is effective only in the linear case. It is especially designed for shared memory architecture with uniform access. (Identical processors, equal access times to memory …).

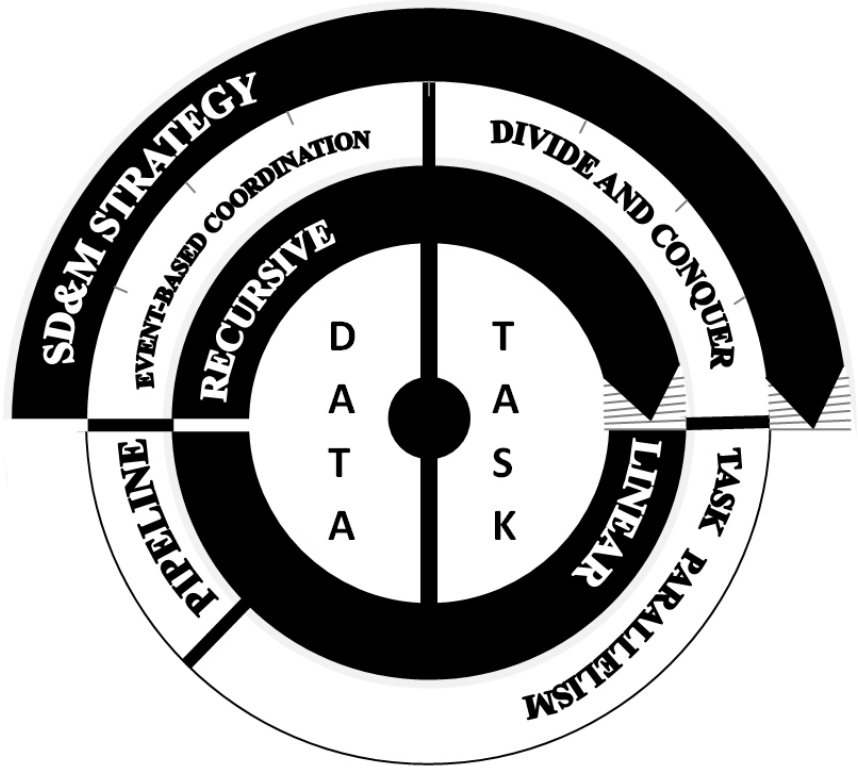


Figure 2 : Circle of decision for the algorithm structure design space

2.4 SD&M 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 conflict that exists between these three forces. These notions will be defined later in this section. Actually, any strategy is facing two major barriers. First, the conflict between efficiency and portability: making a program efficient almost requires that the code take into account the characteristics of the specific system on which it is intended to run, which limits portability. A design that makes use of special features of a particular programming environment (as multi-thread environment) may lead to an efficient program for that particular environment, but 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 *N* processors' potential. This distribution limits the portability of the program since the number of processor is increased.

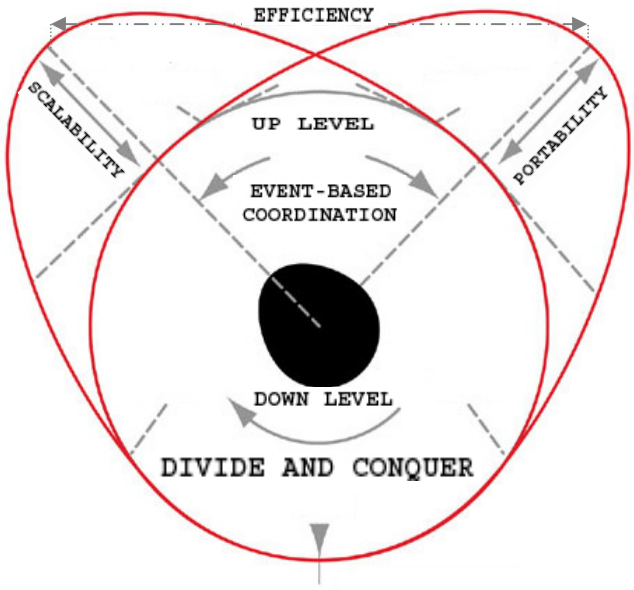


Figure 3 : SD&M Strategy Design

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 feature, as we shown in section 2.1. Shared memory parallel architectures turned out to be best suited for our needs (section 2.3). Therefore, Split Distribute and Merge strategy, that we propose, combines two patterns: Divide and Conquer pattern (***Def.6***) and Event-Based Coordination (***Def.7***). In the following we detail all three phases of SDM strategy.

Definition (6): *Divide and Conquer pattern*

It is based on multi-branched recursion. It solve 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.

Definition (7): *Event-Based coordination*

It is used when dealing with irregular, dynamic or unpredictable data flow.

2.4.1 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 take 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 a computation.

Definition (8): *Scalability*

It is a property which exhibits performance linearly proportional 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 nature choice to begin. So we assume that every program is made up of two parts, sequential and parallel, to establish the following definitions (with $N \geq 2$):

t_s	Processing time of the serial part of a program using one processor.
$t_p(1)$	Processing time of the parallel part of a program using one processor.
$t_p(N)$	Processing time of the parallel part of a program using N CPUs.
$t_T(1) = t_s + t_p(1)$	Total processing time of the serial and parallel part of the program using one processor.
$t_T(N) = t_s + t_p(N)$	Total processing time of the serial and parallel part of the program using N CPUs.
$\alpha(1) = \frac{t_s}{t_s + t_p(1)}$	Non-Scaled percentage of the serial part of the program using one processor.
$\alpha(N) = \frac{t_s}{t_s + t_p(N)}$	Scaled percentage of the serial part of the program using N CPUs
$\beta(1) = (1 - \alpha(1)) = \frac{t_p(1)}{t_s + t_p(1)}$	Scaled percentage of the parallel part of the program using one processor.
$\beta(N) = (1 - \alpha(N)) = \frac{t_p(N)}{t_s + t_p(N)}$	Scaled percentage of the parallel part of the program using N CPUs

Table 2 : Basic definitions of processing time

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

Definition (9): *Speedup-(1^{st} equation)*

$$Speedup = \frac{Work}{Time} = \frac{W}{T}$$

A second formal definition can be given by applying Amdahl's law [37] so the speedup can be defined by the ratio of total processing time of the serial and parallel part of the program using one processor by the total processing time of the same parts using ***N*** processors.

Definition (10): *Speedup-(Amdahl approach)*

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

This formula can be written differently using non-scaled percentage $\beta(1)$ previously defined in ***(tab.2)***:

Definition (11): *Speedup-(2nd equation)*

$$Speedup = \left[\frac{1}{\beta(1) + \frac{1-\beta(1)}{N}}\right]$$

An alternative formulation referred to as Gustafson's law [38] exists. This formulation calibrates the serial percentage according to the total parallel time using N processors.

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

Thus we can define the speedup as follows:

Definition (12): *Speedup-(3d equation)*

$$Speedup = \left[\frac{t_T(1)}{t_T(N)}\right] = [\alpha(N) + N(1 - \alpha(N))] = [N - (N-1)\alpha(N)]$$

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

$$\text{If } \eta_N = \eta_{N'} \Leftrightarrow \frac{Speedup(N)}{N} = \frac{Speedup(N')}{N'} \Leftrightarrow \frac{Speedup(N)}{N} = \frac{Speedup(N')}{N'}$$

$$\Leftrightarrow \frac{W_N}{N * t_T(N)} = \frac{W_{N'}}{N' * t_T(N')} \Leftrightarrow \frac{t_T(N)}{t_T(N')} = \frac{W_N * N'}{W_{N'} * N} = \psi(N, N')$$

Thus it follows that the only parameter that provides a linear performance in proportion to the number of processors (***Def. 8***) is the ratio $\left[\frac{W_N}{W_{N'}}\right]$. Hence the importance of splitting step. Unfortunately such impact can't be shown by applying simply Gustafson approach (***Def. 12***). Scalability will be express only in term of the number of processor.

If $\eta_N = \eta_{N'} \Leftrightarrow \frac{N-(N-1)\alpha(N)}{N} = \frac{N'-(N'-1)\alpha(N')}{N'}$

$$\Leftrightarrow N'\left[N-(N-1)\alpha(N)\right] = N\left[N'-(N'-1)\alpha(N')\right]$$

$$\Leftrightarrow N'*N-\left[N'(N-1)\alpha(N)\right] = N*N'-\left[N(N'-1)\alpha(N')\right]$$

$$\Leftrightarrow \left[N'(N-1)\alpha(N)\right] = \left[N(N'-1)\alpha(N')\right]$$

$$\Leftrightarrow \left[N'(N-1)\frac{t_s}{t_s+t_p(N)}\right] = \left[N(N'-1)\frac{t_s}{t_s+t_p(N')}\right]$$

$$\Leftrightarrow \left[\frac{N'*(N-1)*t_s}{t_s+t_p(N)}\right] = \left[\frac{N*(N'-1)*t_s}{t_s+t_p(N')}\right]$$

$$\Leftrightarrow \left[\frac{(N'*N-N')*t_s}{t_s+t_p(N)}\right] = \left[\frac{(N*N'-N)*t_s}{t_s+t_p(N')}\right]$$

$$\Leftrightarrow \left[\frac{N'*(N-1)}{t_s+t_p(N)}\right] = \left[\frac{N*(N'-1)}{t_s+t_p(N')}\right]$$

$$\Leftrightarrow \left[\frac{N'*(N-1)}{t_T(N)}\right] = \left[\frac{N*(N'-1)}{t_T(N')}\right]$$

$$\Leftrightarrow \left[\frac{t_T(N)}{t_T(N')}\right] = \left[\frac{N'*(N-1)}{N*(N'-1)}\right] == \psi(N,N')$$

2.4.2 Distribution phase

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

The main challenge when performing parallel operations on 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, consist on using a shared work queue in main memory and control access to it via synchronization primitive. 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 randomized stealing [39]. For tasks that share the same memory footprints, randomized locality oblivious work-stealing schedulers do nothing to ensure scheduling of these tasks on workers that share a cache. This significantly limits, not only scalability, but also efficiency (***Def. 13***) for some memory-bandwidth bounded applications on machines that have separate caches.

Definition (13): *Efficiency*

It 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 determination of productivity.

$$Efficiency = \eta_N = \left[\frac{Speedup}{N}\right]$$

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

$$= \left[\frac{W}{Nt_T}\right] \text{according to (}\textbf{\textit{Def. 9}}\text{)}$$

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

The main idea of our approach is to imagine an auto-supplying task system, see ***(fig.4)***. Keeping the local queues filled will be our major goal. Local threads should never have to 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 permanent redistribution of tasks to maintain a certain balance between all processors.

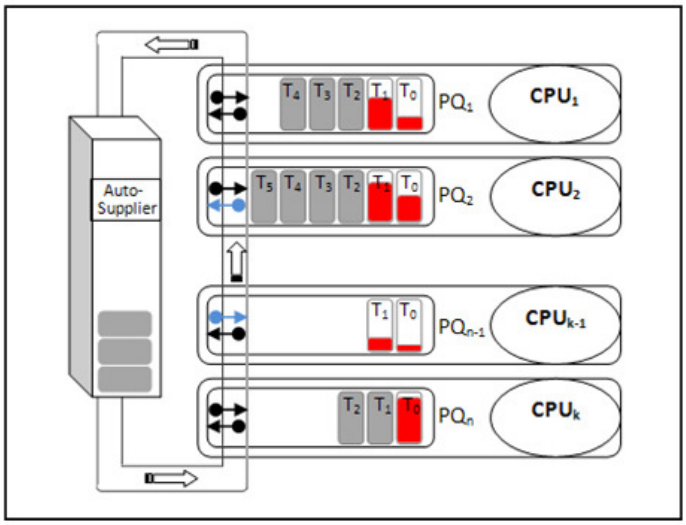
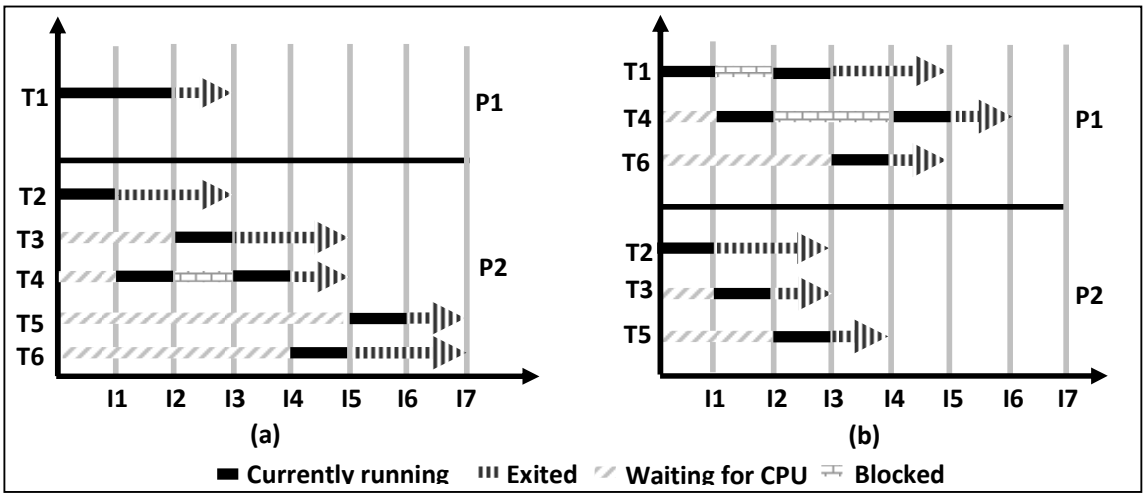


Figure 4 : Auto-Supplying task system

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 "executing" status. In spite of 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 principal memory that supply the processor shortage of work. System maintains a minimum level of work in its queue by importing extra work from others' processors queues. This load balancing feature will keeps all processors busy.

Before detailing our approach, we start by recalling briefly some basic notion of distribution techniques then we introduce our minimal distribution approach that is particularly suitable for topological recursive algorithms where simple point characterization is necessary. Our approach is general and applicable to shared memory parallel machines. Critical cases are also introduced and discussed.

Indeed there are two main types of scheduler. There are those designed for real-time systems (RTS). In this case, the most commonly approaches used to schedule real-time task system are: Clock-Driven, Processor-Sharing and Priority-Driven. Further description of different scheduling approaches can be found in [40,41,42]. According to [42] the Priority-Driven is far superior the other approaches. These schedulers must provide an operational RTS: completed work and delivered results on a timely basis. Other schedulers are designed for Non Real-time system. In this case, schedulers are not subject to the same constraints. Thus, "Symmetric Multiprocessing" scheduler distributes tasks to minimize total execution time without load balancing between processors, see ***(fig. 5. a)***. On multi-core architectures, this can lead to high occupancy rate of one processor while the others are free.



(a) Non-real time symmetric task distribution (b) Task distribution based upon uniformity principle

Figure 5: Illustration of task distribution on multi-core architecture

We propose a novel tasks scheduling approach to prevent improper load distribution while improving total execution time, see ***(fig. 5. b)***. In literature, there are several schedulers that provide a balanced distribution of tasks such as RSDL "Rotating Staircase Deadline" [43] which incorporates a foreground-background descending priority system (the staircase) with run-queue managing minor and major epochs (rotation and deadline). Other scheduler, as CFS "Completely Fair Scheduler" [44], shows consistence. It handles resource allocation for executing processes, and aims to maximize overall CPU utilization while maximizing interactive performance. These schedulers are based on tasks uniformity principle. Through the tasks homogeneity, better distribution can be achieved and total execution time reduced.

Unfortunately, these schedulers are not available in all operating system versions especially for small system. Based on the same principle of tasks uniformity, we propose an adapted scheduling algorithm, simpler to implement and more adapted to topological algorithm implementation.

Let be a basic non-preemptive scheduler 'Basic-NPS', $T=\{t_1,t_2,...,t_k\}$ is the set of all tasks, $T_T=\{t_1,t_2,...,t_i\}$ is the set of tasks to process with $T_T \subset T$, $P=\{p_1,p_2,...,p_n\}$ is the set of all processors and $P_a=\{p_1,p_2,...,p_j\}$ is the set of available processors with $P_a \subset P$.

Basic-NPS ($T_x \Rightarrow P_y$) is able to schedule a set of T_x tasks on P_y processor. Let $\{p\}$ be the maximum of processors that P_y will contain. Then $\{p\}$ can be defined as the maximum of available processors already defined by the set P_a and $\{p\}=\max p_j / p_j \in P_a$. While $([P_a \neq \varnothing] \wedge [T_T \neq \varnothing])$ then $T_x \Rightarrow P_y : T_x \in T_T ; P_y \in P_a$.

In this scheduler, each processor will treat at maximum $m = \max t_i / t_i \rightarrow p_j \leq \max(\frac{|T|}{|P|})$ tasks with $j \in \{1,2,...,n\}$. Then, the worst case to process T is $K(T) = \max\left\{\max_i T_T \rightarrow p_1, ..., \max_{j<...<i} T_T \rightarrow p_k\right\}$. As proof, let suppose that it exist a set $L(T)$ as $\sum L(T) \geq \sum K(T)$. As 'Basic-NPS' manage $L(T)$ and $K(T)$, so we can introduce the following: $|L(T)| \leq m$ and $|K(T)| \leq m$. Thus, if $(\sum L(T) \geq \sum K(T))$ then there is at least one task $\{l\}$, with $k \in K(T)$, such as: $(A \wedge B \wedge C)$ with $A = (l \in L(T)), B = (l \notin K(T)), C = (l > k)$. This is impossible according to the definition of $K(T)$ which was defined as the worst case.

Algorithm 1 : Scheduling policy [Mahmoudi and Akil]

Aux: T : Set of all tasks, P : Set of all processors

1. **While** $(T \neq \varnothing)$ **repeat** :
2. N_T = ***Nbr_active_tasks***() ;
3. N_P = ***Nbr_ available_processors***();
4. **If** $(N_P \neq 0)$ **then**
5. **If** ($N_T < N_P$) **then**
6. **For** each processor N_{Pi} :
7. Generate-new-process (N_{Ti});
8. Identify-class (N_{Ti} , SCHED_FIFO);
9. **Endfor**
10. **Else** : N_{DT} = Desable_tasks (N_P - N_T) ;
11. Insert_desabled_tasks (N_{DT} , T);
12. **For** each processor N_{Pi} :
13. Generate-new-process (N_{Ti});
14. Identify-class (N_{Ti} , SCHED_FIFO);
15. **Endfor**
16. **EndIf**
17. **EndIf**
18. **EndWhile**

'Basic-NPS' policy is described by ***(Alg. 1)***. The first step consists on asking operating system to determine the number of available processor. Depending on this number, algorithm will generate process. One active process will be assigned for each available processor. These new processes will belong to the SHED_FIFO class in order to ensure preemption and especially to avoid context switching. Process will only stop running if work is complete or less frequently when another process, belonging to the same class, with higher priority requesting processor. The global execution will stop if there no more task to process.

This approach, and despite the centralized aspect, does not depend on the number of processors (number of processor is not predefined) or the minimum load of processors witch make this approach more generic and the parallelization strategy more portable. In fact portability is increasingly cited as a desirable goal in parallelization strategy conception.

Definition (14): *Portability*

It is a property which assures that parallel programs are both code portable and performance portable to various parallel machines.

$$\textit{Portability: } \mathrm{P}_n^{\mathrm{P}}(b \to t) = \frac{S_n^T}{S_n^B} * 100\% \quad = \frac{\left[N - (N-1)\alpha(N)\right]^{TS}}{\left[N - (N-1)\alpha(N)\right]^{BS}} * 100\%$$

Despite disagreement about the exact meaning of "Portability", we can consider ***(Def.14)*** as working definition. According to James D. Mooney [45] the primary goal of portability is to facilitate the activity of porting 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.
(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 physical transportation which includes use of compatible media or communication channels between processor, and interpreting and translating file formats, character codes, data representations, processor design. Standard languages and portable compilers bridge the gap between programs and the variety of CPU interfaces that exist in target environments. However, many of these mechanisms still define only part of the environment interface that many applications need. Elements such as file structures, memory management, or especially asynchronous event handling 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.

Here is another major challenge in multi-core multithread architecture programming. In an ideal case, moving from one-core to multi-core should provide ***n*** fold increase in computational power. But practically, it is something that never happened. In fact, all existing computational problems cannot be efficiently parallelized without incurring the costs of inter-processor coordination.

This kind of analysis was evoked in many researches. Let's come back to (***Def.10***) and we focus on Amdahl's law [36]. It captures the notion that the extent to which we can speed up any complex work is limited by percentage of the sequential part in the executed work. Amdahl's Law defines the maximum speed up S that can be achieved by n processors collaborating on an application, where k is the fraction of the work that can be executed in parallel. Assume, for

simplicity, that a single processor completes the work in one second. With n concurrent processors, the parallel part takes $(\frac{p}{n})$ seconds and the sequential part takes $(1-p)$ seconds. Overall, the parallelized computation takes $((1-p)+\frac{p}{n})$ seconds. Thus, $S(n)=1/((1-p)+\frac{p}{n})$. Through this formula, for the given problem and an eight-core machine, Amdahl's law says that even if we parallelize 90% of the solution, but not the remaining 10%, then we end up with only four-fold speedup, and not the expected eight-fold speedup. In fact, these additional parallel parts involve substantial communication and coordination.

In our dynamic parallelization strategy, as we explained later in nest section, each two threads will share only one FIFO queue in order to push neighbors of lowered pixels. Intuitively we are going to opt towards a solution with a simple lock-based shared FIFO queue. Associated push and pop methods will be synchronized by a mutual exclusion lock. Even if this implementation is a correct concurrent FIFO queue, because each method accesses and updates fields while holding an exclusive lock, the method calls take effect sequentially. And according to Amdahl's law, this sequential communication can substantially affect the performance of our program as a whole. In multi-core architecture, such synchronization technique can also be the origin of costly overheads. Even if we opt to second method based on lock-free solution [46] in order to minimize the overheads, it is demanded that at least one thread (of all the threads that are executing the push or pop function at one moment) is progressing (inserting or extracting pixels from or to the FIFO queue). Unfortunately, we do not know in advance how many parallel threads will call push or pop functions. And method calls still take effect sequentially. Other solution is wait-free technique [47], it is required that a process finishes within a finite number of execution steps. Something that we cannot also guarantee because we cannot predict how many points will be characterized and then how many pixels will be inserted in the FIFO queue. Finally we decide to move to spin-wait mechanism [48], illustration is given by ***(fig. 6)***, and a thread waiting to push an item might spin for a brief duration without being added to the queue of waiting threads. As a result, the thread is effectively put to sleep without relinquishing the remainder of its CPU time slot. It is potentially more efficient to spin and wait, instead of using either lock-free or wait-free mechanisms, because those force a thread context switch, which is one of the most expensive operations performed by the operating system.

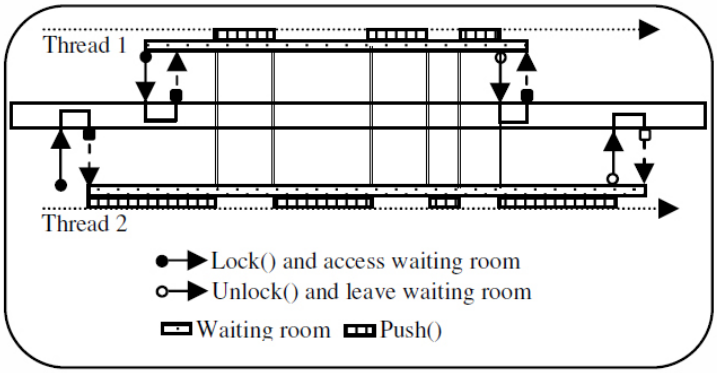


Figure 6: Spin-wait synchronization

2.4.3 The Merging phase

The key problem of each parallelization is merging obtained results. Normally this phase is done at the end of the process when all results are returned by all threads what usually means that only one output variable is declared and shared between all fighting threads. But as we mentioned in section 3.1, we are dealing with a dynamic evolution and if we take into account different steps of simple point detection then pixel characterizations, we can plan the following: The original shared data structure, containing all pixels, is divided into n research zones $\{z_1, z_2 ..., z_n\}$. We associate one thread from the following list $\{T_1, T_2 ..., T_n\}$ to each zone. Each thread can browse freely its zone and if it detects target pixel types, it lowers characterized pixel and it pushes its eight neighbors in one of the available FIFO queues. A queue is said available if only one thread (owner) is using it. One queue cannot be shared by more than two threads so if no queue is available, threads can create a new one and become owners.
Since two threads finished, they directly merge and a new thread is created and then same process is lunched again. New created thread will inherit queue shared between his parents. Thus it can restart research. It is also important to mention that there is no hierarchical order in thread merging, only criteria is finishing time. We mention also that one neighbor cannot be inserted twice. It is a precaution in order to minimize consumed cache. More formal description of merging techniques is given in by (***Algo. 2***).

It is important to highlight similarity and difference that may exist between our merging algorithm and KPN [49]. In effect, both are deterministic and do not depend on execution order. But KPN algorithm may be executed in sequentially or in parallel with the same outcome while our merging algorithm is designed only for parallel execution. KPN support recurrence and recursion while our merging algorithm support only recursion.

In large scale application, KPN showed consistence. Examples include Daedalus project [50] where generated KPN models are used to map process into FPGA architecture. Ambric architectures [51] implement also a KPN model using bounded buffers to create massively DMP Machines based on structural object programming model.

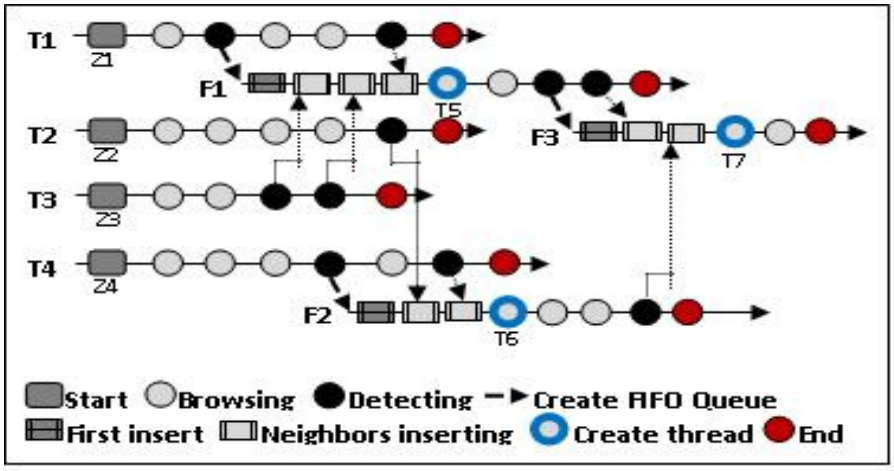


Figure 7: Illustration of merging phase with four threads

In a narrower framework limited to simple point characterization, the implementation of such a model will be very expensive and it would be better to find an easier and more specific algorithm.

An illustration of the merging algorithm with four threads is given by ***(fig.7)***. The original shared data structure is divided into 4 research areas $\{z_1, z_2, z_3, z_4\}$. Threads $\{T_1, T_2, T_3, T_4\}$ will start browsing different zones in parallel.

T_1 is the first to detect target point (constructible, destructible …) so it lowers characterized pixel (in z_1) and it pushes its eight (or four) neighbors in FIFO queue F_1 that it has created before continue browsing. Later, T_3 will detect new target point so it will lower characterized pixel in z_3 then push neighbors in F_1 before continue browsing. T_3 don't need to create new FIFO queue since F_1 is available. T_1 and T_3 will repeat this procedure twice. Since they finish browsing, they merge and new thread T_5 is born. T_5 will start browsing only F_1. Since it detect new target point so it will lower characterized pixel (in $z_5 = z_1 + z_3$) then push neighbors in F_3 that it has created before continue browsing.

Similarly T_2 and T_4 will generate the creation of F_2 and T_6. Here T_6 will eventually merge with T_5 to give birth to T_7. Finally there will be a single thread T_7 which will brows F_3 without detection any target points.

Algorithm 2 : Merging Algorithm [Mahmoudi and Akil]

Aux: Z : Set of research zones, T : Set of threads, $FIFO\_Q$: Set of available FIFO queues and P_T : Target pixel type ; P_D : Detected pixel

1. **For** all zones $(Z_i \in Z)$ **do**
2. **Parallel_browsing** (T_i , Z_i) ;
3. **End_for**
4. **For each** thread $(T_i \in T)$ **do**
5. **If** (pixel_caract(T_i , P_T)==True) **then**
6. **Modify_value**(P_D);
7. **If** (($FIFO\_Q \neq \varnothing$) **then**
8. **Usedstatus**($FIFO\_Q_j$,true) ;
9. **Insert_neighbors**(T_i , P_D, $FIFO\_Q_j$);
10. **Else** : **add_new_fifo** ($FIFO\_Q$);
11. **Usedstatus** ($FIFO\_Q_{j+1}$,False) ;
12. **Insert_neighbors** (T_i , P_D, $FIFO\_Q_{j+1}$);
13. **EndIf ; EndIf ;**
14. **EndFor;**

2.5 Conclusion

In this chapter we have identified needs for a common parallelization strategy of topological operators on shared memory parallel architectures. After studying theoretical basis of parallelization strategies, we presented our approach called SD&M. It is important to mention that some similarity may exist between our split/merging phases and alpha-extension/beta-reduction phases from structural perspective. Actually both approaches intended to put in place more guarantees that the parallelism will actually be met. But uses contexts are different. In effect, Jean Paul Sansonnet [52, 53, 54] team introduced alpha-extension (diffusion) and beta-reduction (merging) notions for stream manipulation in the framework of Declarative Data Parallel language definition and there techniques cannot be applied without a scalar function. While our proposal is restricted to topological characterization in the framework of topological operator's parallelization and no scalar function is required during the application of these two phases.

TOPOLOGICAL WATERSHED

The watershed is now used as a fundamental step in many powerful segmentation procedures. Judged by the great diversity of applications in which this transform appears relevant, watershed can be seen as one of the most popular segmentation tools coming from the field of mathematical morphology. In general, five large classes of algorithms to compute watershed transform can be figured out. The first one is based on flooding approach, the second is based on a path-cost minimization, third class is based on topological approach, fourth class is based on local condition and finally a fifth class based on minimum spanning approach. These classes will be studied in details in the second section of this chapter.

In addition to this study, we will focus also in the parallel implementation of watershed. In fact, the difficulty of such implementation resides in the non-local criteria of the watershed transform. The decision whether a pixel belongs to a basin cannot be based on purely local considerations as Roerdink and al. [2] introduced. He also explains that results given by some algorithms depend on the order in which pixels are treated during execution. In the sequential case, fixing the scanning order can resolve this problem, so that 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 the pixels, and this is unpredictable in the case of asynchronous processors. Throughout this section, we will leave in search of more adapted parallel algorithm to compute watershed transform suited for shared memory parallel machines.

