Algorithmes d'étiquetage en composantes connexes efficaces pour architectures hautes performances

Laurent Cabaret

NNT : 2016SACLS299

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À L'UNIVERSITÉ PARIS-SUD

Ecole doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique
par

M. LAURENT CABARET

ALGORITHMES D'ÉTIQUETAGE EN COMPOSANTES CONNEXES EFFICACES POUR ARCHITECTURES HAUTES PERFORMANCES

Thèse présentée et soutenue à Gif-Sur-Yvette, le 28 Septembre 2016.

Composition du Jury :

M.	S. CONCHON	Professeur Univ. Paris-Sud, LRI	(Président du jury)
M.	D. ETIEMBLE	Professeur Émérite Univ. Paris-Sud, LRI	(Examinateur)
M.	L. LACASSAGNE	Professeur Univ. Paris 6, LIP6	(Directeur de thèse)
Mme	A. MONTANVERT	Professeur Univ. de Grenoble, GIPSA-lab	(Rapporteur)
M.	O. SENTIEYS	Professeur ENSSAT/IRISA Lannion	(Examinateur)
M.	S. SIMON	Professeur Univ. de Stuttgart	(Examinateur)
M.	H. TALBOT	Professeur ESIEE Paris, A2SI	(Rapporteur)

Remerciements

Cette thèse est pour moi un accomplissement personnel que j'avais dû mettre entre parenthèses il y a 15 ans du fait de la santé de mon père. Ainsi à presque 40 ans, la liste de ceux qui m'ont permis d'arriver dans de bonnes conditions jusqu'à la soutenance de ces travaux est plutôt longue et il est certain que mes remerciements seront incomplets. D'avance, veuillez m'en excuser.

Tout d'abord rien n'aurait été possible sans la confiance que m'a accordé mon directeur de thèse monsieur Lionel Lacassagne qui a répondu présent lorsque, le 25 mai 2011, il a reçu l'étrange appel de quelqu'un souhaitant faire une thèse tardive. Il n'est jamais simple de parier sur un parcours tel que le mien et il a su faire preuve d'une implication sans faille, d'un grand sens de la pédagogie pour former à l'informatique l'électronicien que je suis et enfin d'un sens aigu du détail qui a donné toute sa valeur à mon travail. Un grand merci à toi Lionel !

J'exprime ma profonde gratitude à madame Annick Montanvert et monsieur Hugues Talbot pour avoir accepté d'être les rapporteurs de cette thèse et de m'avoir permis par leur remarques constructives de devenir un meilleur chercheur. Je tiens à remercier très chaleureusement, monsieur Sylvain CONCHON pour avoir accepter de présider le jury de thèse, ainsi que messieurs Daniel Étiemble, Olivier Sentieys et Sven SIMON (Danke !) de m'avoir fait l'honneur d'en être membre.

Je tiens à remercier une nouvelle fois monsieur Daniel Étiemble pour sa bienveillance, son implication, sa confiance et ses conseils toujours avisés tant scientifiques que syntaxiques tout au long de mes travaux : travailler avec toi est une expérience inoubliable.

Bien entendu, rien n'aurait été possible sans l'implication de l'École Centrale Paris (d'abord) et de CentraleSupélec (ensuite). Que ce soit les encouragements des collègues et amis pour démarrer ces travaux (Céline, Pascale, Marc, Paolo, Frédéric), le soutien sans faille de messieurs John Cagnol et Lionel Gabet qui m'ont permis d'aménager mon service pour réaliser sérieusement mes travaux de recherche, ainsi bien entendu que toute l'équipe du LISA (Didier, Hanane, Malika, Philippe) qui a pris beaucoup sur elle pour continuer à faire vivre et à développer le laboratoire avec succès.

Je tiens aussi à remercier l'ensemble du LRI pour son accueil chaleureux et plus particulièrement l'équipe Systèmes Parallèles et ses satellites (Andrea, Adrien, Andrès, Antoine, Amal, Christine, Farouk, Ian, Janna, Joël, Joffroy, Katia, Lenaïc, Louiza, Marc, Marie, Sylvain…). Un remerciement très spécial à Stéphanie pour son efficacité et sa bonne humeur et pour m'avoir si souvent aidé à comprendre (et franchir) les subtilités administratives d'une thèse.

Durant ces quatre années, ma famille m'a soutenu sans faillir : Mes remerciements ne seront jamais à la hauteur de ce que j'ai reçu de leur part. Ma mère Michèle et mon père René (qui nous a quitté il y a 15 ans maintenant) qui en plus de leur amour m'ont montré par leur exemple ce qu'était le véritable courage face à la vie et face à la mort. Mon frère Jérôme, ma chère belle-sœur Lætitia ainsi que mes neveux et nièce (Yohan, Laura, Noah) pour leur joie de vivre. Ma femme Christine, qui est le courage, la gentillesse et l'écoute incarnée, capable de porter les soucis des autres et de sourire encore. Je ne saurais en quelques lignes résumer ce que je te dois alors un mot suffira : tout. C'est à ton tour

maintenant de réaliser tes rêves et je serai là pour toi. Mes deux soleils, Manon et Titouan, pour qui le mot thèse doit signifier absence, je veux que vous gardiez toute votre vie à l'esprit que c'est avec et grâce à vous que votre père en arrivé là et qu'il n'est jamais trop tard pour réaliser ses rêves.

Enfin les amis (Nue t'es la, Dorian Boys et les Cachanais) je ne vous oublie pas. Je suis la somme de toutes nos rencontres et même si j'ai dû être lointain pendant ces années vous étiez tous dans mon cœur .

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction

Le traitement d'images pour les machines intelligentes

Montée en puissance des intelligences artificielles

> «Mon intention n'est pas de vous surprendre ou de vous choquer, mais la manière la plus simple de résumer les choses consiste à dire qu'il existe désormais des machines capables de penser, d'apprendre et de créer. En outre, leur capacité d'accomplir ces choses va rapidement s'accroître jusqu'à ce que, dans un futur proche, le champ des problèmes qu'elles pourront aborder soit coextensif à celui auquel s'applique l'esprit humain.»

Cette citation (relevée dans [1]) de 1957 d'Herbert Simon (1916-2001), lauréat du prix Turing en 1975 et du prix Nobel d'économie en 1978, prédisait que la machine serait rapidement l'égal de l'humain dans de nombreux domaines et notamment dans celui de la prise de décision qui était une de ses spécialités. Si l'horizon de sa prédiction a été plus lointain que prévu, il n'en est pas moins vrai que les machines occupent aujourd'hui une place centrale dans les domaines économiques, éducatifs et plus généralement dans notre vie de tous les jours :

- la voiture autonome ou «intelligente» existe déjà [2, 3], et sa diffusion pose plus de problèmes au niveau des infrastructures [4], de la législation et des mentalités [5, 6] qu'au niveau des techniques des systèmes embarqués,
- nos poches contiennent des assistants personnels capables de comprendre (partiellement) notre langage, d'anticiper nos demandes et de nous suggérer ce que nous allons manger, avec qui le faire et dans quel lieu [7].

Ces évolutions récentes reposent sur la conjonction de deux phénomènes : les progrès algorithmiques en intelligence artificielle et la massification des données.

Rôle de la vision artificielle

La vision artificielle, comme les autres branches de l'intelligence artificielle, participe à ces progrès en apportant aux machines une information sur leur environnement. Son impact est notable dans des domaines variés tels que :

- le domaine médical, où le diagnostique assisté par ordinateur permet d'aider le praticien par la mise en évidence d'irrégularités dans des résultats d'imagerie [8, 9],
- les applications industrielles, où le but est d'extraire de l'information pour automatiser et accélérer des processus fastidieux pour les humains [10, 11].
- l'accès interactif à la culture, avec la mise à disposition d'une bibliothèque d'Alexandrie numérique rendue possible par la reconnaissance de caractères (O.C.R) que ce soit pour les caractères imprimés [12] ou manuscrits [13] et même pour les œuvres musicales les plus anciennes (O.M.R) [14],
- l'exploration spatiale, où les délais et la variabilité des communications rendent nécessaire l'autonomie décisionnelle des machines d'explorations [15],

- l'interaction homme-machine, où le contrôle par gestes et l'utilisation de capteurs dédiés ont nécessité la création et l'adaptation de nombreux algorithmes de vision [16],
- la biométrie [17] et la sécurité par le biais du suivi de personnes [18, 19] de la détection de mouvements [20, 21] et de l'analyse de scènes [22] sont des domaines en forte expansion, du fait du contexte sécuritaire actuel, qui nécessitent toujours plus de précision et de rapidité,
- la réalité augmentée dont le champ d'application se développe tant du point de vue médical, industriel que ludique [23].

Enjeu de la massification des données

Dans un monde où la communication visuelle tient le premier plan, pouvoir analyser rapidement des images fixes ou des séquences vidéos pour en extraire de l'information est un enjeu important. La quantité d'images générées et stockées numériquement dans le monde croît en effet exponentiellement. Quelques chiffres permettent de prendre conscience de l'ampleur du phénomène :

- une plateforme dédiée à la photographie (Instagram) s'enrichit chaque jour de 70 millions de photographies et dispose déjà d'une base de 40 milliards d'images,
- le réseau social Facebook dispose d'une base supérieure à 240 milliards d'images qui progresse au rythme de 350 millions par jour (chiffres 2013),
- les bases de données scientifiques ne sont pas en reste et ImageNet [24] propose plus de 14 millions d'images classées en 21841 catégories sémantiques (synsets).

Dans le même temps, la taille des images générées progresse avec la résolution des capteurs comme par exemple les caméras de vidéo-surveillance récente qui ont couramment une résolution HD1080 (1920×1080) à comparer à la résolution standard 4CIF(704×576) et la résolution 7K (7360×4128) qui représente le haut de gamme actuel. Le cas des aéroports de Paris (groupe ADP) illustre cette augmentation des données et du besoin de les traiter toujours plus rapidement. En 2001, la fréquentation des aéroports du groupe ADP était de 71 millions de passagers [25] et est passée à 95,4 millions en 2015 [26]. Dans le même temps, du fait de l'augmentation constante des processus visant à garantir la sécurité des passagers, le nombre de caméras de surveillance est passé de 1000 à 9000 unités et le groupe ADP est actuellement en phase de test de systèmes de reconnaissance faciale [27]. L'objectif d'un tel système est d'avertir le plus rapidement possible (de l'ordre de la minute) de la présence d'un individu recherché afin de faire intervenir le personnel de sécurité.

D'une manière générale, la vision artificielle doit aujourd'hui traiter toujours plus d'images, de plus grande taille, plus rapidement et au plus près de l'utilisateur final du fait de la diversification des supports et des applications. Les algorithmes de traitement d'images doivent donc être agiles et s'adapter aux architectures modernes, de l'assistant personnel (smartphone, tablette) aux architectures les plus parallèles en passant par l'ordinateur de tout un chacun.

Architectures pour la vision artificielle

Depuis les début de la vision artificielle, plusieurs approches architecturales coexistent. Que l'architecture consomme quelques watts ou plusieurs centaines de watts, la performance réside dans l'exploitation maximale de capacités de calculs et de bande passante mémoire de celle-ci. Le développement d'un algorithme efficace passe donc par la prise en compte des mécanismes spécifiques à chaque architecture, des plus généralistes aux plus spécialisées.

- Les processeurs généralistes (GPP [1]) qui équipent le ordinateurs de bureau, les stations de travail, les serveurs les plus courants mais aussi les téléphones mobiles et les tablettes, sont très représentés dans la littérature sur les algorithmes de traitement d'images du fait du grand nombre

1. General Purpose Processor

d'outils de conception supportés et du faible coût des plateformes au regard de leurs performances. De plus, chaque progrès dans l'écosystème de ces architectures est directement bénéfique aux algorithmes de traitements d'images. Ainsi, les progrès réalisés sur les compilateurs, les langages, les mécanismes de caches et les jeux d'instructions ont un impact direct sur la simplicité de développement et les performances des algorithmes de traitement d'images. A l'inverse, l'évolution des architectures généralistes vers un modèle comportant toujours plus de cœurs nécessite de repenser les algorithmes pour exploiter au mieux la puissance des nouveaux modèles de processeurs. Enfin, leur polyvalence permet de les utiliser à la fois pour les algorithmes de bas niveau et ceux de plus haut niveau [28]. Pour des algorithmes spécifiques, les architectures dédiés seront plus efficaces mais nécessiterons dans la plupart des cas de transférer *in fine* les données obtenues vers des architectures généralistes [29].

- Les processeurs graphiques ont été conçu comme des coprocesseurs spécialisés dans le rendu d'images de synthèse. Ils intègrent aujourd'hui plusieurs milliers d'unités de traitements spécialisées organisées pour réaliser de courtes séquences d'instructions identiques sur une grande quantité de données. Les applications nécessitant du calcul régulier avec une intensité arithmétique élevé tels que le filtrage sont très favorablement accélérées sur *GPGPU*[2] et proposent un gain du point de vue de l'efficacité énergétique. A contrario les algorithmes irréguliers qui nécessitent l'utilisation intensive de structures de contrôle sont inefficaces sur ce type d'architectures. Les algorithmes doivent donc être pensées spécifiquement pour ces architectures [30]. Les *GPGPU* restent dépendants d'un système hôte utilisant une architecture généraliste et les temps de transfert très significatifs nécessaires au chargement des données dans la mémoire du processeur graphique et à la récupération des résultats doivent être pris en compte pour une comparaison équitable [31].
- Des architectures spécialisées ont été développées pour faire face aux besoins élevés de puissance de calcul, notamment dans le cadre du traitement d'images, afin atteindre des performances temps-réel. Par exemple, des architectures de type maille permettent d'associer à chaque processeur élémentaire un pixel, une ligne ou un bloc selon les paramètres de l'algorithme. Un des avantages de la structure en mailles est que sa structure régulière préserve les relations spatiales des images pixelisées [32]. Un exemple de structure en maille utilisée pour la vision par ordinateur est la Maille Associative d'Orsay [33]. Basée sur une structure asynchrone, elle permet d'atteindre les performances temps réel nécessaires pour traiter des flux vidéos [34]. Les DSP, les architectures Tilera [35], CELL [36], TSAR (TeraScale Architecture) [37–39] sont d'autres exemples d'architectures spécialisées dont le traitement d'images peux tirer parti. La complexité de programmation de ces architectures, relativement aux architectures généralistes, peut-être levée par l'utilisation de langages dédiés [40].
- Les architectures dédiées conçues pour répondre aux spécificités d'un algorithme particulier, sont généralement implémentées sur FPGA [41] afin d'optimiser finement les ressources en choisissant le juste niveau de parallélisme. Les blocs élémentaires ainsi conçus peuvent être chaînés physiquement, si les ressources du FPGA le permettent, ou temporellement dans le cas des architectures reconfigurable dynamiquement telles qu'Ardoise [42] mais généralement au prix d'une perte de spécialisation. Les avantages de ces architectures dédiées sont : la possibilité d'optimiser la consommation énergétique en ajustant les ressources aux besoins de l'algorithme [43], la possibilité d'utiliser ces développements dans des ASIC poussant ainsi plus loin encore l'optimisation énergétique et le couplage fin avec des processeurs dans le cas des PSoC (Programmable System on Chip). Par exemple dans [44] où une détection de points d'intérêts est réalisé dans une section FPGA et exploitée par le processeur bi-cœur Cortex-A9 embarqué du PSoC Zynq. Un des principaux inconvénient du développement dédié sur FPGA est le haut

2. General Purpose computing on Graphics Processing Unit

niveau de spécialisation nécessaire des concepteurs, associé au temps de développement particulièrement élevé. La génération automatique de code par le biais de la synthèse haut-niveau (HLS) permet de réduire ce temps de développement, pour les applications compatibles, au prix d'une élévation de la consommation. Cette surconsommation peut-être alors modérée par l'utilisation de transformations au niveau algorithmique [45–47].

Dans le cadre des travaux présentés dans ce manuscrit, les algorithmes ont été implémentés et évalués sur des processeurs généralistes et des processeurs graphiques utilisés pour le calcul.