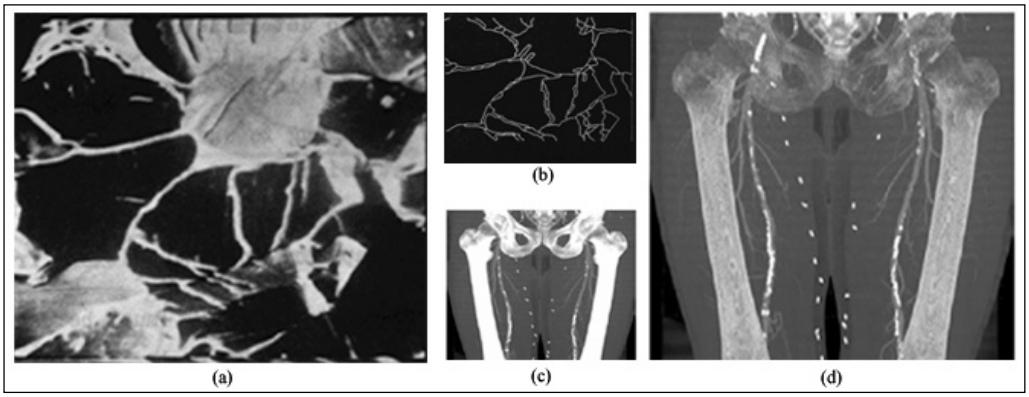
We start by presenting an intensive study of all existing watershed transform in the discrete case in section ***(3.1)***. This study concludes with a classification of different algorithms according to criteria of recursion, complexity, basins computing and topology preservation in section ***(3.2)***. Once best approach identified, we present, in section ***(3.3)***, a new algorithm to compute watershed that is parallel, preserves the topology of the input image, does not need prior minima extraction and suited for ***SMP*** machines. Based on stream notion introduced by Cousty [1], this algorithm does not require any sorting step, or the use of any hierarchical queue. Experimental analyzes such as execution time, performance enhancement and cache consumption are also presented and discussed.

3.1 Watershed transformations

The watershed concept began with Maxwell [55] who introduces the theory behind representing physical characteristics of a land by means of lines drawn on a map. He highlights relationships between the numbers of hills, dales and passes which can co-exist on a surface. Subsequently, through the work of Beucher and al. [56], watershed transform was introduced to image segmentation and nowadays it represents one of the basic foundations of image processing [57].

In this framework, the most simplified description of the watershed approach is to consider a grayscale image as a topographic surface: the gray level of a pixel becomes the elevation of a

point, the basins and valleys correspond to dark areas, whereas the mountains and crest lines correspond to the light areas. If topographic relief is flooded by water, watersheds will be the divide lines of the attraction's domains of rain falling over the region [58] or sources of water springing from reliefs' peaks. Another synopsis has shown consistency is 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. Figure 8 gives a very symbolic description of the mentioned approach. In fact, it shows trends watershed transform use for image processing.



(a) Cleavage fractures in steel, **(b)** contour of **(a)** obtained truth watershed definition introduced by Beusher and al. [56] in 1979, **(c)** Maximum intensity projection of original human lower limb **(d)** Bone tissue removed using mask extended with 3D watershed transform introduced by Straka and al.[59] in 2003.

Figure 8 : Watershed application 1979-2003

Despite its simplicity, this concept has been formalized in different ways giving rise to several definitions of watershed transform. In the discrete case, which is our main interest in this chapter, this problem is amplified since there is no unique definition of the path that the drop of water would follow. This led to a multitude of algorithm to compute watershed transform. Some of these algorithms don't even meet associated watershed definition. We also note that some definitions take the form of algorithm specification which makes the distinction between algorithm specification and implementation very complicated. This problem in literature has been partially resolved by Roerdink and al. [2] ten years ago. Actually authors presented a critical review of several definitions of the watershed transform and the associated sequential algorithms. Even they discuss various issues which often cause confusion in the literature; they don't go further in the classification or comparison of different approaches. They instead focus on parallelization aspect. In other more recent publications, authors tentatively drawn a comparison chart of some watershed transform definition to serve their end goals: showing the relationships that may exist between some discrete definition of watershed [60] or showing that most classical watershed algorithm do not allow the retrieval of some important topological features of the image [61]. In the following we present our analysis of watershed transform (**WT**) in the discrete case: WT based on flooding, WT based on path-cost minimization, watershed based on topology preservation, WT based on local condition and WT based on

minimum spanning forest. For each approach, we start by giving informal definition, then we present processing procedure followed by mathematical foundations and the algorithm of reference. Recent publications based on some approach are also presented and discussed.

3.1.1 Watershed based on flooding

Based on flooding paradigm [62,63,64], the intuitive idea underlying this method comes from geography. Since grayscale image can be seen as topographic surface, the intensity of a pixel can be considered as the altitude of a point. Now, let immerge this surface in still water, with holes created in local minima. Water fills up basins starting at these local minima. As described in ***(Alg. 3)***, the filling of basins is an iterative process that involves gradually raising the water level from Alt_{min} to Alt_{max}. Algorithm must, foreach iteration, fill existing basins (extension regions) and possibly create new basins (new regions). $\mathbf{L_R}$ will denote region list. Dams will be built where waters coming from different basins meet.

For mathematical formulation of the mentioned process, let $f : D \to \mathrm{N}$ be a digital grey value image, with Alt_{min} and Alt_{max} the minimum and the maximum value of. The threshold set of f at level ***Alt*** can be defined as:

Definition (15): *Threshold set of* ***(f)*** *at level* ***(Alt)*** *[Vincent and Soille]*

$$T_{Alt} = \{ p \in D / f(p) \le Alt \}$$

Algorithm 3 : Flooding watershed process

```
1.  for level from Alt_min to Alt_max
2.      // Action 1 : Extend existing region
3.      foreach (R∈ L_R) do Growing [R] until level Alt;
4.      end_for
5.      // Action 2 : Create new region
6.      foreach (Pixel P ∈ level)
7.          if (Pixel P is not associated to any region R) then
8.              Create new region [R] in L_R ;
9.              Add Pixel P to region [R];
10.             Growing [R] until level ;
```

This threshold set define a recursion with a gray level ***Alt*** increasing from Alt_{min} to Alt_{max}, the basin associated with the minima of f are successively expanded. Let X_{Alt} denote the union of the set of basins computed at level ***Alt***. A connected component of the threshold set T_{Alt+1} at level ***Alt+1*** can be either a new minimum or an extension of the basin in X_{Alt} . The geodesic influence zone (IZ) of X_{Alt} within T_{Alt+1} can be computed resulting in an update X_{Alt+1}. Thus we can introduce the following definition. MIN_h will denote the union of all regional minima at altitude ***Alt***.

Definition (16): *Flooding watershed*

$$\begin{cases} X_{Alt\,\min} = \{ p \in D / f(p) = Alt_{\min} \} = T_{Alt\,\min} \\ X_{Alt+1} = MIN_{Alt+1} \cup IZ_{T_{ALT+1}}(X_{Alt}) \end{cases}$$

The watershed $Wshed(f)$ of f is the complement of $X_{Alt_{\max}}$ in D : $Wshed(f) = D / X_{Alt\,\max}$

Vincent and Soille [65] presented an original and efficient implementation ***(Alg. 4)*** of the flooding watershed. This implementation uses **FIFO** queue and it needs two steps:

(1) Sorting pixels in increasing order of grey values (from Alt_{min} to Alt_{max})

(2) Flooding process: All nodes with grey level h are first given an initial label. Then those nodes that have labeled neighbors from the previous iteration are inserted in the queue, and then pixels geodesic influence zones are propagated inside the set of initial pixels.

In their study [2] Roerdink and Meijster have removed two points of inconsistency in Vincent algorithm's recursion. **(i)** Only pixels with grey value h are masked for flooding (line 13), instead of all non-basin pixels of ($level \le h$), as ***(Def.16)*** would require. This explains why labels of '***wshed-pixels'*** (line 15) are also propagated with labels of catchment basins. **(ii)** If a pixel is adjacent to two different basins, it is initially labeled '***wshed***'. But it is allowed to be overwritten at the current grey level by another neighbor's label, if that neighbor is part of a basin (lines 35-36).

They also propose some modification to implement the recursion ***(Def.16)*** exactly. In line 13, all pixels with $im[p] \le h$ have to be masked, the queue has to be initialized with basin pixels only (drop the disjunct lab[q] = wshed in line 15), the resetting of distances (line 50) has to be done in line 14, and the propagation rules in (lines 32-47) have to be slightly changed.

Algorithm 4 : Flooding watershed [Vincent & Soille]

input : Digital grey scale image *G=(D,E,im)*
Output: Labeled watershed image *lab* on *D*

```
1.  #define INIT -1   //initial value of lab image
2.  #define MASK -2   //initial value of each level
3.  #define WSHED 0   //label of the watershed pixels
4.  #define FICTITIOUS (-1,1)   //fictitious pixel ∉ D
5.  curlab ← 0   //curlab is the current label
6.  fifo_init(queue)
7.  for all (P∈D) do
8.    lab[p] ← INIT;
9.    dist[p] ← 0; //dist is a work image of distances
10. end_for
11. SORT pixels in increasing order of grey values ( hmin , hmax)
    // starting flooding process
12. for h = hmin to hmax do //geodesic SKIZ of level h-1 inside level h
13.     for all (P∈D) with im[p]=h do // mask all pixels at level h
            //these are directely accessible because of the sorting step
14.         lab[p] ← MASK;
15.         if (p has a neighbour q) with ((lab[p] > 0 or lab[q]=WSHED)) then
              //initialize queue with neighbours at level h of current basins or watersheds
16.           dist[p] ← 1;
17.           fifo_add(q, queue);
18.         end_if
19.     end_for
20.   curdist ← 1;
21.   fifo_add(FICTITOUS, queue);
22.   loop   //extend basins
23.     p ← fifo_remove(queue);
24.     if (p = FICTITIOUS) then
25.       if (fifo_empty(queue)) then
```

```
26.              BREAK ;
27.           else ( fifo _ add (FICTITOUS, queue) ;
28.                curdist ← curdis + 1;
29.                p ← fifo _ remove(queue);
30.           end_if
31.        end_if
32.        for all  (q∈ N_G(p)) do //labeling p by inspecting neighbours
33.           if (dist[q] < curdist) and (lab[q]>0 or lab[q]=WSHED) then
                 //q belongs to an existing basin or to watersheds
34.              if (lab[q]>0) then
35.                 if ((lab[p]=MASK) or (lab[p]=WSHED)) then
36.                       lab[p] ← lab[q];
37.                    else if  lab[p] ≠ lab[q] then
38.                          lab[p] ← WSHED ;
39.                    end_if
40.              else if (lab[p]=MASK) then
41.                       lab[p] ← WSHED ;
42.                 end_if
43.              else if  ((lab[q]=MASK) and (dist[q]= 0)) then //q is plateau pixel
44.                       dist[q] ← curdis + 1;
45.                       p ← fifo _ add(q, queue);
46.                       end_if
47.        end_for
48.     end_loop
        //detect and process new minima at level h
49.     for all (p∈ D) with (im[p]=h) do
50.        dist[p] ← 0;  // reset distance to zero
51.        if (lab[p]=MASK) then  // p is inside a new minimum
52.           curlab ← curlab + 1; //create new label
53.           fifo _ add( p, queue);
54.           lab[p] ← curlab ;
55.           while not  ( fifo _ remove(queue)) do
56.                    q ← fifo _ remove(queue);
57.                    for all (r∈ N_G(q)) do // inspect neighbours of q
58.                       if (lab[r]=MASK) then
59.                          fifo _ add(r, queue);
60.                          lab[r] ← curlab ;
```

From the introduction about immersion simulation above, we can see that the level-by-level method during the flood procedure is uniform. Unfortunately, this can cause over-segmentation in several cases. Based on this original simulating immersion, Shengcai and Lixu [66] propose, in 2005, a novel implementation using multi-degree immersion. To our knowledge, it is the last update that can be found in the literature, the proposed implementation resists to over-segmentation problem effectively. It starts by redefining the threshold set of f at level ***Alt***. Instead of the original formula ***(Def.15)***, they propose the following one:

Definition (17): *Threshold set of* ***(f)*** *at level* ***(Alt)*** *[Shengcai and Lixu]*

$$T_{Alt} = \{p \in D / f(p) - Diff(p) \le Alt\}$$

In latest definition, $Diff(p)$ refers to immersion level when the flooding reaches a pixel p. Segmentation results are sensitive to this function. In fact, if $Diff(p) = 0$, ***(Def.17)*** can be seen as a special case of ***(Def.15)***. Other extreme case, if $Diff(p)$ reaches its maximum values, all

pixels susceptible to be dumped, will be. According to the user requirement, $Diff(p)$ can be even a constant function or a function computed according to the local characteristic of pixel p. In their paper, Shengcai and Lixu define as the following. $Neighbor(p)$ will denote the set of all p neighbors. And $conn$ will refer to the predefined connectivity: $\{4,8\}$.

Definition (18): *Set of all connected neighbor [Shengcai and Lixu]*

$$Diff(p) = \sum_{q \in Neighbor(p)} \frac{|f(p)-f(q)|}{conn}$$

Obtained results through two implementations of the original and the multi-degree watershed shows that multi-degree immersion method resists the over-segmentation problem effectively. Indeed, the number of detected region in brain crop frame (181*217*181 voxel volume), decreases from 10991, using the old method, to only 35 using the new method. Computation time and consumed memory size are practically the same. More information about implementation can be found in [66].

3.1.2 Watershed based on path-cost minimization

In this class, there are two possible approaches. The first one associates a pixel to a catchment basin when the topographic distance is strictly minimal to the respective regional minimum. While the second one builds a forest of minimum-path trees, each tree representing a basin. In the following we start by introducing watershed by topographic distance [67] before moving to the watershed by image foresting transform [69,70].

Algorithm 5 : Watershed by topographic distance process

1. **Foreach** (marked area ∈ S_M)
2. insert pixels into priority queue Q ;
3. **end_for**
4. **While** ($Q \neq \phi$)
5. Ex_p = extract pixels with highest priority level;
6. **if** (neighbors of $p \in Ex_p$ have the same label Lab) **then**
7. $Lab_p = Lab$;
8. Q = all non-marked neighbors
9. **end_if**
10. **end_while**

Based on the drop of water principle, the intuitive idea behind topographic watershed approach is the steepest descent path principal [67, 68]. A drop of water falling on a topographic relief flows down, as "quickly" as possible, until it reaches a regional minimum.

Let $f : D \rightarrow \mathrm{N}$ be a digital grey value image. Let S_M be the set of markers, pixels where the flooding shall start, are chosen. Each is given a different label Lab .

Topographic watershed process can be described by ***(Alg. 5)***. Let us note that priority level when inserting neighbors (line 2) corresponds to the gray level of the pixel. In line 6, only neighbors that have already been labeled are compared. Finally, only neighbors (in line 8) that are not yet in the priority queue are pushed into the priority queue. The watershed lines set are the complement of labeled points set.

For mathematical formulation of the mentioned process, we follow presentation in [2] which is based on [67]. For the sake of simplicity, we restrict our self to the minimal set of notion that will be useful for our propos. And we start by introducing the topographic distance.

Let us consider $N_G(p)$ as the set of neighbors of pixel p, and $d(p,q)$ as the distance associated to the edge (p,q). Then the lower slope $LS(p)$ of f at a pixel p and the cost for walking from a pixel p to a neighboring pixel q can be defined as following:

Definition (19): *Lower slope of* ***(f)*** *at a pixel* ***p*** *[Roerdink and al.]*

$$LS(p) = \underset{q \in N_G(p) \cup \{p\}}{MAX} \left(\frac{f(p) - f(q)}{d(p,q)} \right)$$

Definition (20): *Cost functions [Roerdink and al.]*

$$\cos(p,q) = \begin{cases} LS(p).d(p,q) / f(p) > f(q) \\ LS(q).d(p,q) / f(p) < f(q) \\ \frac{1}{2}\left(LS(p) + LS(q)\right).d(p,q) / f(p) = f(q) \end{cases}$$

The topographical distance between p and q is the minimum of the topographical distances $T_f^{\pi}(p,q)$ along all paths between p and q : $T_f(p,q) = \underset{\pi \in [p \to q]}{MIN} T_f^{\pi}(p,q)$.

We recall also that the topographical distance along a general path $\pi = (p_0,...,q_l)$ is defined as $T_f^{\pi}(p,q) = \sum_i^{l-1} d(p_i, p_{i+1}) \cos t(p_i, p_{i+1})$.

Finally we can define the topographic watershed for a grey value image f, with f^* the lower completion of f. Note that each pixel, which is not in a minimum, has a neighbor of lower grey value ($f^* = f_{LC}$).

Definition (21): *Topographic watershed [Roerdink and al.]*

Let $(m_i)_{i \in I}$ be the collection of minima of f. The basin $CB(m_i)$ of f corresponding to a minimum $(m_i)_{i \in I}$ is defined as a basin of the lower completion of f :

$$CB(m_i) = \left\{ p \in D, j \in I / \{i\} : f^*(m_i) + T_{f^*}(p, m_i) < f^*(m_j) + T_{f^*}(p, m_j) \right\}$$

And the watershed is the set of points which do not belong to any catchment basin:

$$Wshed(f) = D \cap (\cup_{i \in I} CB(m_i)^c$$

Several shortest paths algorithms for the watershed transform with respect to topographical distance can be found in the literature but the reference algorithm is Fernand Meyer one. In the following we present a variant of Meyer algorithm with integrate the lower slope ***(Def. 19)*** of the input image as introduced Roerdink and al. [2].

Algorithm 6 : Watershed by topographic distance [Meyer]

Input : Lower complete image ***im*** on a digital grey scale image G=(D,E) with cost
Output : Labeled watershed image lab on D

1. #define WSHED 0 *//label of the watershed pixels*
2. *//Uses distance image dist. On output, div[v]=im[v], for all v ∈ D*
3. **for all** ($v \in D$) **do** *//Initialize*
4. $lab[v] \leftarrow 0;$
5. $dist[v] \leftarrow \infty;$
6. **end for**
7. **for all** (*local minima* m_i) **do**
8. **for all** ($v \in m_i$) **do**
9. $lab[v] \leftarrow i;$
10. $dist[v] \leftarrow im[v];$ *//Initialize distance with the values of minima*
11. **end for**
12. **end for**
13. $stable \leftarrow true;$ *//stable is a Boolean variable*
14. **repeat**
15. **for all** pixels ***u*** in forward raster scan order **do**
16. propagate(***u***)
17. **end for**
18. **for all** pixels ***u*** in backward raster scan order **do**
19. propagate(***u***)
20. **end for**
21. **until** *stable*
22. **procedure propagate(*u*)**
23. **for all** ($v \in NG(u)$) in the future (w.r.t scan order) for ***u*** **do**
24. **if** $(dist[u] + \cos t[u,v] < dist[u])$ **then**
25. $dist[v] \leftarrow dist[u] + \cos t(u,v)$
26. $lab[v] \leftarrow lab[u]$
27. $stable \leftarrow false$
28. **else if** $lab[v] \neq WSHED$ **and** $(dist[u] + \cos t[u,v] = dist[v])$ **then**
29. **if** $(lab[v] \neq lab[u])$ **then**
30. $lab[v] = WSHED$
31. $stable \leftarrow false$

The second approach to compute a watershed based on path-cost minimization, as we introduced in the beginning, consists on building a forest of minimum-path trees where each tree represent a basin. This approach is described in the framework of image foresting transform [69]. The IFT defines a minimum-cost path forest in a graph, whose nodes are the image pixels and whose arcs are defined by an adjacency relation between pixels. The cost of a path in this graph is determined by a specific path-cost function, which usually depends on local image properties along the path, such as color or gradient. The roots of the forest are drawn from a given set of seed pixels. For suitable path-cost functions, the IFT assigns one minimum-cost path from the seed set to each pixel, in such a way that the union of those paths is an oriented forest, spanning the whole image. The IFT outputs three attributes for each pixel: its predecessor in the optimum path, the cost of that path, and the corresponding root. Returned solution is usually obtained in linear time and requires a variant of Dijkstra [71], Moore [72] or Dial's shortest-path algorithm [73].

For mathematical formulation of the IFT-watershed, we start by defining some basic notions of image foresting transform as introduced in [69]. Actually, an image ***ImgIn*** can be seen as a pair (J, I) where J refers to a finite set of pixels and I refers to a mapping that assigns to each pixel $(p \in J)$, a pixel value $I(p)$ in some arbitrary value space. Distinct binary relation

between pixels of J will define an adjacency relation A. Once the adjacency has been fixed, ***ImgIn*** can be interpreted as a directed graph, whose nodes are image pixels and whose arcs are pixel pairs in A.

Before moving to path cost definition, let's remember that a sequence of pixels $\pi = \langle t_1, t_2, ..., t_k \rangle$ where $(t_i, t_{i+1}) \in A$ for $(1 < i < k-1)$ constitute a path. In the following we will denote by $org(\pi)$ the origin t_i of π and by $dst(\pi)$ the destination t_k of π. Now let assume a given function f that assigns to each path π a path cost $f(\pi)$, in some totally ordered set υ of cost values. We introduce the max-arc path-cost function $f_{\max}$ that will be used later. Note that $h(t)$ and $w(s,t)$ are fixed.

Definition (22): *Max-arc path-cost function [Falcão and al.]*

$$f_{\max}(\langle t \rangle) = h(t)$$

$$f_{\max}(\pi \cdot \langle s,t \rangle) = \max\{f_{\max}(\pi), w(s,t)\}$$

For IFT use, a specific function $f^S(\pi)$ can be defined since the search to paths start in a given set $(S \subset J)$ of seed pixels.

$$f^S(\pi)\begin{cases} f(\pi) if\ (org(\pi) \in S) \\ +\infty \quad otherwise \end{cases}$$

Now, we can introduce the spanning forest concept. We remember that a predecessor map is a function P that assigns to each pixel $t \in J$ either some other pixel$\in J$ or a distinctive marker $\mathbf{M} \notin J$. Thus, a spanning forest (**SF)** can be seen as a predecessor map which contains no cycles.

Definition (23): *Spanning forest [Falcão and al.]*

For any pixel $t \in J$, a spanning forest P defines a path $P^*(t)$ recursively as $\langle t \rangle$ if $P(t) = \mathbf{M}$ and $P^*(s) \cdot \langle s,t \rangle$ if $P(t) = s \neq \mathbf{M}$, we denote by $P^0(t)$ the initial pixel of $P^*(t)$.

Algorithm 7 : IFT Algorithm [Falco and Al.]

Input: ***Img*** $= (J, I)$: Image, $A \subset JxJ$: Adjacency relation, f :path-cost function

Output: P : optimum path forest, α, β : two sets of pixels with $\alpha \cup \beta = J$.

1. Set $\alpha \leftarrow \{\ \}$, $\beta \leftarrow J$ *//Initialize*
2. **for all pixels** $t \in J$ **do** *//Initialize*
3. Set $P(t) \leftarrow \mathbf{M}$
4. **end for**
5. **while** $\beta \neq \phi$ **do** *//Compute*
6. **remove** from β a pixel s such that $f(P^*(s))$ is minimum,
7. **add** s **to** α
8. **for** each pixel $t$such that $(s,t) \in A$
9. **If** $f(P^*(s) \cdot \langle s,t \rangle) < f(P^*(t))$ **then**
10. **Set** $P(t) \leftarrow s$;

Falcão and al. algorithm describes IFT computing. Its algorithm is based on Dijkstra's procedure [71] for computing minimum-cost path from a single source in a graph and returns an optimum-

path forest P for a seed-restricted cost function f^s or any pixel $t$with finite cost $f^s(P^*(t))$. Pixels will belong to a tree whose root is a seed pixel.

The IFT-watershed assumes that seeds pixel correspond to regional minima of the image or to markers that can be considered as imposed minima. The max-arc path-cost function $f_{\max}$ is the same as ***(Def. 22)***. We remember that $h(t)$ is fixed but arbitrary handicap cost for any paths starting at pixel t. We remember also that $w(s,t)$ is the weight of arc $\langle s,t\rangle \in A$, ideally, higher on the object boundaries and lower inside the objects.

Definition (24): *Arc weight functions*

$w_1(s,t) = |J(s) - J(t)|$

$w_2(s,t) = G(t)$

There are two usual arc weight functions ***(Def 24)***: ***(w_1)*** where $J(s)$ refers to the intensity of pixel of s. In that case IFT-Watershed is said by dissimilarity and ***(w_2)*** where $G(t)$ is the morphological gradient of ***ImgIn*** at pixel t. In that case IFT-Watershed is said on gradient.

In [74] Lotufo and al. introduced IFT-watershed from markers, ***(Algo. 8)***, which can be computed by a single IFT where the labeled markers are root pixels. Proposed algorithm use hierarchical FIFO queue (HFQ). In this case, the root map can be replaced by the label map which corresponds to the catchment basins and the used path-cost function is given by the following formula. Note that, the maximum of the set $\{I(p_j)/\, j\in(1,n)\}$ is computed only if ***(p_1)*** is already a marker pixel. And $I(p_i)$ refer to the value of the pixel p_i in the image I.

Definition (25): *Path-cost functions [Lotufo and al.]*

$$f_m(< p_1, p_1, \ldots, p_1 >) = \begin{cases} \max\{(I(p_1), I(p_2), \ldots, I(p_n)\} \\ +\infty \end{cases}$$

Algorithm 8 : IFT – Watershed from markers [Lotufo]

Input : I : input image, $wshed$: labeled marker image

Output : $wshed$: watershed catchment basins

Aux : C : cost map, initialized to infinity; $FIFO$:hierarchical FIFO queue

1. **for all pixels** $wshed(p) \neq 0$ **do** *//Initialize*
2. $C(p) \leftarrow I(p)$
3. Insert p in $FIFO$ with cost $C(p)$
4. **end for**
5. **while** $FIFO$ **do** *//Propagation*
6. $p \leftarrow$ remove from $FIFO$
7. **for each** $q \in N(p)$
8. **If** $(C(q) > \max\{C(p), I(q)\})$ **then**
9. $C(q) \leftarrow \max\{C(p), I(q)\}$;
10. Insert q in the $FIFO$ with cost $C(q)$;
11. $wshed(p) \leftarrow wshed(q)$

3.1.3 Topological watershed

The original concept behind topological watershed [75] is to define a "topological thinning" that transforms the image while preserving some topological properties, namely the number of connected components of each lower cross-section as we will explain in the following.

Before introducing global process to compute topological watershed, ***(Algo. 9)***, we define some basic notions that will be resumed in next paragraph for mathematical formulation : Let F be grayscale image and λ be a grey level, the lower cross-section $\overline{F}_\lambda$ is the set composed of all the points having an altitude strictly lower than λ. A point x is said to be ***'w-destructible'*** for F if its altitude can be lowered by one without changing the number of connected components of $\overline{F}_\lambda$, with $k = F(x)$. A map G is called a ***'w-thinning'*** of F if it may be obtained from F by iteratively selecting a ***'w-destructible'*** point and lowering it by one. Thus a basic definition of the topological watershed and a global description of its computing process can be given by ***(Def. 26)*** and ***(Alg. 9)***. Note that this process is repeated on loop until no more ***'w-destructible'*** point remains.

Definition (26): *Topological watershed [Couprie and Bertrand]*
A topological watershed of F is a ***'w-thinning'*** of F which contains no W-destructible point. The major feature of this transform is to produce a grayscale image.

Algorithm 9 : Topological watershed process

1. For all p in E, check number of connected components of the lower cross-section at level p which are adjacent to p.
2. Lower the value of p by one if this number is exactly one

For deep mathematical formulation, we follow description provided in [64]. We start by defining a simple point in a graph, in a sense which is adapted to the watershed, and then we extend this notion to weighted graphs through the use of lower sections [75].

Couprie and al. define a transform that acts directly on the grayscale image, by lowering some points in such a manner that the connectivity of each lower cross-section $\overline{F}_\lambda$ is preserved. The regional minima of the result, which have been spread by this transform, can be interpreted as the catchment basins. The formal definition of topological watershed ***(Def. 28)*** relies on the following particular notion of simple point.

Definition (27): *Simple point [Couprie and Bertrand]*

Let $G = (E, \Gamma)$ be a graph and let $X \subset E$.

A point $x \in X$ is simple (for $\overline{X}$) if the number of connected components of $\overline{X} \cup \{x\}$ is equal to the number of connected components of X. In other words, x is simple (for $\overline{X}$) if x is adjacent to exactly one connected component of $\overline{X}$.

Definition (28): *Topological watershed [Couprie and Bertrand]*

Let $F \in F(E)$, $x \in E$ and $k = F(x)$. The point x is destructible for F if x is simple for $\overline{F}_k$. We say that $W \in F(E)$ is a topological watershed of F if W may be derived from F by iteratively lowering destructible points by one until stability (that is, until all points of E being non-destructible for W).

Actually checking whether a point is ***'w-destructible'***, or not, cannot be done locally if only available information are graph (E,Γ) and a function F. Couprie and al. [76] propose a new algorithms ***(Algo. 10)*** making possible to perform this test on all the vertices of a weighted graph in linear time, and also to check directly how low the ***'w-destructible'*** point may be lowered until it is no more w-destructible, thanks to the component tree which may be built in quasi-linear time. In the following, we introduce Couprie's functions to identify ***'w-destructible'*** point.

Algorithm 10 : w-destructible function [Couprie and al.]

Input : $F, C(\overline{F}), \Psi$;

1. $V \leftarrow$ Set of element of $C(\overline{F})$ pointed by $\Psi(q)$ for all q in $\Gamma^{-1}(p)$;
2. **If** $(V = \phi)$ **then** return $[\infty, \phi]$;
3. $[k_m, c_m] \leftarrow$ **HighestFork** $(C(\overline{F}), V)$;
4. **If** $[k_m, c_m] = [\infty, \phi]$ **then** return $\min(V)$;
5. **If** $(k_m \leq F(p))$ **then** return $[k_m, c_m]$ **else** return $[\infty, \phi]$;

Previous algorithm gives correct results with regard to the definition ***(Def. 28)*** and is linear in time complexity with respect to the number of neighbors of p. Checking whether a point is ***'w-destructible'*** or not, involves the computation of the highest fork of different elements of the set ***V(p)***,see ***(Algo. 11)***. This may require a number of calls to ***BLCA*** (Binary lowest common Ancestor) which is quadratic with respect to the cardinality of ***V(p)***: every pair of elements of ***V(p)*** has to be considered.

Algorithm 11 : HighestFork Function [Couprie and al.]

Input : C : *a component tree,* V *: a set of components of C*

1. $[k_1, c_1] \leftarrow \min(V)$; *// let $[k_2,c_2]...[k_n,c_n]$be the other elements of V*
2. $k_m \leftarrow k_1$;
3. $c_m \leftarrow c_1$;
4. **for** i**from** 2 **to** n **do**
5. $[k, c] \leftarrow$ BLCA $(C, [k_i, c_i], [k_m, c_m])$;
6. **If** $[k, c] \neq [k_i, c_i]$ **then** $k_m \leftarrow k_1$;
7. $c_m \leftarrow c_1$;
8. **If** $(k_m = k_1)$ **then return** $[\infty, \phi]$ **else return** $[k_m, c_m]$;

The ***HighestFork*** function returns the highest fork of the set V, or the indicator $[\infty, \phi]$ if there is no highest fork. This algorithm makes ***(n-1)*** calls of the ***BLCA*** operator, where n is the number of elements in V.

Let ***C*** be a component tree, let ***V*** be a set of components of ***C***, we denote by ***min(V)*** an element of ***V*** which has the minimal altitude. For the following algorithm, we assume that C is represented in a convenient manner for ***BLCA***. Thus, we must propose a criterion for the selection of the remaining ***'w-destructible'*** points, in order to avoid multiple selections of the same point.

Couprie and al. introduce the idea to give the greatest priority to a ***'w-destructible'*** point which may be lowered down to the lowest possible value. They prove that an algorithm which uses this strategy never selects the same point twice. A priority queue could be used to select ***'w-destructible'*** points in the appropriate order. Here, we present their specific linear watershed algorithm which may be used when the grayscale range is small.

Algorithm 12 : Topological watershed [Couprie]

Input: $F, C(\overline{F}), \Psi$;

Output: F ;

1. **For** k **from** $k_{\min}$ **to** $(k_{\max} - 1)$ **do** $L_k \leftarrow \phi$
2. **For all** $(p \in E)$ **do**
3. $[i, c] \leftarrow W - Destructible(F, p, C(\overline{F}), \Psi)$
4. **If** $(i \neq \infty)$ **then**
5. $L_{i-1} \leftarrow L_{i-1} \cup \{p\}$;
6. $K\{p\} \leftarrow i - 1$;
7. $H\{p\} \leftarrow$ pointer to $[i, c]$;
8. **end if**
9. **end for**
10. **For** k **from** $k_{\min}$ **to** $(k_{\max} - 1)$ **do**
11. **While** $(\exists p \in L_k)$ **do**
12. $L_k = L_k / \{p\}$;
13. **If** $(K(p) = k)$ **then**
14. $F(p) \leftarrow k$;
15. $\Psi(p) \leftarrow H(p)$;
16. **For all** $(q \in \Gamma(p), k < F(q))$ **do**
17. $[i, c] \leftarrow W - Destructible(F, q, C(\overline{F}), \Psi)$;
18. **If** $(i = \infty)$ **then** $k(p) \leftarrow \infty$;
19. **Else if** $(k(p) \neq (i - 1))$ **then**
20. $L_{i-1} \leftarrow L_{i-1} \cup \{q\}$;
21. $k(p) \leftarrow (i + 1)$;
22. $H\{q\} \leftarrow$ pointer to $[i, c]$;

3.1.4 Watershed transform based on local condition

There is a big similarity between this approach and the drop of water one. Actually basin surface increases in a progressive manner. The local condition of label continuity is iteratively applied along the steepest descent path that reaches the basin minimum. The downhill algorithm, the hill climbing algorithm and the toboggan algorithm are based on this approach. More details of the first two algorithms are given by [67,68] and the toboggan algorithm will be detailed later in this section. Differences between these three algorithms lie in the processing strategy and data structure as shown in [2].

For mathematical formulation, we follow description provided in [60]. In witch Audigier and al. start by presenting the following catchment basin formulation $CB_{LC}(m_i)=\{v\in \mathrm{V}, \mathrm{L}(v)=\mathrm{L}(m_i)\}$, since local condition watershed assigns to each pixel the label of some minimum m_i. Thus watershed can be defined as follow. We recall that the condition of '$\mathbf{P}_{steepest}(v)\neq\{\ \}$' means that '$v$' has at least one lower neighbor.

Definition (29): *Watershed based on local condition [Audigier and Lotufo]*

For any lower complete image L_{CB}, a function L assigning a label to each pixel is called watershed segmentation if:

a) $L(m_i)\neq L(m_j)\forall i\neq j/\{m_k\}$ is the set of minima of L_{LC}.

b) For each pixel v with $\mathbf{P}_{steepest}(v)\neq\{\ \}, \exists p\in \mathbf{P}_{steepest}(v), \mathrm{L}(v)=\mathrm{L}(p)$.

As we mentioned earlier, we will introduce the toboggan algorithm [77, 78] as a reference of the local condition watershed approach. Actually this algorithm is referred as a drainage analogy. It seeks to identify the steepest descent from each pixel of the gradient magnitude of the input image to a local minimum of the topographic surface. Then pixels that belong to the same minima are merged by assigning them a unique label. Sets of pixels having the same label will define catchment basins. The resulting watershed regions are divided by a boundary path which will build the watershed lines.

Let consider $G: D\rightarrow R^{+}$ as a gradient magnitude image, where D is the indexing domain of the image. D can be decomposed into a finite number of disjoint level sets since pixels are sorted in the increasing order. Sets can be denoted by: $D_h=\{p\in D\,|\,G(p)=h\}$. Lin and al [77] define the following pixels classes: Class C_1 refers to all pixels p in D_h with an altitude strictly greater than the altitude of its lowest neighbor. Class C_2 refers to all pixels p in D_h belonging to a connected component with one or more catchment basin and $p\notin C_1$. Finally, class C_3 refers to all pixels p in D_h belonging to a connected component without any catchment basin. Thus we can give a global description of the computing process ***(Algo. 13)*** followed by the toboggan algorithm ***(Algo. 14)***.

Algorithm 13 : Toboggan watershed process

1. Records the sliding directions for all $(p \in C_1) \cup (p \in C_2)$ in D
   a. Records the lowest neighbours of all $(p \in C_1)$ in D.
   b. Grown region from all $(p \in C_1)$
2. Assign label for all $(p \in C_3)$
3. Assign label to each unlabeled image by first tobogganing then backtracking using best first search.

Algorithm 14 : Toboggan Algorithm [Lin and al.]

Input : Img : a gradient magnitude image;

Output: L : a label image, Q : empty FIFO queue ;

1. **For all** $(p \in D)$ **do** *//Simulation of sliding for all C_1 pixels*
2. $h = G(p)$
3. $h_{MIN} = \min\{G(q), q \in Neighbor(p)\}$
4. **If** $(h > h_{MIN})$ **then**
5. $S = \{(q \mid G(q) = h_{MIN}) \&\& (q \in Neighbor(p))\}$
6. $SlidingList(p) = S$
7. $Q \leftarrow p$
8. $GrowingDist(p) = 0$
9. **Else if :**
10. $SlidingList(p) = \phi$
11. **End if**
12. **End for**
13. **While** $Q \neq \phi$ **do** *//Simulation of keep- sliding for all C_2 pixels*
14. $p \leftarrow Q$
15. $d = GrowingDist(p) + 1$
16. $h = G(p)$
17. **For all** $(q \in Neighbor(p))$ **and** $(G(q) = h)$ do
18. **If** $(SlidingList(q) = \phi)$ **then**
19. Append (p) to $SlidingList(q)$
20. $GrowingDist(q) = d$
21. $Q \leftarrow q;$
22. **Else If** $(GrowingDist(q) = d)$ **then**
23. Append (p) to $SlidingList(q)$
24. **End if**
25. **End while**
26. **For all** $(p_0 \in D)$ **and** $(SlidingList(p_0) = \phi)$ **do** *// labeling C_3 pixels*
27. **If** $L(p_0)$ is not assigned **then**
28. $L(p_0)$ =new_label
29. $h = G(p_0)$
30. **While** $Q \neq \phi$ **do**
31. $p \leftarrow Q$
32. **For all** $(p \in Neighbor(p))$ **and** $(G(q) = h)$ **do**

```
33.                 If L(q) is not assigned then
34.                     (L(q) = L(p_0))
35.                     Q ← p
36.                 End if
37.             End for
38.         End while
39.     End if
40. End for
41. For all (p ∈ D) do // Tobogganing – Depth first search
42.   Resolve(p)
43. End For
```

Algorithm 15 : Resolve function [Lin and al.]

Input : Pixel site p

```
1.   If L(p) is not assigned then
2.     S = SlidingList(p)
3.     For all (q ∈ S) do
4.       Resolve(q)
5.     End for
6.     If S has a unique label α then
7.       L(p) = α
8.     Else
9.       L(p) = RIDGE_label
10.    End if
11.  End if
```

3.1.5 Watershed transform based on minimum spanning forest

The original idea is very close to the second case of the path cost minimization based watershed that consist on building a spanning forest from a graph. Actually, the beginning was with Meyer [79] who proposes to compute watershed transform from a weighted neighborhood graph whose nodes are the catchment basins corresponding to the minima of the image. Arcs of the graph, that separate neighbor catchment basins, are weighted by the altitude of the pass between these basins. Extracted minimum spanning forests define partitions that are considered solution of watersheds. It's important to mention that returned solutions are multiple. Authors established also the links between the minimum spanning forest and flooding from marker algorithms. Trough Meyer's bases, Cousty and al. [1] introduce the watershed-cuts and establish the optimality of this approach by showing the equivalence between the watershed-cuts and the separations induced by minimum spanning forest relative to the minima.

For mathematical foundations, we will follow notations in [1] to present some basic definitions to handle with minimum spanning forest cuts and watershed-cuts.

Let G be graph with $G = (V(G), E(G))$. $V(G)$ is a finite set of vertex of G. Unordered pairs of $V(G)$, called also edge of G, constitute the element of $E(G)$ set.

Let denote the set of all maps from E to $\mathbb{Z}$ by F and we consider that any maps of F weights the edges of G. Let $\mathrm{F} \subseteq F$ and $u \in E(G)$, $\mathrm{F}(u)$ will refers to the altitude of u and $\mathrm{M(F)}$ will refers the graph whose vertex set and edge set are, respectively, the union of the vertex sets and edge sets of all minima of F.

Let X and Y be two sub-graphs of G. We say that Y is a forest relative to X if Y is an extension of X and for any extension $\mathrm{Z} \subset \mathrm{X}$ of X, we have $\mathrm{Y=Z}$ whenever $\mathrm{V(Z)=V(Y)}$.

(i) Y is said a spanning forest relative to X (for G) if Y is a forest relative to X and if $\mathrm{V(Y)=V}$. In this case, there exists a unique cut S for Y. It is composed by all edges of G whose extremities are in two distinct components of Y. Since Y is an extension of X, it can be seen that this unique cut S (induced by Y) is also a cut for X.

(ii) Y is said a minimum spanning forest relative to X (for F, in G) if Y is a spanning forest relative to X and if the weight of Y is less than or equal to the weight of any other spanning forest relative to X. In this case, S is considered as a minimum spanning forest cut for X.

Trough these equivalences, Cousty demonstrate that the set $\mathrm{S} \subseteq \mathrm{E}$ is a minimum spanning forest cut for $\mathrm{M(F)}$ if and only if S is a watershed cut of F, that can be computed by any minimum spanning tree algorithm. And he proposes a linear algorithm to compute it using a new ***'stream'*** notion that we will not detail later. Only the stream algorithm will be introduced. Now, before presenting the watershed-cuts algorithm we just recall the definition of the minimal altitude of an edge.

Definition (30): *Minimal altitude of an edge [Cousty and al.]*

Let denote by $F^{-}(x)$ the map from V to $\mathbb{Z}$ such that for any $x \in V$, $F^{-}(x)$ is the minimal altitude of an edge which contains x. Then a path $\pi = \langle x_0, ..., x_l \rangle$, is considered as a path of steepest decent for F (in G) if for any $i \in [1, l]$, $F(\{x_{i-1}, x_i\}) = F^{-}(x_{i-1})$.

Algorithm 16 : Watershed-cuts algorithm [Cousty and al.]

Input : (V,E,F) : Edge-weighted graphs;
Output: Ψ: a flow mapping of F

1. **Foreach** $(x \in \mathrm{V})$ **do** $\psi(\mathrm{x}) \leftarrow \mathrm{NO\_LABEL}$;
2. $nb\_labs \leftarrow 0$
3. **Foreach** $(x \in \mathrm{V})$ **such that** $(\psi(\mathrm{x})=\mathrm{NO\_Label})$ **do**
4. $[\mathrm{L,Lab}] \leftarrow \mathrm{Stream(V,E,F,\psi,x)}$;
5. **If** $(lab = -1)$ **then**
6. nb_labs++
7. **Foreach** $(y \in \mathrm{L})$ **do** $\psi(\mathrm{y}) \leftarrow \mathrm{nb\_labs}$;
8. **Else Foreach** $(y \in \mathrm{L})$ **do** $\psi(\mathrm{y}) \leftarrow \mathrm{labs}$;

Previous algorithm computes a watershed-cut using a stream function that is described as the following:

Algorithm 17 : Stream function [Cousty and 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).

```
1.  L ←{x}
2.  L'←{x} // the set of sources not yet explored of L
3.  While there exists (y ∈ L') do
4.      L' ← L' \ {y};
5.      breadth_first ← TRUE ;
6.      While (breadth_first) and (∃ {y,z} ∈ E / z ∉ L and F({y,z})= F(y)) do
7.        If (Ψ(z) # No_label) then
8.                              return [L,Ψ(z)] // exist an Θflow L already labeled
9.        Else if ( F⁻(z) < F⁻(y) ) then
10.               L ← L ∪ {z}; // z is the only well of L
11.               L' ← {z}; // switch the in-depth exploration first
12.               breadth_first ← FALSE
13.           Else
14.               L ← L ∪ {z}; // therefore z is a well of L
15.               L' ← L' ∪ {z}; // continue exploration in width first
16. return [L,-1]
```

3.2 Classification of watershed algorithms

In this section we will learn from different syntheses present in Roerdink [2] and Audigier [60] works. The following table summarizes some characteristic of introduced watershed transforms. Selected criteria are justified by our objective to identify the most suitable algorithm for parallel implementation.

The starting point is the definition space; we note that IFT-Watershed and MSF-Watershed definitions are limited to the discrete space while the other watersheds definitions are spread into continue space. IFT-Watershed, MSF-watershed and LC-Watershed form the 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 unique solution while all other definitions return multiple solutions. Note that set of solutions returned by the IFT-Watershed can be unified by creating litigious zones when solutions differ [60]. All six algorithms, that don't exactly include their definitions, return unique solution but don't preserve the number of connected components of the original input image. Actually, Vincent-Soille, Meyer and Lin's algorithm don't preserve important topological features. Only Lotufo, Couprie and Cousty's algorithm are correct from this point of view.

Regarding computing process, only Flooding-Watershed needs pixel's sorting while others transforms will pass this costly step. But this does not preclude associated algorithms to use hierarchical structures when implementing. Except Cousty's algorithm that doesn't need any hierarchical queue.

Watershed based on					
Flooding	*Path-cost minimization*		*Topology*	*Local condition*	*MSF*
	TD	*IFT*			
Vincent & Soille [65]	Meyer [67]	Lotufo [70]	Couprie [75]	Lin [77]	Cousty [1]
Defined in	Disc. cont. Space	Disc. cont. space	Only on discret. space	Disc. cont. space	Disc. cont. space	Only on discrete space
Classified as	Line WT	Line WT	Region WT	Line WT	Region WT	Region WT
Gives unique solution	Yes	Yes	No	No	No	No
Preserve topology	No	No	Yes	Yes	No	Yes
Requires a sorting step	Yes	No	No	No	No	No
Use of h.queue	Yes	Yes	Yes	Yes	Yes	No
Minima computing	Yes	Yes	No	No	-	No
Is linearity	Linear	-	linear	Linear*	-	Linear

Table 3 : Comparison between main watershed transform

Vincent-Soille and Meyer's algorithms impose also a prior minima computation, which is not the case of the others. For complexity, observe that Vincent and Soille algorithm runs in linear 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 the priority function is limited and the number of neighbors of a point is small constant, Couprie's algorithm runs also in linear time with O(n + m) complexity. Lotufo and Cousty's algorithm run also in linear time. Cousty's algorithm is executed at most O(|E|) times.

Trough this analysis, ***(Algo. 16)*** holds best characteristics. The fact that sorting step is not required, hierarchical queue is not used and minima are not computed, make it an excellent candidate for parallelization on shared memory architecture.

3.3 Construction of parallel topological watershed

In this section, we start by introducing some basic definitions of stream notion which is crucial to the flooding paradigm. Then, we introduce in detail our parallel watershed-cut. Illustration of parallel compution process is given. Execution time and cache consumption are performed and analyzed. Efficiency and scalability are also presented and discussed.

3.3.1 Basic notions and definitions

Based on Cousty approach [1], we define and illustrate 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 steepest descent for F. We recall that steepest descent principal is already defined on section 3.1.2.

Now, Let 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 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 $L_1 \prec L_2$, if there exist a top x of L_1 and a bottom y of L_2, and there is from y to x a path L, with steepest decent for F. If $L_1 \prec L_2$ the L is a stream. If there is no stream under L, L is considered as an $\prec -stream$. Now any stream L which contains $\prec -streams$ is itself an $\prec -streams$.

Basic illustration of stream notion is given by ***(fig. 9): (a)*** the red graphs superimposed 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 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 $\prec -streams$ and they contain the set $\{a,b\}$ which is the vertex set of minimum of F.

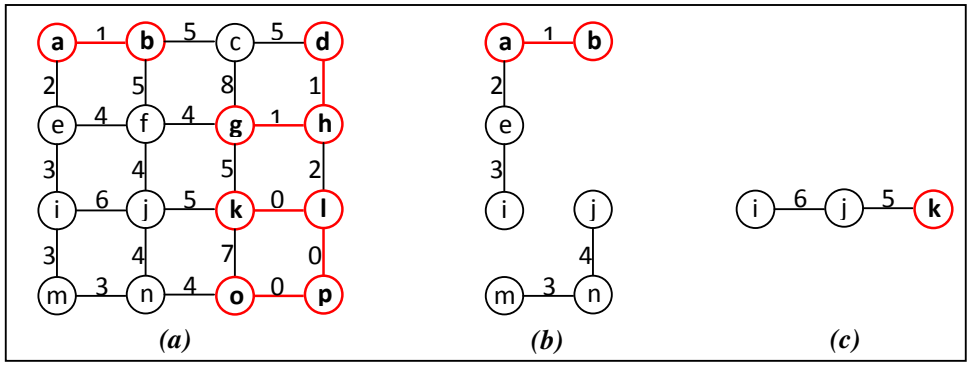
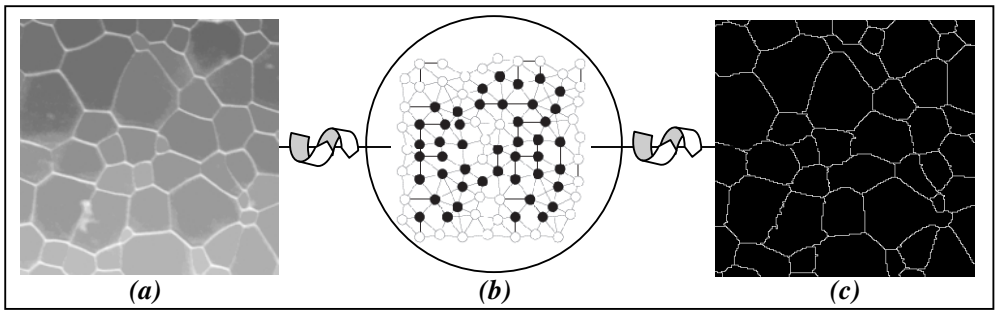


Figure 9 : Stream notion illustration following Cousty approach [1].

Streams extracted by Cousty function are $\prec -streams$. In the following we recall the link that exist between $\prec -streams$ and minima. Let L be a stream. If L is $\prec -stream$ 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 $\prec -streams$, 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 $\prec -streams$ minimal for the inclusion relationship.

We will now move on to define flow family notions. Actually the vertices of a graph can be arranged in the following manner with the aim of partition the vertex set of G from $\prec -streams$ of F. Let $\zeta = \{L_1, ..., L_n\}$ be a set of n $\prec -streams$. ζ is said a flow family if $\cup\{\zeta_i \mid i \in \{1,...,n\}\} = V$ and for any two family 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.



(a) Input image ***(b)*** associated weighted graph ***(c)*** output watershed

Figure 10 : watershed computing principal

Trough these notions 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 $\prec -stream$ 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 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 algorithm 18, that is based on parallel extraction of streams, able to produce a flow-cut hence a watershed. A general illustration is given by ***(fig.10)***.

3.3.2 Parallel stream computing

For that propose, following algorithm will assigns, in 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 than streams can be completely computed in parallel, see ***(fig. 11)***. For N point (x_1, x_2… x_n), their associated flows are simultaneously extracted: (L_1, L_2… L_n).

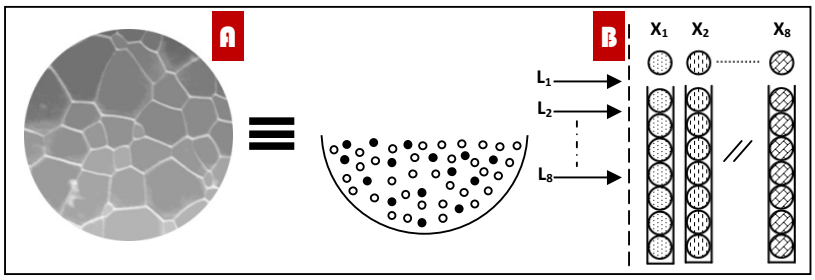
Each flow 'L_i' is composed of point not yet labeled and whose source is x_i. Stream function proposed by Cousty, pleaded in line 5 ***(Alg. 18)***, is launched N times. It allows the extraction of $L_{i \in \{1,2,3...n\}}$. Intuitively, it explores the path of greatest slope, by mixing iterations first in-depth and width of the approaches.

The main invariants of this function are:

(i) The set 'L' is, for each iteration, a stream (flow).
(ii) The set L' (line 2 - stream function) includes all wells of L not yet explored.

The stream function ***(Alg. 17)*** 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 $\prec -stream$. 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 steepest descent. Thus, there is an $\prec -stream$ 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 $\prec -stream$. Otherwise, if only depth-first iterations were used, Stream could be stuck on plateaus (connected sub-graphs of G with constant altitude) since some bottoms of L would never be explored.



(a) Partition of input image ***(b)*** Parallel stream computation
Figure 11 : Flow compute illustration

At the end of flow function executing, a family , of ***N*** streams (***L1***, ***L2***… ***Ln***) whose elements must be labeled is generated. The initial procedure [in the iterative case] is to assign a new label (***nb_labs***) to each '***Li***' element if the latter is a $\prec -stream$. If it is not the case, the old returned label ***lab***, of the $\prec -stream$ '***Hi***', included in '***Li***', is assigned to the different elements of '***Li***'. Now if we want to launch this procedure in parallel manner, ***N/2*** flows can be treated at ones.

Algorithm 18 : Parallel watershed-cut [Mahmoudi and Akil.]

Input : (V, E, F) : Edge-weighted graphs;
Output : Ψ : Flow partition of F

1. foreach x ∈ V do Ψ(x) ← No-Label ; **// No data dependency - FULL PARALALISM**
2. nb_labs ← 0 ; **// Global shared attributed label**
3. i ←0 ; treated stream
4. foreach (x ∈ V) such as (Ψ(x) = = No-Label) do **// lunch N process in parallel**
5. $[L_i, lab_i]$ ← **Stream** (V, E, F, Ψ,x_i) ; **// to get associated stream for each x_i**
6. nb-fusion = i ;
7. while (nb_fusion != 1)
8. for (j =0 ; j <= nb_fusion ; j+=2) do **// lunch (nb_fusion/2) process at once**
9. if ($L_j \cap L_{j+1}$) = ∅ then **s-labeling** ($[L_j, lab_j]$, nb_labs) ;
10. **s-labeling** ($[L_{j+1}, lab_{j+1}]$, nb_labs) ;
11. else **f-labeling** ($[L_j, lab_j]$, $[L_{j+1}, lab_{j+1}]$, nb_labs) ;
12. nb-fusion = nb-fusion / 2 ;

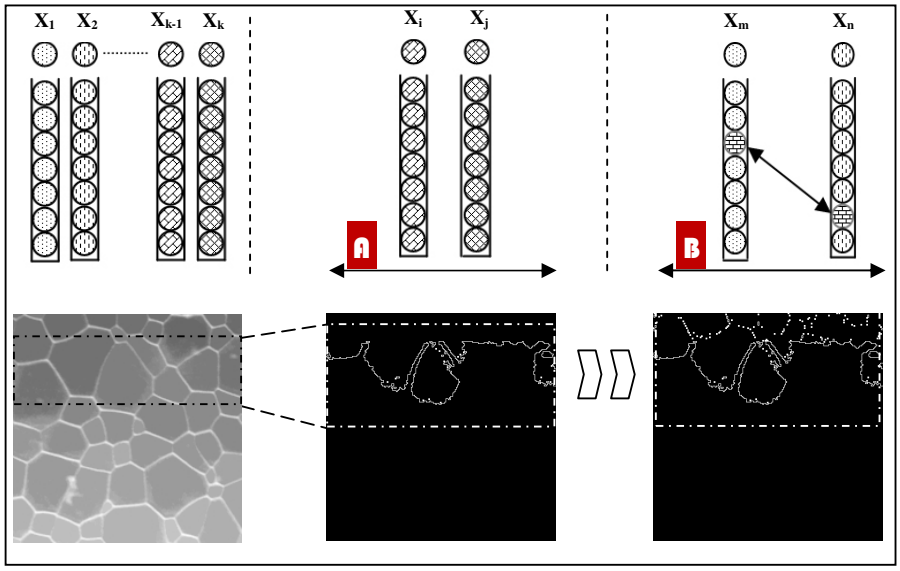
The procedure, in the parallel case, is based on the idea of labeling and merging two obtained flows at once. If two flows (to merge) '***Li***' and '***Li+1***' contain no common summit, ($L_i \cap L_{i+1}$)=∅, meaning there are no common wells between the two sources '***xi***' and '***xi+1***', in this case the merging is simple, for each flow '***Li***' and '***Li+1***', see ***(fig 12.a).*** Note that ***s-labeling*** function ***(Algo. 19)*** launches only the initial procedure [used previously in the iterative case].

Algorithm 19 : Function s-labeling [Mahmoudi and Akil]

Input: (L, lab, nb_labs)

1. **if** (lab = -1) **then** // L is $\prec -stream$
2. nb_labs ++ ;
3. **foreach** (y ∈ L) **do** Ψ(y) ← nb_labs ;
4. **else**
5. **foreach** (y ∈ L) **do** Ψ(y) ← lab ;
6. **return** NULL

If the two flows (to merge) 'L_i' and 'L_{i+1}' contain 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 12.b).*** In this case, merging is more complicated. We developed ***f-labeling*** procedure ***(Algo. 19)*** able to make fusion in the following special cases: ***(i)*** 'L_i' and 'L_{i+1}'are two $\prec -streams$, ***(ii)*** 'L_i' and 'L_{i+1}'are two *streams* including two $\prec -streams$, ***(iii)*** 'L_i' is an $\prec -stream$ and 'L_{i+1}' is a *stream* including an $\prec -stream$.



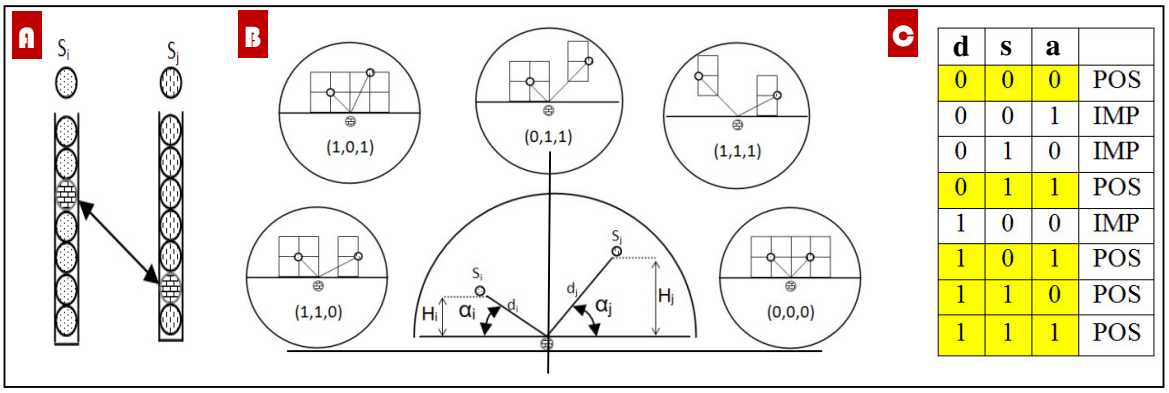
(a) Merging streams with common wells ***(b)*** Merging streams with common wells

Figure 12 : Merging illustration

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? Proposed solution is inspired from the flooding paradigm. Indeed, we start by studying all possible cases of merging two water streams gushing from different sources, see (***fig. 13***). Our goal is to identify which stream will be the first to reach the well. This latest will mark the well by its own label. The starting point is the steepest decent 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, topographic slope particularly influence the runoff. The inclination of the slope is surely the most important

topographical aspect. Normally, its impact is limited on short slope. It is more visible on longer slope even if runoff needs a certain distance to reach its maximum velocity. Mathematic formulation of flow medium speed can be is given by the Chezy formula: $V_c = c\left(h * s\right)^{1/2}$ introduced in 1769. '***C***' refers to roughness coefficient of Chezy. '***S***' refers to the slope, and '***h***' refers to the altitude of the source.



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

Figure 13 : Merging techniques

If we draw the truth table with these three factors (d: distance, s: slope, s: 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 separates sources from 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 followed paths are different for the two streams.

The question now is: does one of these five situations necessarily appear when merging. If we are dealing with two $\prec -stream$, this problem does not arise because we are forced to generate a new label for identified wells (line 1, ***Algo. 20***). Also, if one of the two streams includes an $\prec -stream$, it means there exist a label already generated that we can assign to the common wells (line 5, ***Algo. 20***). Finally, if both streams include $\prec -stream$ 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 gray level of a pixel represent its altitude. Slope and distance between sources and wells can be computed trough pixels coordinate. According to fixed conditions, roughness coefficient is equal to one.

Algorithm 20 : Function f-labeling [Mahmoudi and Akil]

Input: (L_a, lab_a, L_b, lab_b, nb_labs)

1. **// L_a AND L_b ARE two $\prec -stream$**
2. if (laba = -1) && (labb = -1) then
3. nb_labs ++ ;
4. **Attrib_lab**(L_a,L_b,nb_lab) ;
5. **// L_a OR L_b INCLUDES an $\prec -stream$ already labeled**

```
6.  else if ((lab_a ≠ -1) && (lab_b = -1))
7.          Attrib_lab(L_a,L_b,lab_a) ;
8.       else if ((lab_a = -1) && (lab_b ≠ -1))
9.             Attrib_lab(L_a,L_b,lab_b) ;
10.           // L_a AND L_b INCLUDE two ≺ −stream already labeled
11.           else if (AVspeed(L_a)> AVspeed(L_b))
12.                  Attrib_lab(L_a,L_b,lab_a) ;
13.                else Attrib_lab(L_a,L_b,lab_b) ;
14. Return NULL

15. Function Attrib_lab(L_1,L_2,lab) :
16.     foreach (z ∈ {L_1 ∩ L_2}) do Ψ(z) ← lab ;
17.     foreach (x ∈ L_1 ) such as (Ψ(x) = = No-Label) do Ψ(x) ←lab;
18.     foreach (y ∈ L_2 ) such as (Ψ(y) = = No-Label) do Ψ(y) ←lab;
19. Return Null
```

3.4 Performance testing

In this section we present an overall assessment of the parallel watershed operator. We begin by presenting test conditions. Then, obtained results in terms of execution time and cache use are presented and discussed. Based on these results, we compute efficiency and scalability of our implementation. We enhance discussion on scalability by computing the amount of work required to reach the average speed. Unfortunately, portability of our application will not be assessed for purely technical reasons.

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	7.928	29.249	193.950	466.478	1.614.014
Empty intersection	7.901	28.955	192.612	463.997	1.096.307
Full intersection	27	294	1.338	2.481	6.139

Table 4 : Tested Image [parallel watershed]

For profiling we used a microscopic view of a cross-section of a uranium oxide ceramics (see ***fig. 11.a***). To choose the right size, we compared number of streams intersections during merging step for each image. Obtained results, see ***table 4***, show that cut-size (***640*640***) is the most appropriate for profiling. Indeed, for cuts with less size, number of full intersection (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, total intersection number is very high which may cause much confusion when profiling cache. (Determinate instructions number)

		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

Table 5 : Used processors features [parallel watershed alg.]