L'étiquetage en composantes connexes

Contexte applicatif

Le but de la vision artificielle est de permettre à un système informatique d'analyser des images pour les «comprendre». Dans ce cadre, le traitement des composantes connexes joue un rôle de pont entre les algorithmes de bas niveau, dont l'objectif est la mise en valeur d'une caractéristique de l'image (filtrage, mise en valeur des contours ou des régions, extraction de zones d'intérêt, ...) et les algorithmes de plus haut niveau, chargés d'analyser les données ou de prendre des décisions. Les travaux présentés dans ce manuscrit portent sur l'étiquetage en composantes connexes des images binaires en deux dimensions qui sont une des étapes clefs d'un grand nombre d'applications et d'algorithmes de vision artificielle. Cette famille d'algorithmes s'inscrit dans le cadre plus général du traitement des composantes connexes qui recouvre différentes formes selon que l'objet à traiter est une image 2D, 3D, binaire, en niveau de gris ou en couleur, ou selon l'objectif recherché. En plus de l'étiquetage, on trouve en effet : la décomposition en arbre de composantes connexes et la segmentation par composantes connexes. Le traitement des composantes connexes est ainsi présent dans l'ensemble du champ d'application de la vision artificielle.

- Dans le cas d'une application automobile embarquée, il s'agit pour le système de comprendre son environnement pour prendre des décisions avec une précision suffisante et dans le respect des contraintes temporelles. Dans [3], les auteurs utilisent l'étiquetage en composantes connexes après l'étape de création d'une carte des pixels en mouvement, pour produire une liste des régions en mouvement. Un filtrage basé sur la logique floue est ensuite appliquée à cette nouvelle liste.
- Dans le cadre de la reconnaissance optique de caractères, la description du moteur libre Tesseract [12] qui traite des images déjà binarisées, indique que l'étiquetage en composantes connexes intervient dès la première étape pour fournir au moteur les contours des caractères.
- Dans le domaine médical, il est nécessaire d'isoler des régions d'intérêt dans des images 2D ou 3D. Dans [48], les auteurs appliquent une structure de données hiérarchique des composantes connexes, appelée arbre de composantes, à des images 3D issues de la tomoscintigraphie par émission de positons (PET). L'utilisation de cet arbre permet une segmentation robuste et en temps réel.
- Dans le domaine de la biométrie, l'identification d'individus par vue de profil présentée dans [17], utilise l'étiquetage pour compter et filtrer les composantes connexes en fonction de leur taille.
- Dans le domaine de la vidéosurveillance, la recherche de composantes connexes peut se faire sur l'image segmentée mais aussi sur d'autres informations. Dans [18], l'étiquetage en composantes connexes est appliquée à l'information de mouvement pour analyser les zones de plus fortes variations.

- Dans le domaine de l'interaction homme-machine, [16] utilise l'étiquetage après une phase de segmentation basée sur la profondeur des pixels mesurée avec un capteur dédié (Kinect) pour trier les régions par taille afin d'identifier les différents éléments de la main de l'utilisateur.
- Dans un cadre industriel, [49] présente une application de mesure en temps réel des caractéristiques d'un nuage de gouttes de peinture. Dans ce cadre, l'analyse des composantes connexes permet d'extraire les caractéristiques de chaque goutte.

L'étiquetage en composantes connexes peut aussi être intégré dans un autre algorithme :

- l'algorithme de Swedsen-wang, utilisé pour la détection de transition de phase dans les simulations de ferromagnétisme, utilise à chaque itération un double étiquetage du fond et du premier plan et ceci pour des images 2D comme les images 3D [50].
- un algorithme de suivi, basé sur le filtrage de Kalman et le mélange de gaussienne [51], utilise l'étiquetage en composantes connexes pour fournir au filtre de Kalman des informations statistiques sur les régions (objets) mises en évidence dans la phase de mélange de gaussienne.

Dans la suite du manuscrit, la notion d'étiquetage en composantes connexes fera référence exclusivement à l'étiquetage des images binaires en deux dimensions.

Enjeu de la rapidité

Le critère de base de l'étiquetage en composantes connexes étant la connexité des pixels, tous les algorithmes basés sur le même critère de connexité doivent donner le même résultat. La forme de ce résultat peut varier selon les applications (image étiquetée ou description synthétique des composantes connexes), mais deux pixels connexes dans l'image à étiqueter seront toujours dans la même composante connexe. Il en découle que ce qui différencie les algorithmes entre eux n'est pas le résultat mais le temps mis à l'atteindre. Accélérer l'étape d'étiquetage permet d'une part d'accélérer automatiquement tous les algorithmes basés sur ce mécanisme et d'autre part de proposer un complément cohérent pour traiter les images issues de segmentations rapides [52]. Évaluer un algorithme d'étiquetage en composantes connexes, c'est donc avant tout mesurer sa rapidité, qui dépend aussi des caractéristiques intrinsèques du matériel utilisé.

Depuis sa création en 1966, le domaine a vu naître de nombreux algorithmes et a fait l'objet, depuis le début des années 2000, de nombreuses publications spécifiquement dédiées aux implémentations sur architectures généralistes [53–64]. Ces algorithmes récents sont tous décrits comme étant les plus rapides, mais les méthodologies de tests hétéroclites rendent les comparaisons délicates. De plus, ils sont majoritairement séquentiels et ne sont donc plus adaptés aux architectures modernes qui sont majoritairement multi-cœurs. En 2000, Lionel Lacassagne, directeur de cette thèse, a créé un algorithme d'étiquetage en composantes connexes novateur, car pensé spécifiquement pour les architectures RISC. Cet algorithme nommé *Light Speed Labeling* (LSL) était alors le plus rapide sur ce type d'architecture. L'évolution des architectures et l'apparition de nouveaux algorithmes nous a poussés à réévaluer sa pertinence. C'est le point de départ de nos travaux.

Architectures pour l'étiquetage en composantes connexes

En plus des architectures généralistes, qui feront l'objet de la majorité des résultats présentés dans ce manuscrit, des algorithmes ont été proposés spécifiquement pour d'autres types d'architecture.

- Les architectures *GPGPU* s'accommodent difficilement du caractère irrégulier des algorithmes directs et il est nécessaire de penser spécifiquement les algorithmes en régularisant au maximum les traitements. La section 6.4.9.1 présentera les travaux dans ce domaine.

- Les architectures spécialisées de type réseau associatif, ayant un modèle de construction très adaptés aux images en deux dimensions, présentent l'avantage d'utiliser une topologie similaire à celle des algorithmes d'étiquetage en résolvant un problème global par une série de traitements locaux. Ainsi dans [65], les auteurs ont recensé les algorithmes dédiés et les exemples d'implémentations sur des architectures de type maille. Leur grande diversité, illustre l'affinité entre l'étiquetage en composantes connexes et la topologie en maille (au sens large). Des algorithmes ont été proposés pour :
  - les mailles 2D de même dimensions que l'image,
  - les mailles 2D de dimensions inférieures à l'image, pour lesquels chaque nœud traite plusieurs pixels, ce qui accélère les traitements en limitant les communications,
  - les mailles reconfigurables, pour lesquels l'activation des nœuds est conditionnée par la nature (fond ou premier plan) du pixel correspondant,
  - les architectures pyramidales (empilement de mailles de taille décroissante),
  - les mailles d'arbres (MOT).

  Enfin, la Maille Associative d'Orsay (asynchrone) détenait, jusqu'à nos travaux, le record en terme de débit d'étiquetage soit 40Gp/s [33] (nombre de pixels traités par seconde).
  Des implémentations pour des architectures basées sur d'autres modes d'interconnexions ont aussi été proposées : les *hypercubes* [65] (Connexion Machine 1 et 2) ainsi que les *fat tree* (Connexion Machine 5) [66].
- Les architectures dédiées, à base de FPGA, sont régulièrement utilisées pour réaliser une unité de traitement dédié à l'étiquetage en composantes connexes dans le cadre de systèmes embarqués nécessitant des performances temps réel [67]. Elles sont insérées après l'unité de traitement bas niveau pour soulager le système principal. Les algorithmes classiques nécessitent beaucoup de mémoire rapide et majoritairement double port. La rareté relative de cette mémoire et la nécessité de réduire le chemin critique afin d'augmenter la fréquence de fonctionnement ont poussé au développement d'algorithmes spécifiques aux FPGA. Dans [68], les auteurs présentent un algorithme en une passe qui ne produit pas l'image étiquetée mais seulement les descripteurs des différentes composantes connexes ce qui lui permet de réduire les besoins en termes de mémoire. Des travaux ultérieurs ont permis d'abaisser encore la quantité de mémoire nécessaire [69] ou la quantité de ressource matérielle [70]. Afin de réduire la complexité de conception, il est possible d'utiliser des outils de synthèse à partir de langages de haut niveau. Dans [71], les auteurs ont utilisé Handel-C pour implémenter un étiquetage en deux passes et ont ainsi pu analyser rapidement les variations de performances obtenues en fonction des modifications algorithmiques de haut niveau. Des propositions basés sur des mémoires adressable par contenu [72] ont mis en évidence l'avantage de la personnalisation avancée possible sur les FPGA en répondant de manière élégante à la problématique posée par l'utilisation de la procédure Union-Find.

Propos et organisation du manuscrit

La démarche qui a prévalu lors des travaux de thèse peut se résumer en trois catégories mutuellement bénéfiques.

- Comprendre : la littérature est riche sur l'étiquetage en composantes connexes et chaque article a apporté sa pierre à l'édifice. Nous avons implémenté un grand nombre d'algorithmes et mis en place une plateforme unique permettant de les exécuter dans un contexte maîtrisé pour en comprendre les nuances en fonction de l'architecture des processeurs.

- Évaluer : nous avons proposé une méthodologie de tests reproductible sur laquelle la communauté pourra se baser pour mettre en évidence les qualités et défauts des algorithmes. Cela nous a permis de nous affranchir des différences de matériels, de systèmes d'exploitation, d'images étudiées.
- Améliorer : cette étude fine a permis de mettre en lumière les spécificités des algorithmes et a été la base des contributions suivantes : création d'une version parallèle du LSL, proposition d'une infrastructure logicielle de parallélisation des algorithmes directs en général, création de nouveaux algorithmes dérivés de la famille des algorithmes itératifs adaptés au contexte du multi-cœur et proposition d'un algorithme d'étiquetage spécifiquement dédié aux GPU.

La suite du document présentera les travaux et résultats selon le cheminement suivant :

- Le premier chapitre présentera les principes qui sous-tendent l'étiquetage en composantes connexes. Tout d'abord d'un point de vue général puis approfondi *via* la présentation du premier algorithme d'étiquetage moderne créée par Azriel Rosenfeld en 1966.
- Dans la deuxième chapitre, nous décrirons plusieurs algorithmes séquentiels «deux passes» ainsi que la méthodologie de test qui nous a permis de comparer les différents algorithmes.
- Le troisième chapitre présentera les résultats de la campagne de tests réalisée et mettra en lumière le comportement des différents algorithmes et de leurs variantes.
- Après cet état des lieux de l'étiquetage, nous proposerons dans le quatrième chapitre une version parallèle de l'algorithme LSL ainsi qu'une méthode originale de parallélisation des algorithmes directs d'étiquetage en composantes connexes pour les architectures multi-cœurs basée sur OpenMP.
- Le cinquième chapitre présentera et analysera les performances des algorithmes ainsi parallélisés et mettra en évidence les limites des différents algorithmes face à l'augmentation du nombre de cœurs.
- L'avènement d'architectures offrant un très grand nombre de cœurs (Xeon phi / GPU) est l'occasion de faire le point sur les algorithmes itératifs. C'est ce que nous proposerons dans le sixième chapitre. Après une étude des propriétés des algorithmes itératifs, nous proposerons deux nouveaux algorithmes ainsi qu'une implémentation inédite et performante pour GPU.
- Enfin, le septième chapitre sera consacré à l'étude des performances des algorithmes présentés sur des architectures à grand nombre de cœurs.

Chapitre 1

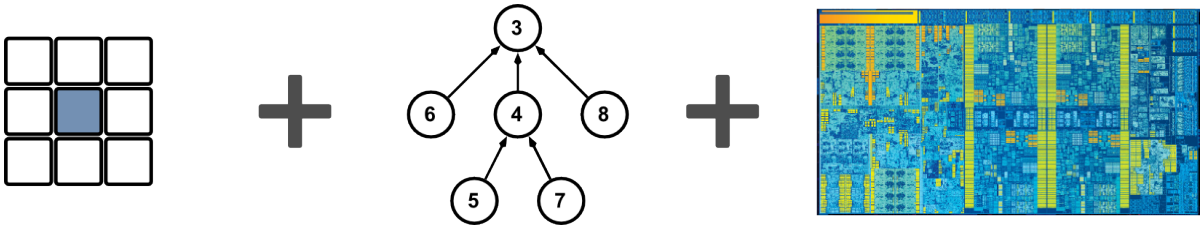
J'attachais une grande importance aux présentations. C'était souvent la seule image claire que nous laisserions aux gens.

–La Horde du contrevent, Alain Damasio

Fondamentaux de l'étiquetage en composantes connexes d'images binaires

La compréhension et le développement d'algorithmes d'étiquetage en composantes connexes moderne mettent en jeu :

- les notions de topologie numérique,
- les techniques de manipulations des graphes,
- la connaissance fine des architectures qui exécutent les algorithmes.



Dans ce chapitre, nous allons présenter les notions de topologie numérique, les structures formelles et pratiques ainsi que les techniques de manipulation de graphes utilisées par l'étiquetage en composantes connexes des images binaires. Une fois ces bases posées, nous poursuivrons par la description des premiers algorithmes d'étiquetage en composantes connexes proposés entre 1966 et 1983 et par une première analyse des spécificités architecturales de ces algorithmes. Enfin, nous poserons les bases de l'analyse en composantes connexes.

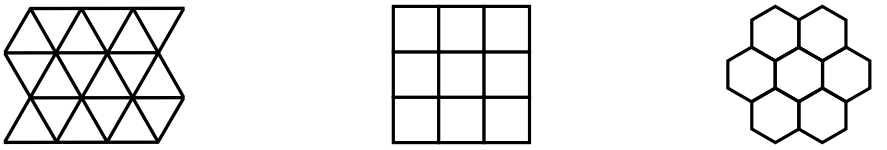
1.1 Notions de topologie pour l'étiquetage en composantes connexes

1.1.1 Topologie numérique

La topologie numérique telle que définie par Azriel Rosenfeld [73] est l'étude des propriétés et caractéristiques des images à 2 ou 3 dimensions telles que la connexité et les frontières. Elle est donc un sous-ensemble de la topologie principalement utilisée dans le cadre d'algorithmes d'analyse des images dits de «bas niveau» tels que les algorithmes de squelettisation, de détection de contours ou de régions, d'étiquetage en composantes connexes et de tous les algorithmes de morphologie mathématique. Les définitions suivantes prennent place dans le cadre de la topologie numérique des images à 2 dimensions.

1.1.2 Pavage

Le pavage d'une image est la décomposition de celle-ci en cellules élémentaires (pavés ou pixels). La figure 1.1 représente les pavages réguliers plans existants [74] : le pavage triangulaire (fig. 1.1a), le pavage carré (fig. 1.1b) et le pavage hexagonal (fig. 1.1c) qui correspondent respectivement à des pixels triangulaires, carrés et hexagonaux.



(a) Pavage triangulaire (b) Pavage carré (c) Pavage hexagonal

Fig. 1.1 – Pavages plans à base de polygones réguliers

1.1.3 Maillage

Le maillage est la construction duale du pavage. Il est composé de sommets (ici les centres des pavés) et d'arêtes qui relient les sommets des pixels qui partagent un côté (voisins). Les pavages à base triangulaire, carrée et hexagonale donnent respectivement des maillages à maille hexagonale (fig. 1.2a), carrée à 4 voisins (fig. 1.2b) et triangulaire (fig. 1.2d). En étendant le terme de *voisins* aux pixels qui partagent un côté ou un sommet, on obtient le maillage carré à 8 voisins(fig. 1.2c).