Wall-clock execution times for numbers of threads equal to 1, 2, 4, 8, 16 and 32 were determined. Different characteristics of used architectures are presented in table 5. The minimum value of 2 timings was taken as most indicative of algorithm speed. Results of implementation on the different architecture are shown in the following table.

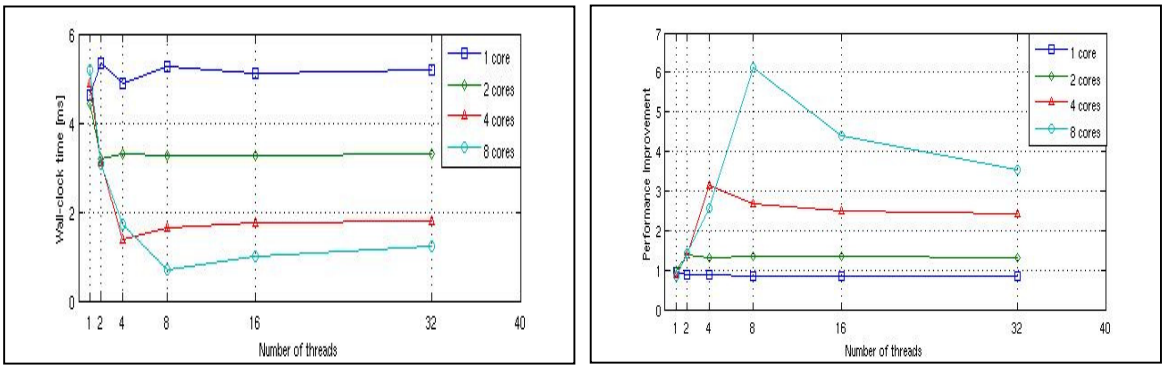
	1 CPU	2 CPUs	4 CPUs	8 CPUs
1 Thread	**4638**	4448	4898	5190
2 Threads	5321	**3182**	3114	3092
4 Threads	4898	3303	1384	1709
8 Threads	5253	3253	1639	**713**
16 Threads	5129	3278	1744	990
32 Threads	5190	3303	1794	1235

Table 6 : wall clock (ms) – [parallel watershed Alg.]

We note that 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 (definition 11 chapter 2). A remarkable result about speedup is also shown in table 4. 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 architecture is stability of execution time on each n-core machine since the code uses n or more threads. For better illustration we establish execution time and speedup curve (see ***fig. 14***).

	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

Table 7 : Performance improvement [parallel watershed Alg.]



(a) (b)

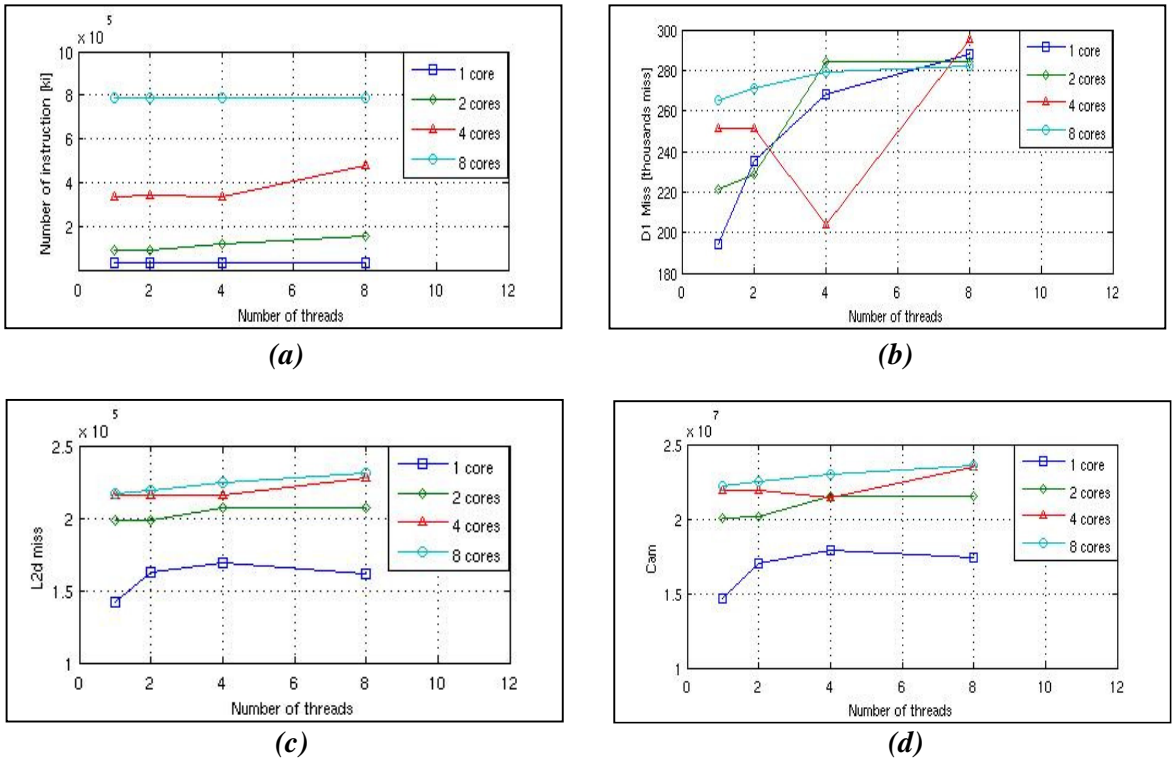
Figure 14 : (a) execution time (b) performance improvement [parallel watershed Algo.]

In the following we present our experimental analysis. We consider a commonly used Intel processor configuration. Number of processor varies from one to eight. The frequency varies between 1,73 GHz and 3,4 GHz as shown in ***table 5***. The ***L1*** caches have at least a 32-byte block size, while capacity vary between 16 Kbytes and 32 Kbytes, and for the associativity, only eight ways is considered. The ***L2*** caches have at least a 64-byte block size, while capacities vary between 512 Kbytes and 8 Mbytes, and the associativity varies between two and sixteen ways.

As a result of this experiment, ***fig. 15 (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 performance, as more processors are utilized. This area is generally identified as cache-efficiency zone. Balanced workloads offer higher locality and better exploit the cache and hence expand the cache efficiency zone to the right and up. An outstanding example is given by the following table which summarizes number of instruction, ***L1*** and ***L2*** data misses on four architectures using SMP scheduling policy. We note that number of instruction increase from an average of ($34x10^6$) instr. on 1 ***CPU*** to ($790x10^6$) instr. on 8 ***CPUs***.

To highlight cache performance, we compute wait status which refers to the delay experienced by processor when accessing external ***L2*** caches each time that information is missing in ***L1***. Since ***L1miss*** is followed either by an ***L2hit*** (success) or ***L2miss***, wait status can be computed by following 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). $WS_{cm} = ((D1miss - L2Dmiss) * 10) + (L2dmiss * 100)$. To estimate lost time during memory access, we simply multiply the wait status by P4 660 frequency (3.6 GHz) and E5405 frequency (2 GHz). Thus we realize that estimated lost time on 8CPU is insignificant compared to lost time on 1CPU. This result is very visible on the ***E8400*** and ***E5335*** architectures. For the ***E5405*** architecture, result is less visible due to cache structure: While ***E5405*** is considered as eight CPUs architectural, but physically they are two ***Quads*** on the same chip (***L2*** = 2x4Mb).



***(a)* Number of instruction *(b)* Data 1 Miss *(c)* L2d miss *(d)* Evaluation of wait status**

Figure 15 : Cache profiling [parallel watershed]

		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

Table 8 : Cache profiling [parallel watershed]

In the sequel, we turn to efficiency evaluation, using ***def. 13*** *(with ts = 360ms)*, in order to describe exploitation degree of each processor in used SMP machines. As introduced in [80], this profiling will highlight limitations introduced by parallel watershed implementation on SMP machines. Indeed, the efficiency decreases of 30% when switching from mono core architecture to dual cores architecture. Despite a slight increase on quad cores architecture, the efficiency is 20% lower than that measured with 1Cpus. More details are shown in table 9, see also ***fig. 16***.

	Intel P4- 660	Intel Dual C. E8400	Intel C2 Quad E5335	Intel Xeon E5405
EFFICIENCY RATE	0,94	0,69	0,79	0,76

Table 9 : Efficiency improvement [parallel watershed Algo.]

Causes for losses of efficiency can be explained by the following reasons, partially introduced in [84] 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 is not located in their local data stores. ***(iii)*** Set-up delays due to the set-up of control and processing logic and the network for inter-PE communication.

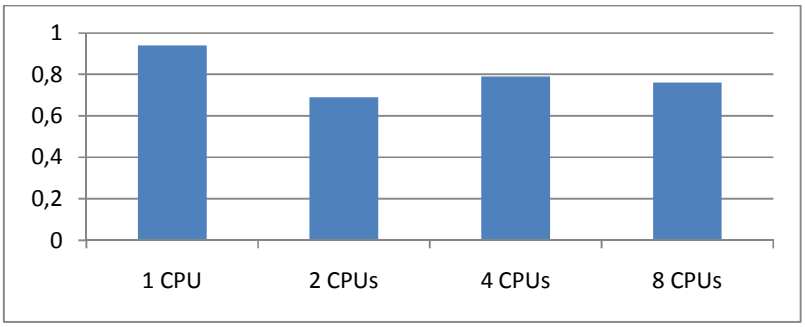


Figure 16 : Efficiency improvement [parallel watershed algo.]

In further evaluation, we extend speedup profiling of parallel watershed computing into scalability analysis. According to Intel theoric study [81], very high scalability can be achieved on multicore architecture. By way of example, dual-core architecture offers a scalability of roughly ***80%*** for the second processor, depending on the OS, application, compiler, and other factors. That means the first processor may deliver ***100%*** of its processing power, but the second processor typically suffers 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 mono-core architecture, but instead 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 ration between achieved speedup and the number of processor, will be very useful to determinate scalability. We can also extend this definition into the maximum average speed which is defined as ratio of maximum achieved speedup by number of processor. $Max(Av_{us}) = \frac{\max(A_{speed})}{(N)}$. Obtained results are presented in the following table:

	Intel P4- 660	Intel Dual C. E8400	Intel C2 Quad E5335	Intel Xeon E5405
1 Thread	0,940	-	-	-
2 Threads	-	0,685	-	-
4 Threads	-	-	0,787	-
8 Threads	-	-	-	0,763
16 Threads	-	-	-	-
32 Threads	-	-	-	-

Table 10 : Maximum Average Unit Speed [parallel watershed]

Concerning scalability, see ***Def. 9***, it can be written as $\psi(N,N') = \frac{N' \times W}{N \times W'}$. In this formula, W refers to amount of work of our algorithm when N processors are employed and W' refers to amount of work of our algorithm when N' processors are employed to maintain the average speed. In ideal situation, W' is equivalent to $\frac{N' \times W}{N}$ thus $\psi(N,N')$ be equal to 1. Unfortunately, this never happen in real situation, actually $W' > \frac{N' \times W}{N}$, thus $\psi(N,N') < 1$.

To calculate different values of efficiency foreach architecture, we must first determine the necessary amount of work W', as shown in ***Table 11***, to reach Average Speed Unit Av_{us}. Note that chosen Average Unit Speed is 0,787 on 4 CPUs using 4 Threads (Associated W = 334.020.732).

		1 CPU		2 CPUs	
1 Thread	Av_{us}	0,940	0,787	-	
	Work (W')	34.598.772	28.967.270		
2 Threads	Av_{us}	-		0,685	0,787
	Work (W')			90.875.188	104.406.968
4 Threads	Av_{us}	-		-	
	Work (W')				
8 Threads	Av_{us}	-		-	
	Work (W')				

		4 CPUs	8 CPUs	
1 Thread	Av_{us}	-	-	
	Work (W')			
2 Threads	Av_{us}	-	-	
	Work (W')			
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

Table 11 : Average speed Unit [parallel watershed Algo.]

The scalability results for parallel watershed processing are shown in ***table 12***. Our experiments demonstrate a very good scalability across all tested architectures.

	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

Table 12 : Scalability profiling [parallel watershed Algo.]

As the number of thread increase, a linear speedup has been observed (see ***fig. 17***). Also, the speedup improves as the problem sizes increases. Note that when number of thread exceed number of cores, total execution time dramatically reduces. The difference between each efficiency curve with the ideal curve (constant efficiency equal to 1) decreases as the number of thread increases.

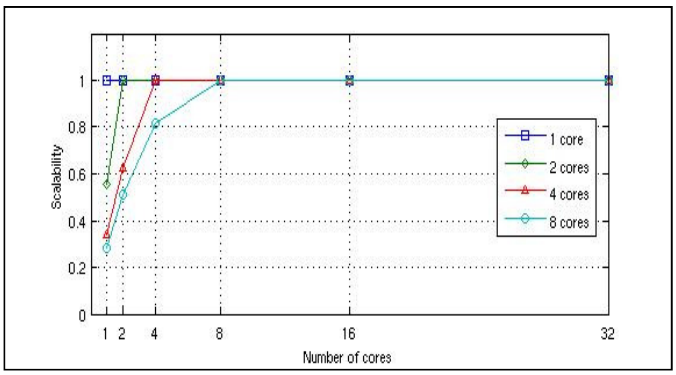


Figure 17 : Performance improvement for parallel watershed

3.5 Conclusion

In this chapter, we have presented an intensive study of all existing watershed transform in the discrete case: WT based on flooding, WT based on path-cost minimization, watershed based on topology preservation, WT based on local condition and WT based on minimum spanning forest.

First contribution is the global nature of the proposed study. In fact, for each approach, we give informal definition, then we presented processing procedure followed by mathematical foundations and the algorithm of reference. Recent publications based on some approach are also presented and discussed.

Second contribution concerns classification of watershed algorithms according to criteria of recursion, complexity, basins computing and topology preservation.

Third contribution concern a new algorithm to compute watershed that is parallel, preserves the topology of the input image, does not need prior minima extraction and suited for SMP machines. This algorithm does not require any sorting step, or the use of any hierarchical queue. A global description of the computing process is given by ***fig. 18***. Through this illustration, we show links between parallel watershed-cut and the SD&M strategy application. In fact, splitting step is applied directly on input graph when selecting sources. Unlike conventional technique of division such as pixel division, or block division, the source selection is completely random. Associated steam computing is fully parallel (read mode data accesses). Then distribution depends only on the available processors. This flexibility in data manipulation allowed us to obtain very good results especially in terms of efficiency ***fig. 14 (b)*** without using the 'Basic-NPS' scheduler. Finally, the merging step contains procedures of s-labeling and f-labeling. Through these two functions, we have remained confident in our approach for merging streams

two by two (algorithm already introduced in section ***2.4.3***. Experimental analyzes such as execution time, performance enhancement, cache consumption, efficiency and scalability are also presented and discussed.

Note that our algorithm can't be applied directly over 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 [82, 83].

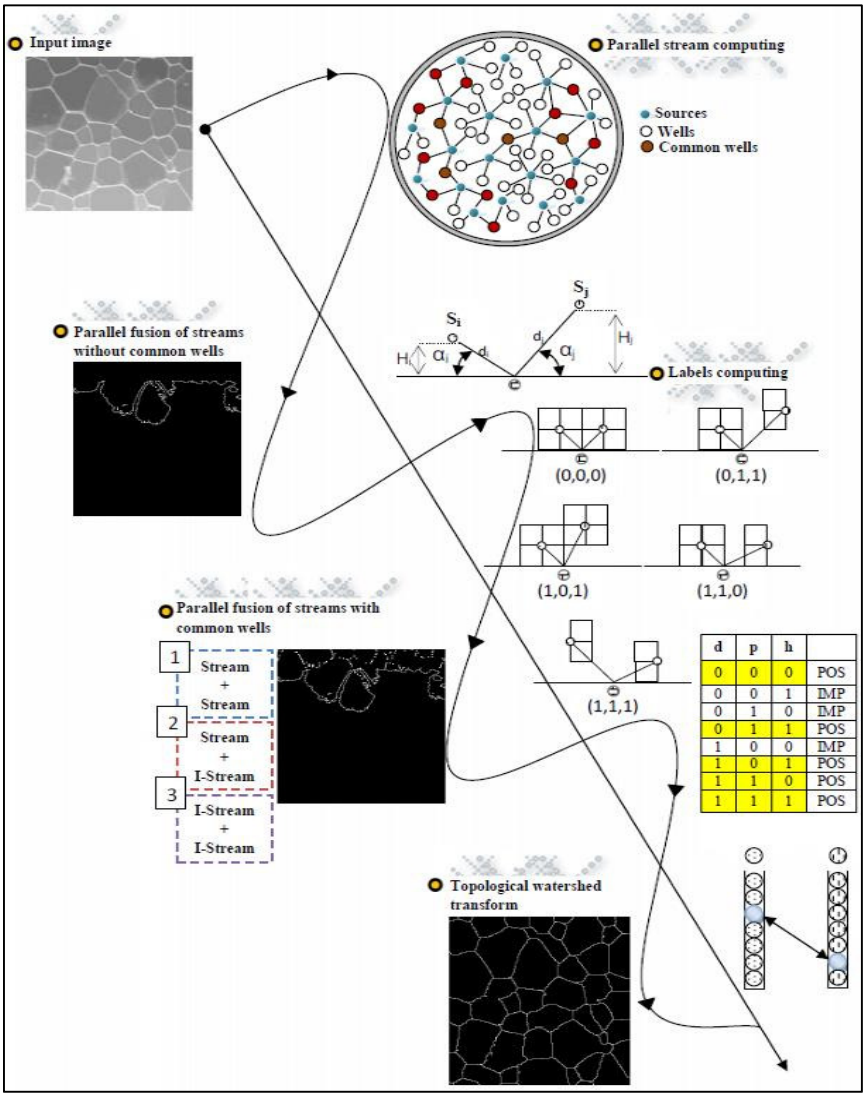


Figure 18 : Segmentation chain based on parallel watersheds-cut

TOPOLOGICAL THINNING

In many computer vision applications, standard techniques of pattern recognition are thinning algorithms. As a preprocessing stage, these algorithms have been used for the recognition of handwriting or printed characters, fingerprints, chromosomes and biological cell structures, etc. [84]. Topological thinning and skeletonization are ones of the most cardinal operators for this kind of preprocessing. In literature, several 2D parallel thinning methods can be found, see [85, 86, 87, 88, 90]. Proving that such an algorithm always preserve topology is not an easy task, even in 2D. We say that an algorithm preserves topology if obtained skeleton through thinning method has the necessary information to reconstruct the original image. The proofs found in the literature are often combinatorial and hardly extendable to 3D, a fortiori to higher dimensions.

Couprie [89] present a study of fifteen parallel thinning algorithms, see ***table 13***, based on the framework of critical kernels. He proves that ten among these fifteen algorithms indeed guarantee topology preservation, and give counter-examples for the five other ones. He also investigates, for some of these algorithms, the relation between the medial axis and the obtained homotopic skeleton.

			Topology	
			Preserved	***Not preserved***
1	D. Rutovitz [91]	1966		■
2	T. Pavlidis [87, 88]	1981	■	
3	R.T. Chin, H.K. Wan, D.L. Stover and R.D. Iverson [84]	1987	■	
4	C.M. Holt, A. Stewart, M. Clint and R.D. Perrott [89]	1987	■	
5	Y.Y. Zhang and P.S.P. Wang [94]	1988		■
6	R.W. Hall [85]	1989	■	
7	R.Y. Wu and W.H. Tsai [95]	1992		■
8	Z. Guo and R.W. Hall [90] (first version)	1992	■	
9	Z. Guo and R.W. Hall [90] (Second version)	1992	■	
10	Z. Guo and R.W. Hall [90] (Third version)	1992	■	
11	B.K. Jang and R.T. Chin [97]	1992		■
12	B.K. Jang and R.T. Chin [91]	1993	■	
13	U. Eckhardt and G. Maderlechner [92]	1993	■	
14	S.S.O. Choy, C.S.T. Choy and W.C. Siu [100]	1995		■
15	T. Bernard and A. Manzanera [83]	1999	■	
16	M. Couprie, F. N. Bezerra and G Bertrand [8]	2001	■	

Table 13 : Classification of thinning algorithm according to topology preservation

According to the above study, algorithms proposed by Pavlidis in 1981 [90, 92], by Chin and al. in 1987 [87], by Holt and al. in 1987 [93], by Hall in 1989 [88], by Guo and Hall in 1992 [96] (3 variants), by Jang and Chin in 1993 [98], by Eckhardt and Maderlechner in 1993 [99], and by Bernard and Manzanera in 1999 [86] preserve topology. However, algorithms proposed by Rutovitz [91], by Zhang and Wang [94], by Wu and Tsai [95], by Jang and Chin [97] and by Choy and al. [100] produce a Skeleton that don't allows to reconstruct the original image. We propose to extend this study by adding a new algorithm proposed by Couprie and al. [8]. This algorithm is also able to act directly over grayscale image without modifying topology.

Based on this first '***classification***' of 2D thinning algorithm, we go further in this study and we start by proposing new classification, in section ***4.2***, that takes into account not only the criterion of topology preservation but also the proximity of the obtained skeleton from the medial axis, necessary execution time and cache consumption. Since suited algorithm for parallelization is identified and convinced that best performance can be achieved, we propose a concurrent implementation of a powerful topological thinning algorithm that is originally introduced by Couprie [8]. In section ***4.3***, we start by introducing theoric background, we give also some illustration of original algorithm before presenting parallel lambda skeleton procedure. Based on ***SD&M*** strategy, Distributed work during thinning process is done by a variable number of threads. Tests on ***2D*** grayscale image (***512x512***), using shared memory parallel machine (***SMPM***) with ***8 CPUs*** cores (***2 × Xeon E5405*** running at frequency of ***2 GHz***), showed an enhancement of ***6.2*** with a maximum achieved cadency of ***125 images/s*** using 8 threads.

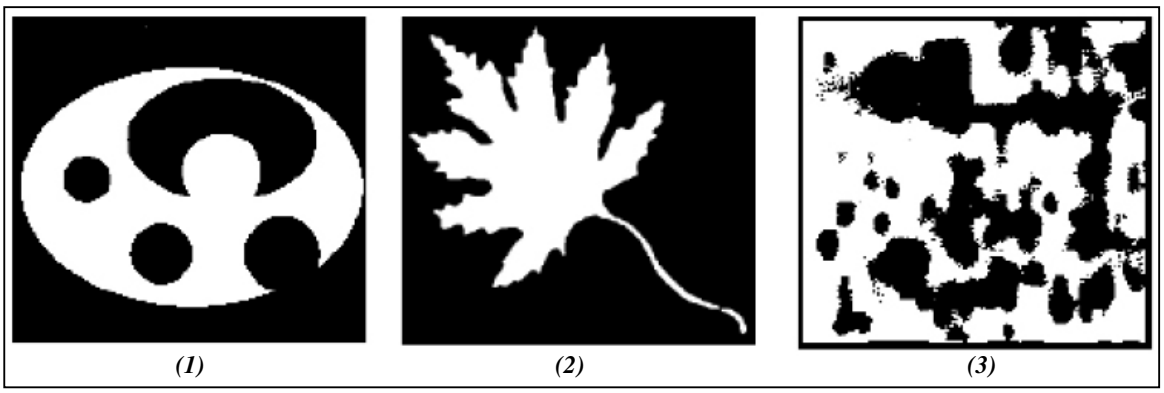


Figure 19: Used shapes for thinning algorithms comparison [89]

4.1 Classification of thinning algorithms

In order to make a quantitative evaluation of selected algorithms, we start by presenting two selection criteria based on Jang and Chin's study [101]. Authors propose to compute ratio between the number of pixels contained in the maximal disks (obtained from the skeleton) and the effective number of pixels in the original image. This measure, see ***(Def. 31)***, is used to determine the proximity of the skeleton from the medial axis as proof of skeletal connectivity and convergence. $Area()$ function is a pixel's counter, S' refers to the number of pixels contained in the maximal disks and S refers to the effective number of pixels (in the original image).

Definition (31): *Proximity of the skeleton from medial axis (M_m)*

$$M_m = \frac{Area(S')}{Area(S)}$$

Inspired by Jang and al. study, we adopt this criterion of skeleton proximity from medial axis with some modification, see ***(Def.32)***. Actually, we privilege the ratio between number of pixels of the skeleton which belong to the medial axis ***(Ai)*** and number of pixels in the skeleton ***(Ni)***. Our goal is to identify the algorithm that returns the most centered skeleton. Thus, most faithful

algorithm is the one whose ***C1*** value is closest to 1. An initial assessment, see ***table 14***, using three different images shown in ***fig.19***, has allowed us to draw first curve (see ***fig. 20***).

Definition (32): *Skeleton connectivity criteria* (C_1)

$$C_1 = \frac{Area(A_i)}{Area(N_i)}$$

		Shape (1)		*Shape (2)*		*Shape (3)*	
		N_1	A_1	N_2	A_2	N_3	A_3
1	T. Pavlidis [87, 88]	847	564	2829	1359	4241	2172
2	R.T. Chin, H.K. Wan, D.L. Stover and R.D. Iverson [84]	544	153	1572	334	3057	778
3	C.M. Holt, A. Stewart, M. Clint and R.D. Perrott [89]	590	466	1713	1079	2780	1444
4	R.W. Hall [85]	591	467	1773	1103	3060	1557
5	Z. Guo and R.W. Hall [90] (first version)	658	484	1993	1122	3508	1903
6	Z. Guo and R.W. Hall [90] (Second version)	591	468	1775	1104	3264	1863
7	Z. Guo and R.W. Hall [90] (Third version)	560	437	1664	993	3149	1750
8	B.K. Jang and R.T. Chin [91]	704	564	2394	1359	3787	2178
9	U. Eckhardt and G. Maderlechner [92]	724	564	2434	1359	3895	2171
10	T. Bernard and A. Manzanera [83]	678	534	1929	1219	3528	2018
11	M. Couprie [8]	707	545	1882	1114	3831	2069

Table 14 : Evaluation of pixels' Skeleton (A_i) **and** (N_i) **[89]**

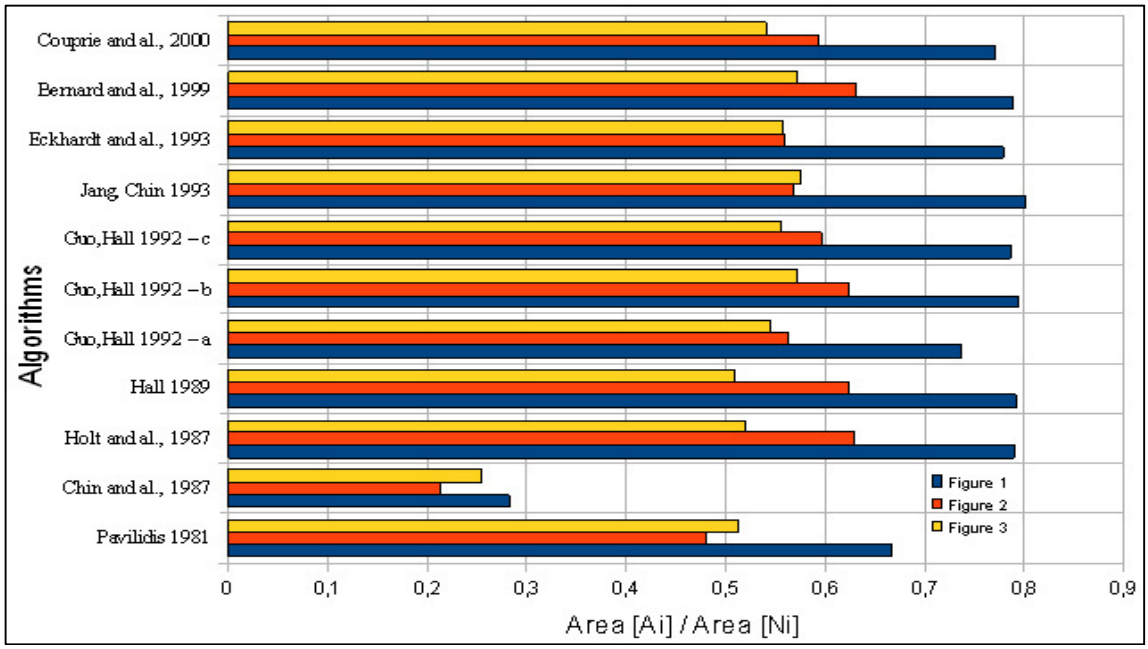


Figure 20: Algorithm classification according to Skeleton connectivity criteria (C_1)

Unfortunately this quantitative measure is not sufficient to evaluate algorithms. Actually, even if ***(C1)*** tends to 1, the gap between the number of skeleton's pixels belonging to medial axis and the number of reference pixels of medial axis can be great. For example, let's consider two algorithms. The first one provides ***50*** pixels belonging to medial axis, the total number of skeleton's pixels is also ***50*** and the number of reference pixels is ***100***. The second algorithm provides respectively ***100***, ***150*** and ***100***. Thus, ***(C1)*** for the first algorithm is equal to 1 even if medial axis doesn't include many points from the medial reference axis; therefore, the deviation

is wide ***0.5***. For the second algorithm, ***(C1)*** is over ***1*** even if all pixels of the median reference axis are included in reference median axis. To resolve this problem, we propose a second criterion ***(C2)*** to identify how mush generated Skeleton is symmetric and close to reference medial axis, see ***Def. 33***. Thus we compute the ratio between number of pixels of the skeleton which belong to the medial axis ***(Ai)*** and number of pixels of reference medial axis ***(R_i)***.

Definition (33): *Skeleton symmetry criteria (C_2)*

$$C_2 = \frac{Area(A_i)}{Area(R_i)}$$

A second assessment using same images showed in ***fig.19***, has allowed us to draw second curve (see ***fig. 21***). Measurements are presented in ***table 14*** and ***table 15***.

	Shape (1)	***Shape (2)***	***Shape (3)***
Medial axis (reference)	564	1359	2178

Table 15 : Evaluation of Medial axis (reference) pixels (R_i) [89]

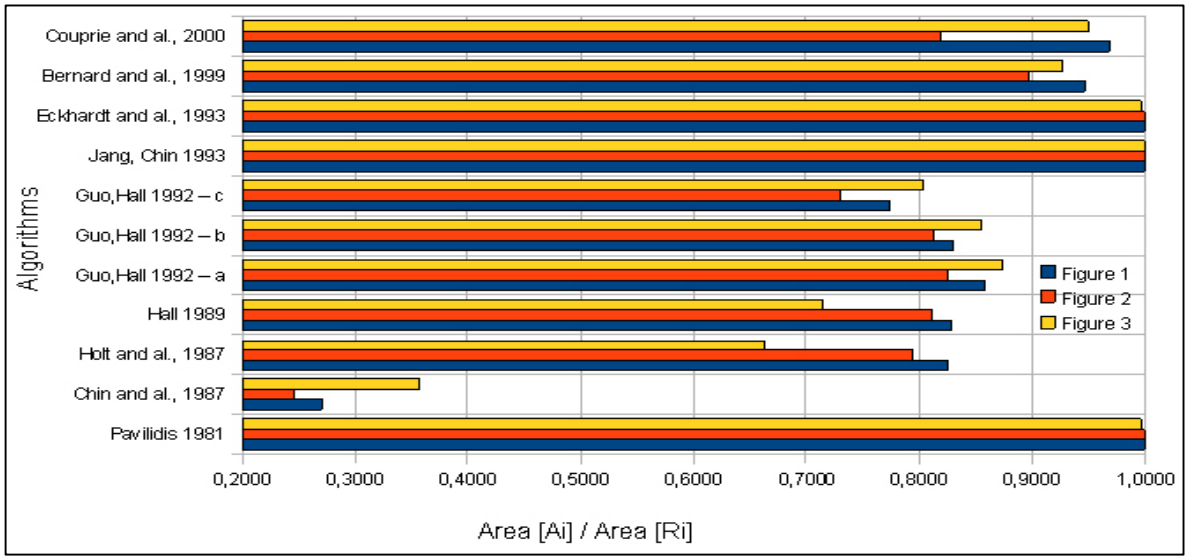


Figure 21: Algorithm classification according to Skeleton symmetry criteria *(C_2)*

1	Bernard, Manzanera 1999
2	Jang, Chin 1993
3	Couprie and al. 2000
4	Eckhardt, Maderlechner 1993
5	Guo, Hall 1992 (b)

Table 16 : Top five thinning algorithm according to Skeleton connectivity and symmetry

According to ***fig. 20*** and ***fig. 21***, first algorithm proposed by Pavlidis [87] has a very good factor ***(C_2)*** except that it has a bad factor ***(C_1)*** which doesn't exceed ***0.7***. Proposed algorithm by Chin [84] has the worst classification according to two criteria so it can be automatically rejected. Although, Holt [89] and Hall [85] algorithms have similar factors, Couprie [86] and Jang [91] shows that Hall algorithm is an enhanced version of Holt algorithm, therefore we chose it for assessment. The three versions of Guo [90] algorithm have similar results but second version

(Guo, Hall 1992 - b) has a better classification in the two graphs thus this version will be also selected. Jang [91], Eckhardt [92] and Bernard [83] have also a very good classification in both graphs and can be kept. Finally we can conclude by the upper comparison chart that ranks algorithm by qualitative descending factor [***(C1) AND (C2)***].

In the following, we conduct an initial evaluation of selected algorithms. Tested images are those shown in ***fig. 19 (1) (2) (3)***. Respective sizes are ***(800x600)***, ***(755x755)***, and ***(755x755)***. We turn sequential algorithms on mono core architecture ***Intel Pentium 4***, more details about used ***P4*** processor are given in the following table.

		Intel P4	Intel Dual C. T1400	Intel Quad Q9550	Intel Xeon E5405
Number of processor		1	2	4	2 x 4
SMT		Yes	Yes	Yes	Yes
Frequency		3,40 GHz	1,73 GHz	2,83 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	512 kb	6Mb	2 x 6Mb
	Asso.	8-way	2-way	24-way	24-way
	Block size	64byte	64byte	64byte	64byte
RAM size		1Gb	2Gb	2Gb	8Gb
Linux version		2.6.21	2.6.3	2.6.29	2.6.18
Valgrind version		3.2.3	3.4.1	3.4.1	3.4.1

Table 17 : Used processors features [Thinning alg.]

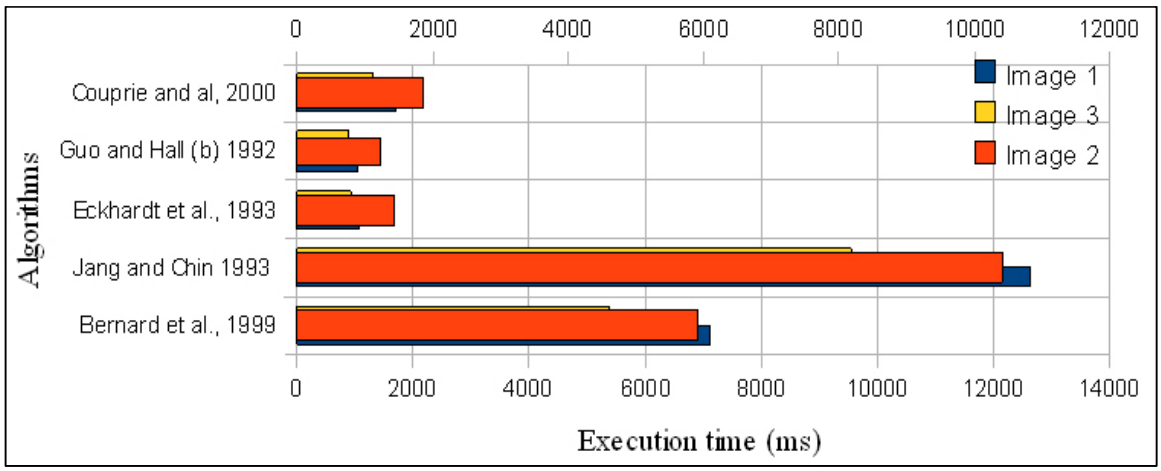
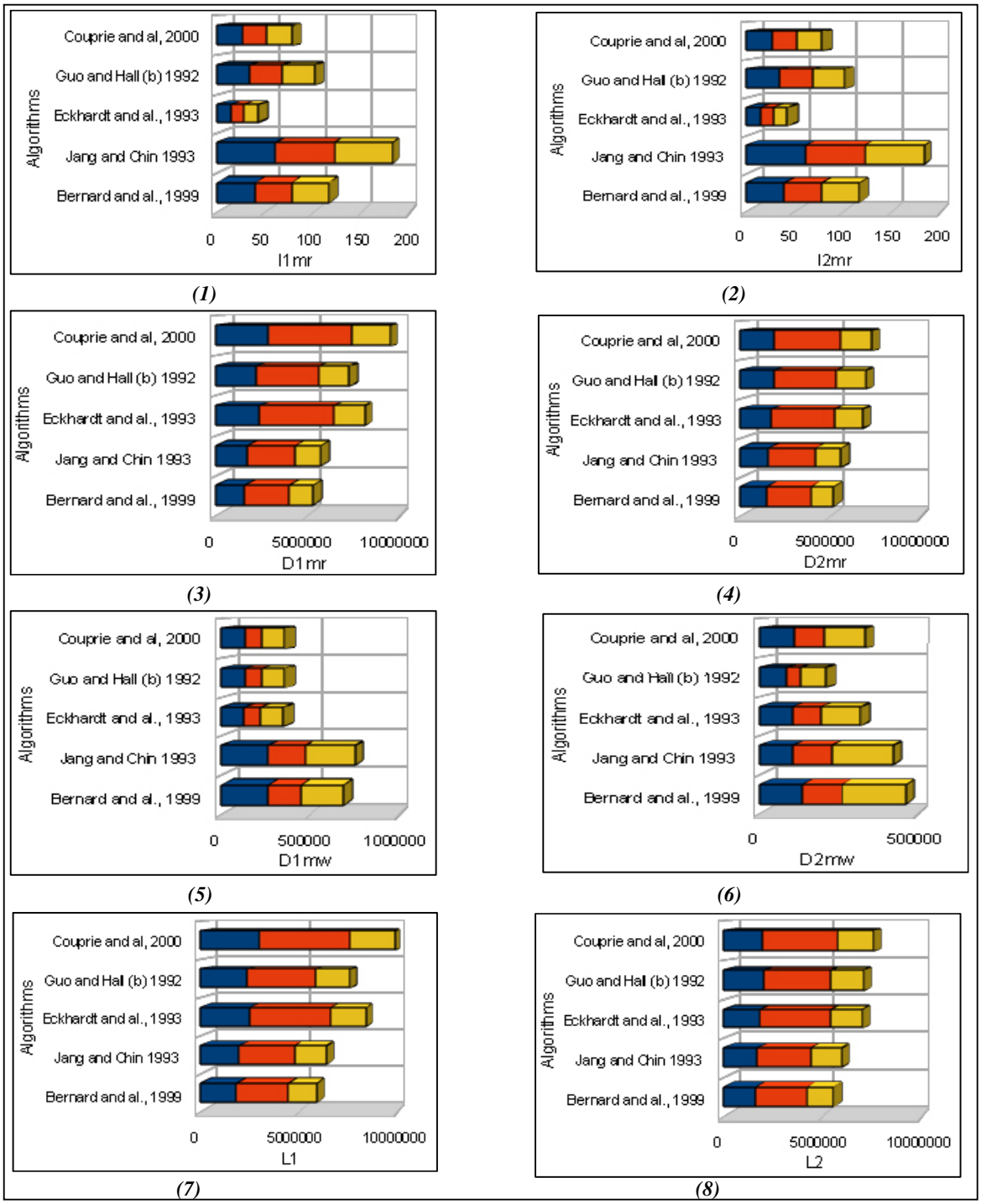


Figure 22: Execution time - serial versions on monocore machine [Thinning Algo.]

Obtained results, see ***fig. 22***, show that Guo and Hall algorithm is the most efficient in terms of execution time. In fact, algorithms of Couprie, Guo and Eckhardt can be grouped together because they display almost same performance for all tested images. Jans and Chin algorithm is the most expensive. It consumes six times more than Guo algorithm. Concerning cache consumption, see ***fig. 23***) results are different. In fact, Jang and Bernard algorithms show better performance despite significant loss of instructions at L1and L2 levels (High execution time).

We note good data management when reading or writing. Algorithms that suffer from some problems of data management are rather those of Couprie, Guo and Eckhardt. These results are most visible especially when computing cache consumption for image 2.



(1) L1 Instruction miss *(2)* L2 Instruction miss *(3)* L1 Data read miss *(4)* L2 Data read miss *(5)* L1 Data write miss *(6)* L2 Data write miss *(7)* Total L1 cache miss *(8)* Total L2 cache miss, (■ Image 1 ■ Image 2 ■ Image 3)

Figure 23: Cache profiling of serial versions on monocore machine [Thinning Algo.]

If we combine implementation features of these algorithms (execution time and cache use) with classification criteria previously presented ***(C_1 && C_2)***, we can draw following classification on ***table 18***. Even if Guo and Hall algorithm has best performance, first three algorithms can be classified in the same family. In fact, they have many similarities and close performances. Second family is composed of two algorithms, Jang and Bernard. Despite their low cost of cache consumption, execution times are quite high. Based on Couprie study, we will develop more all of these five algorithms to identify the most suitable one for parallel implementation on ***SMPM***. For the sake of simplicity, we restrict ourselves to the minimal set of notions that will be useful for our purpose.

1	Guo, Hall 1992 (b)
2	Eckhardt, Maderlechner 1993
3	Couprie and al. 2000
4	Jang, Chin 1993
5	Bernard, Manzanera 1999

Table 18 : Classification of thinning algorithm

Guo and Hall, see ***(Algo. 21)***, introduced the notion of delatable pixel x_{GH} . Based on Boolean expressions defined bellow, they classify a pixel as delatable if some conditions are hold: Let $x_{GH} \in G^2$, let $X \subset G^2$,

The pixel(x_{GH}) is said delatable if and only if: ***(i)*** D(x_{GH})=1, ***(ii)*** G(x_{GH})=0, ***(iii)*** B(x_{GH})>2 and ***(iv)*** Neighborhood of (x_{GH}) does not match any the following masks.

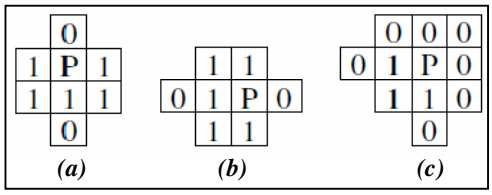


Figure 24: Guo and Hall masks

Algorithm 21 : Thinning algorithm – Second version [Guo and Hall]

Input: set X
Output: set X

1. **repeat**
2. $Y \leftarrow$ set of pixels X which are GH-deletable
3. $X \leftarrow X \setminus Y$
4. **Until** $Y = \phi$

Eckhardt and Maderlechner, see ***(Algo. 22)***, introduced the notion of simple and perfect pixels. First, they establish the following definitions: (i) ***Interior pixel***: it is a pixel in X having all its four strong neighbors in X ***(ii) Boundary pixel***: it is a pixel in X which is not ***interior pixel***. ***(iii) Inner boundary pixel***: it is a boundary pixel which has an interior pixel as strong neighbor.

Thus, a pixel (p) in X can be classified as ***simple***, if it is a ***boundary pixel*** and if there exist exactly one strong connected component of pixels of X in the neighborhood of (p) which is strongly connected to (p).

Finally, ***inner boundary pixel*** (p) is classified as ***perfect***, if there exists a strong neighbor $\Gamma_i(p)$ of (p) which is interior and such that $\Gamma_j(p) \notin X$ with $j = (i+4) \bmod 8$.

Algorithm 22 : Thinning algorithm [Eckhardt and Maderlechner]

Input: set X

Output: set X

1. **repeat**
2. $Y \leftarrow$ set of pixels in X witch are both simple and perfect
3. $X \leftarrow X \setminus Y$
4. **Until** $Y = \phi$

Couprie and al., see ***(Algo. 23)***, introduced new notions and operators in the frame work of cross-section topology. In particular, the notion of $\lambda - destructible$ pixels which allows to selectively simplify the topology, based on a local contrast parameter λ.

To achieve this simplification; they introduce the notion of ***λ-destructible*** point which is more flexible then the notion of destructible point. In fact, a point is said to be a ***λ-deletable*** point (for F), λ being a positive integer, if it is either a ***λ-destructible*** point, or a ***peak point*** such that $F(x) - \alpha^-(x, F) \leq \lambda$. We remind that a point x is said ***λ-destructible*** if it satisfies one of the two following conditions:

(i) x is ***destructible*** or x is ***k-divergent***.

(ii) At least ***k-1*** connected components c_i of $\Gamma^-(x, F)$ are such that $F(x) - F^-(c_i) \leq \lambda$, with $i = \{1, \dots, k-1\}$.

Let $X \subset Z^2$ and $x \in X$, x is an end point (for X) if $\#(\Gamma_n^*(x) \cap X)=1$. Let $F \subset \varphi$ and $x \in Z^2$, x is an end point (for F) if it is an end point for the set F_k with $k = F(x)$. A point is said to be ***λ-end*** point (for F) if it is an end point for F and if: $F(x) - \alpha^-(x, F) > \lambda$.

Algorithm 23 : Thinning algorithm [Couprie, Bezerra and Bertrand]

Input: $F \in \varphi$,

Output: F

1. **repeat until stability**
2. Among all points which are $\lambda - deletable$ and $\lambda - end$
3. Select a point x of minimal value
4. $F(x) = \alpha^-(x, F)$

Jan and chin, see ***(Algo. 24)***, use the medial axis (of the input object) when computing Skeleton to enhance connectivity, unit-width convergence, medial axis approximation, noise immunity, and efficiency.

Let $X \subset G^2$, let $x \in X$, $r \in \mathbb{N}$, the ball $B_4(x,r)$ is maximum for X if $B_4(x,r) \subseteq X$ and if there is no other ball included in X which contain $B_4(x,r)$. Thus, medial axis of X is the set of the centers of all the maximal balls for X. Jan and Chin masks are showed in ***fig. 25***.

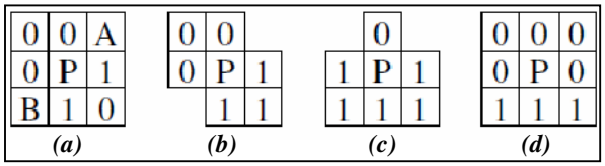


Figure 25: Jang and Chin masks

Algorithm 24 : Thinning algorithm [Jan and Chin]

Input: set X

Output: set X

1. $A \leftarrow$ medial axis of X
2. **repeat**
3. $Y \leftarrow$ set of pixels in X which match Jang and Chin masks
4. $Y \leftarrow Y \setminus A$
5. $X \leftarrow X \setminus Y$
6. **Until** $Y = \phi$

Bernard and Manzanera, see ***(Algo. 25)***, introduce a parallel iterative thinning procedure that respect homotopy, mediality, thickness, rotation invariance and noise immunity. Used masks by thinning procedure are presented in the following:

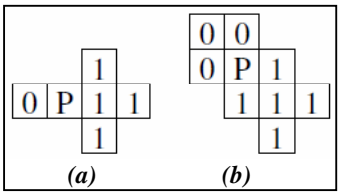


Figure 26: Bernard and Manzanera masks

Algorithm 25 : Thinning algorithm [Bernard and Manzanera]

Input: set X

Output: set X

1. **repeat**
2. $Y \leftarrow$ set of pixels in X which match Bernard and Manzanera masks
3. $X \leftarrow X \setminus Y$
4. **Until** $Y = \phi$

Algorithmic structures, previously presented, are very similar. An iterative process is always launched until stability. In each iteration, some pixels are selected according to a set of criteria then output image is updated. The only algorithm that requires pre-treatment is Jang and Chin one (pixels of the medial axis are calculated). Unlike other algorithms, Couprie procedure doesn't exempt selected pixels from the output image but it changes their values. This will allow greater independence in the data processing if we ever plan to launch a parallel processing. For this reason, we selected the latter algorithm to propose a parallel version suitable for (***SMPM***).

4.2 Parallel lambda-skeleton algorithms

In this section, some basic notions of topological operators are first summarized; Illustration of the original algorithm is also introduced. Then, new adapted version of thinning is introduced. Experimental analyzes results of different implementations are also presented and discussed.

4.2.1 Theorical background

First, we recall some basic notions of grayscale images. A 2D grayscale image may be seen as a map F from Z^2 to Z. For each point $x \in Z^2$, $F(x)$ is the graylevel value of x. We denote by φ the set composed by all maps from Z^2 to Z. Let $F \in \varphi$, the section of F at the level k is the set F_k composed of all point $x \in Z^2$ such that $F_k \geq k$. As for the binary case, if we use the n-adjacency for the section F_k of F , we must use $\overline{n}$-adjacency for the section $\overline{F_k}$ with $(n, \overline{n})$ = ***(8,4)*** or ***(4,8)***.

We remind that for two points $x(x_1, x_2)$, $y(y_1, y_2) \in Z^2$, we consider that y is 4-adjacent to x if $|y_1 - x_1| + |y_2 - x_2| \leq 1$, and y is 8-adjacent to x if $\max(|y_1 - x_1|, |y_2 - x_2|) \leq 1$. In the following, we consider the two neighborhoods relations Γ_4 and Γ_8 defined by, for each point $x \in Z^2$,

$\Gamma_4(x) = \{y \in Z^2 \mid y \text{ is 4 adjacent to } x\}$,
$\Gamma_8(x) = \{y \in Z^2 \mid y \text{ is 8 adjacent to } x\}$.

For more general presentation, we will define $\Gamma_n^*(x) = \Gamma_n(x) \backslash \{x\}$. We will also denote by $\overline{F}$ the complementary map of F. We note that the complementary sets of the section of F are section of $\overline{F}$. In all the rest of this paragraph, we will note n=8 for the section of F , thus we must use $\overline{n}$=4 for $\overline{F}$. It is also important to mention that a non-empty connected component X of a section F_k of F is a (regional) maximum for F if $X \cap F_{k+1} = \emptyset$ and a set $X \subset Z^2$is a regional minimum for F if it is a regional maximum for $\overline{F}$. Let $F \in \varphi$, the point $x \in Z^2$ is destructible (for F) if x is a simple for F_k, with $k = F(x)$. We remind that a point x is said simple for $X \subset Z^2$ if $T(x, X) = 1$ and $\overline{T}(x, X) = 0$.

$T(x, X)$ and $\overline{T}(x, X)$ are the two connectivity numbers defined as follows ($\# X$stands for the cardinal of X): $T(x, X) = \#C_n[x, \Gamma_8^*(x) \cap X]$; $\overline{T}(x, X) = \#C_{\overline{n}}[x, \Gamma_8^*(x) \cap \overline{X}]$;

So we can define the four neighborhoods:

$$\Gamma^{++}(x, F) = \{y \in \Gamma_8^*(x); F(y) > F(x)\}$$
$$\Gamma^{+}(x, F) = \{y \in \Gamma_8^*(x); F(y) \geq F(x)\}$$
$$\Gamma^{--}(x, F) = \{y \in \Gamma_8^*(x); F(y) < F(x)\}$$
$$\alpha^{-}(x, F) = \begin{cases} max\{F(y), y \in \Gamma^{--}(x, F), if \Gamma^{--}(x, F) \neq \emptyset\} \\ F(x) \qquad otherwise \end{cases}$$

We define also some associated connectivity numbers:

(i)
(ii)
(iii)

Furthermore, the connectivity numbers allow the classification of the topological characteristics of a point:

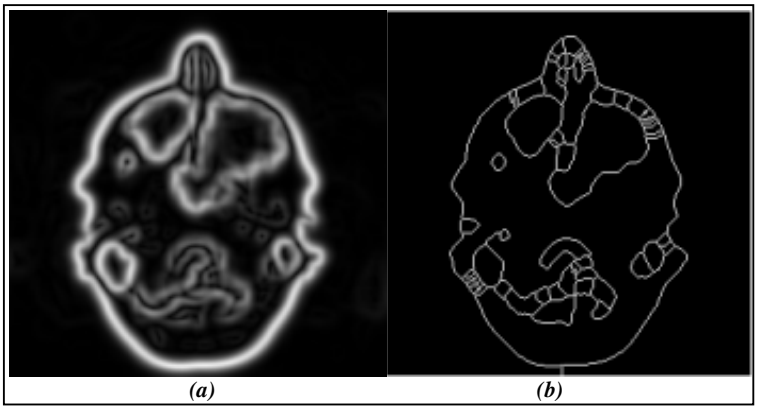
(i) is a peak point if .
(ii) is a -divergent if .

A point is said to be a -deletable point (for), being a positive integer, if it is either a -destructible point, or a peak point such that . We remind that a point is said -destructible if it satisfies one of the two following conditions: is destructible or is -divergent and at least k-1 connected components of ¯ are such that , with .

Let and , is an end point (for) if =1. Let and , is an end point (for) if it is an end point for the set with . A point is said to be -end point (for) if it is an end point for and if: .

4.2.2 Illustration of original algorithm

, we say that is a skeleton of if is obtained from by iteratively selecting a destructible and non-end point in and lowering it down to , until stability. In order to get a filtered skeleton, that is to eliminate non significant branches and regional minima, Bertrand and Couprie allow -deletable and not -end to be lowered. It is important to mention that each time that a pixel is lowered, its eight neighbors must be reexamined to be sure that topology is still preserved. In Figure 1, we illustrate this method on a gradient image ***(a)*** obtained from a 2D grayscale image of an MRI brain section by Deriche gradient operator. ***(b)*** is obtained by a filtered thinning with Full algorithm is already introduced in last section ***(Algo. 23)***.



(a) After Deriche gradient operator; ***(b)*** filtered skeleton with

Figure 27 : Filtered Skeleton illustration [Thinning alg.]

4.2.3 Parallel thinning algorithm

Now, we present a parallel version of the thinning according to strategy previously discussed. Let the map F from Z^2 to Z represent the input grayscale image. For each point $x \in Z^2$, $F(x)$ is the graylevel value of x. We denote by φ the set composed by all maps from Z^2 to Z.

Let $F \in \varphi$, the section of F at the level k is the set F_k composed of all point $x \in Z^2$ such that $F_k \geq k$.

Let *T* be the set of type sought in the characterization of pixels. For thinning algorithm: $T = \{\lambda-\text{deletable and not } \lambda-\text{end points}\}$. It is important to mention that points from *T* can also be end-point and isolate-point for crest restoring.

We will refer to global search space by *Ime*, and associated map (from Z^2 to Z) to each sub-space Ime_i is F_i. For each point $x \in Z^2$, $F_i(x)$ is the graylevel value of x in the search space Ime_i.

The following dynamically parallel λ–Skeleton algorithm (it is adapted for two concurrent threads, but it can be easily extended to N threads) starts by dividing the search space. m_{inf} and m_{sup} define sub-region bounds. Since the distributed work starts, each thread will lower each characterized pixel and then push its eight neighbors in E_{sn}. E_{sn} is the set of all selected neighbors and it is shared between only two threads. E_{sn} is the new defined set to explore since threads finished. Newly characterized pixels are pushed in a private set called E_{ki}. The pixel set assigned to the newly generated thread is nothing else than E_{sn} and the associated search space is $((Ime_i \cup Ime_{i+1}) \cup E_{ki} \cup E_{ki+1})$.

Algorithm 26 : Dynamically Parallel λ –Skeleton [Mahmoudi and Akil]

Input : ***m***:colums, ***n***:lines, ***b***:image

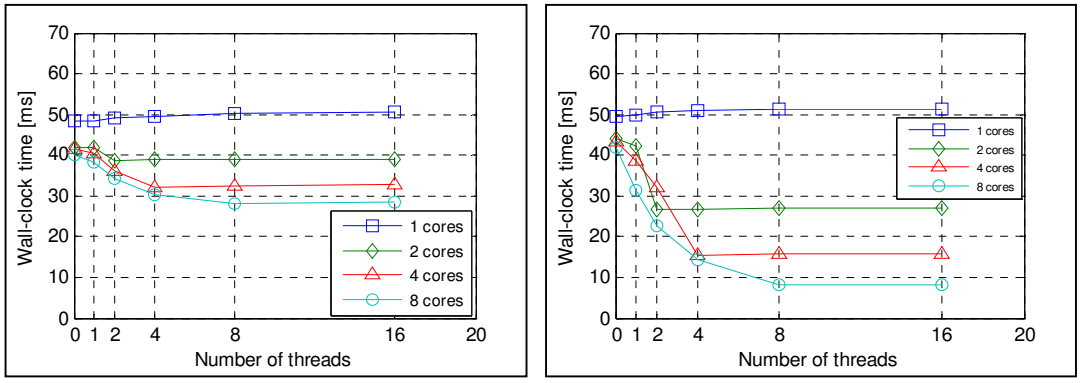
1. For all $p \in Ime$ do
2. if $(m_{inf} < Ime_i(p) < m_{sup})$then $E_i \leftarrow E_i \cup \{p\}$;
3. Repeat until stability
4. $E_{sn} \leftarrow \emptyset$;
5. While $(k \neq 0)$then
6. For all $p \in E_i$ do
7. if $(p \in T)$ then $F_i(x) \leftarrow \alpha^-(x, F_i(x))$;
8. $E_{sn} \leftarrow E_{sn} \cup \{\text{eight p neighbors}\}$;
9. else $E_{ki} \leftarrow E_{ki} \cup \{p\}$;
10. endif
11. For all $p \in E_{i+1}$ do
12. if $(p \in T)$ then $Ime_{i+1}(x) \leftarrow \alpha^-(x, Ime_{i+1}(x))$;
13. $E_{sn} \leftarrow E_{sn} \cup \{\text{eight p neighbors}\}$;
14. else $E_{ki+1} \leftarrow E_{ki+1} \cup \{p\}$;
15. endif
16. $E_i \leftarrow E_{sn}$;
17. $Ime_i \leftarrow Ime_i + Ime_{i+1}$;
18. $Ime_i \leftarrow Ime_i \cup E_{ki} \cup E_{ki+1}$;
19. if $(E_i = \emptyset)$ then $k \leftarrow 0$;
20. clean $\{E_i, E_{ki}, E_{ki+1}\}$;
21. end while

4.2.4 Experimental analysis

The proposed parallel λ -Skeleton algorithm was implemented in C in two variants: the first implementation, based on a simple lock-based shared FIFO queue, using OpenMP critical directive. The second is based on a spin-wait FIFO queue, already introduced in section ***2.4.2***. Wall-clock execution times for numbers of threads equal to 1, 2, 4, 8, and 16, for each one of these implementations, were determined. The efficiency $\Psi(n)$, already defined in section ***2.4.1***., is also measured.

Times were performed on eight-core (***2× Xeon E5405***) shared memory parallel computer of the Faculty of Electrical Engineering and Communication of Brno University, on Intel ***Quad-core Xeon E5335***, on Intel ***Core 2 Duo E8400*** and Intel mono-processor ***Pentium 4 660***. Each processor of the ***Xeon E5405*** and ***E5335*** runs at 2 GHz and both of the two machines have 4 GB of RAM. The E8400 processor runs at 3 GHz. The Pentium processor runs at 3.6 GHz (see ***Table 17***). The last two machines have 2 GB of RAM. The minimum value of 5 timings was taken as most indicative of the speed of the algorithm. The measurements were done on 2D grayscale image ***(512*512)*** of real brain MRI. Results of the two implementations are shown in ***fig. 28 (a)(b)***.

On the eight-core machine, wall-clock execution time for the first implementation using a lock-based shared FIFO queue drops from an average of ***40.211*** ms for a single thread down to ***28.458*** ms at 8 threads. For the second implementation using spin-wait FIFO queue, wall-clock execution time drops from an average of ***41.889*** ms for a single thread down to ***8.282*** ms at 8 threads. As expected, the speed-up for the second implementation using Private-Shared FIFO queue is higher than for the one using lock-based shared FIFO queue, because context changing were nearly eliminated.

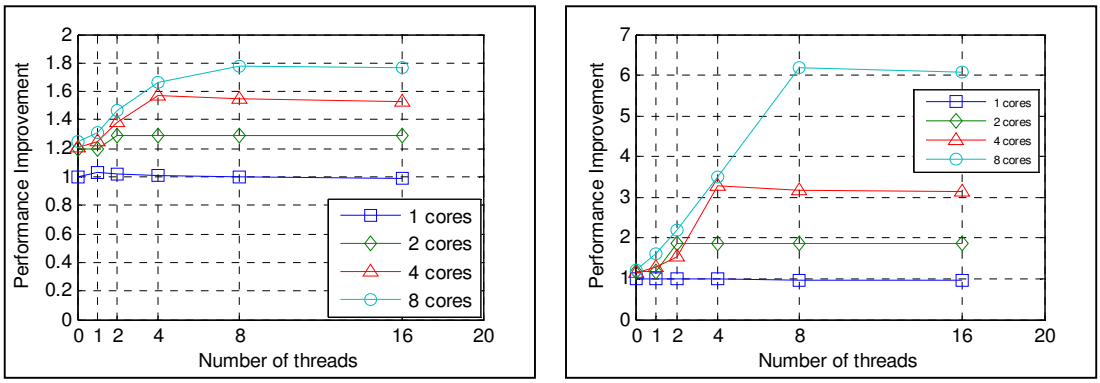


(a) Using a lock based shared FIFO queue ***(b)*** Using a spin-wait shared FIFO queue

Figure 28 : Execution time [parallel thinning]

A remarkable result shown in ***(fig. 29)*** is the fact that the speed-up increases as we increase the number of threads beyond the number of processors in our machine (eight cores). For the first implementation, the speedup at 8 threads is ***1.7 ± 0.05***. However, for the second implementation

the speedup has increased to ***6.2 ± 0.01***. Another common result between ***fig. 28 (a)*** and ***fig. 28 (b)*** is stability of execution time on each *n*-core machine since the code uses *n* or more threads.