(a) Maillage hexagonal (b) Maillage carré à 4 voisins (c) Maillage carré à 8 voisins (d) Maillage triangulaire

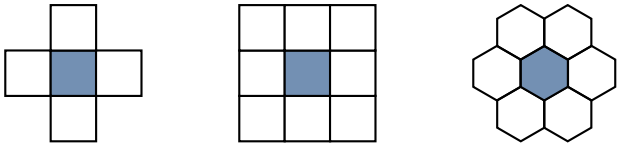
Fig. 1.2 – Maillages correspondants

1.1.4 Connexité

La notion de maillage permet de mettre en évidence la connexité. Celle-ci désigne le critère retenu pour définir la notion de *voisin*. En pratique, seuls les pavages à base carrée et hexagonale et leurs connexités associées sont utilisées. La figure 1.3 présente ces trois connexités :

- la 4-Connexité ($4C$) : dans une trame carrée, seuls les pixels dans l'axe vertical et l'axe horizontal sont considérés comme connexes (fig. 1.3a),
- la 8-Connexité ($8C$) : dans une trame carrée, tous les pixels sont considérés comme connexes (fig. 1.3b),
- la 6-Connexité ($6C$) : dans une trame hexagonale, tous les pixels sont considérés comme connexes (fig. 1.3c).

Les connexités $4C$ et $8C$ sont adaptées aux images numériques à trame carrée (ou rectangulaire), les images à trame hexagonale natives sont obtenues à partir de capteurs spécifiques et présentent des propriétés isotropiques très avantageuses en morphologie mathématique. Il est possible de construire des images à trame hexagonale à partir d'une trame carrée par décalage et interpolation.



(a) 4-connexité (b) 8-connexité (c) 6-connexité

Fig. 1.3 – Représentation des connexités d'une image en deux dimensions, le pixel bleu représente le pixel courant et les pixels blancs représentent les voisins au sens de la connexité considérée

Les travaux détaillés dans ce manuscrit ont pour cadre des images à trame carrée et sont donc volontairement limités aux connexités $4C$ et $8C$. Les algorithmes de $4C$ ne nécessitent qu'un sous-ensemble de règles par rapport aux algorithmes $8C$ et ils ne présentent pas de spécificités algorithmiques particulières. La majorité des algorithmes décrits et proposés dans ce manuscrit le seront sous leur forme $8C$. Il est intéressant de noter qu'une grande partie des algorithmes d'étiquetage en composantes connexes $8C$ peuvent être adaptés à la $6C$ par le retrait d'une seule règle.

1.1.5 Sens de parcours de l'image

Selon le sens de parcours de l'image ou *balayage*, tous les pixels présents autour du pixel courant ne présentent pas les mêmes propriétés. Certains auront déjà été traités lors du traitement du pixel courant et d'autres non. Le sens de parcours de l'image a donc une influence sur les algorithmes et la terminologie.

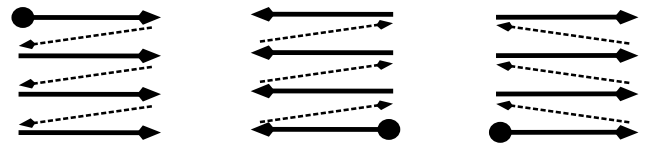
Il existe plusieurs balayages possibles d'une image. Dans le cadre de ce manuscrit, trois balayages différents ont été utilisés (fig. 1.4).

- le balayage *direct* (fig. 1.4a) : le parcours de l'image commence en haut à gauche de l'image et les pixels sont parcourus de la gauche vers la droite ligne par ligne jusqu'au bas de l'image.
- le balayage *inverse* (fig. 1.4b) : le parcours de l'image commence en bas à droite de l'image et les pixels sont parcourus de la droite vers la gauche ligne par ligne jusqu'au haut de l'image.
- le balayage *remontant* (fig. 1.4c) : le parcours de l'image commence en bas à gauche de l'image et les pixels sont parcourus de la gauche vers la droite ligne par ligne jusqu'au haut de l'image.

La majorité des algorithmes modernes sont des algorithmes directs, c'est-à-dire utilisant un balayage direct.

1.1.6 Voisinage

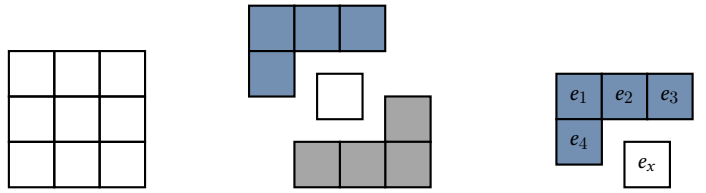
Le voisinage de taille n d'un pixel est l'ensemble des pixels qui lui sont connexes avec une distance d telle que $d < n$ au sens de la connexité choisie. Ainsi la figure 1.5a représente un voisinage de taille 1



(a) Balayage direct (b) Balayage inverse (c) Balayage remontant

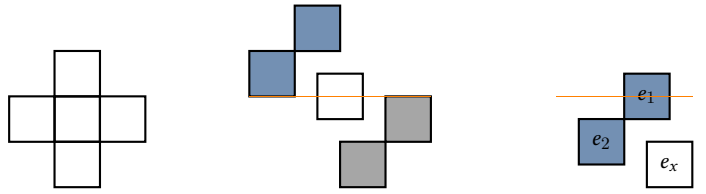
Fig. 1.4 – Sens de balayage : le point rond représente l'origine du balayage, les flèches pleines le sens de parcours des lignes et les flèches en pointillé représentent l'enchaînement des lignes

(aussi appelé 3×3) pour une connexité $8C$ alors que la figure 1.6a représente le même voisinage mais pour une connexité $4C$. Au sein de ce voisinage on peut distinguer trois ensembles : *passé*, *présent* et *futur* représentés dans les figures 1.5 ($8C$) et 1.6 ($4C$). En effet, lors d'un balayage direct de l'image (cf. 1.1.5), le passé est l'ensemble des pixels traités avant le pixel courant et le futur l'ensemble de ceux qui le seront par la suite. Le voisinage passé est donc l'ensemble des pixels appartenant au voisinage sélectionné et qui ont déjà été traités. Les figures 1.5c ($8C$) et 1.6c ($4C$) représentent le voisinage passé (bleu) ainsi que le pixel courant (blanc). C'est cet ensemble qui sera utilisé sous le nom de *masque* dans la majorité des algorithmes d'étiquetage en composantes connexes. Lors d'un balayage inverse, les voisinages passé et futur sont échangés.



(a) Voisinage 3x3 (b) Passé, présent et futur (c) Passé et présent

Fig. 1.5 – Décomposition du voisinage d'un pixel en passé - présent - futur (8-connexité)



(a) Voisinage 3x3 (b) Passé, présent et futur (c) Passé et présent

Fig. 1.6 – Décomposition du voisinage d'un pixel en passé - présent - futur (4-connexité)

1.2 De la composante connexe à l'image étiquetée

1.2.1 Composantes connexes

Chaque algorithme étudié dans ces travaux comporte des variations dans la méthode et les structures de données, mais tous permettent d'obtenir les composantes connexes de l'image. Cela est dû au fait qu'il respectent, explicitement ou non, un ensemble de propriétés. La représentation formelle de ces propriétés permet de mettre en évidence les invariants des algorithmes d'étiquetage en composantes connexes.

Une composante connexe $\mathcal{A}$ est un ensemble des pixels pour lesquels il existe une relation de connexité, ce qui équivaut à dire que pour tout couple de pixels (p_a, p_b) de la composante connexe, il existe un chemin ζ_{p_a,p_b} interne à $\mathcal{A}$ permettant de relier les deux pixels [74].

- Soit une image binaire quelconque I de taille $H \times W$.
- Soit $p_{i,j}$ un pixel de I tel que :
  $\forall (i,j) \in \mathbb{N}^2$tel que $i < W$ et $j < H, p_{i,j} = I[i][j]$.
- Soit $p^e_{i,j}$ une entité «pixel étiqueté» qui associe $p_{i,j}$ et *e l'étiquette* du pixel, à savoir un numéro qui le rattache à une composante connexe.

On définit alors la composante connexe $\mathcal{A}$ telle que

$$\mathcal{A} = \begin{cases} P_a = \{p^a_{i_0,j_0}, ..., p^a_{i_{n-1},j_{n-1}}\} \\ E_a = \{a_0, ..., a_m\} \\ R_a = min(a_k) \end{cases} \tag{1.1}$$

avec :

- P_a la liste des n pixels étiquetés appartenant à la composante connexe.
- E_a la liste des m étiquettes appartenant à la composante connexe,
- R_a la plus petite étiquette (min) [1] appartenant à la composante connexe aussi appelée racine.

I comporte un nombre fini K de composantes connexes
Afin de manipuler cette structure, définissons trois opérateurs :

- L'opérateur *Nouvelle* tel que si $p_{i,j}$ est un pixel isolé de l'image et a_0 une étiquette non attribuée :

$$Nouvelle(p_{i,j}) = \mathcal{A} \text{ avec } \mathcal{A} = \begin{cases} P_a = \{p^{a_0}_{i,j}\} \\ E_a = \{a_0\} \\ R_a = a_0 \end{cases} \tag{1.2}$$

- L'opérateur *Ajout* tel que si $p_{i,j}$ est un pixel connexe uniquement à $\mathcal{A}$ au moment de son traitement :

$$Ajout(\mathcal{A}, p_{i,j}) = \begin{cases} P_a = \{p^a_{i_0,j_0}, ..., p^a_{i_{n-1},j_{n-1}}, p^a_{i,j}\} \\ E_a \\ R_a \end{cases} \tag{1.3}$$

- L'opérateur *Union* tel que si $\mathcal{A}$ et $\mathcal{B}$ sont deux composantes connexes distinctes reliées par $p_{i,j}$:

$$\begin{aligned} Union(\mathcal{A}, \mathcal{B}) &= Union(a, b) \text{ avec } \forall a \in E_a, \forall b \in E_b \\ &= Union(a_0, b_0) \\ &= C \end{aligned} \tag{1.4}$$

avec

$$C = \begin{cases} P_c = P_a \cup P_b = \{p^a_{i_0,j_0}, ..., p^a_{i_{n-1},j_{n-1}}, p^b_{i_0,j_0}, ..., p^b_{i_{n-1},j_{n-1}}, p^c_{i,j}\} \\ E_c = E_a \cup E_b = \{a_0, ..., a_m, b_0, ..., b_m\} \\ R_c = min(R_a, R_b) \end{cases} \tag{1.5}$$

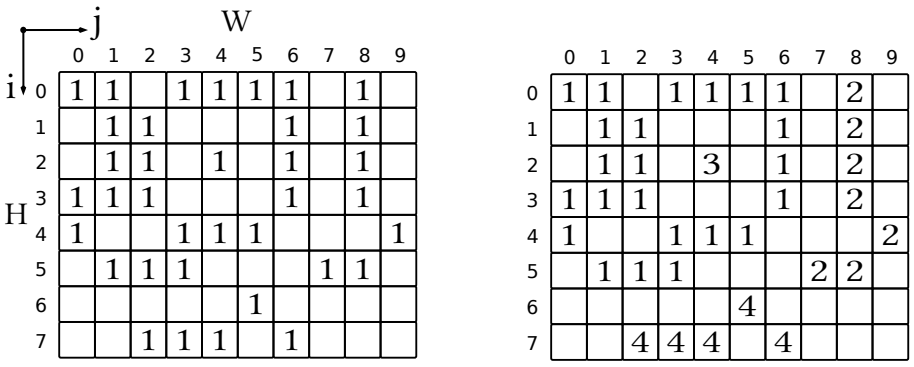
1. En fait R_a doit être un représentant unique $\mathcal{A}$ et doit appartenir effectivement à E_a. Il est donc possible de remplacer l'opérateur *min* par l'opérateur *max*. Cependant la très grande majorité des algorithmes se base sur l'opérateur *min*.

1.2.2 Structures de l'étiquetage en composantes connexes pour les images binaires

Le but de l'étiquetage en composantes connexes pour les images binaires n'est pas de construire une liste des composantes connexes mais d'attribuer à chaque pixel d'une image binaire *sa* composante connexe. Cela implique une série de mécanismes liés aux opérateurs *Nouvelle*, *Ajout* et *Union* et adaptés aux structures propres de l'étiquetage en composantes connexes.

Si chaque algorithme d'étiquetage en composantes connexes est unique et possède donc ses propres mécanismes qui seront mis en évidence au fur et à mesure dans les deux premiers chapitres, certaines structures sont communes et leur connaissance est nécessaire à la bonne compréhension des mécanismes généraux de l'ECC. Nous avons regroupé ici une description de ces structures et leurs représentations associées.

- *Image binaire - $I[H][W]$* (fig. 1.7a) : c'est la donnée d'entrée des algorithmes d'ECC. Elle prend la forme d'une table d'octets de taille $H \times W$.
- *Étiquette - e* : c'est un nombre affecté à un ou plusieurs pixels qui symbolise son appartenance à une classe d'équivalence. L'étiquette d'un pixel peut évoluer au cours de l'exécution de l'algorithme.
- *Image d'étiquettes - $E[H][W]$* (fig. 1.7b) : c'est le produit des algorithmes d'ECC. Dans le manuscrit, du fait des tailles d'images rencontrées [2], elle prendra la forme d'une table de mots de 4 octets de taille $H \times W$. Le nombre maximal d'étiquettes non nulles est donc $ne_{max} = 2^{32} - 1$.
- *Table d'équivalence - T[N]* : Lors de l'exécution de l'ECC, si le mécanisme d'*Union* est invoqué, celui-ci connecte des étiquettes entre elles. La table d'équivalence est la structure utilisée par la majorité des algorithmes pour recenser les connexions entre les étiquettes. Un compteur (régulièrement noté *ne* dans la suite du manuscrit) permet de connaître le numéro de la dernière composante connexe créée.
- *Ancêtre - a* : a est l'ancêtre de e si $a = T[e]$.
- *Racine - r* : r est une racine si $r = T[r]$.



(a) I : Image binaire

(b) E : Image finale des étiquettes

Fig. 1.7 – Structures de données de l'étiquetage en composantes connexes

Avec ces définitions, il est possible de retranscrire les opérateurs *Nouvelle*, *Ajout* et *Union* :

- *Nouvelle* : l'action de créer une nouvelle composante connexe se réalise en incrémentant le compteur *ne*.

2. de 1024×1024 à 8192×8192 pixels

- *Ajout* : l'action d'ajouter un pixel à une composante connexe se traduit par l'affectation du numéro de celle-ci au pixel. Il n'existe donc pas une liste de pixels correspondant à une étiquette. Mais il est au contraire possible de connaître la composante connexe correspondant à un pixel.
- *Union* : l'action d'unir deux composantes connexes consiste à manipuler T pour représenter cette connexion. Par exemple, connecter la composante 1 à la composante 2 correspond à l'opération $T[2] \leftarrow 1$.

Une fois ces bases posées, il est possible de concevoir une version naïve afin de présenter les mécanismes de l'étiquetage en composantes connexes.

1.2.3 Première intuition

Dans une image binaire, nous considérerons par convention les pixels à «0» comme formant le fond et les pixels à «1» comme formant le premier plan (le choix symétrique est possible). L'idée principale de l'étiquetage en composantes connexes étant de grouper tous les pixels du premier plan qui sont en contact (fig. 1.7a) sous une même étiquette (fig. 1.7b), une première approche est de parcourir l'image pixel à pixel et, si le pixel courant est un point du premier plan, d'étudier son voisinage passé (car déjà étiqueté). L'étiquetage en composantes connexes utilise un traitement local pour résoudre un problème global. Selon les étiquettes des pixels présents dans le voisinage, trois cas sont possibles :

- aucun voisin : utilisation de l'opérateur *Nouvelle*,
- une seule racine dans le voisinage (qui peut être commune à plusieurs pixels) : utilisation de l'opérateur *Ajout*,
- plusieurs racines différentes dans le voisinage : utilisation de l'opérateur *Union*.