(a) Using a lock based shared FIFO queue ***(b)*** Using a spin-wait shared FIFO queue

Figure 29 : Performance improvement [parallel thinning]

For better readability of our results, we tested the efficiency of our algorithm on various architectures using the $\Psi(n)$ formula introduced earlier with fixed serial time equal to ***48.247 ms***. For parallel time we use best parallel time obtained using 8 threads. As can be seen in ***fig. 30***, second implementation is more efficient that the first one in all architectures. It is also suitable to return to Amdahl's law, introduced in section 4, in order to explain obtained results. In fact the global speed up formula is $S(n) = \frac{T(1)}{T(n)}$. Then the defined efficiency $\Psi(n) = T_s/(n * T_p)$ can be written as $\Psi(n) = T_s/(n * T_p) = \frac{S(n)}{n} = \frac{T(1)}{n * T(n)}$. According to Amdahl's law $S(n) = \frac{1}{1-p+\frac{p}{n}}$, efficiency can be written as follows: $\Psi(n) = \frac{1}{n*(1-p)+p}$. Thus if the number of cores increases, the speedup also increases (more work can be done simultaneously with more threads). On the other hand the efficiency will decrease.

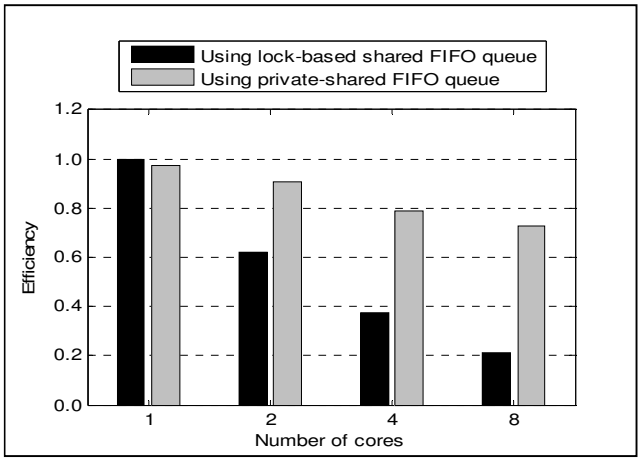


Figure 30 : Efficiency improvement [parallel lambda Skeleton]

4.3 Conclusion

In this chapter, we have presented an intensive study of sixteen thinning algorithms in the frame work of critical kernels: Pavlidis [90, 92], Chin and al. [87], Holt and al. [93], Hall [88], Guo and Hall [96] (3 variants), Jang and Chin [97, 98], Eckhardt and Maderlechner [99], Bernard and Manzanera [86], Rutovitz [91], Zhang and Wang [94], Wu and Tsai [95], Choy and al. [100] and Couprie and al. [8].

First contribution is limited to the classification of these algorithms according to five selection criteria: ***(i)*** preservation of topology, ***(ii)*** skeleton connectivity, ***(iii)*** skeleton symmetry, ***(iv)*** execution time and ***(v)*** cache consumption. Through this classification, we identified Couprie's algorithm as the most suitable algorithm for parallelization on shared-memory architectures.

Second contribution concern an adapted algorithm to compute skeleton that is parallel, preserves the topology of the input image and suited for ***SMP*** machines. ***Fig. 31*** makes link between SD&M strategy and dynamic lambda skeleton algorithm: first step is dividing research area into different sub-region bounds. Since distribution start, each thread will lower each characterized pixel and then push its eight neighbors in available FIFO queue. Each queue contains the set of all selected neighbors and it is shared between only two threads. Unfortunately, conventional synchronization techniques such that lock-based shared FIFO queue have not given good results and performance of our algorithm remained modest. Therefore we have applied our approach based on spin-wait FIFO queue, already introduced in section ***2.4.2*** for better performance. Dynamic lambda skeleton algorithm becomes five times faster than original version proposed by Couprie and al.[8]. Tests on ***2D*** grayscale image (***512x512***), using shared memory parallel machine (***SMPM***) with ***8 CPUs*** cores (***2 × Xeon E5405*** running at frequency ***of 2 GHz***), showed an enhancement of ***6.2*** with a maximum achieved cadency of ***125 images/s*** using 8 threads.

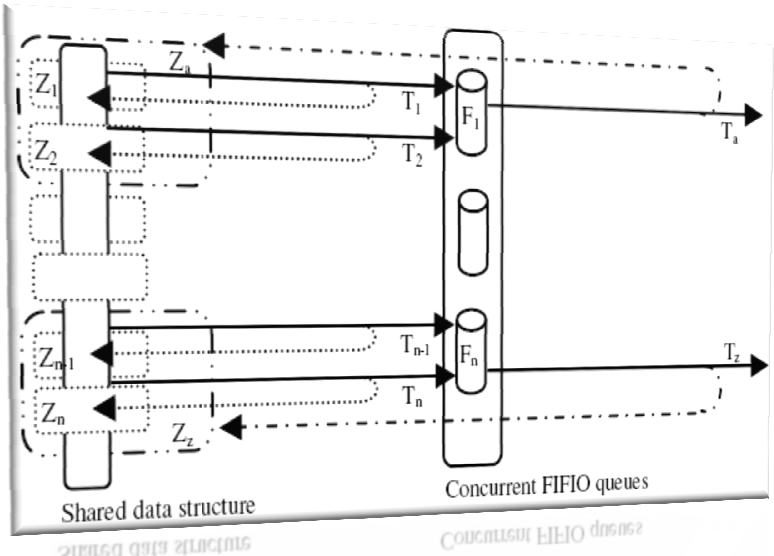


Figure 31 : Illustration on dynamic lambda Skeleton process

TOPOLOGICAL SMOOTHING

Smoothing filter is the method of choice for image preprocessing and pattern recognition. For example, the analysis or recognition of a shape is often perturbed by noise, thus the smoothing of object boundaries is a necessary preprocessing step. Also, when warping binary digital images, we obtain a crenellated result that must be smoothed for better visualization. The smoothing procedure can also be used to extract some shape characteristics: by making the difference between the original and the smoothed object, salient or carved parts can be detected and measured.

Smoothing shape has been extensively studied and many approaches have been proposed. The most popular one is the linear filtering by Laplacien smoothing for 2D-vector [102] and 3D mesh [103]. Other approach by morphological filtering can be applied directly to the shape [104] or to curvature plot of the object's contour [105]. Unfortunately none of these operators preserve the topology (number of connected components) of the original image. In 2004, Couprie and Bertrand [22] introduced a new method for smoothing 2D and 3D objects in binary images while preserving topology. Objects are defined as sets of grid points, and topology preservation is ensured by the exclusive use of homotopic transformations defined in the framework of digital topology [106]. Smoothness is obtained by the use of morphological openings and closings by metric discs or balls of increasing radius, in the manner of alternating sequential filters [107]. The authors' efforts have brought about two major issues such as preserving the topology and the multitude of objects in the scene to smooth out without worrying about memory management, latency or cadency of their filter. Inspired by their approach, we propose a new algorithm for topological smoothing that is parallel and preserves topology.

This chapter is organized as follows: in section 5.1, some basic notions of topological operators are summarized; the original smoothing filter is introduced. In section 5.2, the new parallel smoothing method is introduced. Evaluations of acceleration, efficiency and success rate of cache memory access are presented and discussed in section 5.3. Finally, we conclude with summary in section 5.4.

5.1 Theoretical background

In this section, we recall some basic notions of digital topology [106] and mathematical morphology for binary images [58]. We define also the homotopic alternating sequential filters [22]. For the sake of simplicity, we restrict ourselves to the minimal set of notions that will be useful for our purpose. We start by introducing morphological operators based on structuring elements which are balls in the sense of Euclidean distance, in order to obtain the desired smoothing effect.

We denote by $\mathbb{Z}$ the set of relative integers, and by E the discrete plane $\mathbb{Z}^2$. A point $x \in \mathrm{E}$ is defined by (x_1, x_2) with $x_i \in \mathbb{Z}$. Let $x \in \mathrm{E}$, $r \in \mathbb{N}$, we denote by $B_r(x)$ the ball of radius r

centered on x, defined by $B_r(x)=\{y\in E, d(x,y)\le r\}$, where d is a distance on E. We denote by B_r the map which associates to each x in E the ball $B_r(x)$. The Euclidean distance d on E is defined by: $d(x,y)=\left[\mathrm{A}^2-\mathrm{B}^2\right]^{1/2}$ with $\mathrm{A}=(x_1-y_1)$ and $\mathrm{B}=(x_2-y_2)$.

An operator on E is a mapping from $P(E)$ into $P(E)$, where $P(E)$ denotes the set of all subsets of E. Let r be an integer, the dilation by B_r is the operator δ_r defined by $\delta_r(X)=\bigcup_{x\in X}B_r(x)\ \ \forall X\in P(E)$. The ball B_r is termed as the structuring element of the dilation. The erosion by B_r is the operator ε_r defined by duality: $\varepsilon_r=*\delta_r$.

Now, we introduce notion of simple point which is fundamental for the definition of topological operators in discrete spaces. We give a definition of local characterization of simple points in $E=\mathbb{Z}^2$. Let consider two neighborhoods' relations Γ_4 and Γ_8 defined for each point $x\in E$ by:

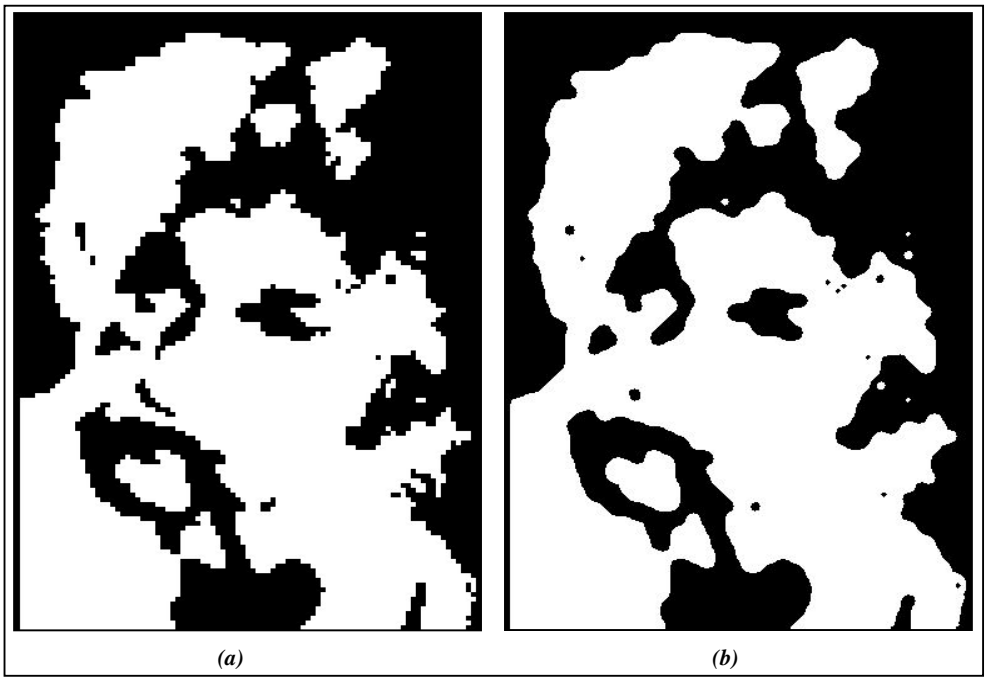
$$\Gamma_4(x)=\{y\in E;|y_1-x_1|+|y_2-x_2|\le 1\},$$
$$\Gamma_8(x)=\{y\in E;\max|y_1-x_1|,|y_2-x_2|\le 1\}.$$

For general case, we define $\Gamma_n^*(x)=\Gamma_n(x)\setminus\{x\}$ with $n\in\{4,8\}$. Thus y is said n-adjacent to x if $y\in\Gamma_n^*(x)$. We say also that two points x and y of X are n-connected in X if there is an n-path between these two points. The equivalence classes for this relation are n-connected components of X. A subset X of E is said to be n-connected if it consists of exactly one n-connected component. The set of all n-connected components of X which are n-adjacent to a point x is denoted by $C_n[x,X]$. To guarantee correspondence between X topology and $\overline{X}$ topology, we use n-adjacency for X and $\overline{n}$-adjacency for $\overline{X}$, with $(n,\overline{n})$ equal to (8; 4) or (4; 8).

Informally, a simple point p of a discrete object X is a point which is inessential to its topology. In other words, we can remove p from X without changing its topology. A point $x\in X$ is said simple if each n-component of X contains exactly one n-component of $X\setminus\{x\}$ and if each $\overline{n}$-component of $\overline{X}\cup\{x\}$ contains exactly one $\overline{n}$-component of $\overline{X}$. Let $X\subset E$ and $x\in E$, two connectivity numbers defined as follows (#X = cardinality of X): $T(x,X)=\#C_n\left[x,\Gamma_8^*(x)\cap X\right];\overline{T}(x,X)=\#C_{\overline{n}}\left[x,\Gamma_8^*(x)\cap\overline{X}\right]$.

The following properties allow us to locally characterize simple points [106,108] hence to implement efficiently topology preserving operators: $x\in E$ is simple for $X\subseteq E\leftrightarrow T(x,X)=1$ and $\overline{T}(x,X)=1$.

The homotopic alternating sequential filter is a composition of homotopic cuttings and fillings by balls of increasing radius. It takes an original image X and a control image C as input, and smoothes X while respecting its topology and geometrical constraints implicitly represented by C. A simple illustration is given by ***fig. 32***. Smoothed image ***(b)*** is obtained using Homotopic Alternative Sequential filter (***HAS***) with a radius equaled to five and four connectedness (Γ_4). More examples can be found in [22].



(a) Input image ***(b)*** Smoothed image

Figure 32 : Smoothing illustration

Based on this filter, Authors [22] introduce a general smoothing procedure with a single parameter to control smoothing degree. Let $C \subseteq X$, $r \in \mathbb{N}$ and $D \subseteq \overline{X}$ with X any finite subset of E. The homotopic alternating sequential filter $(HASF)$ of order n, with constraint sets C and D, is defined as follows:

$$HASF_n^{C,D} = HF_n^D \circ HC_n^C \circ ... HF_1^D \circ HC_1^C$$

In the previous formula, HC_n^C (i) refers to homotopic cutting of X by B_n with constraint set C and HF_n^D (ii) refers to homotopic filling of X by B_n with constraint set D. These two homotopic operators can be defined as follows:

$$HC_n^C(X) = *H(Y,V) \text{ With } \begin{cases} Y = H(X, \varepsilon_n(X) \cup C) \\ V = (\delta_n(Y) \cap X) \end{cases} \text{ (i)}$$

$$HF_n^D(X) = H(Z,W) \text{ With } \begin{cases} Z = *H(X, \delta_n(X) \cap \overline{D}) \\ W = (\varepsilon_n(Y) \cup X) \end{cases} \text{ (ii)}$$

We recall that $H(Z,W)$ is a homotopic constrained thinning operator. It gives the ultimate skeleton of Z constrained by W. The ultimate skeleton is obtained by selecting simple point in increasing order of their distance to the background thanks to a pre-computed Euclidian distance map [109]. We recall also that $*H(Y,V)$ is an homotopic constrained thickening operator. It thickens the set of Y by iterative addition of points which are simple for $\overline{Y}$ and belong to the set V until stability.

5.2 Parallel smoothing filter

In this section we start by analyzing overall structure of original algorithm. Then we continue with the parallelization of Euclidean distance, thinning and thickening algorithm. We conclude by a performance analysis of each operator. Obtained execution time, efficiency, speedup and cache misses will be introduced and discussed.

As we have shown previous section, smoothing algorithm receives as input a binary image and maximum radius. It uses two procedures for homotopic opening and closing, see ***fig. 33 (a) (b)***. The call is looped to ensure an ongoing relationship between input and output. The opening process is a consecutive execution of erosion, thinning, dilatation and thickening. While closure procedure ensures the same performance of the four consecutive functions with single difference: the erosion instead of dilatation. Thinning and thickening ensure the topological control of erosion and dilatation. This control is based on researching and removing of all destructible points (already defined in section ***4.2.1***). When destructible point is deleted, its neighbors are reviewed to ensure that they are not destructible either.

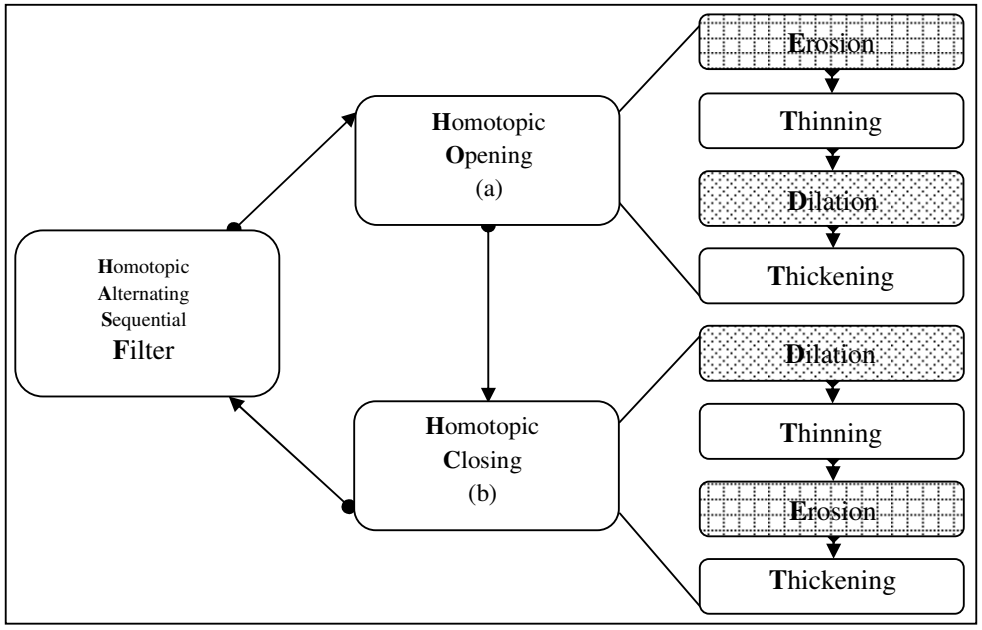


Figure 33 : Overall structure [Original smoothing algorithm]

A preliminary assessment of first implementation code, see ***Table 19***, shows that Euclidean distance computing (**EucDis**) takes more time than topological point characterization (**Topcar**). For an image of (***200*200***), computation time of E.D with an infinite radius is ***46.67%*** while point characterization of ***2.4*** million points occupies only ***18.15%***. If we limit radius between ***5*** and ***10***, computation time of E.D. continues to increase. It can reach ***64.44%*** of total time with a radius equal to ***5***. However time for topological characterization is only ***8.89%*** for ***1*** million points. These finding remain the same if we increase image size. Beyond (***512*512***), computing time of point characterization becomes considerable.

	200 x 200			168 x 168		
	r=5	r=10	r=∞	r=5	r=10	r=∞
EucDis (%)	64.44	54.93	46.67	59.25	49.79	35.25
TopCar (%)	8.89	13.89	18.15	11.58	16.50	24.03

***(EucDis)*: Euclidian distance function; *(TopCar)*: topological characterization function**

Table 19 : Time execution rate [Smoothing algorithm]

5.2.1 Study on Euclidean distance algorithms

During previous evaluation, ***4SED*** [109] algorithm was used for Euclidean distance computation. So we are looking for another algorithm that is faster, and parallelizable. New algorithm must have an Euclidean distance computation error less than, or equal to, that produced by ***4SED*** in order to maintain homotopic characteristics of the image.

In literature, several algorithms for Euclidean distance computing exist. Lemire [110] and Shih [111] algorithms are bad candidates because Lemire's algorithm does not use Euclidean circle as structuring element. Then homotopic property will not be preserved. Shih's algorithm has a strong data dependency which penalizes parallelization. In [112], Cuissenaire propose a first algorithm for Euclidian distance computing, called ***PSN*** "Propagation Using a Single Neighborhood" that uses only four neighbors (on element structure). He also proposes a second algorithm, called ***PMN*** "Propagation Using Multiple Neighborhood" that uses eight neighbors. In [113], he also proposes a third algorithm with $o(n^{3/2})$ complexity, which offers an accurate computation of the Euclidean distance. Only drawback of this third algorithm is computation time which is very important and goes beyond the two algorithms mentioned above. Even if computing error produced by ***PSN*** is greater than computing error produced by ***PMN***, it is comparable to that produced by ***4SED***. Low data dependence and ability to operate on 3D images, makes ***PSN*** algorithm a potential candidate to replace ***4SED***. Meijster [9] proposes an algorithm to compute exact Euclidean distance. Algorithm complexity is $o(n)$ and it operates in two independent, but successive, steps. First step is based on looking over columns then computing distance between each point and existing objects. Second step includes same treatment looking over lines. It is important to note that strong independence between different processing steps and computing error equal to zero makes Meijster algorithm another potential candidate to replace ***4SED***. Algorithm is also able to operate on 3D images. Theory analysis of Meijster and Cuissenaire algorithms can be found in Fabbri's work [114].

In the following, we propose first analysis of selected algorithms (Danielsson-***4SED*** version [109], Cuisenaire-***PSN*** version [112] and Meijster [9]) based on their implementation in order to compare between them. We have implemented ***4SED*** algorithm using a fixed size stack. This stack uses a FIFO queue and it has small size while ***4SED*** algorithm does not need to store temporal image. Results are directly stored into the output image, we will retain this implementation because ***4SED*** assessment serve only as reference for comparison. For ***PSN*** implementation, we used stacks with dynamic sizes. Memory is allocated using small blocks defined at stack creation. When an object is added to queue, algorithm will use available memory of last block. If no space is available, a new block is allocated automatically. Block size is proportional to image size (***N x M / 100***). Finally we used a simple memory structure to implement Meijster algorithm. A simple matrix was used to compute distance between points and object of each column and three vectors were used to compute distance in each line. We recall that this comparison is done in order to select the best algorithm among three candidates.

Figure 34 describes obtained results by different implementations on single processor architecture ***P4***. During this evaluation we used binary test image (***200x200***). We have also varied ball radius. We used Valgrind software to evaluate different designs. Callgrind tool returns the cost of implementing of each program by detecting IF (Instruction Fetch).

Results show that ***PSN*** algorithm is the most expensive in all cases (for any radius). Meijster algorithm is moderately faster than ***4SED***. Their curves are practically parallel and their returned values are proportional. However, difference between 4SED and PSN curves is more visible, it become larger when we increase radius. The output images returned by Meijster algorithm hold the best visual quality while Euclidean distance computation error is almost zero thus our efforts will be brought on Meijster algorithm parallelization.

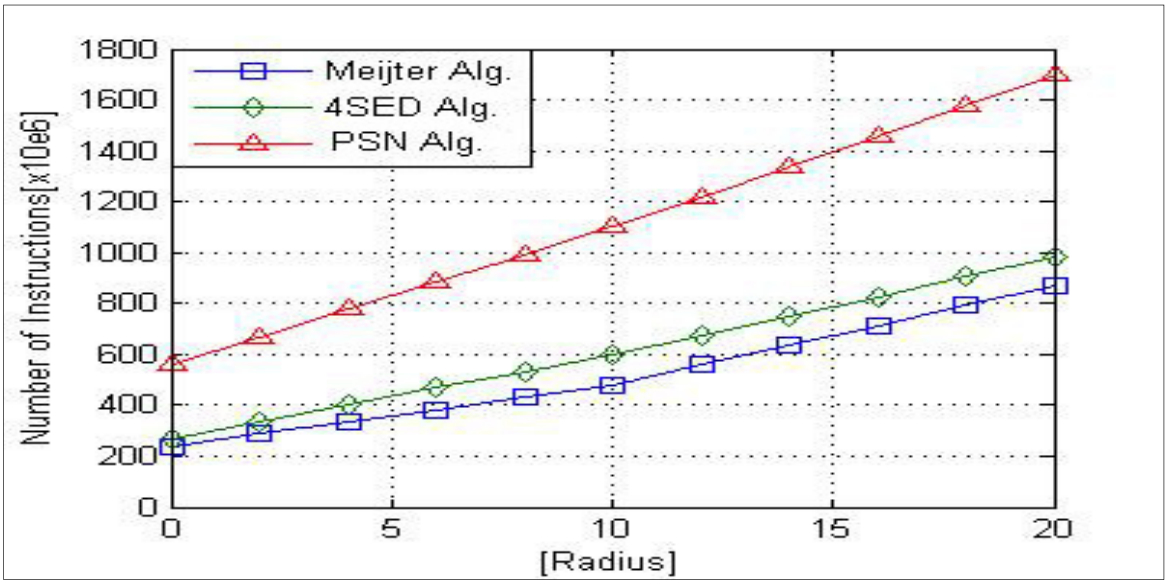


Figure 34 : Execution time [Danielsson, Cuisenaire and Meijster Algo.]

5.2.2 Parallelization of Meijster algorithm

We denote by I input image with m columns and n rows. We denote by B an object included in I. The idea is to compute, for each point $p \in I \wedge p \notin B$, separating distance between p and the closest point b with $b \in B$ and $\forall (0 \le b \le m)$, $b = (b_x, b_y)$. This amount to compute the following matrix: $dt[p_x, p_y] = \sqrt{EDT(p)}$ with $EDT(p) = \min(p_y - b_x)^2 + G(p_x, b_y)^2$.

If we assume that minimum distance of an empty group K is ∞ and $\forall z \in K$, we have $(z_y + \infty) = \infty$ then $EDT(p)$ formula can be written as follow: $\forall b_x < n$, $\forall b_y \le m$, $EDT(p) = \min(p_y - b_x)^2 + G(p_x, b_y)^2$ with $G(p_x, y) = \min |p_x - b_x| : b = (b_x, y)$. Thus we can split the Euclidian distance transform procedure into two steps. The first step is to scan columns and compute EDT for each column y. Second step consists on repeating the same procedure for each line.

In the following we start by detailing these two steps: In the first step $G(p_x, y)$ can be computed through the two following sub functions with $\forall 0 \le b_x \le n$:

$G_T(p_x, y) = \min p_x - b_x : b = (b_x, y)$,

$G_B(p_x, y) = \min b_x - p_x : b = (b_x, y)$.

To compute $G_T(p_x, y)$ and $G_B(p_x, y)$, we scan each column y from top to bottom using the two following formula: $G_T(p_x, y) = G_T(y, p_x - 1) + 1$ $G_B(p_x, y) = G_B(y, p_x + 1) + 1$. Thus sequential algorithm of the first step can be written as follows. Complexity order is $o(n \times m)$.

Algorithm 27 : E.D.T algorithm – 1st Step – Original version [Meijster]

Input : m:colums, n:lines, b:image

1. **Forall** $y \in [0..m-1]$ **do**
2. **If** $(0, y) \in B$ **then** $g[0..y] = 0$
3. **else** $g[0..y] = \infty$
4. **endif**
5. /* G_T */
6. **for** $(x = 1)$ **to** $(n - 1)$ **do**
7. **if** $[x, y] \in B$ **then** $g[x..y] = 0$
8. **else** $g[x, y] = g[x + 1, y] + 1$
9. **endif**
10. **endfor**
11. /* G_B */
12. **for** $(x = n - 2)$ **downto** (0) **do**
13. **if** $g[x + 1, y] < g[x, y]$ **then**
14. $g[x, y] = g[x + 1, y] + 1$
15. **endif**
16. **Endfor; endforall**

Let's move to the second step. We start by defining $f(p,y)=(p_y-y)^2+G(p_x,y)^2$.Then we can define $EDT(p)=\min f(p-y), \forall 0\le y\le m$. For each row u, we note that there is, for the same point p, the same value of $f(p,y)$ for different values of y, so we can introduce the concept of "***region of column***".

Let S be the set of y points such that $f(p,y)$ is minimal and unique. The formula of S, $\forall 0\le y\le u$, is $S_p(u)=\min y: f(p,y)\le f(p,i)$. $\forall 0\le i\le u\wedge u\le m$. Let T be the set of points with coordinate greater than, or equal to, horizontal coordinate of the intersection with a region: $T_p(u)=Sep_{p_x}(S_p(u-1),u)+1$.

Let $Sep(i,u)$ be the separation between regions of i and u, defined by the following with $Dif=(G(p_x,u)^2-G(p_x,i)^2)$:

$$f(p,i)\le f(p,u)$$
$$\Leftrightarrow (p_y-i)^2+G(p_x,i)^2\le (p_y-u)^2+G(p_x,u)^2$$
$$\Leftrightarrow Sep_{p_x}(i,u)=(u^2-i^2+Dif)/2(u-1)=p_y.$$

Thus lines will be processed, from left to right then from right to left. During the first term, from left to right, two vectors S and T will be created. These two vectors will contain respectively all regions and all intersections. During the second treatment, from right to left, we compute f for each value of S. For each respective values of T, f is computed. ***Algorithm 28*** is associated to second step. For the first term, complexity order is $q+2(m-u)$ whereas complexity order of the second term is only m.

The independence of data processing between rows and columns is the key to apply of ***SD&M*** parallelization strategy. In the first stage, column processing, we can define data interdependence by the following equation:

$$G(p_x,y)=\min\{G_T(p_x,y),G_B(p_x,y)\}$$
$$\Leftrightarrow G_T(p_x,y)=\begin{Bmatrix}0 & if\,(p_x,y)\in B\\ G_T(p_x,y) & else\end{Bmatrix}$$
$$\Leftrightarrow G_B(p_x,y)=\min\{G_B(p_x+1,y),G_T(p_x,y)\}$$

It follows that values of each column y of G, depends only on lines: p_x, p_x+1 and p_x-1. Similarly, at the second stage, we can introduce the following interrelationship: $Edt(p)=f(p,S_p(q))$. Then $\forall(0\le y\le u),(0\le i\le u)\Lambda(u<m)$, $S_p(u)=\min y: f(p,y)\le f(p,i)$. Thus, if $(u=T_p(q))$ so $q=(q-1)$ which imply the following: $T_p(u)=Sep_{p_x}(S_p(q),u)+1$.

Algorithm 28 : E.D.T algorithm - 2nd Step – Original version [Meijster]

Input : **b**:image, **g**: G_Table, **m**: columns, **n**:lines

1. **Forall** $x \in [0..n-1]$ **do**
2. $q = 0$
3. $s[0] = 0$
4. $t[0] = 0$
5. */* First part */*
6. **for** $(u = 1)$ **to** $(m-1)$ **do**
7. $A = (q \geq 0) \Lambda [f((x,t[q]),s[q])]$
8. $B = f((x,t[q]),u)$
9. **while** $(A > B)$ **then** $q \leftarrow (q+1)$
10. **end while**
11. **if** $(q < 0)$ **then** $(q \leftarrow 0)$
12. $(s[0] \leftarrow u)$
13. **else** $w \leftarrow Sep(s[q],u,x)+1$
14. **if** $(w < m)$ **then** $q \leftarrow (q+1)$
15. $s[q] \leftarrow u$
16. $t[q] \leftarrow w$
17. **endif**
18. **endif**
19. **endfor**
20. */* Second part */*
21. **for** $(u = m-1)$ **to** (0) **do**
22. $Edt[x,u] = f((x,u),s[q])$
23. **if** $(u = t[q])$ **then** $q \leftarrow (q-1)$
24. **endif**
25. **Endfor**
26. **end Forall**

According to this formalization, values of $f(p,i)$ and $Sep_x(i,u)$ are independent of modified data. So using two vectors S and T, a private variable q for each line ensures complete independence in writing. We start applying the splitting step by sharing the columns and lines processing between multiple processors. A thread can process one or more columns and the number of threads used will depend on the number of processors. The results returned by all threads in this first stage will be merged in order to start lines processing.

In the following we introduce the parallel version of Meisjter algorithm for both steps. Associated algorithm complexity is $o((n \times m)/N)$. $(n \times m)$ refers to image size and N refers to the number of processors.