Au fur et à mesure du parcours de l'image, les composantes connexes évoluent en fonction des cas rencontrés. Ce n'est que lorsque le dernier pixel de l'image a été traité que l'image est entièrement étiquetée et que l'information de connexité est complète. Cependant chaque pixel s'est vu attribuer une étiquette temporaire qui reflète l'état de l'arbre au moment de son traitement. Sachant que cet état évolue potentiellement à chaque nouveau pixel rencontré, il est donc nécessaire de réaliser un nouveau parcours de l'image pour affecter à chaque pixel la racine de sa composante connexe (réétiquetage) afin de permettre une lecture du résultat par un humain. Cette façon de faire n'est pas unique et la littérature propose de nombreux algorithmes que nous allons classifier en catégories.

1.2.4 Les catégories d'étiquetage en composantes connexes

Les algorithmes d'étiquetage en composantes connexes peuvent être classifiés différemment selon le centre d'intérêt. Nous allons ici en proposer trois qui sont complémentaires et permettent de passer en revue l'essentiel des techniques :

- La forme du voisinage considéré.
- Le nombre de passes (ou itérations) sur l'image.
- La gestion des équivalences.

1.2.4.1 Catégorisation selon la forme du voisinage

Dans les chapitres 2 à 5, nous illustrerons les caractéristiques des algorithmes directs en nous basant sur une sélection d'algorithmes représentatifs : Rosenfeld[73], Suzuki[58], Grana[60], RCM[63], HCS_2[64], HCS[61], Light Speed Labeling (LSL)[53].

Ces algorithmes diffèrent par le voisinage pris en compte pour étudier la connexité. La figure 1.8 présente les différents masques correspondant à ces algorithmes.

Nous distinguerons les algorithmes «pixels» qui se basent sur un voisinage lié à chaque pixel dont le représentant le plus courant est celui de Rosenfeld (utilisé par *Rosenfeld* et *Suzuki*), des algorithmes «segments» qui se basent sur les segments horizontaux connexes de l'image au sein d'une même ligne (c'est le cas de *LSL*) fig. 1.8f). Parmi les algorithmes pixels, il existe une variante dénommée algorithmes «blocs» qui étend (cas de *Grana* - fig. 1.8d et HCS_2 - fig. 1.8c) ou réduit (cas de *RCM* - fig. 1.8b) la taille du voisinage avec toujours pour but de modifier le ratio pixel à traiter / pixel généré. Le cas de HCS (fig. 1.8e), est remarquable en ce sens qu'il utilise un masque différent selon qu'il est dans ou en dehors d'un segment de premier plan, ce qui fait de lui un algorithme hybride pixel/segment.

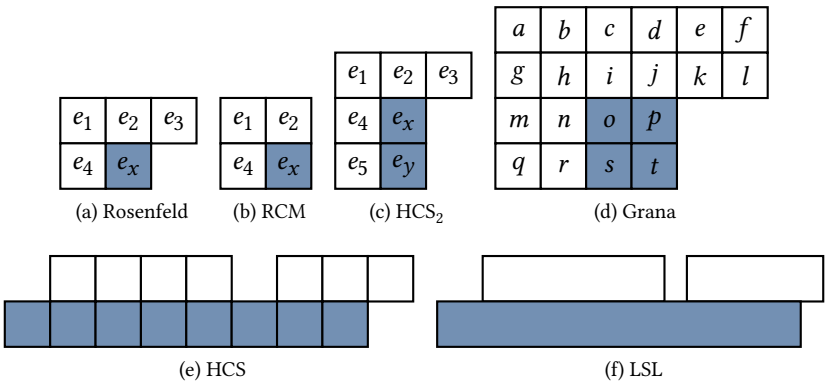


Fig. 1.8 – Masque des algorithmes directs

1.2.4.2 Catégorisation selon le nombre de passes sur l'image

Parcourir l'image en traitant chaque pixel dans l'ordre direct est une opération dont le temps d'exécution est proportionnel au nombre de pixels de l'image. Il en découle que réduire le nombre de passes est un enjeu très important.

Selon l'article [63], les algorithmes d'étiquetage en composantes connexes peuvent être classifiés selon trois familles :

- Les algorithmes multi-passes ou itératifs : il est nécessaire d'appliquer l'algorithme jusqu'à atteindre la stabilisation du résultat [75].
- Les algorithmes directs en deux passes qui représentent la très grande majorité : après une première passe de classification des pixels de l'image, il est nécessaire de consolider les classes d'équivalence puis de reparcourir l'image pour réaffecter à chaque pixel la racine de sa classe d'équivalence afin de rendre le résultat lisible par un humain. Dans cette catégorie, certains auteurs font la proposition de supprimer l'étape de réétiquetage et de fournir comme sortie l'image étiquetée et la table d'équivalence. C'est alors aux algorithmes suivants dans la chaîne de traitement d'exploiter ces données. Cette variante est couramment nommée algorithme une passe.
- Les algorithmes de contour-tracing qui ne parcourent pas l'image de façon linéaire. Ils recherchent un pixel de premier plan non étiqueté dans l'image. Dès qu'ils en rencontrent un, ils suivent le contour (zone de transition premier plan/arrière plan) jusqu'à revenir au point de départ. Tous les pixels contenus à l'intérieur de ce contour sont alors étiquetés avec la même étiquette et le parcours reprend dans le même ordre jusqu'à un nouveau point non étiqueté. Si à la fin de cette unique passe toute l'image est en effet étiquetée, le résultat peut-être différent des autres algorithmes car la plupart ne tiennent pas compte des éventuels trous dans une

composante connexe et de l'imbrication possible de plusieurs composantes connexes (cas des composantes 1 et 3 de la figure 1.7b).

Il faut noter que plusieurs méthodes d'étiquetage en composantes connexes ne peuvent pas être catégoriser sur ce critère. Par exemple, les algorithmes se basant sur les *quadtree*[76, 77] dont l'efficacité pour l'étiqueter une image est faible[78], mais qui sont très adaptés aux séquences d'images où seule une portion de l'image est mise à jour[79].

1.2.4.3 Catégorisation selon la méthode de gestion des étiquettes

Chaque masque peut être associé de manière orthogonale à une méthode de gestion des étiquettes. La section 1.3 présentera les principes d'une telle gestion ainsi qu'une des méthodes utilisées : l'algorithme *Union-Find* classique. Il existe plusieurs variations algorithmiques autour de cet algorithme[80, 81] ainsi que plusieurs variations dans son implémentation utilisant une table unique ou des systèmes de listes[58]. Nous verrons que dans le cas des méthodes itératives[82, 83], il est possible de se passer d'une structure de données spécifique à cette gestion.

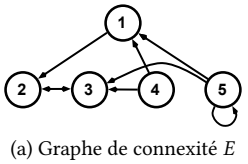
1.3 Structures de données et manipulations des graphes

Dans une composante connexe A telle que définie en 1.2.1, E_a est une simple liste d'étiquettes. Pour respecter la faisabilité de l'opération *Union* ainsi que la détermination de la racine R, une relation d'ordre est nécessaire entre les m étiquettes qui la composent. E_a possède donc les attributs d'un graphe et nous le nommerons donc *graphe de connexité*. C'est cette structure de graphe qui nous permet de réaliser l'opération *Ajout* sans avoir à maintenir formellement la liste P.

Nous allons ici présenter de manière progressive les outils et mécanismes qui ont conduit à la représentation sous forme de table d'équivalences et à l'utilisation de l'algorithme *Union-Find*.

1.3.1 Dualité graphe de connexité / matrice d'adjacence

La manière classique et complète de représenter un graphe est l'utilisation d'une matrice d'adjacence. Dans cette matrice, les arêtes du graphe sont symbolisées par une valeur «1» à la ligne et à la colonne correspondants aux sommets adjacents. La matrice Ad est alors équivalente à la représentation du graphe E (fig. 1.9a).



(a) Graphe de connexité E

$$\overbrace{\text{Origine}}$$

$$\text{Cible}\left\{\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}\right.$$

(b) Matrice d'adjacence Ad

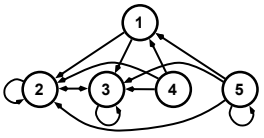
Fig. 1.9 – Dualité graphe de connexité / matrice d'adjacence

1.3.2 Fermeture transitive

Comme vu en 1.2.3, lorsque la construction du graphe de connexité est terminée, il est nécessaire d'assigner à chaque pixel de P l'étiquette R de la racine de sa composante connexe au lieu des éti-

quettes intermédiaires. Du fait de l'utilisation d'un matrice d'adjacence, il est nécessaire de réaliser une fermeture transitive du graphe pour établir une connexion directe entre e et R.

La fermeture transitive d'un graphe consiste à créer des arcs entre chacun des sommets qui sont reliés par des chemins indirects. Cette opération permet donc d'avoir une structure permettant de passer directement d'un sommet à tout autre sommet qui lui est connexe dans le graphe. Le résultat de la fermeture transitive est donc un nouveau graphe (fig. 1.10) qui contient tous les liens possibles entre tous les nœuds du graphe. C'est cette propriété qui nous intéresse afin de passer de n'importe quelle étiquette à la racine de sa composante connexe.



(a) Graphe «fermé» Gf

$$\overset{\text{Origine}}{\begin{matrix}\\ \text{Cible}\end{matrix}}\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Matrice d'adjacence «fermée» Adf

Fig. 1.10 – Un graphe et sa fermeture transitive

Dans le cas de l'étiquetage en composantes connexes, la fermeture transitive n'est pas entièrement utilisée. Ce qui nous intéresse est uniquement de passer d'une étiquette à sa racine correspondante. Dans la littérature de l'étiquetage en composantes connexes, le terme fermeture transitive recouvre donc l'opération de fermeture réduite aux liens vers les racines.

1.3.3 Algorithme de Floyd-Warshall

Afin de réaliser cette fermeture transitive, l'algorithme de Floyd-Warshall [84] est le plus répandu. Il permet de construire la fermeture transitive d'un graphe orienté ou non orienté.

Algorithme 1 : Fermeture transitive - Algorithme de Floyd Warshall

Input : Ad une matrice d'adjacence de taille $N \times N$
Result : Adf la matrice d'adjacence de la fermeture transitive

```
1 Adf = Ad
2 for k = 1 to N do
3     for i = 1 to N do
4         for j = 1 to N do
5             Adf_ij = min(Adf_ij, Adf_ik + Adf_kj)
```

L'application de cet algorithme nous permet de calculer la matrice de la figure 1.10b qui est le dual du graphe de la figure 1.10a. La complexité de cet algorithme est en $O(n^3)$ et l'occupation mémoire de la matrice est d'ordre n^2 avec n le nombre d'étiquettes nécessaire à la construction de l'étiquetage (n est variable selon le contenu de l'image et peut être bien plus grand que le nombre final de composantes connexes (cf. 2.3)). Du fait de son empreinte mémoire et des besoins en capacités de calcul, l'algorithme de Floyd-Warshall n'est donc en aucun cas une solution viable pour l'étiquetage en composantes connexes.

1.3.4 Forêts enracinées

Dans le cas particulier de l'étiquetage en composantes connexes, les graphes sont en fait des forêts enracinées. Une forêt enracinée est un ensemble fini d'arbres enracinés et un graphe est un arbre enraciné si et seulement si :

- il est connexe,
- il a un unique sommet sans prédécesseur (la racine),
- et tous ses autres sommets ont exactement un prédécesseur.

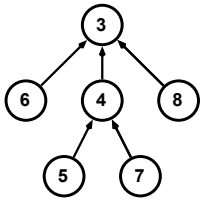
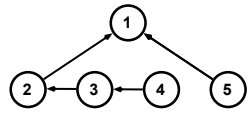
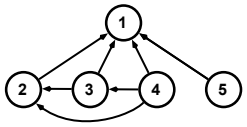


Fig. 1.11 – Exemple d'arbre enraciné

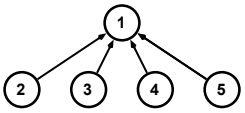
Les graphes étant orientés dans le sens des étiquettes décroissantes, les matrices d'adjacence ont des propriétés particulières qui vont permettre de gagner en compacité. En choisissant comme convention que le sommet d'origine est représenté par la ligne et que le sommet d'arrivée est représenté par la colonne, la matrice devient triangulaire supérieure. Dans ce cas, il est intéressant de remarquer que ce que nous recherchons réellement pour notre application est une forme réduite de la fermeture transitive qui correspond en fait au sous-ensemble du graphe fermé composé des arêtes qui relient chaque étiquette à la racine de sa composante connexe.



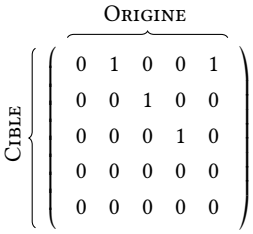
(a) Graphe de connexité



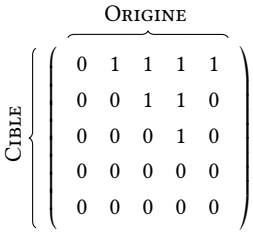
(b) Graphe de connexité fermé



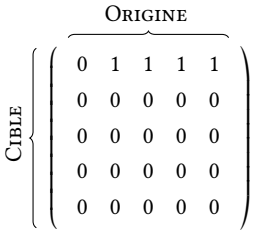
(c) Graphe de connexité fermé réduit aux racines



(d) Matrice d'adjacence



(e) Matrice d'adjacence fermée



(f) Matrice d'adjacence fermée réduite aux racines

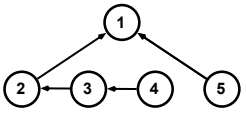
Fig. 1.12 – Dualité graphe / matrice d'adjacence dans le cas des graphes orientés

1.3.5 Représentation par table d'équivalences

Une autre façon de représenter l'information contenue dans la matrice d'adjacence est d'utiliser une table d'équivalences. Là où la matrice d'équivalence indiquait une équivalence par un « 1 » aux coordonnées représentant l'arête, la table d'équivalence associe à une étiquette l'étiquette de son ancêtre (figs. 1.13b et 1.13d). Cela n'est possible que dans le cas de forêts enracinées dans lesquelles

chaque sommet ne peut pointer que vers une seule étiquette, ce qui est le cas des graphes obtenus à la fin du processus d'étiquetage en composantes connexes. Cependant, là où la matrice d'adjacence représente sans difficultés les connexions d'une étiquette à plusieurs ancêtres, la table d'équivalence ne peut faire correspondre qu'un seul ancêtre à une étiquette. Cela introduit une difficulté dans la phase de construction du graphe. En effet, si dans un voisinage on constate que l'étiquette 3 qui pointait jusque-là vers 2 est connexe à 1, le fait de faire pointer $3 \longrightarrow 1$ fait perdre l'information $3 \longrightarrow 2$ car il n'existe plus de mécanisme direct pour représenter cette information. La solution retenue est d'associer l'ensemble des étiquettes vers la plus petite des étiquettes.

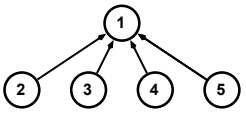
Cet exemple illustre le fait que l'utilisation d'une table d'équivalences en remplacement d'une matrice d'équivalence implique de gérer finement l'union des différentes classes d'équivalence (opérateur *Union*) tout au long de la première passe, pour éviter des situations de rupture de graphes.



(a) Graphe d'adjacence

e	0	1	2	3	4	5
$T[e]$	0	1	1	2	3	1

(b) Table d'équivalence



(c) Graphe d'adjacence fermé

e	0	1	2	3	4	5
$T[e]$	0	1	1	1	1	1

(d) Table d'équivalence fermée

Fig. 1.13 – Dualité graphe / table d'équivalence

1.3.6 Algorithme *Union-Find*

Nous venons de voir que la validité de l'étiquetage dépend de la validité de l'opérateur *Union*. Une des procédures les plus couramment utilisées est la procédure *Union-Find* qui est composée de deux étapes : *Find* et *Union*.

- *Find* : permet de trouver la racine de l'arbre à laquelle appartient une étiquette.
- *Union* : fusionne deux arbres.

Algorithme 2 : Algorithme Find

Input : e une étiquette, T une table d'équivalence
Result : r, la racine de e

```
1 r ← e
2 while T[r] ≠ r do
3     r ← T[r]
```

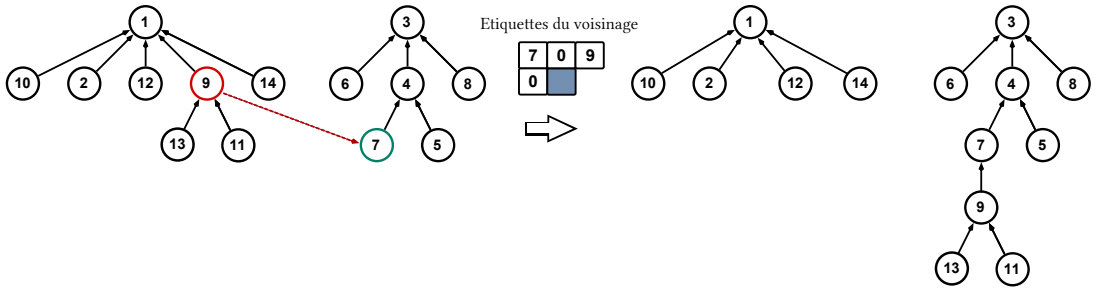
Unir deux arbres sans remonter à leurs racines est une opération qui peut introduire des coupures dans l'arbre du fait de la représentation sous forme de table d'équivalences.

Considérons deux arbres A et B (fig. 1.14). Supposons que les nœuds 9 et 7 soit en contact dans l'image, il est alors nécessaire de fusionner les deux graphes. La figure 1.14b représente le résultat correct où après avoir recherché la racine de chaque arbre, on fait pointer la plus grande racine vers

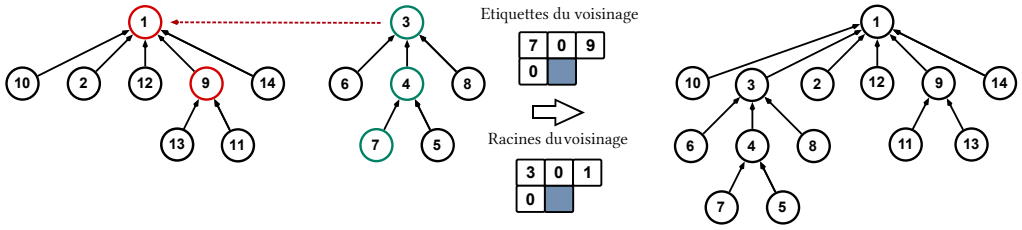
Algorithme 3 : Algorithme Union

Input : e_1, e_2 deux étiquettes, T une table d'équivalence
Result : r, la plus petite des racines des étiquettes

```
1 r1 ← Find(e1, T) r2 ← Find(e2, T)
2 if r1 < r2 then
3 |   r ← r1, T[r2] ← r
4 else
5 |   r ← r2, T[r1] ← r
```



(a) Fusion eronnée : l'association directe des deux étiquettes fait perdre la connexion entre les deux branches



(b) Fusion correcte : l'association des racines conserve la connexion entre les deux branches

Fig. 1.14 – Fusion de deux arbres orientés : sans remontée à la racine (haut), avec remontée à la racine (bas)

la plus petite. La figure 1.14a représente un résultat erroné où l'on fait pointer directement le nœud de plus grande valeur vers le nœud de plus petite valeur et en perdant, ce faisant, toute connexité entre les deux graphes. Le graphe a été «cassé».

1.4 Algorithmes pionniers

Afin de comprendre les algorithmes modernes, il est utile d'étudier les algorithmes des pionniers de l'étiquetage en composantes connexes que sont Rosenfeld & Pfaltz (1966), Haralick & Shapiro (1981), Lumia & Shapiro & Zuniga (1983) et Ronse & Devijver (1984). Chacun à leur manière, ils ont proposé une réponse adaptée aux capacités mémoire de leur époque.

1.4.1 Rosenfeld & Pfaltz

Dans le domaine de l'étiquetage en composantes connexes (comme dans bien des champs du traitement d'image), il y a un avant et un après Azriel Rosenfeld (19 Février 1931 - 22 Février 2004). En 1966, il a publié avec John Pfaltz la première proposition qui tente de répondre efficacement à la problématique de l'étiquetage en composantes connexes dans [73]. L'algorithme avait été implémenté en Fortran sur un IBM7090/94 [3] comportant une mémoire de 32768 mots de 36 bits. La mémoire était donc une ressource très précieuse.

Dans l'article original, la procédure de gestion des équivalences est complexe et incomplètement décrite et se base sur l'état de l'art des algorithmes d'union de graphes de son époque [85] qui est l'ancêtre direct de la procédure Union-Find telle que décrite par Tarjan. Dans la littérature, l'algorithme de Rosenfeld est donc maintenant décrit avec la procédure Union-Find : c'est donc le choix fait aussi dans ce manuscrit.

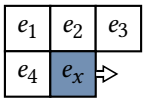


Fig. 1.15 – Masque de Rosenfeld

La proposition était de réaliser le traitement en deux passes avec une étape intermédiaire de fermeture transitive du graphe (selon le schéma décrit en 1.2.3) :

- Première passe : Scanner l'image pixel par pixel et en fonction de son voisinage passé déterminer l'étiquette et enregistrer au fur et à mesure l'ensemble des équivalences rencontrées dans une table qui représente le graphe (Algo. 4 en 8C , algo. 39 en 4C).
- Réaliser la fermeture transitive des racines du graphe de connexité afin de supprimer les composantes temporaires (Algo. 5),
- Seconde passe : Scanner l'image pour affecter à chaque pixel l'étiquette de la racine de l'arbre auquel il appartient (Algo. 6).

L'algorithme est décomposé étape par étape dans la figure 1.16 : chaque sous-figure représente l'état de l'image étiquetée ainsi que de la table d'équivalence lors d'une étape clef de l'algorithme.

Considérons l'image binaire de départ (fig. 1.16a), le nombre d'étiquettes créées est $n_e = 0$, la table d'équivalence est initialisée avec $T[e] = e$. L'image est parcourue pixel à pixel dans le sens direct. Si le pixel courant est un pixel de fond on lui assigne l'étiquette 0 (fig. 1.16b) sinon :

- Si le voisinage ne comporte pas d'étiquette non nulle (fig. 1.16c, 1.16d, 1.16e), il y a création d'une nouvelle étiquette ($n_e = n_e + 1$) et affectation de cette étiquette au pixel courant. On remarquera que les bords sont considérés comme des étiquettes nulles.

3. http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP7090.html

Algorithme 4 : Rosenfeld 8C avec Union-Find

```
   Input : I[H][W]
   Result : E[H][W], T
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          if I[i][j] ≠ 0 then
4              e1 ← E[i-1][j-1] e2 ← E[i-1][j] e3 ← E[i-1][j+1] e4 ← E[i][j-1]
5              if e1 = e2 = e3 = e4 = 0 then
6                  ex ← ne++
7              else
8                  r1 = Find(T,e1) r2 = Find(T,e2) r3 = Find(T,e3) r4 = Find(T,e4)
9                  ε ← min+(r1, r2, r3, r4)
10                 if (r1 ≠ 0 and r1 ≠ ε) then Union(T, e1, ε)
11                 if (r2 ≠ 0 and r2 ≠ ε) then Union(T, e2, ε)
12                 if (r3 ≠ 0 and r3 ≠ ε) then Union(T, e3, ε)
13                 if (r4 ≠ 0 and r4 ≠ ε) then Union(T, e4, ε)
14                 ex ← ε
15         else
16             ex ← 0
17         E[i][j] ← ex
```

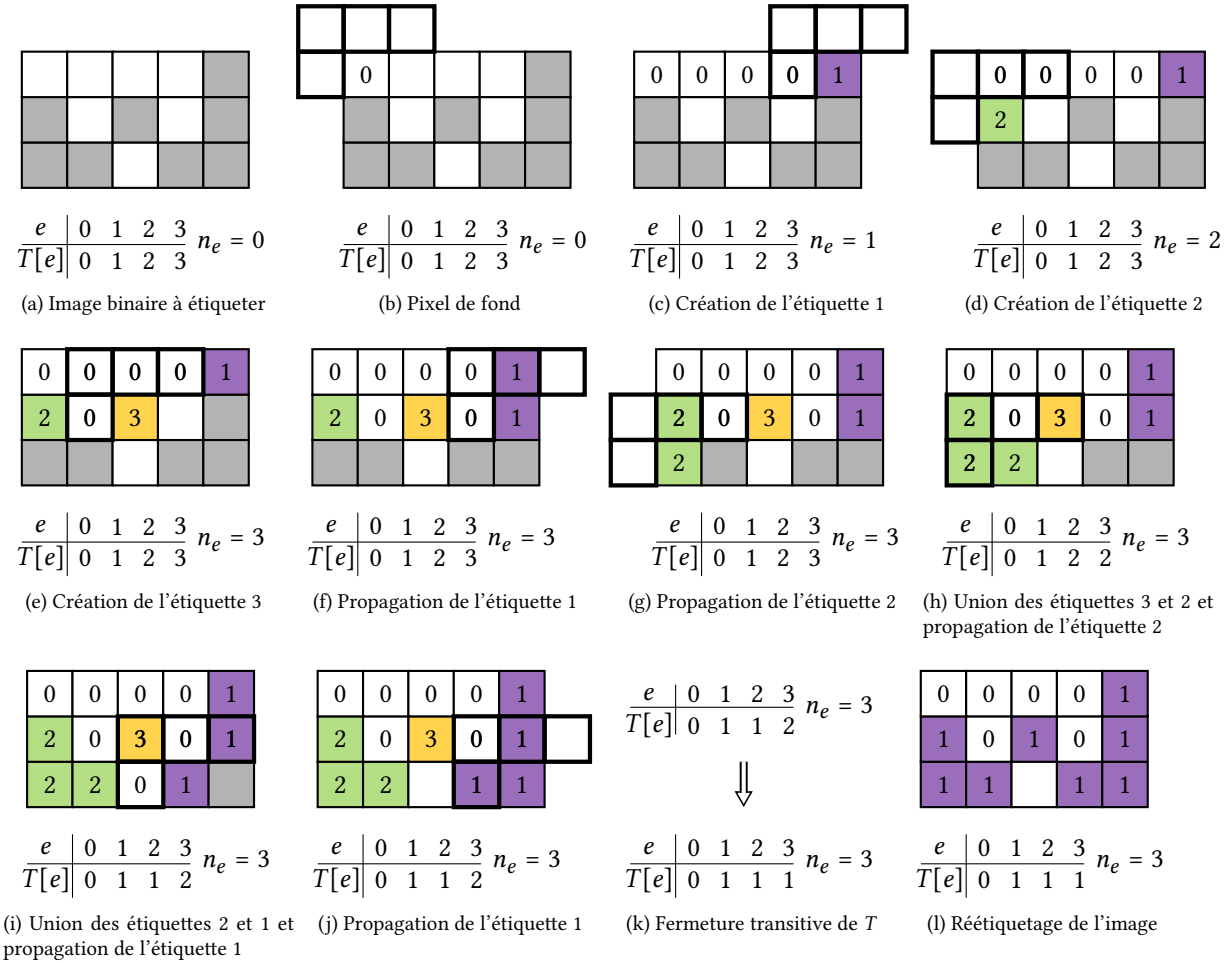
Algorithme 5 : Fermeture transitive (des racines de la forêt)

```
   Input : T avec e associée à son ancêtre, ne_max la plus grande étiquette de l'image
   Result : T avec e associée à sa racine
1  for i = 0 to ne_max do
2      T[e] ← T[T[e]]
```

Algorithme 6 : Réétiquetage selon la table d'équivalence

```
   Input : E[H][W], T
   Result : E[H][W]
1  for i = 0 to N − 1 do
2      for j = 0 to M − 1 do
3          E[i][j] ← T[E[i][j]]
```

- Si le voisinage comporte une unique étiquette non nulle (fig. 1.16f, 1.16g, 1.16j), cette étiquette est propagée au pixel courant.
- Si le voisinage comporte plusieurs étiquettes non nulles différentes (fig. 1.16h, 1.16i), la table d'équivalence est modifiée selon l'algorithme Union-Find et la plus petite des racines (obtenue en utilisant l'opérateur min^+) est affectée au pixel courant.



(a) Image binaire à étiqueter (b) Pixel de fond (c) Création de l'étiquette 1 (d) Création de l'étiquette 2

(e) Création de l'étiquette 3 (f) Propagation de l'étiquette 1 (g) Propagation de l'étiquette 2 (h) Union des étiquettes 3 et 2 et propagation de l'étiquette 2

(i) Union des étiquettes 2 et 1 et propagation de l'étiquette 1 (j) Propagation de l'étiquette 1 (k) Fermeture transitive de T (l) Réétiquetage de l'image

Fig. 1.16 – Rosenfeld : Étapes clefs du déroulement de l'étiquetage

Une fois la première passe terminée, l'algorithme 5 permet d'obtenir la table d'équivalence fermée (fig. 1.16k). La seconde passe permet alors, par application de la table d'équivalence (Algo. 6 - T est utilisée comme une LUT), d'obtenir l'image finale (fig. 1.16l).

1.4.2 Haralick & Shapiro

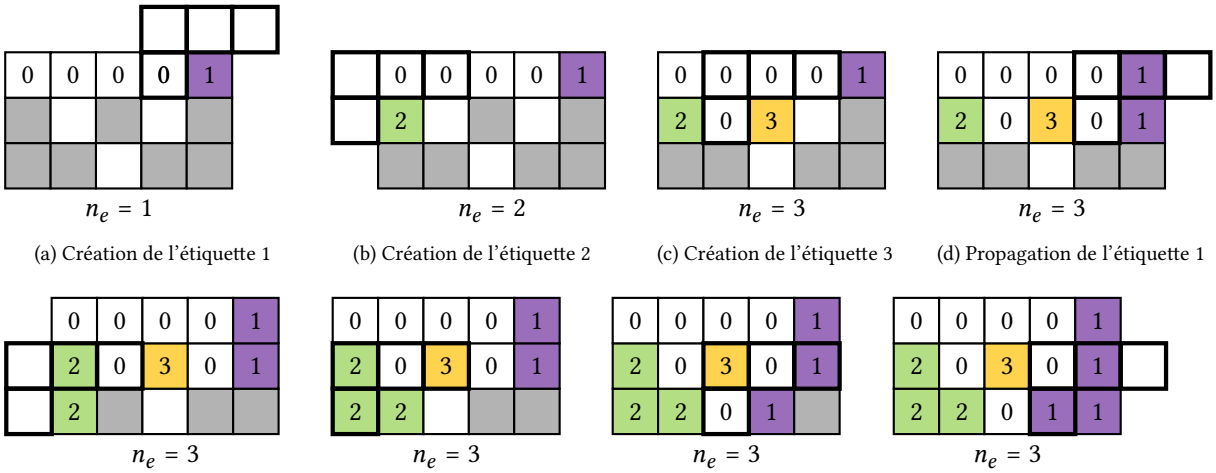
La limitation de la quantité de mémoire sur les ordinateurs a poussé R. M. Haralick, à proposer en 1981 un algorithme d'étiquetage en composantes connexes itératif récursif [82]. L'idée est de n'utiliser aucune structure supplémentaire à l'image des étiquettes et de réaliser l'étiquetage par une succession de propagations utilisant un balayage direct et inverse.

Haralick utilise le terme récursif pour les algorithmes tenant compte des modifications précédentes de l'image des étiquettes au sein d'une même passe. Cette classe d'algorithmes est aussi connu sous le terme «en place». Comme indiqué dans la section 1.2.2, l'image binaire I et l'image des éti-

(a) Masque direct (b) Masque inverse

Fig. 1.17 – Masques d'Haralick 8C

quettes E sont deux structures différentes : c'est ce qui explique que l'algorithme n'est pas directement exécuté dans I.