Algorithm 29 : E.D.T algorithm - 1st Step – parallel version [Mahmoudi and Akil]

1. **For** $(y = t, y < m, y = y + t_{\max})$ **do**
2. **If** $(0, y) \in B$ **then** $g[0, y] \leftarrow 0$
3. **else** $g[0, y] \leftarrow \infty$
4. **endif**
5. /* G_T */
6. **for** $(x = 1)$ **to** $(n - 1)$ **do**
7. **if** $[x, y] \in B$ **then** $g[x, y] \leftarrow 0$
8. **else** $g[x, y] \leftarrow g[x+1, y] + 1$
9. **endif**
10. **Endfor**
11. /* G_B */
12. **for** $(x = n - 2)$ **downto** (0) **do**
13. **if** $(g[x+1, y] < g[x, y])$ **then**
14. $g[x, y] \leftarrow g[x+1, y] + 1$
15. **endif**
16. **endfor**
17. **Endforall**

Proposed parallel version of Meijster algorithm was implemented in C using OpenMP directives. Speedup for numbers of threads equal to 1, 2, 4, 8, and 16 were determined. The efficiency measure $\Psi(n)$ is given by the following formula with n the number of processors: $\Psi(n)$ = seq. time /(n *p. time) (ii), already presented in section ***2.4.1.***

Times were performed on eight-core (***2× Xeon E5405***) shared memory parallel computer, on Intel Quad-core ***Xeon E5335***, on Intel Core 2 Duo ***E8400*** and Intel mono-processor ***Pentium 4 660***. The minimum value of 5 timings was taken as most indicative of algorithm speed. More information about architectures characteristics are given in ***Table 20***.

Algorithm 30 : E.D.T algorithm - 2nd Step – parallel version [Mahmoudi and Akil]

1. **For** $(x = t, x < n, x = x + t_{\max})$ **do**
2. $q = 0$; $s[0] = 0$;
3. $t[0] = 0$;
4. /* First part */
5. **for** $(u = 1)$ **to** $(m - 1)$ **do**
6. $A \leftarrow (q \geq 0) \Lambda \left[f((x, t[q]), s[q]) \right]$
7. $B \leftarrow f((x, t[q]), u)$
8. **while** $(A > B)$ **do** $q \leftarrow (q + 1)$
9. **end while**
10. **if** $(q < 0)$ **then** $(q \leftarrow 0)$
11. $(s[0] \leftarrow u)$
12. **else** $w \leftarrow Sep(s[q], u, x) + 1$
13. **if** $(w < m)$ **then** $q \leftarrow (q + 1)$

```
14.                    s[q] ← u
15.                    t[q] ← w
16.                endif
17.            endif
18.          Endfor
19.          /* Second part */
20.          for (u = m−1) downto (0) do
21.              Edt[x,u] ← f((x,u), s[q])
22.              if (u = t[q]) then q ← (q−1)
23.              endif
24.          endfor
25.      end Forall
```

The measurements were done on 2D binary image (***512*512***). If we can get a satisfactory outcome for this standard, it will be the same for smaller size images. View cache size limits, larger image will not be tested. ***Figure 35*** shows that number of instructions to compute Euclidian distance drops from an average of $9.5x10^8$ using ***4SED*** algorithm down to $7.6x10^8$ ms with Meijster algorithm. Despite the passage from a sequential version running on single core to a parallel version running on 8 processors, acceleration is only multiplied by ***1.6*** as shown in ***fig. 36 (a)***.This can be explained by the choke point between columns processing and lines processing. Waiting time between these two treatments significantly penalizes acceleration. ***Figure 36 (b)*** shows that efficiency variation depends on the number of threads. It is also proportional to the number of processors. Moving to ***3***, ***5*** or ***7*** threads (odd number) decreases significantly the efficiency which reaches its maximum each time that the number of threads is equal the number of processors.

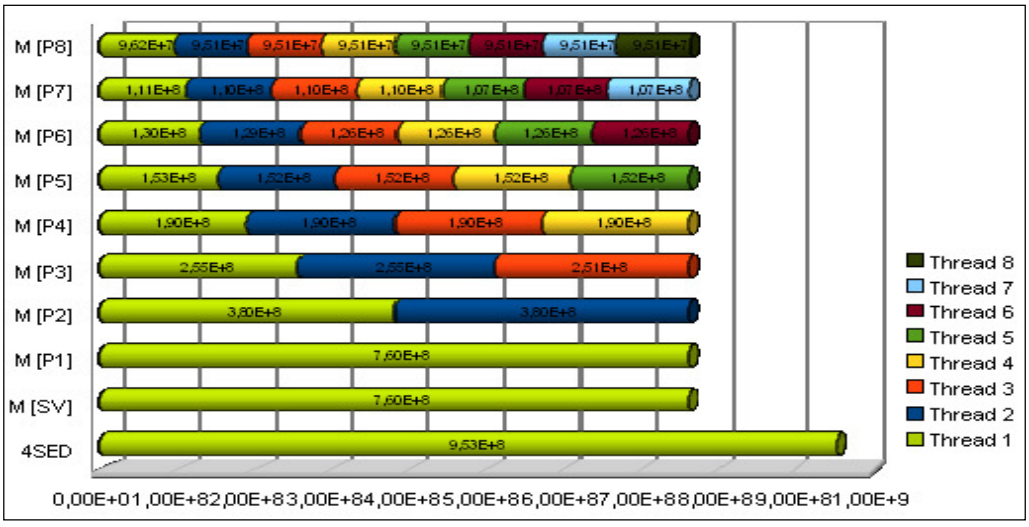
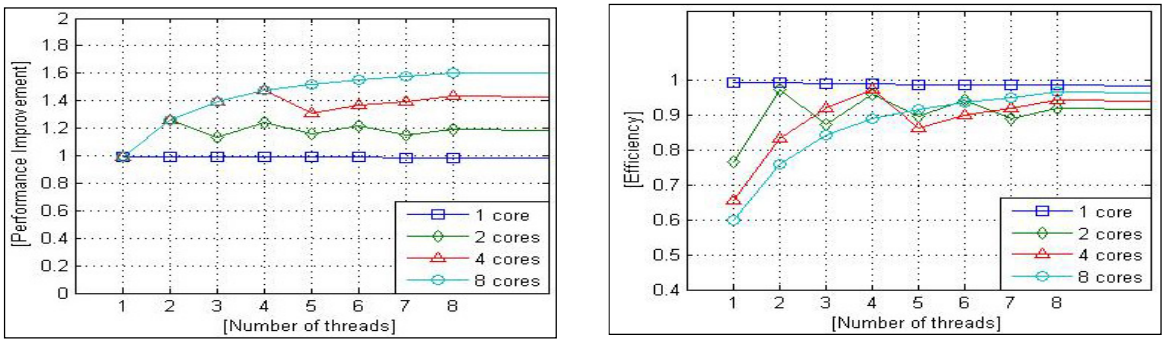


Figure 35 : Evaluation of instruction distribution (Meijster Alg.)



(a) ***(b)***

Figure 36: (a) Performance evaluation ***(b)*** Efficiency evaluation [Meisjter Algo.]

5.2.3 Thinning and thickening computation

Algorithms of thinning and thickening are almost the same. The only difference between them is the following: in thinning algorithm, destructible points are detected then their values are lowered. In thickening algorithm, constructible points, are detected then their values are increased. For parallelization, we will apply the same techniques introduced in previous chapter. Inspired from Couprie approach, we propose a similar version using two loops. Target points are initially detected then their value lowered or enhanced according to appropriate treatment. The set of their eight (or four) neighbors are copied into a "***buffer***" and rechecked. This treatment is repeated until stability. In the following, we present an adapted version of Couprie's thinning algorithm.

Algorithm 31 : Adapted Version Thinning Algo. [Mahmoudi and Akil]

1. **while** ($input[x]$ is destructible) **do**
2. $push(x, stack1)$
3. $x \leftarrow x+1$
4. **EndWhile**
5. $output \leftarrow input$
6. **While** $(stack1 \neq \varnothing) \wedge (\max_{iter} > 0)$ **do**
7. **While** $(stack1 \neq \varnothing)$ **do**
8. $x \leftarrow pop(stack1)$
9. **if** ($output[x]$ is destructible) **then**
10. $output[x] \leftarrow reduce\_pt(x)$
11. $push(x, stack2)$
12. **endif**
13. **end while**
14. **While** $(stack2 \neq \varnothing)$ **do**
15. $x \leftarrow pop(stack2)$
16. $v \leftarrow neighbors(x)$
17. $i \leftarrow 0$
18. **While** $(i < 8)$ **do**
19. **if** $(v[i] \notin stack1)$ **then**

```
20.            push(v[i], stack1)
21.          endif
22.        EndWhile
23.      EndWhile
24.      max_iter ← max_iter − 1
25.    EndWhile
```

Unfortunately direct application of introduced parallel processing is not possible with the set of all points. Some points, called critical points, cannot be eliminated in parallel because initial topology of the image may be broken. ***Figure 37*** illustrates this case: Critical points of an input image ***(a)*** are identified in ***(b)***. If these points are deleted in one iteration ***(c)*** topology necessary is broken ***(d)***. To resolve this problem, we propose that research areas assigned to each thread must be composed of at least six lines (of the image). Each thread will use two buffers to treat each three lines thus four buffers are used to treat six lines as shown in ***fig 37 (e)***.

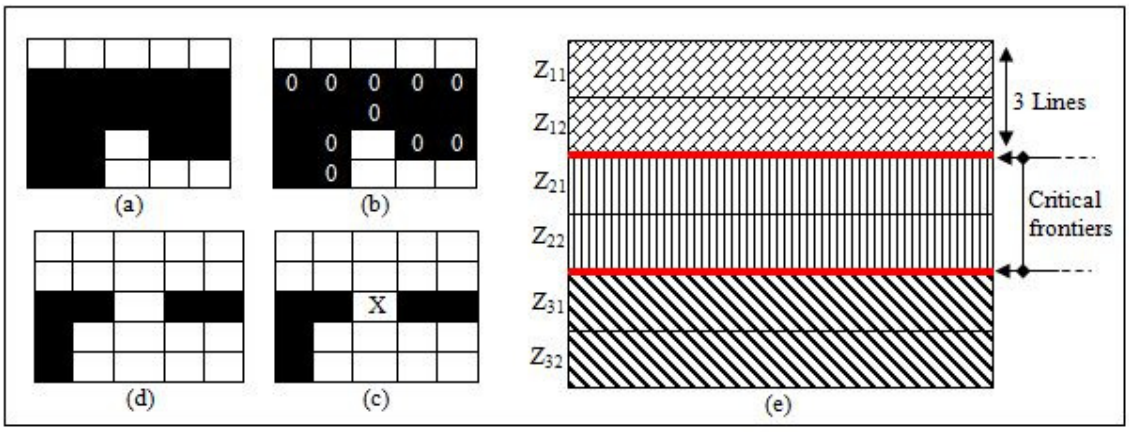
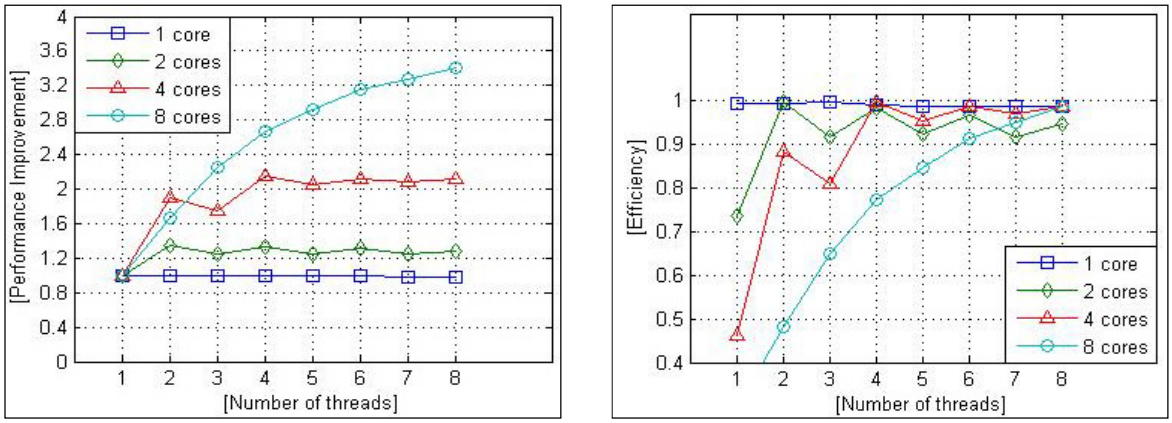


Figure 37 : (a) (b) (c) (d) Critical point illustration (e) Research Area Assignment

Through this organization threads can start running in parallel on Z_{11}, Z_{21} and Z_{31}. Once processing is completed threads can restart running on Z_{12}, Z_{22} and Z_{32}. In some cases, a neighbor of a destructible point is detected on the border of a contiguous area. To prevent that such neighbor escape to recheck, it must be injected to buffer of the right thread. Let's suppose that a point $p \in Z_2$ is considered as destructible by ***T2***, so its value will be lowered and its four neighbors $\{v_1, v_2, v_3, v_4\}$ should be rechecked. Neighbors $\{v_1, v_2, v_4\}$ belong to Z_2 so they will be push in T2 buffers. The neighbor $\{v_3\}$ belongs to Z_3 so it will stack T3 buffers.

Performance evaluation of introduced adapted version of Couprie's algorithm is shown in ***fig.38 (a) (b)***. On eight cores architecture, acceleration does not exceed ***3.4***. Such moderate result can be explained by critical borders processing. Regarding efficiency, the best performance is achieved when the number of thread is equal to the number of processors. If this equality is not ensured, the efficiency decreases. The problem threads' add number still persists. The next step

is to combine the parallel version of Meijster algorithm and the adapted version of Couprie's algorithm to build the parallel processing of topological smoothing.



(a) *(b)*

Figure 38 : ***(a)*** Performance Evaluation ***(b)*** Efficiency Evaluation [Couprie's Algorithm].

5.3 Global analysis

In this section, we present a global evaluation of the parallel smoothing operator. We start by presenting performance evaluations in terms of acceleration and efficiency. Then we evaluate cache memory consumption.

5.3.1 Execution time

We implemented two versions of the proposed parallel topological smoothing algorithm, the first one using '***Symmetric Multiprocessing'*** scheduler and the second one using '***basic-NPS***' scheduler. Wall-clock execution times for numbers of threads equal to ***1***, ***2***, ***4***, ***8*** and ***16*** were determined. The minimum value of ***2*** timings was taken as most indicative of algorithm speed. The measurements were done on 2D binary image (***512*512***). Results of the second implementation on the eight-core are shown in the following figure.

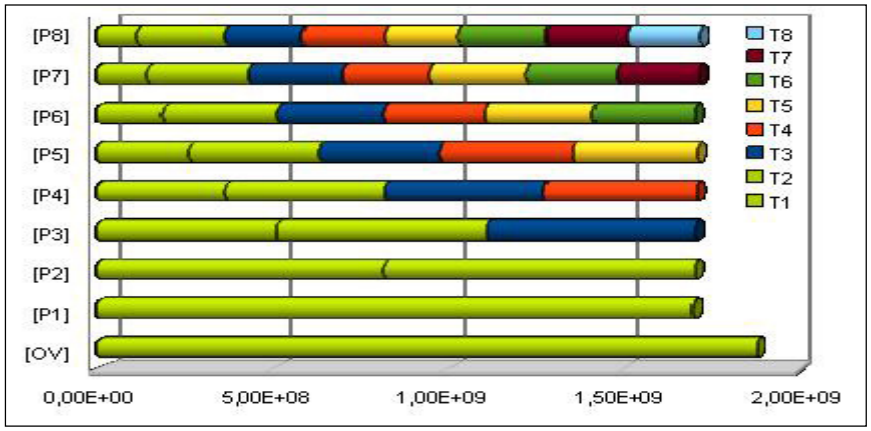
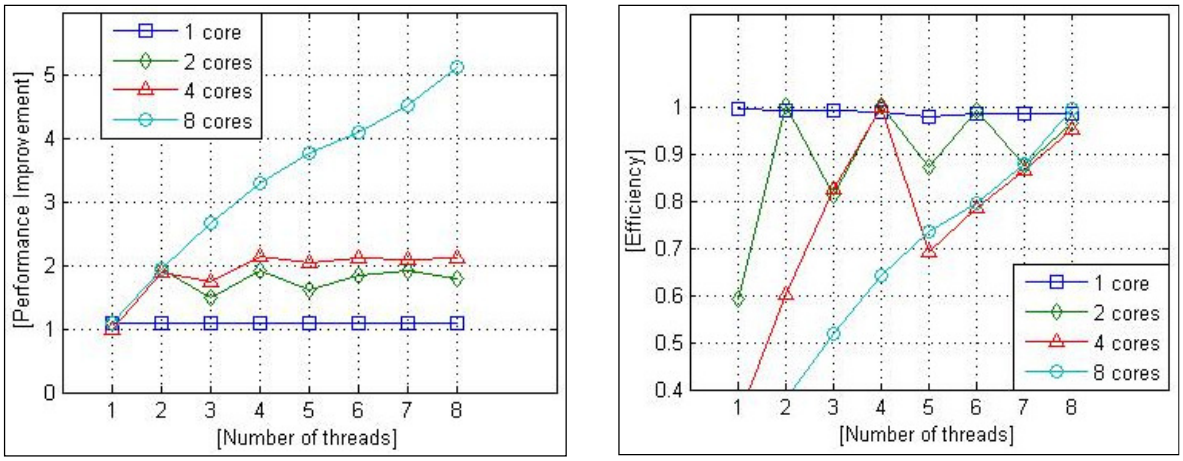


Figure 39: Tasks distribution using 'Basic-NPS' [parallel topological smoothing]

We note that number of instructions drops from an average of ***1879 x10^8*** FI with a single thread down to ***1652 x10^8*** ms with 8 threads. As expected, the speed-up for the second implementation using '***basic-NPS***' scheduler is higher than for the one using "***Symmetric Multiprocessing***" scheduler, thanks to balanced distribution of tasks. A remarkable result about speedup is also shown in ***fig. 40 (a)***. In fact, speed-up increases as we increase the number of threads beyond the number of processors in our machine (eight cores). In the first implementation, using "***Symmetric Multiprocessing***" scheduler, the speedup at 8 threads is ***1.9±0.01***. However, for the second implementation, using '***basic-NPS***' scheduler, the speedup has increased to ***5.2±0.01***. Another common result between different architecture is stability of execution time on each n-core machine since the code uses n or more threads. For better readability of our results, we tested also efficiency of our algorithm on various architectures (see ***fig. 40 (b)***) using the $\psi(n)$ formula introduced earlier. For parallel time ratio we used best obtained time with 8 threads ('basic-NPS' scheduler).



(a) ***(b)***

Figure 40 : ***(a)*** Global Performance improvement ***(b)*** Global efficiency improvement

5.3.2 Cache memory evaluation

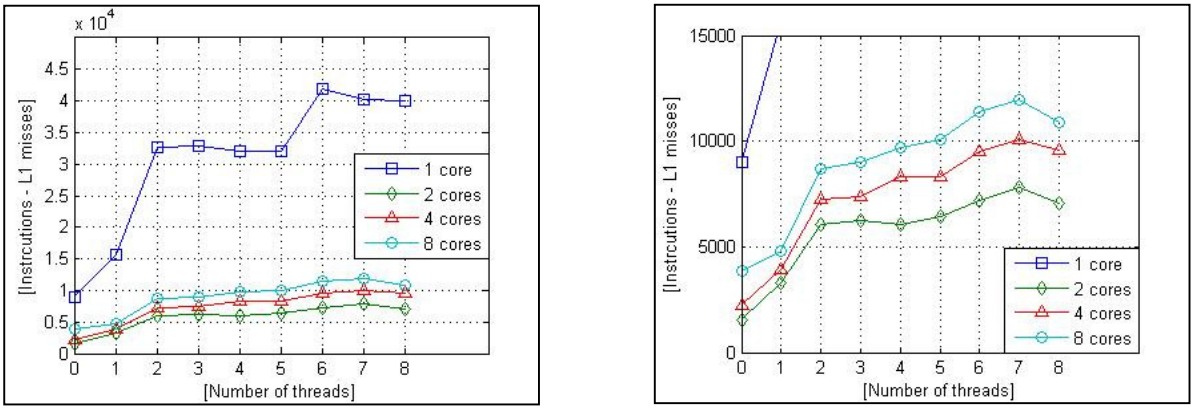
As memory access is a principal bottleneck in current-day computer architectures, a key enabler for high performance is masking the memory overhead. If we starts from basic theory that two classic cache design parameters dramatically influence the cache performance: the block size and the cache associativity. So the simplest way to reduce the miss rate is to increase the block size even it increases the miss penalty. The second solution is to decrease associatively in order to decrease hit time thus to retrieve a block in an associative cache, the block must be searched inside of an entire set since there is more than one place where the block can be stored.

Unfortunately, we are dealing with non-reconfigurable architectures with caches whose associativity and block size are predefined by the manufacturer. Nowadays, new approaches to reduce cache miss are developed such as taking advantage of locality of references to memory or using aggressive multithreading so that whenever a thread is stalled, waiting for data, the system can efficiently switch to execute another thread. Despite their power, the application of both approaches remains limited. In fact, applications of locality approach still experimental

even with Larrabee technology introduced by Intel. And the aggressive multithreading approach has been specially designed for graphics processing engines, which manage thousands of in-flight threads concurrently. So it is not recommended for general SMP machines with limited number of processors and threads. With all these limitations, the most intuitive solution is to rely on the scheduling. Thanks to the '***basic-NPS***' scheduler, we have balanced the charges then prevent context switching thus we minimize caches misses.

		Intel P4	Intel Dual C. T1400	Intel C2 Quad Q9550	Intel Xeon E5405
Num. of processor		1	2	4	2 x 4
SMT		Yes	Yes	Yes	Yes
Frequency		3,4 GHz	1,73 GHz	2,83 GHz	2,00 GHz
L1 Instr. Cache	**Size**	16Kb	32Ko	32Ko	32Ko
	Asso.	8-way	8-way	8-way	8-way
	Block size	32byte	32byte	32byte	32byte
L1 Data Cache	**Size**	16Kb	32Ko	32Ko	32Ko
	Asso.	8-way	8-way	8-way	8-way
	Block size	64byte	64byte	64byte	64byte
L2 Cache	**Size**	2Mb	512Kb	6Mb	6Mb
	Asso.	8-way	8-way	8-way	8-way
	Block size	64byte	64byte	64byte	64byte
RAM size		1Gb	2Gb	2Gb	8Gb

Table 20 : Hardware configuration [parallel smoothing alg.]



(A-1) ***(A-2)***

Figure 41 : ***(A-1)*** Instruction - L1 misses; ***(A-2)*** zoom on (A-1) [parallel topological smoothing]

In the following we present our experimental analysis. We consider a commonly used Intel processor configuration (More details are given by ***Table 20***). Number of processor varies from one to eight. The frequency varies between ***1,73 GHz*** and ***3,4 GHz***. The ***L1*** caches have at least a ***32-byte*** block size, while capacity vary between ***16 Kbytes*** and ***32 Kbytes***, and for the associativity, only eight ways is considered. The ***L2*** caches have at least a ***64-byte*** block size, while capacities vary between ***512 Kbytes*** and ***6 Mbytes***, and the associativity varies between two and twenty four ways. The scheduler relies on our basic-NPS scheduling policy. As a result of this experiment, see ***fig. 41 (A-1)***, we found that three 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 performance, as more processors

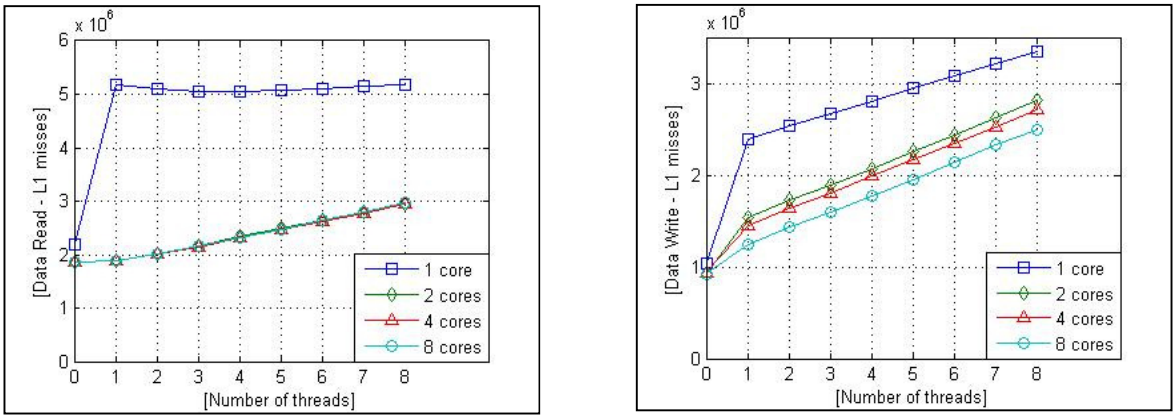
are utilized. This area is generally identified as cache-efficiency zone. At some point, the cache becomes too small for the growing stream of access requests, so memory latency is no longer masked by the cache and instruction cache misses reduce more moderately. As the number of available threads again increases, the multithread efficiency zone (on the right) is reached, where adding more threads improves performance up to the maximal performance of the machine, or up to the bandwidth wall. Balanced workloads offer higher locality and better exploit the cache and hence expand the cache efficiency zone to the right and up.

An outstanding example is given by the following table which summarizes number of ***L1*** instruction misses on ***Intel Dual Core T1400*** architecture using ***SMP*** scheduling policy and ***Basic-NPS*** scheduling policy. We note that number of instruction misses drops from an average of ***18844 L1 Instr.*** misses (using ***SMP***) with two threads down to ***6030 L1 Instr.*** misses (using ***Basic-NPS***) usually with two threads. Here success rate is largely above the average of ***50%***. The same rate will be practically maintained when increasing the number of threads.

Number of threads		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Instr. L1 misses	**Sym. Multi. Scheduler**	10298	18844	19476	18638	19726	20058	20324	18946
	Basic-NPS scheduler	3307	6030	6262	6035	6437	7202	7804	7085

(Symmetric Multiprocessing scheduler vs. Basic-NPS scheduler)

Table 21 : L2 – Instructions Misses [Topological smoothing]



(A) ***(B)***

Figure 42 : ***(A)*** Data Read ***(B)*** Data Write - L1 misses [parallel topological smoothing]

Moreover, the shape of the performance curve depends on how fast the cache hit rate degrades as a function of the number of threads. Any success access to ***L1*** will eliminate an attempt to access to ***L2*** thus performance curve, ***fig. 43 (A-1)(A-2)***, will evaluate in the same way. By reducing the number of cache miss from instruction cache, processor or thread of execution has not to wait (stall) until the instruction is fetched from main memory which immediately impact execution time.

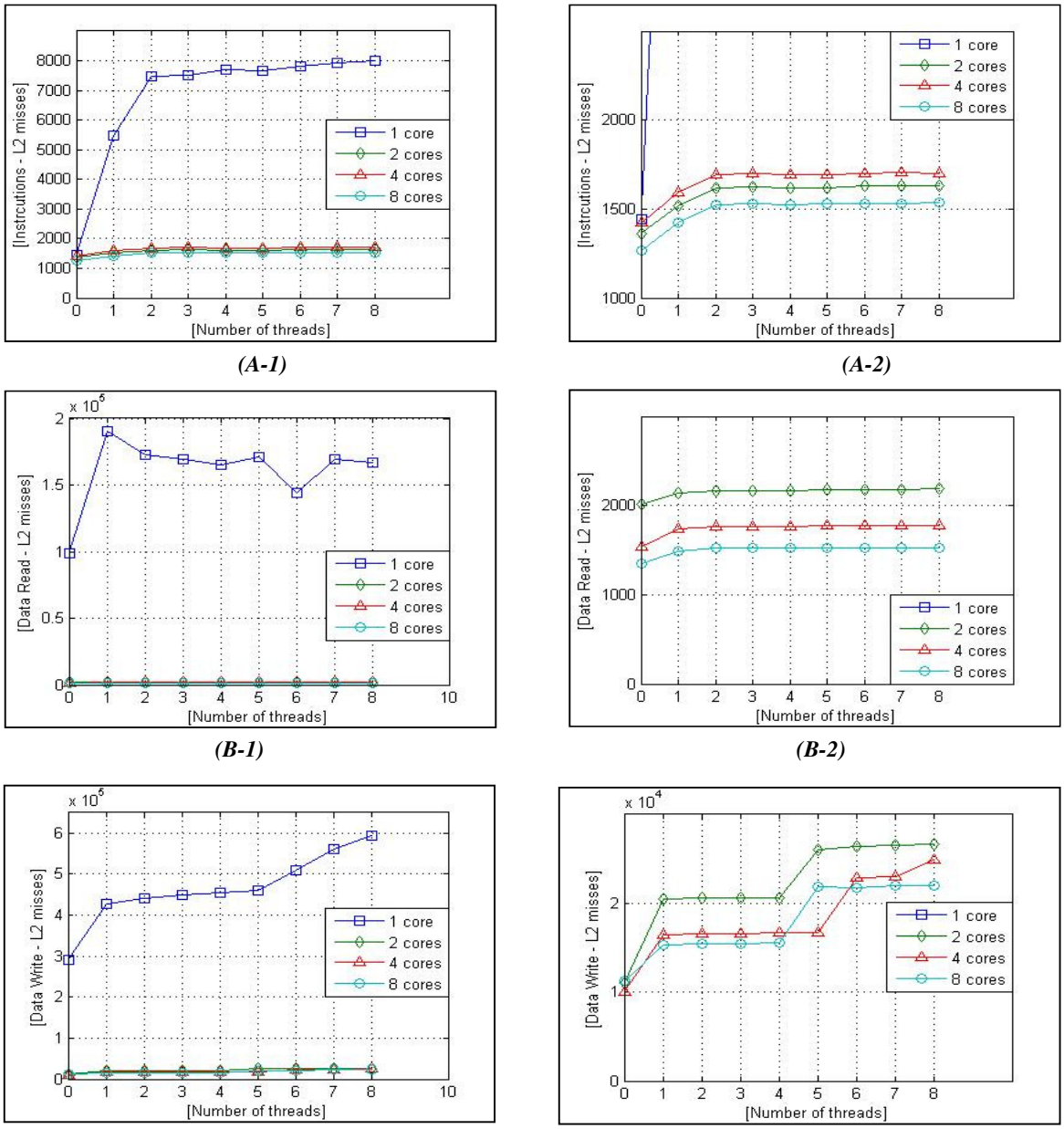


Figure 43 : ***(A-1)*** Instruction ***(B-1)*** Data Read ***(B-1)*** Data Write - L2 misses

Figures 42 (A) and ***fig. 43 (B-1)*** show so much load balancing and implicitly context switching between processes can affect performance in terms of reading data from caches. However, improvement in writing data, see ***fig. 42 (B)*** and ***fig. 43 (C-1)***, in two caches remains modest. When there are more computation instructions per memory access, performance climbs more steeply with additional threads. This is because as more instructions are available for each memory access, fewer threads are needed to fill the stall time resulting from waiting for memory.

5.4 Conclusion

In this chapter, we have presented a new parallel computation method for topological smoothing through combining parallel computation of Euclidean Distance Transform using Meijster algorithm and parallel Thinning–Thickening processes using an adapted version of Couprie's algorithm.