(a) Création de l'étiquette 1 (b) Création de l'étiquette 2 (c) Création de l'étiquette 3 (d) Propagation de l'étiquette 1

(e) Propagation de l'étiquette 2 (f) Propagation de l'étiquette 2 (g) Propagation de l'étiquette 1 (h) Propagation de l'étiquette 1

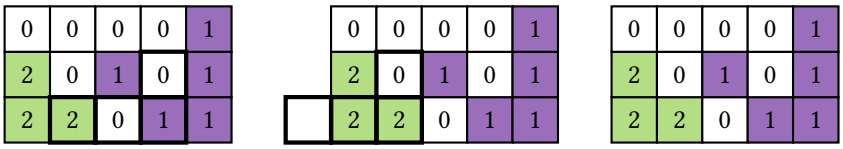
Fig. 1.18 – Haralick : Première passe - balayage direct

La figure 1.18 décrit le balayage direct de la première passe. Celui-ci est différent des autres passes car il contient l'initialisation à la volée de l'image des étiquettes E. Pour chaque pixel de premier plan de l'image, le minimum positif du voisinage passé (masque direct fig. 1.17a) est calculé et affecté au pixel courant. La procédure est identique à celle proposée par Rosenfeld à l'exception de l'absence de table d'équivalences. La relation d'équivalence «est connexe à» est donc perdue et seul les itérations jusqu'à stabilisation garantissent la validité du résultat final. La première divergence apparaît dans la figure 1.18f. Lorsque les étiquettes 3 et 2 sont présentes dans le voisinage, l'étiquette 2 est propagée mais l'information de connexité entre 3 et 2 est négligée. Cette situation se répète dans la figure 1.18g.

La phase directe est suivie par une phase inverse (fig. 1.19) qui utilise le masque inverse (fig. 1.17b) pour propager le minimum positif de la même manière. L'étiquette 3 est bien remplacée par l'étiquette 1 (fig. 1.19a) mais celle-ci ne peut être propagée à l'étiquette 2 (fig. 1.19b) et la première passe se termine sur le résultat incomplet de la figure 1.19c. L'image des étiquettes ayant été modifiée dans cette passe, le critère de stabilité n'est pas respecté et une passe supplémentaire doit être exécutée.

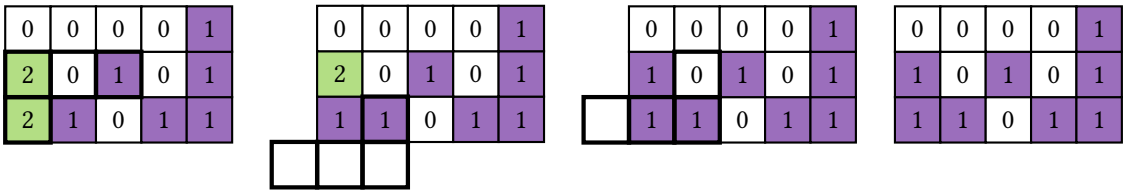
Dans la seconde passe (fig. 1.20), la phase directe n'apporte qu'une propagation (fig. 1.20a) mais celle-ci permet à la phase inverse de terminer l'étiquetage. Une troisième passe sera nécessaire pour atteindre la stabilité et l'étiquetage sera alors terminé.

Pour cette image très simple, seulement trois passes sont nécessaires. Pour des images plus complexes, le nombre d'itérations peut croitre très rapidement avec la taille de l'image jusqu'à rendre cette classe d'algorithme inutilisable pour des images de grande taille. C'est le cas des spirales présentées dans la section 6.2.2.



(a) Propagation de l'étiquette 1 (b) Propagation de l'étiquette 2 (c) Fin de la première passe

Fig. 1.19 – Haralick : Première passe - balayage inverse



(a) Phase descendante : Propagation de l'étiquette 1 (b) Phase ascendante : Propagation de l'étiquette 1 (c) Phase ascendante : Propagation de l'étiquette 1 (d) Résultat de la phase ascendante et de la troisième passe : l'image est stabilisée

Fig. 1.20 – Haralick - deuxième passe et stabilisation

Algorithme 7 : Haralick - première passe balayage direct 8C

Input : I une image binaire de taille $H \times W$ (0 pour le fond, 1 pour le premier plan)
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          if I[i][j] ≠ 0 then
4              e1 ← E[i-1][j-1]   e2 ← E[i-1][j]   e3 ← E[i-1][j+1]   e4 ← E[i][j-1]
5              if e1 = e2 = e3 = e4 = 0 then
6                  ex ← ne++
7              else
8                  ε ← min+(e1, e2, e3, e4)
9                  ex ← ε
10         else
11             ex ← 0
12         E[i][j] ← ex
```

Algorithme 8 : Haralick - balayage inverse 8C

Input : E une image partiellement étiquetée de taille $H \times W$ résultant d'une phase directe
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = H − 1 to 0 do
2      for j = W − 1 to 0 do
3          ex ← E[i][j]
4          if ex ≠ 0 then
5              e4 ← E[i][j+1]   e3 ← E[i+1][j-1]   e2 ← E[i+1][j]   e1 ← E[i+1][j+1]
6              ε ← min+(ex, e1, e2, e3, e4)
7              E[i][j] ← ε
```

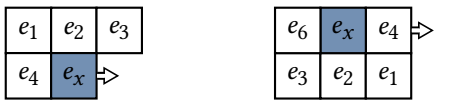
Algorithme 9 : Haralick : passes directes suivantes 8C

Input : E une image partiellement étiquetée de taille $H \times W$ résultant d'une phase inverse
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1 for i = 0 to H − 1 do
2     for j = 0 to W − 1 do
3         e_x ← E[i][j]
4         if e_x ≠ 0 then
5             e_1 ← E[i-1][j-1]   e_2 ← E[i-1][j]   e_3 ← E[i-1][j+1]   e_4 ← E[i][j-1]
6             ε ← min+(e_x, e_1, e_2, e_3, e_4)
7             E[i][j] ← ε
```

1.4.3 Lumia & Shapiro & Zuniga

En 1983, faisant le constat que les deux algorithmes précédents étaient inadaptés aux images de grande taille car ils provoquaient des défauts de pages, Lumia & Shapiro & Zuniga [86] ont proposé un algorithme deux passes hybride qui utilise T_L une table d'équivalences locale à l'échelle de la ligne courante.



(a) Masque direct (b) Masque remontant

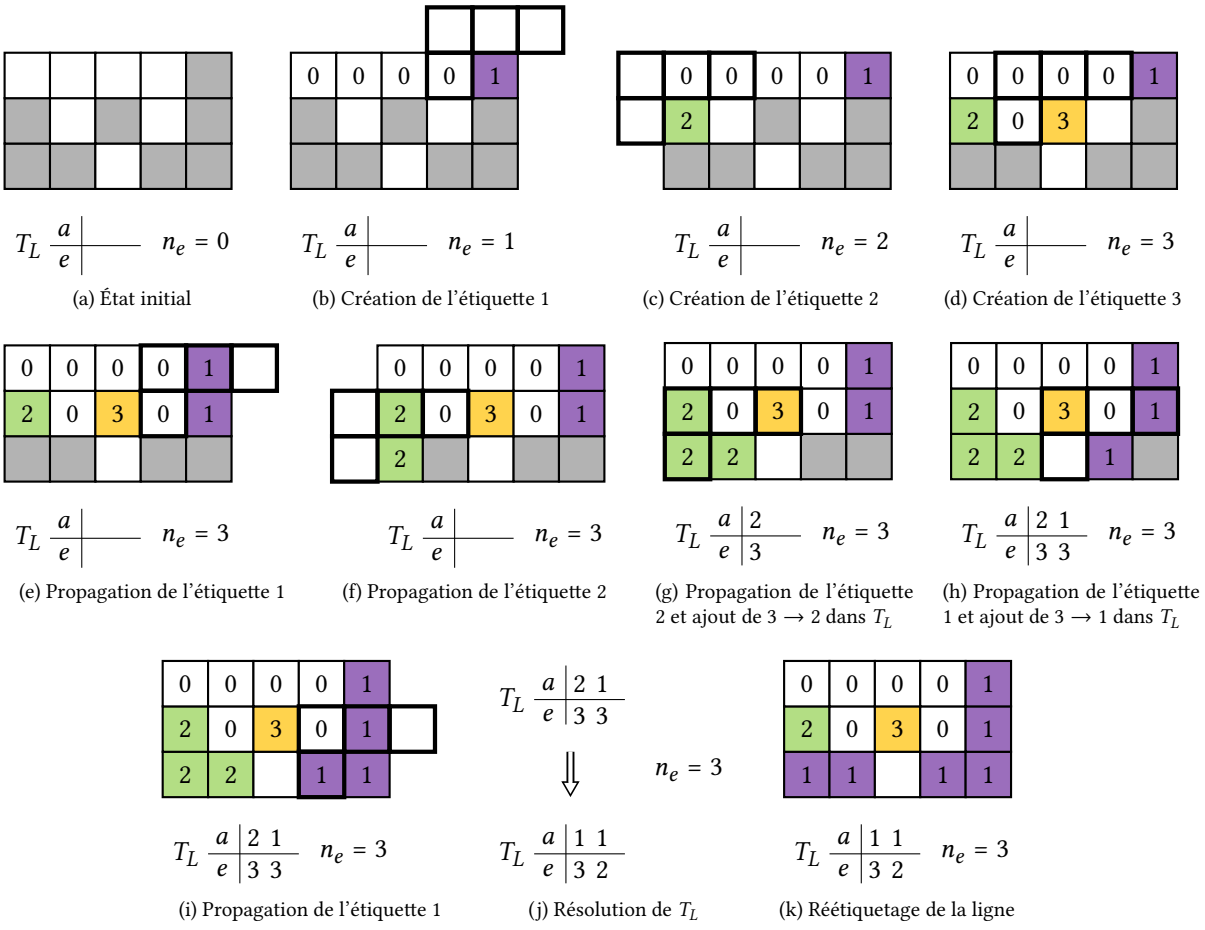
Fig. 1.21 – Masques de Lumia

La première passe (fig. 1.22) est très similaire à celles proposées par Haralick et Rosenfeld. Sa spécificité est que, au cours du traitement d'une ligne, lorsqu'une opération d'union est nécessaire, les deux étiquettes correspondantes sont stockées dans une petite table (figs. 1.22g et 1.22h). À la fin de la ligne, on réalise la fermeture transitive de la table (fig. 1.22j) avant de réétiqueter la ligne avec les racines des composantes connexes (fig. 1.22k). Tout comme pour la version originale de Rosenfeld, la méthode de fermeture de la table n'est pas décrite dans l'article et sa réalisation demande de parcourir plusieurs fois la table locale. Mais contrairement à l'algorithme de Rosenfeld, le caractère local de la table ne permet pas de remplacer la méthode de gestion des étiquettes par une procédure Union-Find globale.

Une seconde passe (fig. 1.23) remontante, utilisant le masque remontant spécifique de Lumia (fig. 1.21b), est alors réalisée.

Les auteurs indiquent que là où leur algorithme étiquette une image 4096×2500 en une heure sur un *VAX 11/780*[4], celui proposé par Haralick en mettrait 17 heures et celui de Rosenfeld 167 heures. Cet algorithme n'a pas connu de descendance directe du fait de cette table locale inadaptée à la procédure *Union-Find*. L'algorithme original est donc fourni en annexe pour mémoire mais sa compréhension fine n'est pas indispensable à la compréhension des algorithmes modernes. Dans le cadre de nos travaux, l'apport principal de l'algorithme est justement ce principe de table locale permettant une meilleure gestion de la mémoire. Cette table sera reprise et adaptée dans l'algorithme LSL (sec. 2.4.9) et est une des clefs de son efficacité.

4. https://fr.wikipedia.org/wiki/VAX

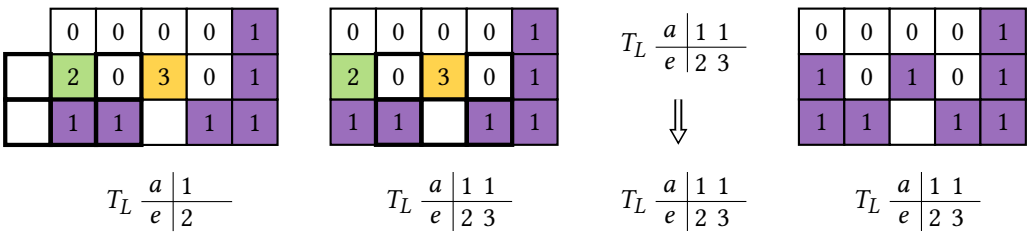


(a) État initial (b) Création de l'étiquette 1 (c) Création de l'étiquette 2 (d) Création de l'étiquette 3

(e) Propagation de l'étiquette 1 (f) Propagation de l'étiquette 2 (g) Propagation de l'étiquette 2 et ajout de 3 → 2 dans T_L (h) Propagation de l'étiquette 1 et ajout de 3 → 1 dans T_L

(i) Propagation de l'étiquette 1 (j) Résolution de T_L (k) Réétiquetage de la ligne

Fig. 1.22 – Étapes de l'algorithme de Lumia - passe sens direct

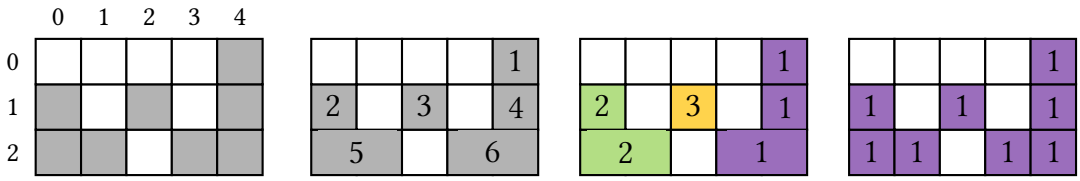


(a) Ajout de 2 → 1 dans T_L (b) Ajout de 3 → 1 dans T_L (c) Résolution de T_L (d) Réétiquetage de la ligne

Fig. 1.23 – Étapes de l'algorithme de Lumia - passe remontante

1.4.4 Ronse & Devijver

Dans [87], C. Ronse & P.A. Devijver ont décrit le premier algorithme «segment» pour l'étiquetage en composantes connexes. Plutôt que de traiter l'image pixel à pixel, les auteurs considèrent qu'ils sont en possession d'une liste des segments horizontaux de l'image (tab. 1.24e et fig. 1.24b). Cette liste est obtenue à partir d'un matériel spécifique spécialisé dans l'extraction de segments. Pour les images binaires, la représentation d'un segment par le couple formé par ses coordonnées ou par une de ses coordonnées et la longueur du segment est l'équivalent du codage RLC (Run Length Coding). Le fond étant le complément du premier plan toute l'information est contenu dans cette représentation. À partir de cette liste, l'algorithme affecte une étiquette à chaque segment et produit la liste des équivalences (tab. 1.24f) qu'il maintient à l'aide d'un système de listes chaînées (non décrites ici car n'ayant pas été réutilisée). Une passe de relecture des étiquettes des segments au travers de la table contenant les équivalences permet de produire l'image finale des étiquettes. Cette seconde passe est l'équivalent du réétiquetage de l'algorithme de Rosenfeld.



(a) Image d'origine

(b) Image décomposée en segments

(c) Segments étiquetés avant résolution

(d) Image étiquetée

N°	Ligne	Début	Fin	Étiquette
1	0	4	4	1
2	1	0	0	2
3	1	2	2	3
4	1	4	4	1
5	2	0	1	2
6	2	3	4	1

(e) Représentation segment + étiquettes avant résolution des équivalences

3→2
2→1

(f) Équivalences résolues

Fig. 1.24 – Représentation segment

La représentation en segments est l'apport principal de cet algorithme car elle permet dès que les segments sont de taille supérieure à 2 de gagner en compacité et de gérer le voisinage et les connexités à l'échelle du segment limitant ainsi la création d'étiquettes supplémentaires. Elle sera reprise dans l'algorithme LSL (sec. 2.4.9).

1.5 Contraintes algorithmiques et architecturales

Le propos de nos travaux étant d'étudier l'adéquation entre les algorithmes d'étiquetage en composantes connexes et les architectures modernes pour proposer les algorithmes et leurs implémentations les plus efficaces selon les architectures, nous nous intéressons dans cette section aux caractéristiques en lien avec les performances. En posant les bases de l'étiquetage moderne, les précurseurs ont défini les principales caractéristiques de l'étiquetage en composantes connexes qui restent valables aujourd'hui.

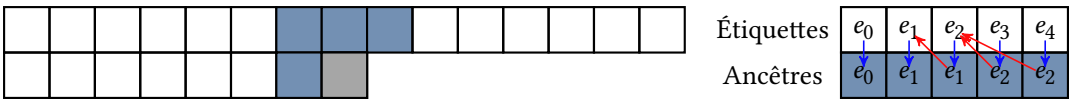
- Ces algorithmes sont basés sur une topologie hybride.
- Le ratio nombre de calculs / nombre d'accès mémoire (intensité arithmétique) est faible.
- Ils sont tous créateurs d'étiquettes supplémentaires.

Ces caractéristiques sont la clef de compréhension des algorithmes modernes qui, chacun à leur façon, contribuent à améliorer un point ou un autre.

1.5.1 Topologie des algorithmes d'étiquetage en composantes connexes : un mélange de données éparses et denses

L'étiquetage en composantes connexes fait intervenir deux types de structures des données

- Une structure dense : les pixels de l'image sont par nature contigus et sont parcourus dans le sens direct qui est le sens naturel pour le processeur et ses caches (fig. 1.25a).
- Une structure éparse : bien que la table d'équivalence soit ordonnée, les étiquettes rencontrées peuvent être très distantes dans la table et *a fortiori* la nécessaire remontée dans l'arbre à la recherche de la racine de la composante connexe n'est absolument pas linéaire et ne tire que très peu profit des caches de bas niveau car elle nécessite une remontée des étiquettes les plus grandes vers les plus petites (fig. 1.25b).



(a) Topologie mémoire fortement connexe

(b) Topologie mémoire dispersée

Fig. 1.25 – Topologies mémoires de l'étiquetage en composantes connexes

Il apparaît clairement que l'étiquetage en composantes connexes est fortement dépendant des données et qu'un enjeu des améliorations successives est de limiter entre autres l'impact de cette dépendance.

1.5.2 Problématique de l'intensité arithmétique

En nous basant sur l'algorithme de Rosenfeld, il est possible de caractériser le nombre de calculs, d'accès mémoire et de tests par pixel. Le but ici n'est pas de donner des valeurs exactes (qui sont dépendantes des données) mais d'illustrer que les opérations de chargement et de contrôle sont prépondérantes.

- Si le pixel est un pixel de fond : un chargement, un test, un rangement dans un registre et un rangement en mémoire sont nécessaires.
- Si le pixel n'a aucune étiquette dans son voisinage : il faut ajouter quatre chargements (le chargement du voisinage), quatre tests (les voisins sont-ils tous nuls ?), un chargement (*ne*), une incrémentation d'un registre.
- Si le pixel a un ou plusieurs pixels dans le voisinage :
  - S'ils appartiennent tous à la même composante connexe : il faut ajouter un nombre indéterminé de chargements et de tests pour atteindre la racine de la composante connexe pour chaque pixel du voisinage, trois tests pour déterminer le minimum et huit tests pour s'assurer qu'il n'y a pas de mise à jour à réaliser.
  - Sinon il faut ajouter, pour chaque pixel dont la racine est supérieure au minimum des racines, un nombre indéterminé de chargements et de tests ainsi qu'un rangement dans un registre.

Il apparaît que la majorité des actions sont des opérations de chargements/rangements et des opérations de contrôle pour très peu de calculs. Les performances globales de l'algorithme de Rosenfeld sont donc liées à celles de la mémoire. L'enjeu des améliorations présentées dans le chapitre suivant sera entre autres d'améliorer ce rapport pour diminuer l'impact des opérations de contrôle, de chargements et de rangements.

1.5.3 Étiquettes supplémentaires

Dans le vocabulaire de l'étiquetage en composantes connexes, il existe trois types d'étiquettes :

- Les étiquettes finales qui sont les racines des arbres et qui représentent donc le nombre réel de composantes connexes de l'image.
- Les étiquettes temporaires qui sont l'ensemble des étiquettes nécessaires à l'étiquetage complet de l'image.
- Les étiquettes supplémentaires qui sont toutes les étiquettes temporaires qui ne sont pas des racines à la fin de l'étiquetage.

Selon l'image et l'algorithme, le nombre d'étiquettes supplémentaires peut être très élevé. Dans la section 2.3, nous étudierons leur évolution en fonction de l'algorithme utilisé et de l'image considérée. C'est l'existence des étiquettes supplémentaires qui conduit à la nécessité de réaliser des opérations d'*Union* et qui fait que les arbres ont une profondeur supérieure à 1. Réduire ce nombre et la hauteur de l'arbre de chaque composante connexe aura un impact sur les performances des algorithmes.

A titre d'exemple, pour l'algorithme de Rosenfeld (et tous ceux qui utiliseront le masque correspondant), deux formes minimales sont génératrices d'étiquettes supplémentaires : les marches d'escalier (fig. 1.26a) et le V (fig. 1.26b). Toute forme dérivant de ces formes minimales sera génératrice d'étiquettes supplémentaires. Il est intéressant de remarquer que les algorithmes segment seront insensibles aux marches d'escaliers car les équivalences sont évaluées pour l'ensemble du segment.

(a) Forme en marche d'escalier (b) Forme en V

Fig. 1.26 – Formes génératrices d'étiquettes supplémentaires pour le masque de Rosenfeld

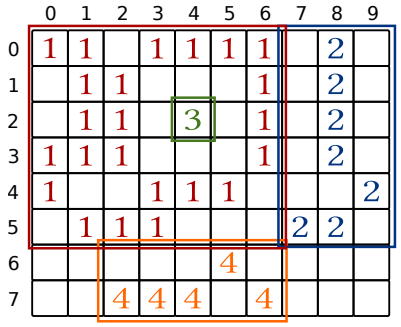
1.6 Analyse en composantes connexes

Dans la mouvance de l'étiquetage en composantes connexes, l'analyse en composantes connexes (ACC) est l'opération qui extrait des informations sur les différentes composantes connexes d'une image binaire (fig. 1.27a et tab. 1.27b).

1.6.1 Descripteurs d'une composante connexe

Dans un chaîne de traitement d'images, l'image des étiquettes est la donnée d'entrée d'algorithmes spécialisés capables d'analyser les composantes connexes pour en extraire des caractéristiques utiles au traitement global. Par exemple dans une chaîne d'O.C.R. ou de suivi d'objets, l'algorithme cherchera à isoler les différentes composantes connexes et à en extraire une *signature* pour les identifier. Dans la suite du manuscrit, nous nommerons ces caractéristiques des *descripteurs* d'une composante connexe.

Les descripteurs usuellement recherchés (fig. 1.27a) sont :



(a) Représentation des descripteurs

cc	BB_{min}	BB_{max}	S	S_i	S_j
1	(0,0)	(5,6)	23	64	52
2	(0,7)	(5,9)	7	20	56
3	(2,4)	(2,4)	1	2	4
4	(6,2)	(7,6)	5	34	20

(b) Descripteurs

Fig. 1.27 – Extractions des descripteurs des composantes connexes

- la boîte englobante $[i_{min}, i_{max}] \times [j_{min}, j_{max}]$,
- le moment d'ordre 0 : S,
- les moments d'ordre 1 : S_x et S_y.

1.6.2 Calcul des descripteurs

Calculer ces descripteurs, en même temps que l'image des étiquettes se construit (à la volée), est une solution à la problématique de la prédominance des opérations de contrôle et peut permettre de s'affranchir de la phase de réétiquetage.

- la boîte englobante : le calcul des coordonnées de la boîte englobante repose sur la comparaison entre les coordonnées de la boîte englobante en cours de construction et entre les coordonnées minimales (l_{min}, h_{min}) et maximales (l_{max}, h_{max}) de l'entité ajoutée (pour les pixels $l_{min} = l_{max}$ et $h_{min} = h_{max}$, pour les segments $h_{min} = h_{max}$ et l_{min} = début du segment et l_{max} = fin du segment).
- Le moment d'ordre 0 : le nombre de pixels de la composante connexe est obtenu en cumulant le nombre de pixels de l'entité (pour les pixels 1, pour les segments la longueur du segment).

$$S = \sum_{k=1}^{N} 1 = N$$

- Les moments d'ordre 1 : S_i et S_j qui permettent d'obtenir les coordonnées du centre de masse des composantes connexes.

$$S_x = \sum_{k=1}^{N} j_k, \qquad S_y = \sum_{k=1}^{N} i_k$$

1.7 Conclusion

Ce chapitre a mis en évidence la diversité des approches et la dépendance des différents algorithmes aux données à traiter. Afin de réaliser une comparaison équitable et signifiante de ces algorithmes et afin d'analyser l'efficacité des différentes approches, il est nécessaire de disposer d'une méthodologie d'évaluation permettant d'étudier les algorithmes dans le détail et de l'appliquer aux algorithmes modernes les plus représentatifs. Les algorithmes directs seront abordés dans les chapitres 2 à 5 et les algorithmes itératifs dans les chapitres 6 et 7.

Chapitre 2

Mais le hasard est un allié aussi fugitif que mortel. Il te tue avec la même facilité qu'il te sauve. Apprend à réduire ce fauve à la dimension d'un chat. Circonscris la turbulence. Les meilleurs aéromaîtres caressent un chaton et jouent à la pelote avec lui. Un chaton, pas un tigre.

–La Horde du contrevent, Alain Damasio

Etat de l'art des algorithmes séquentiels d'étiquetage en composantes connexes

2.1 Introduction

Le chapitre précédent a mis en évidence les fondements de l'étiquetage en composantes connexes ainsi que les contraintes des premiers algorithmes. La nature de l'étiquetage en composantes connexes impose que tous les algorithmes produisent les mêmes données. Cela signifie que les pixels connexes doivent être groupés dans une même classe d'équivalence et ce n'est pas sur ce point que les améliorations peuvent survenir. Faire évoluer un algorithme ou en créer un nouveau doit donc conduire à une progression des performances selon un critère précis dépendant de son contexte d'utilisation : rapidité d'exécution, empreinte mémoire limitée, stabilité et prédictibilité du temps d'exécution.

La littérature est très prolixe depuis le début des années 2000 en nouveaux algorithmes qui se revendiquent tous comme étant les plus rapides mais qui se comparent rarement entre eux. Lorsque des mesures de performance sont proposées dans les articles, elles le sont dans des cadres fortement différents ce qui empêche toute comparaison directe.