Introduced smoothing filter is parallel, preserve the topology and suited for ***SMP*** implementation. ***SD&M*** strategy was applied twice. First time, when computation ***E.D.T***, the splitting step starts by sharing columns and rows to scan between different processes. A thread can process one or more columns (rows). Number of threads will depend only on number of processors. Second time, when computing thinning (thickening), splitting step starts by dividing research area into different sub-region bounds. Since distribution start, each thread will lower each characterized pixel and then push its eight neighbors in available FIFO queue. Each queue contains the set of all selected neighbors and it is shared between only two threads. Unfortunately, obtained results using ***SMP*** scheduling policy, are not sufficient especially for cache consumption. For this reason, we move to ***PSN*** scheduler when distributing work. Finally we apply the same approach of fusion threads in pairs when computing ***E.D.T***, thinning and thickening.

CONLUSION

This chapter presents the conclusions of the thesis, an extended summary of the research work and restatements of contributions. Future work section summarizes the next steps to follow into the research of parallelization strategy.

6.1 Contribution

The main idea behind the present thesis is basically to identify the best approach to parallelize image processing operators based on topological transform. This approach, formally described as a strategy, was designed for shared memory parallel machines.

Parallelization strategy is the science of planning and marshalling architecture resources for their most efficient and effective use. Its main objective is to provide a mechanism to design parallel algorithm by identifying existing concurrency, modifying their structure before any further implementation (with right techniques) takes place. The idea is to take into consideration algorithm characteristics and performance issues during parallel algorithm design from the beginning. So, introduced strategy (***SD&M***) aims to give a full description of parallel topological algorithm design approach, helping parallel designers to obtain powerful and enhanced computation method of topological transforms on shared memory parallel machines. If we consider a given topological operator, the parallelization strategy describes how such algorithm can be modified with respect of the following algorithmic characteristics:

(i) Topology : structure allowing full topology preservation
(ii) Processing: structure providing parallel processing functionalities and best work distribution. During parallel processing, structure assuring coordination and communication are also used.

Thus, using ***SD&M*** strategy, a powerful parallel operator based on topological transform will be the result of restructuring and splitting original amount of work between different processor. Work distribution and processing activities take into account coordinating and communication.

In more formal way, SD&M strategy can be described as the combination of divide and conquer patterns and event-based coordination patterns hence the name that we have assigned. Note that introduced strategy represents the last stitch in the decomposition chain of algorithm design patterns and it provides a fine-grained description of topological operators' parallelization. It covers recursive algorithms and it is especially designed for shared memory architecture with uniform access. Although the cost of communication (Memory-processor and inter-processors) is high enough, shared memory architectures appear to be among adapted platforms for this type of processing. Actually, these architectures have the advantage of allowing immediate sharing of data with is very helpful in the conception of any parallelization strategy. They are non-dedicated architecture using standard component (processor, memory, buses ...) so economically

reliable and they also offer some flexibility of use in many application areas, particular image processing.

Based on this strategy, we proposed a series of parallel topological algorithm (new or adapted version). In the following we present our main contributions:

Major contribution concern a new algorithm to compute watershed that is parallel, preserves the topology of the input image, does not need prior minima extraction and suited for SMP machines. This algorithm does not require any sorting step, or the use of any hierarchical queue. Links between parallel watershed-cut and the SD&M strategy application can be described as follow: The splitting step is applied directly on input graph when selecting sources. Unlike conventional technique of division such as pixel division, or block division, the source selection is completely random. Associated steam computing is fully parallel (read mode data accesses). Then 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 contains procedures of s-labeling and f-labeling. Through these two functions, we have remained confident in our approach for merging streams two by two. Experimental analyzes such as execution time, performance enhancement, cache consumption, efficiency and scalability are also presented and discussed. Note that proposed parallel watershed-cut algorithm was preceded by an intensive study of different watershed transform in the discrete case: WT based on flooding, WT based on path-cost minimization, watershed based on topology preservation, WT based on local condition and WT based on minimum spanning forest. For each approach, we give informal definition, then we presented processing procedure followed by mathematical foundations and the algorithm of reference. Recent publications based on some approach are also presented and discussed. This study led us to classification of watershed algorithms according to criteria of recursion, complexity, basins computing and topology preservation.

Second contribution concern an adapted algorithm to compute skeleton that is parallel, preserves the topology of the input image and suited for ***SMP*** machines. Links between parallel watershed-cut and the SD&M strategy application can be described as follow: first step is dividing research area into different sub-region bounds. Since distribution start, each thread will lower each characterized pixel and then push its eight neighbors in available FIFO queue. Each queue contains the set of all selected neighbors and it is shared between only two threads. Conventional synchronization techniques such that lock-based shared FIFO queue have not given good results and performance of our algorithm remained modest. Therefore we have applied our approach based on spin-wait FIFO queue for better performance. Dynamic lambda skeleton algorithm becomes five times faster than original version proposed by Couprie and al.[8]. Tests on ***2D*** grayscale image (***512x512***), using shared memory parallel machine (***SMPM***) with ***8 CPUs*** cores (***2 × Xeon E5405*** running at frequency ***of 2 GHz***), showed an enhancement of ***6.2*** with a maximum achieved cadency of ***125 images/s*** using 8 threads. Note that proposed parallel lambda-skeleton algorithm was preceded by an intensive study of sixteen thinning algorithms in the frame work of critical kernels. We conclude this study by the classification of these algorithms according to five selection criteria: ***(i)*** preservation of topology, ***(ii)*** skeleton

connectivity, ***(iii)*** skeleton symmetry, ***(iv)*** execution time and ***(v)*** cache consumption. Through this classification, we identified Couprie's algorithm as the most suitable algorithm for parallelization on shared-memory architectures.

Third contribution concern a new parallel computation method for topological smoothing through combining parallel computation of Euclidean Distance Transform using Meijster algorithm and parallel Thinning–Thickening processes using an adapted version of Couprie's algorithm. Introduced smoothing filter is parallel, preserve the topology and suited for ***SMP*** implementation. ***SD&M*** strategy was applied twice. First time, when computing ***E.D.T***, the splitting step starts by sharing columns and rows to scan between different processes. A thread can process one or more columns (rows). Number of threads will depend only on number of processors. Second time, when computing thinning (thickening), splitting step starts by dividing research area into different sub-region bounds. Since distribution start, each thread will lower each characterized pixel and then push its eight neighbors in available FIFO queue. Each queue contains the set of all selected neighbors and it is shared between only two threads. Unfortunately, obtained results using ***SMP*** scheduling policy, are not sufficient especially for cache consumption. For this reason, we move to ***PSN*** scheduler when distributing work. Finally we apply the same approach of fusion threads in pairs when computing ***E.D.T***, thinning and thickening.

6.2 Perspectives

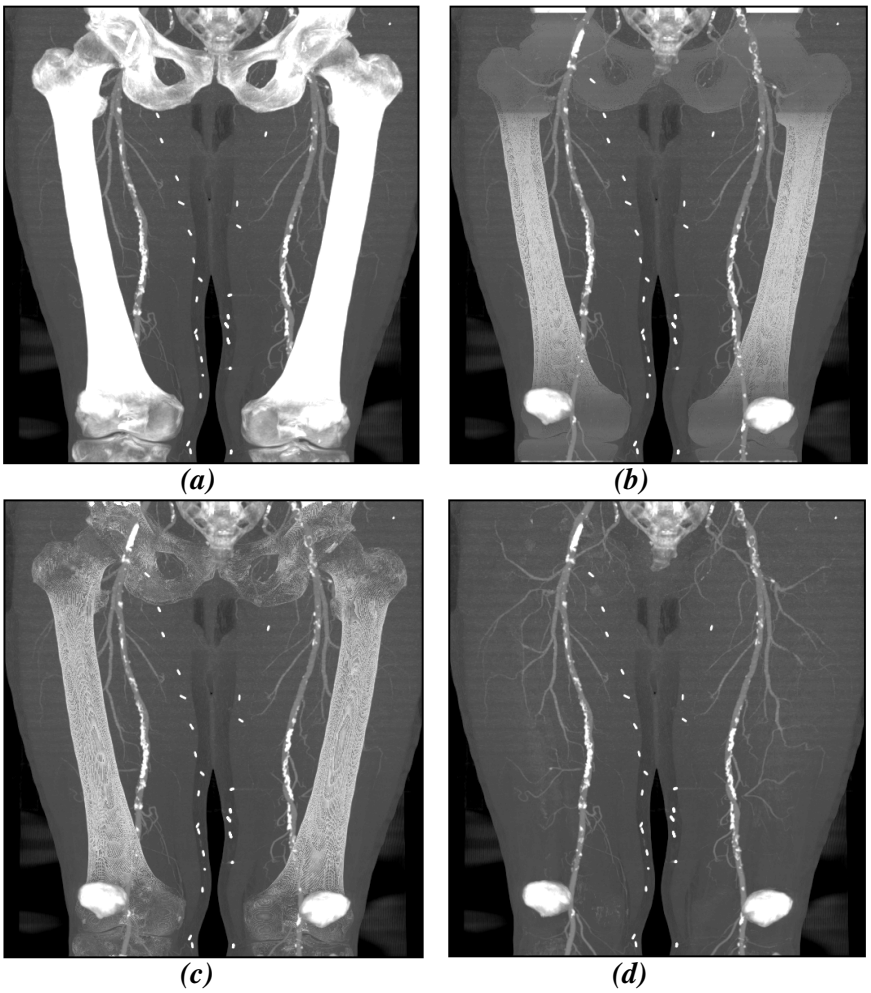
A possible extension of present work is the use of produced parallel algorithms and powerful multicore/multithread architectures to improve modern 3D medical imaging software in which segmentation procedure plays a crucial role.

Applications based on multiple detector–row computed tomography (CT) provide huge high-resolution volumetric datasets which is extremely hard to inspect without computer aided pre-processing. Straka and al. [59] work aimed at visualization and treatment planning of peripheral arterial occlusive disease by means of CT-*angiography*. In fact, for the visualization and the diagnostic assessment of vascular disease it is useful if tissues in images are segmented. To highlight data volume and complexity of such application, note that CT-*angiography* of the peripheral arteries is performed using multiple–detector row computed tomography. Thus obtained series contain about 2000 transverse images (512x512-12bit/pixels). The analysis of such data volume using currently medical workstation takes about four hours.

A first contribution to solve this problem is based on algorithmic solution proposed by Straka [59]. The basic idea of his method, for bone tissue labeling in computed tomography data, is reflected in the combined application of two techniques: a-prior knowledge derived from a density based probabilistic atlas is used to locate characteristic parts of bones and a watershed transform to identify spatially coherent sub volumes, regardless of their density. First step consist a pre-computed atlas of bone density information to assign a bone-probability to each voxel. Second step consist on partition the whole volume in homogenous regions using the

watershed transform, followed by classification of 3D watershed regions using the previously acquired bone mask. Third step, masks are deleted to cope with partial volume effect.

Even if this method is designed specifically for bone extraction (robustness and efficiency of the application is shown in ***fig. 45***), Authors [59] believe that it is applicable for a large class of software where object that might have variable densities throughout the dataset need to be identified.



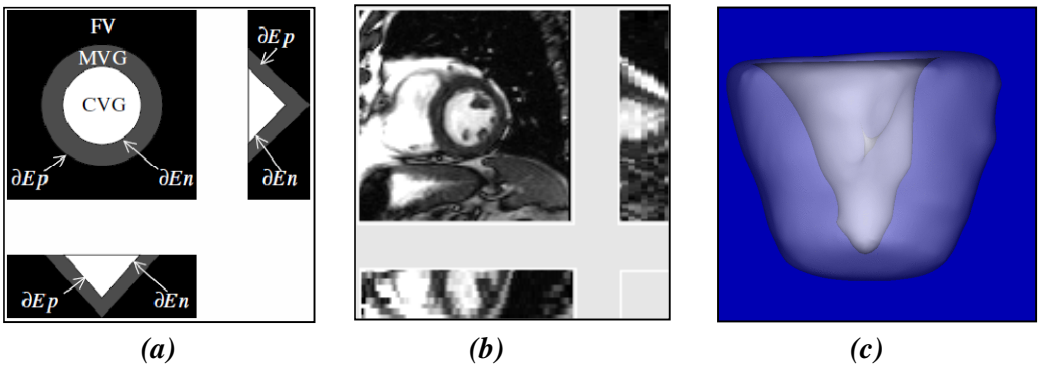
(a) Maximum Intensity Projection (MIP) of the original dataset, ***(b)*** MIP, bone tissue partially removed using mask obtained with probabilistic atlas. ***(c)*** MIP, bone tissue removed using mask extended with 3D watershed transform, ***(d)*** MIP, bone tissue removed using the dilated mask. (Patella and sacro-coccygeal bone not modeled).[59]

Figure 44 : Illustration of 3D Watershed Transform application for Medical Image Segmentation

Other application on modern 3D medical imaging turns around cardiology software. In many clinical applications, it is essential to obtain precise information on the size and the function of the left ventricle (LV). Trough 3D images acquisition at different times of the heart cycle, Magnetic Resonance (MR) imagery offers a complete and perfect morphological left ventricle characterization. High precision in extracted measures make Magnetic Resonance imagery the method of reference in left ventricle study and analysis. But like any application with high precision in medical imaging, generated volume of data is very large. Thus, segmentation of such datasets is not only complex but also very expensive.

Based on this segmentation framework, Jean Cousty [1,115] provides a new automated method to segment the left ventricular myocardium in 4D (3D+t) cine-MR images, see ***fig. 45***. For object recognition, he used Exact Euclidean distance transforms to take into account prior geometric properties and Homotopic transforms to guarantee topological soundness of the segmentations. In order to assure time continuity of the successive 3D objects, he used a watershed transform that can be applied directly on the 4D sequence considered as a whole. To this end, he considered the watershed cuts in the watershed transform, and a temporal component of a 3D+t image gradient.

In the future, we plan to study this application and revisit used algorithms: replacing the watershed algorithm by the parallel watershed-cut to assess, in practice, the contribution of this alternative.



(a) Schematic view of three orthogonal sections of the objects of interest in left ventricle images ***(b)*** Three orthogonal section of 3D MR images of the left ventricle. ***(c)*** A three dimensional rendering of these objects of interest [115]

Figure 45 : Illustration of 4D Watershed Transform application for Medical Image Segmentation

BIBLIOGRAPHY

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

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

[3] Bernard, T.M. and Manzanera, A.: Improved low complexity fully parallel thinning algorithm. Proceedings of the 10th International Conference on Image Analysis and Processing. (1999) pp. 215.

[4] Jang, B. and Chin, R.T.: Reconstructable parallel thinning. International journal of pattern recognition and artificial intelligence, vol. 7. (1993) pp. 1145-1181.

[5] Eckhardt, U. and Maderlechner G.: Invariant thinning. Edit. Inst. für Angewandte Mathematik. (1993) 37 pages.

[6] Guo, Z. and Hall, R.W.: Fast fully parallel thinning algorithms. CVGIP: Image Understanding Journal, vol. 55. (1992) pp. 317-328.

[7] Hall, R.W.: Fast parallel thinning algorithms: Parallel speed and connectivity preservation. ACM Vol. 32 (1). (1989) pp. 124-131.

[8] Couprie, M., Bezerra, F. N. and Bertrand, G.: Topological operators for grayscale image processing. Journal of Electronic Imaging, vol. 10. (2001) pp. 1003-1015.

[9] Meijster, A., Roerdink, J.B.T.M., and Hesselink, W.H.: A general algorithm for computing distance transforms in linear time. Mathematical Morphology and its Applications to Image and Signal Processing. (2000) pp. 331-340.

[10] Carriero, N., and Gelernter, D.: How to Write Parallel Programs. A Guide to the Perplexed. Yale University, Department of Computer Science, New Heaven, Connecticut. (1988)

[11] Chandy, K. M., and Taylor, S.: An Introduction to Parallel Programming. Jones and Bartlett Publishers, Inc., Boston. (1992).

[12] Darlington, J. and To, H. W.: Building Parallel Applications without Programming. Department of Computing, Imperial College. United Kingdom. In Abstract Machine Models, Leeds. (1993)

[13] Arjona, J. L. O.: Architectural Patterns for Parallel Programming, Models for Performance Estimation. University College London. (2006).

[14] Andrews, G.R.: Concurrent Programming: Principles and Practice. The Benjamin/Cummings Publishing Company. (1991).

[15] Foster, I.: Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering. Addison-Wesley Publishing Co. Reading, Massachusets. (1994).

[16] Culler, D., Singh, J. P., and Gupta, A.: Parallel Computer Architecture. A Hardware/Software Approach. Morgan Kaufmann Publishers (1997).

[17] Dijkstra, E.W.: Co-operating Sequential Processes. F. Genyus. Ed. Programming Languages. Academic Press, New York, 1968.

[18] Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM vol. 21 (8). (1978).

[19] Brinch-Hansen, P.: Distributed Processes: A Concurrent Programming Concept. Communications of the ACM, vol.21 (11). (1978).

[20] Bertrand, G., Everat, J. C., and Couprie, M.: Topological approach to image segmentation. In SPIE Vision Geometry V, vol. 2826. (1996) pp. 65-76.

[21] Bertrand, G., Everat, J. C., and Couprie, M.: Image segmentation through operators based on topology. Journal of Electronic Imaging. (1997) pp. 395-405.

[22] Couprie, M., and Bertrand, G.: Topology preserving alternating sequential filter for smoothing 2D and 3D objects. Journal of Electronic Imaging, vol. 13. (2004) pp. 720-730.

[23] Bertrand, G.,: On Topological Watersheds. Journal of Mathematical Imaging and Vision, vol. 22. (2005) pp. 217 – 230.

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

[25] Seinstra, F. J., Koelma, D., and Geusebroek, J. M.: A software architecture for user transparent parallel image processing on MIMD computers. International Euro-Par conference. (2001) pp. 653-662.

[26] Mattson, T. G., Sanders, B. A., and Massingill., B.: Patterns for parallel programming. Addison-Wesley Professional, 1st edition. (2004) 384 pages.

[27] Feldmann, S., Sgall, J., Teng, S-H.: Dynamic scheduling on parallel machines. 32nd Annual Symposium on Foundations of Computer Science. (1991) pp. 111-129

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

[29] Quinn., J. M.: Parallel Computing: theory and practice. McGraw-Hill Inc, 2nd Edition. (1994) 446 pages.

[30] Flynn, M.: Some Computer Organizations and Their Effectiveness. Computers, IEEE Transactions on, vol. C-21 (9). (1972) pp. 948-960.

[31] Steen, Aad. J., and Dongarra, Jack. J.: Overview of recent supercomputers. NCF report of Utrecht University. (2005) 90 pages.

[32] Buzbee, B. L.: Applications of MIMD machines. In Computer Physics Communications, Vol. 37(1-3). pp. 1-5.

[33] Foster, I.: Designing and Building Parallel Programs. Addison Wesley, 1st Edition. (1995) 430 pages.

[34] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann, 1st Edition. (2000) 231 pages.

[35] Reinders, J.: Intel Threading Building Blocks. O'Reilly Media Inc. (2007) 303 pages.

[36] Ramaswamy, S., Sapatnekar, S., and Banerjee, P.: A framework for exploiting task and data parallelism on distributed memory multicomputers. Parallel and Distributed Systems, IEEE Transactions on, vol.8. (1997) pp.1098-1116.

[37] Amdahl, Gene. M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conference Proceedings, Vol. 30. (1967) pp. 483–485.

[38] Gustafson, John L.,: Reevaluating Amdahl's Law. Communications of the ACM vol.31. (1988) pp. 532-533.

[39] Acar, U. A., Blelloch, G. E., and Blumofe, R. D.: The data locality of work stealing. In Proceedings of the 20th ACM symposium on Parallel algorithms and architectures. (2000) pp. 1–12.

[40] Natarajan, S.: Imprecise and Approximate Computation. Springer, 1st Edition. (1995) 200 pages.

[41] Tilborg, A. and Koob, G.M.: Foundations of Real-Time Computing: Scheduling and Resources Management. Springer, 1st Edition. (1991) 340 pages.

[42] Leung, J., and Zhao, H.: REAL-TIME SCHEDULING ANALYSIS. Report of Computer Science Dep. New Jersey Ins. (2005) 134 pages.

[43] Kolivas, C.: RSDL completely fair starvation free 64 interactive cpu scheduler. lwn.net, (March 2007).

[44] Molnar, I.: Modular Scheduler Core and Completely Fair Scheduler. lwn.net, 2007.

[45] James D. Mooney. “Strategies for Supporting Application Portability”

[46] Anderson, J. H., Ramamurthy, S., and Jeffay, k.: Real Time computing with lock-free shared Object. ACM Transaction on Computer System vol.15. (1997) pp.134 -165.

[47] Herlihy, M.: Wait-Free Synchronization. ACM transaction on programming languages and system, vol. 13. (1991) pp. 124-149.

[48] Anderson, T. E.: The performance of spin locks alternatives for shared money multiprocessors. IEEE Transactions on Parallel and Distributed Systems, Vol. 1. (1990) pp. 6-16.

[49] Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In IFIP Congress 74 Proc. North-Holland Publishing Co. (1974).

[50] Nikolov, H., Thompson, M., Stefanov, T., Pimentel, A. D., Polstra, S. Bose, R., Zissulescu, C., and Deprettere, E. F.: Daedalus: Toward Composable Multimedia MP-SoC Design. In ACM/IEEE Int Proc. Design Automation Conference. (2008), pp. 574-579.

[51] Halfhill, T.: Ambric's New Parallel Processor. Microprocessor Report. mdronline.com. (2008)

[52] Giavitto, L., and Sansonnet, P. : Introduction à 8 ½. Rapport interne LRI Orsay. (1994).

[53] Giavitto, L., and Sansonnet, P. : 8 1/2 : data-parallélisme et data-flow. Techniques et Sciences Informatiques Vol. 12. (1993).

[54] Mahiout, A., J.-L. Giavitto, L., and Sansonnet, P. : Distribution and scheduling data-parallel dataflow programs on massively parallel architectures. Software for Multiprocessors and Supercomputers, Office of Naval Research USA & Russian Basic Research Foundation. (1994).

[55] Maxwell, J.: On hills and dales. Philosophical Magazine, vol. 4(40). (1870) pp. 421-427.

[56] Beucher, S., and Lantuéjoul, C.: Use of watersheds in contour detection. International Workshop on Image Processing Real-Time Edge and Motion Detection Estimation. (1979).

[57] Beucher, S., and Meyer, F.: The morphological approach to segmentation: the watershed transformation. Mathematical Morphology in Image Processing. (1993) pp. 433-481.

[58] Serra, J.: Image Analysis and Mathematical Morphology. Academic Press. (1982) 610 pages.

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

[60] Audigier, R.: and Lotufo., R. A.: 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) pp. 277–288.

[61] Najman, L., and Couprie, M.: Watershed algorithms and contrast preservation. Lecture Notes in Computer Science, vol. 2886. (2003) pp. 62-71.

[62] Soille, P. : Morphological Image Analysis. Springer-Verlag Berlin and Heidelberg GmbH & Co. K. (1999).

[63] Dougherty, E., and Lotufo, R.: Hands-on Morphological Image Processing. SPIE Publications. (2003).

[64] Najman, L., Couprie, M., and Bertrand, G.: Watersheds, mosaics and the emergence paradigm. Discrete Applied Mathematics, vol. 147. (2005) pp. 301-324.

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

[66] Shengcai, P., and Lixu., G.: A Novel Implementation of Watershed Transform Using Multi-Degree Immersion Simulation. In 27th Annual International Conference of the Engineering in Medicine and Biology Society. (2005) pp. 1754 – 1757.

[67] Meyer, F.: Topographic distance and watershed lines. Signal Processing, vol. 38. (1993) pp. 113-125.

[68] Najman, L., and Schmitt, M.: Watershed of a continuous function. Signal Processing, vol. 38. (1993) pp. 68-86.

[69] Falcão, A. X., Stolfi, J., and Lotufo, R. A.: The Image Foresting Transform: Theory, Algorithms, and Applications. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26. (2004) pp.19-29.

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

[71] Dijkstra, E.W.: A Note on Two Problems in Connection with Graphs. Numeriche Mathematik, vol 1. (1959) pp. 269-271.

[72] Pape, U.: Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem. Mathematical Programming, vol. 7. (1974) pp. 212-222.

[73] Dial, R. B., Glover, F., Karney, D., and Klingman, D.: A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. In Networks Journal, vol 9. (1979) pp. 215-248.

[74] Lotufo, R.A., Falcão, A. X., and Zampirolli, F. A.: IFT-Watershed from Gray-Scale Marker. 15th Brazilian Symposium on Computer Graphics and Image Processing, Vol 2. (2002).

[75] Couprie, M., and Bertrand, G.: Topological grayscale watersheds transform. SPIE Vision Geometry V Proceedings, vol. 3168. (1997) pp. 136-146.

[76] Couprie, M., Najman, L., and Bertrand, G.: Quasi-linear algorithms for the topological watershed. Journal of Mathematical Imaging and Vision, vol. 22. (2005) pp. 231-249.

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

[78] Mortensen, E.N., and Barrett, W.A.: Toboggan-based intelligent scissors with a four-parameter edge model. In IEEE Conf. Computer Vision and Pattern Recognition. (1999) pp. 452–458.

[79] Meyer. F.: Minimum Spanning Forests for Morphological Segmentation. Proc. 2nd International Conference on Math. Morphology. and Its Applications to Image Processing. (1994) pp. 77-84.

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

[81] Fruehe J.: Planning Considerations for Multicore Processor Technology. Dell Power Solutions. (2005).

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

[83] Khalimsky, E.: Topological structures in computer science. J. Appl. Math. Simulation vol. 1 (1). (1987) pp. 25-40.

[84] Jain, Anil K.: Fundamentals of Digital Image Processing. Prentice-Hall 1st Edition. (1988).

[85] Arcelli, C., Cordella, L.P., and Levialdi., S.: Parallel thinning of binary pictures. In Electronic Letters, vol. 11(7). (1975) pp. 148-149.

[86] Bernard, T. M. and Manzanera, A.: Improved low complexity fully parallel thinning algorithm. 10th International Conference on Image Analysis and Processing. (1999).

[87] Chin, R. T., Wan, H. K., Stover, D. L., and Iverson, R. D.: A one-pass thinning algorithm and its parallel implementation. Computer Vision, Graphics, and Image Processing Journal, 40(1). (1987) pp. 30-40.

[88] Hall, R. W.: Fast parallel thinning algorithms: Parallel speed and connectivity preservation. Communications of the ACM Magazine, 32(1). (1989) pp 124-131.

[89] Couprie, M.: Note on fifteen 2d parallel thinning algorithms. UMLV Internal Report, IGM. (2005).

[90] Pavlidis, T.: An asynchronous thinning algorithm. Computer Graphics and Image Processing, vol. 20(2). (1982) pp. 133-157.

[91] Rutovitz, D.: Pattern recognition. Journal of the Royal Statistical Society. (1966) pp. 504-530,.

[92] Pavlidis, T.: A flexible parallel thinning algorithm. In Proc. IEEE Comput. Soc. Conf. Pattern Recognition, Image Processing. (1981) pages 162-167.

[93] Holt, C.M., Stewart, A., Clint, M., and Perrott, R.H.: An improved parallel thinning algorithm. Communications on ACM, 30(2). (1987) pp.156-160.

[94] Zhang, Y.Y., and Wang, P.S.P.: A modified parallel thinning algorithm. In Pattern Recognition, 9th International Conference on, (1988) pp. 1023-1025.

[95] Wu, R.Y., and Tsai, W.H.: A new one-pass parallel thinning algorithm for binary images. Journal of pattern recognition letter. (1992) pp. 715-723.

[96] Guo, Z., and Hall, R.W.: Fast fully parallel thinning algorithms. CVGIP Image Understanding, vol.55(3). (1992) pp. 317-328.

[97] Jang , B.K., and Chin, R.T.: One-pass parallel thinning: Analysis, properties, and quantitative evaluation. Pattern Analysis and Machine Intelligence, IEEE Transactions on. (1992) pp.1129-1140.

[98] Jang, B., and Chin, R.T.: Reconstructable parallel thinning. International journal of pattern recognition and artificial intelligence Journal, vol. 7. (1993) pp. 1145-1181.

[99] Eckhardt, U., and Maderlechner. G.: Invariant thinning. Int. J. Pattern Recogn. Artif. Intell. Vol. 7. (1993) pp. 1115-1144.

[100] Choy, S.S.O., Choy, C.S.T., and Siu, W.C.: New single-pass algorithm for parallel thinning. Journal Computer Vision and Image Understanding vol. 62(1). (1995) pp. 69-77.

[101] Jang, B. K., and Chin, R. T.: One-pass parallel thinning: Analysis, properties and quantitative evaluation. Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 14(11). (1992) pp 1129-1140.

[102] Taubin, G.: Curve and surface smoothing without shrinkage. Fifth International Conference on Computer Vision. Proceedings. (1999) pp. 852-857.

[103] Liu, X., Bao, H., Shum, H-Y. and Peng, Q.: A novel volume constrained smoothing method for meshes. Graphical Models, vol.64. (2002) pp. 169-182.

[104] Asano, A., Yamashita, T. and Yokozeki, S.: Active contour model based on mathematical morphology. Fourteenth International Conference on Pattern Recognition, Proceedings. (1998) pp. 1455-1457.

[105] Leymarie, F. and Levine, M.D.: Curvature morphology. Proceedings of Vision Interface. (1989) pp.102-109.

[106] Yung Kong, T. and Rosenfeld, A.: Digital topology: introduction and survey. Computer Vision, Graphics and Image Processing, Vol. 48, (1989) pp. 357-393.

[107] Sternberg, S. R.: Grayscale Morphology", Computer Vision Graphics and Image Understanding, vol. 35. (1986) pp. 333-355.

[108] Bertrand, G.: Simple points, topological numbers and geodesic neighborhoods in cubic grids. Pattern Recognition Letters, vol.15. (1994) pp. 1003-1011.

[109] Danielsson, P.E.: Euclidean distance mapping. Computer Graphics and Image Processing vol. 14. (1980) pp. 227-248.

[110] Lemire, D.: Streaming Maximum-Minimum Filter Using No More than Three Comparisons per Element. Nordic Journal of Computing, vol.13(4). (2006) pp 328-339.

[111] Shih, F. Y. and Wu, Y.: Fast Euclidean distance transformation in two scans using a 3x3 neighborhood. Computer Vision and Image Understanding, no. 94. (2004) pp. 195-205.

[112] Cuisenaire, O. and Macq, B.: Fast Euclidean Distance Transformation by Propagation Using Multiple Neighborhoods. Computer Vision and Image Understanding vol.76(2), (1999) pp. 163-172.

[113] Cuisenaire, O. and Macq, B.: Fast and exact signed Euclidean distance transformation with linear complexity. IEEE Intl Conference on Acoustics, Speech and Signal Processing. (1999) pp. 3293-3296.

[114] Fabbri, R., Costa, L. F., Torelli, J. C. and Bruno, O M.: 2D Euclidean distance transform algorithms: A comparative survey. ACM Computing Surveys vol.40. (2008).

[115] Cousty, J., Najman, L., Couprie, M., Clément-Guinaudeau, S., Goissen, T., and Garot. J.: "Automated, Accurate and Fast Segmentation of 4D cardiac MR images". F.B. Sachse and G. Seemann (Eds), Procs. of Functional Imaging an Modeling of the Heart. (2007) pp. 474-483.