Afin de comprendre l'influence des différentes optimisations présentées, il a donc été nécessaire de proposer un jeu de données unifié et d'y soumettre chaque algorithme afin de fournir des résultats directement comparables. Ce jeu de données et ses spécificités seront présentés dans la section 2.2. Une fois cette procédure établie, nous présenterons une sélection des différentes améliorations algorithmiques modernes dans la section 2.4 .

2.2 Construction d'un jeu de données unifié

2.2.1 Images épreuves

Afin de proposer des comparatifs cohérents et pertinents des différents algorithmes et de mettre en évidence les spécificités de chaque variante tout en permettant à l'ensemble de la communauté de l'étiquetage en composantes connexes de comparer nos résultats aux leurs et afin aussi de mettre en évidence l'adéquation des algorithmes aux architectures mises en œuvre, nous avons fait le choix de créer une procédure de tests de performance ayant quatre caractéristiques :

- la procédure doit être reproductible,
- les paramètres de la procédure doivent avoir une influence directe et lisible sur les performances des algorithmes,
- la procédure doit mettre en évidence le comportement des algorithmes dans le cas d'applications courantes comme dans des cas complexes.
- la procédure doit exprimer les résultats dans des unités permettant de rendre compte des performances de l'algorithme plutôt que de celles de la machine sur laquelle les tests ont été réalisés.

L'état de l'art concernant les jeux de données repose principalement sur la notion de bases de données d'images. On retrouve SIDBA [88] dans de nombreuses contributions, ainsi qu'une quantité non négligeable d'images réputées complexes qui n'apparaissent que dans une publication. Quelques travaux se basent sur des images aléatoires aux propriétés (densité, modèle de générateur, graine) inconnues.

Afin de concilier à la fois la pertinence du test tout en offrant la possibilité aux équipes ayant déjà réalisé des travaux de comparer leurs résultats aux nôtres, nous avons choisi une approche double. D'une part, les tests seront effectués sur la base de données la plus utilisée (SIDBA) pour une comparaison directe avec la littérature. D'autre part, des tests seront réalisés sur un système reproductible et discriminant d'images aléatoires basé sur trois paramètres : la taille, la densité et la granularité.

2.2.2 Taille des images

La taille de l'image de test est un facteur qui permet de mettre en évidence le comportement de chaque algorithme face à la gestion de la mémoire (gestion des différents niveaux de caches, empreinte mémoire totale) et à l'impact de celle-ci sur les performances. Dans l'ensemble du document, nous avons fait varier la taille des images pour rester dans un cadre cohérent en fonction des architectures utilisées. En effet une architecture 4×15 cœurs n'est pas employée dans le même contexte qu'une architecture 4 cœurs ou qu'un processeur embarqué. Selon les cas (séquentiel/parallèle/embarqué/-serveur), les images aléatoires varient donc de 1024×1024 à 8192×8192 pixels.

Le même problème se pose pour la base de données SIDBA. La taille des images utilisées dans cette base est de 800×600. Cette taille ne permet pas d'évaluer avec pertinence les performances des différentes parallélisations et ne sont pas ou plus représentatives d'une application moderne. Nous avons fait le choix d'étendre la base SIDBA par homothétie. SIDBA4 sera donc la base d'images de taille 3200×2400 issue d'une mise à l'échelle exacte de la base SIDBA (chaque pixel est remplacé par un bloc 4×4).

Dans un environnement de production, la taille moyenne des images à traiter est variable selon l'application considérée mais la tendance générale est à l'augmentation de celle-ci avec la diffusion à grande échelle de capteurs de résolution supérieure à 16Mp via les smartphones, tablettes, appareils photo. La table 2.1 recense quelques résolutions remarquables et donne un aperçu de la variété des résolutions auxquelles l'étiquetage en composantes connexes doit pouvoir être appliqué.

Nom	Hauteur(px)	Largeur(px)	Total(px)
QVGA	320	240	307K
VGA	640	480	307K
SVGA	800	600	480K
XGA	1024	768	786K
HD	1280	720	921K
ARGUS-IS [89]	368 capteurs de 5Mpx		1,9M
FHD	1920	1080	2,1M
WQHD	2560	1440	3,7M
UHD (4K)	3840	2160	8,3M
FUHD (8K)	7680	4320	33,2M
Nasa Gigapan Tile	7621	7391	56,3 M
Nasa Gigapan	7680	2,5M	19,1 G
QUHD (16K)	15360	8640	132,7M

TABLE 2.1 – Résolutions remarquables et quantité de données correspondante

2.2.3 Métriques

La figure 2.1a représente le temps de traitement d'une série d'images de taille croissante de $N^2 = 32 \times 32$ pixels à $N^2 = 8192 \times 8192$ (avec N la base de l'image) sur deux architectures Intel :

- SDB - SandyBridge i7-2600 1x4 cœurs 3.4GHz - DDR3 1067MHz (un cœur utilisé ici)
- SKL - Skylake i7-6700 1x4 cœurs 4GHz - DDR4 3466MHz (un cœur utilisé ici)

Chaque courbe met en évidence l'évolution affine du temps de traitement total de l'image en fonction du nombre de pixels de l'image (ici N^2) et permet de se rendre compte du caractère temps réel ou non des résultats. Cependant la comparaison des deux courbes apporte plus d'informations sur la différence de fréquence des deux architectures que sur leurs capacités intrinsèques de traitement. Deux campagnes de tests réalisés sur des architectures même faiblement différentes ne donneront donc pas de résultats comparables ce qui rend très complexe la comparaison entre les travaux de recherche des différentes équipes.

Afin de générer des résultats (courbes, tableaux) pertinents, nous proposons donc de mettre en place deux métriques communes à tous les tests en exprimant à chaque fois que cela est possible les résultats en termes de cycles par points (*cpp*) et si nécessaire en nombre de pixels traités par seconde (débit) exprimé en Gp/s. L'intérêt du *cpp* est qu'il permet de comparer les performances d'une architecture au sein d'une même classe d'architectures. Tandis que le nombre de pixels traités par seconde permet de comparer différentes classes d'architectures indépendamment de leur fréquence ou de leur degré de parallélisme.

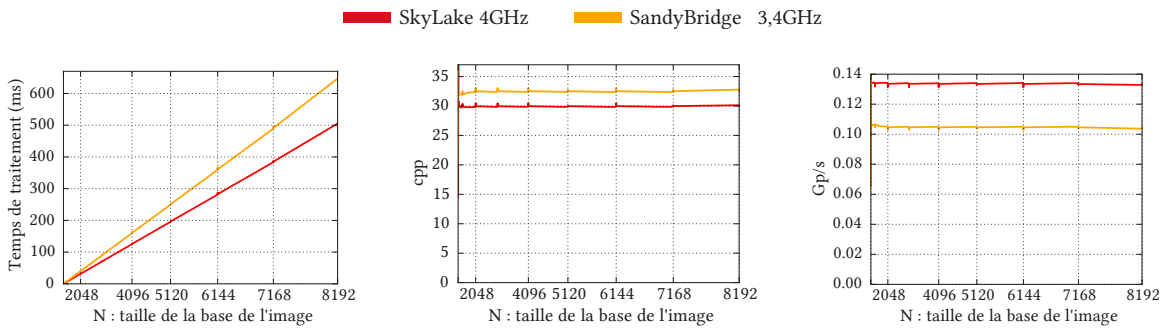
La figure 2.1b représente la même campagne de tests que celle de la figure 2.1a avec comme métrique le *cpp* (Gp/s pour la figure 2.1c). Ces représentations rendent plus simple la comparaison des algorithmes pour une architecture fixée et celle des architectures pour un algorithme donné.

2.2.4 Reproductibilité des images aléatoires

Dans le cadre d'une démarche de recherche reproductible [90], nous avons cherché à minimiser les aléas sur la création des images aléatoires.

2.2.4.1 Choix du générateur pseudo-aléatoire

Pour chaque pixel de l'image, un générateur pseudo-aléatoire tire une valeur réelle entre 0 et 100. Cette valeur est seuillée par la valeur de la densité recherchée pour déterminer si le pixel appartient au premier plan («1») dans le cas où la valeur est inférieure au seuil ou à l'arrière plan («0») sinon.



(a) La métrique «temps de traitement» informe sur le caractère temps réel et sur la position relative des algorithmes sur une machine donnée

(b) La métrique «cycles par pixel» permet de s'affranchir de la taille de l'image et de la fréquence du processeur

(c) La métrique «pixels par seconde» permet de s'affranchir de la taille de l'image et de comparer des architectures très différentes

Fig. 2.1 – Métriques et dépendance à la taille N^2 de l'image et à la fréquence du processeur

Afin d'assurer la reproductibilité des tests, il est nécessaire de fixer et de rendre publique la graine d'amorçage du générateur pseudo-aléatoire. Cependant, selon les types et versions des systèmes d'exploitation, selon les compilateurs et les architectures, le comportement du générateur peut varier. La solution proposée ici est de se baser sur un générateur pseudo-aléatoire indépendant de l'environnement : le générateur *Mersene Twister* [91].

Celui-ci a l'avantage d'avoir une période très longue ($2^{19937} - 1$) et d'être très répandu dans la communauté scientifique. Son code source est disponible en C (langage utilisé pour nos travaux) et il est le générateur par défaut pour les langages Python, Ruby, R, PHP ainsi que pour l'environnement MATLAB. Il est disponible pour C++ depuis la version C++11.

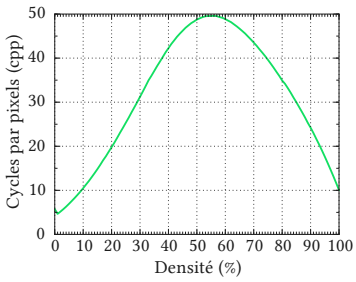
2.2.4.2 Choix de la graine d'amorçage

La reproductibilité du test est nécessaire mais le choix de la graine ne doit pas biaiser les résultats. Afin de s'assurer que le choix de la graine d'initialisation ne favorise pas un algorithme par rapport à un autre, nous avons réalisé une étude sur l'influence de celle-ci sur les performances des algorithmes. Cette étude a mis en évidence que la variabilité des résultats en fonction de la graine était en moyenne inférieure à ±1% et dans le pire des cas contenue dans une enveloppe de ±2.5% autour de la valeur moyenne. L'influence de la graine sur le temps moyen d'étiquetage est dans le pire des cas inférieure à 2%.

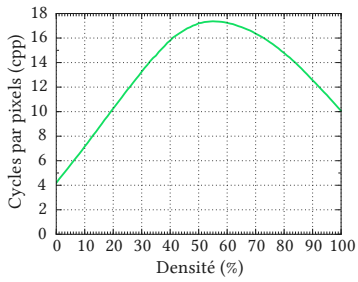
2.2.5 Densité

La densité d de l'image de test est le nombre de pixels de premier plan par rapport au nombre total de pixels de l'image. La densité est de fait un élément qui contribue à la difficulté d'étiquetage de l'image car dans la plupart des algorithmes, l'analyse du voisinage n'est effectuée que pour les pixels de premier plan. Dans l'ensemble des tests de performance, nous avons fait varier la densité d de 0 à 100% avec un pas de 1%. La figure 2.2a représente l'évolution du nombre de cycles par pixel (*cpp*) pour l'algorithme de Rosenfeld pour une taille d'image de 1024×1024 sur la machine SKL. L'évolution du cpp en fonction de la densité est globalement symétrique par rapport à la densité $d = 56\%$. Cela met en évidence la difficulté que représente la gestion de la table d'équivalences. En effet pour une densité nulle ($d = 0\%$) la valeur du cpp est de 4,2 alors que pour une densité maximale ($d = 100\%$) la valeur du cpp est de 10,1. Entre ces deux valeurs, le cpp augmente fortement avec un maximum à 49,6. Le nombre de pixels à traiter n'est donc pas le seul facteur déterminant : c'est leur agencement qui augmente la difficulté. La simple étude en densité est donc nécessaire mais non suffisante : c'est pourquoi nous proposons en complément le paramètre de la granularité.

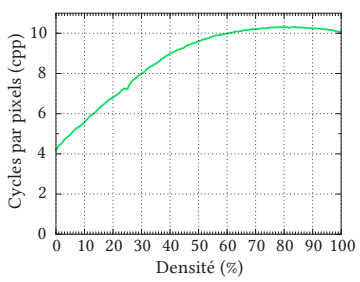
2.2.6 Granularité



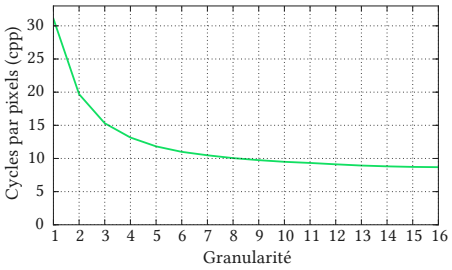
(a) Évolution du *cpp* pour g=1, le ratio $cpp_{max} : cpp_{100\%} \approx 5$



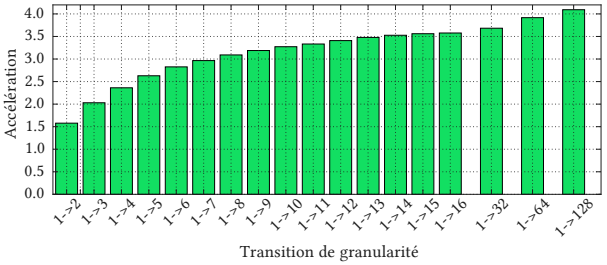
(b) Évolution du *cpp* pour g=4, le ratio $cpp_{max} : cpp_{100\%} \approx 1.75$



(c) Évolution du *cpp* pour g=16, le ratio $cpp_{max} : cpp_{100\%} \approx 1.03$



(d) Évolution du *cpp* moyen pour $g \in [1; 16]$

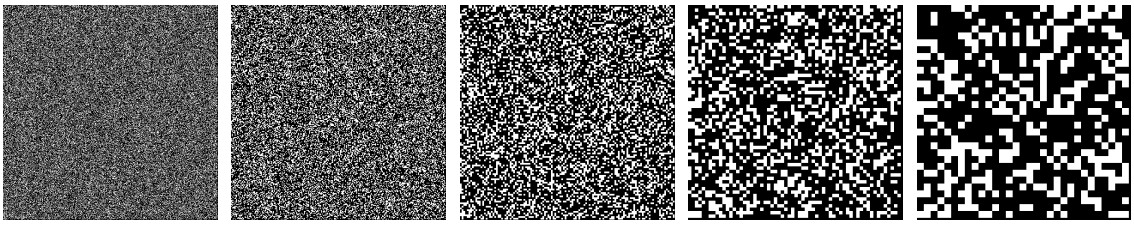


(e) Taux d'accélération du temps de traitement de l'image en passant de g=1 à $g \in [2; 16]$ et à $g \in \{32, 64, 128\}$

Fig. 2.2 – Évolution du *cpp* en fonction de la densité pour $g = 1$, $g = 4$ et $g = 16$ et de *cpp* moyen sur l'ensemble des densités en fonction de la granularité

La granularité g de l'image de test est la taille (base) du bloc utilisé pour la génération de l'image aléatoire. Dans le cas d'une image de granularité g=1, chaque pixel fait l'objet d'une détermination pseudo-aléatoire de son état alors que dans le cas d'une image de granularité g=16, ce sont des blocs de $16 \times 16 = 256$ pixels qui sont affectés à chaque fois.

La figure 2.3 représente les images de test pour une densité de 35% et une granularité g variant entre 1 et 16



(a) $g = 1$ (b) $g = 2$ (c) $g = 4$ (d) $g = 8$ (e) $g = 16$

Fig. 2.3 – Images aléatoires de densité 35% pour une granularité $g \in 1, 2, 4, 8, 16$ et une taille d'image de 1024×1024

Le choix de faire varier la granularité g de l'image de test repose sur le caractère fortement topologique des algorithmes. Pour une même densité de pixels de premier plan dans l'image, la concentration de ces points d'intérêt dans un nombre réduit de composantes connexes a une implication forte sur les performances des algorithmes. Les courbes des figures 2.2b et 2.2c représentent l'évolution du cpp pour des granularités plus élevées. On constate que très logiquement les valeurs pour les

densités $d = 0\%$ et $d = 100\%$ sont inchangées mais que le *cpp* moyen décroît et que la courbe de *cpp* se rapproche de la droite reliant $cpp_{0\%}$ et $cpp_{100\%}$. Cette diminution est continue mais son effet est décroissant en fonction de la granularité. L'accélération moyenne obtenue en passant d'une image de granularité g=1 une granularité $g = k$ est représentée dans la figure 2.2e et montre que chaque augmentation de la granularité augmente les performances. Nous avons limité notre jeu de données aux granularités. On peut constater que le passage de g=1 à g=16 représente $\approx 87\%$ du gain de performances obtenu lors du passage de g=1 à g=128 pour seulement $\approx 12\%$ de la plage de valeurs. Les valeurs obtenues pour g=16 informent sur la tendance des performances pour des images plus structurées. Afin de garder le jeu de données utilisable, nous nous limiterons dans la suite de nos travaux à l'étude des granularité de 1 à 16.

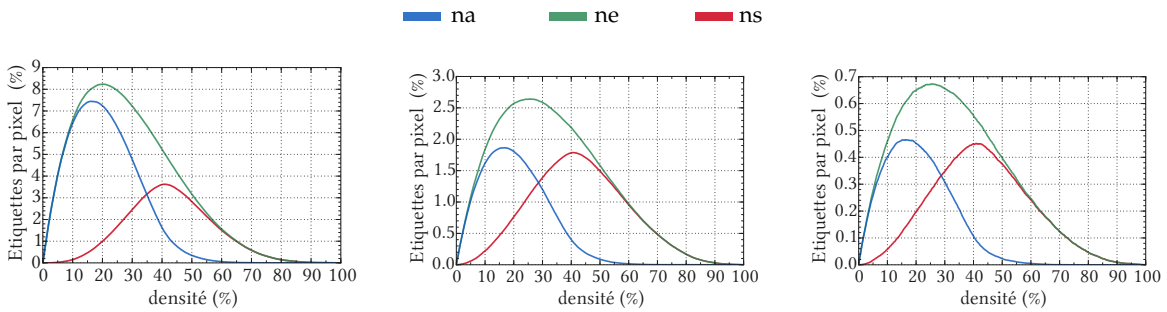
Le rôle de la base SIDBA sera alors de confirmer les performances sur des images plus structurées. Pour chaque élément de SIDBA, nous avons évalué la granularité équivalente g_e de l'image. Il s'agit de comparer le *cpp* de l'image avec les *cpp* des images aléatoires de densité équivalente. Pour les images *city1* et *city2*, $7 < g_e < 8$, pour *home4*, $g_e \approx 12$, pour *room1* g_e n'est pas calculable car le *cpp* est plus faible que pour toutes les images aléatoires utilisées. Pour les autres images, $16 < g_e < 128$.

2.3 Analyse des caractéristiques du jeu de données

Le jeu de données proposé est reproductible. Il fournit des images aléatoires très complexes à étiqueter ($g = 1$) qui permettent d'évaluer le comportement des algorithmes dans les pires cas mais aussi des images aléatoires mettant en évidence le comportement des algorithmes pour des images plus réalistes ($g \geq 8$) et des images réelles. Afin de comprendre les formes des courbes de *cpp* en fonction de la densité ou de la granularité, il est important de connaître l'influence des différents paramètres sur la structure des données.

2.3.1 Influence des paramètres du jeu de données sur le nombre d'étiquettes

Si la taille (N^2), la densité (d) et la granularité (g) sont les paramètres commandables du jeu de données, le nombre de pixels de premier plan ($d \times N^2$), le nombre de composantes connexes (na), le nombre d'étiquettes temporaires (ne) et le nombre d'étiquettes « supplémentaires » ($ns = ne - na$) sont des indicateurs de la difficulté d'étiquetage de l'image.



(a) g=1

(b) g=2, ns augmente en proportion

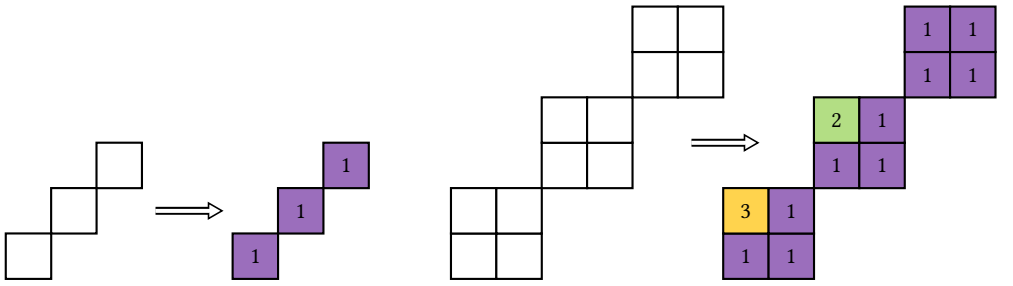
(c) g=4, homothétie de la courbe g=2 (rapport 1/4)

Fig. 2.4 – Mise en évidence de l'évolution du nombre de composantes connexes (na), du nombre d'étiquettes temporaires (ne) et du nombre d'étiquettes supplémentaires ($ns = ne - na$) en fonction de la granularité

Afin de s'affranchir de la taille de l'image, les courbes de la figure 2.4 sont exprimées en nombre d'étiquettes par pixel de l'image. L'algorithme de référence utilisé est l'algorithme de Rosenfeld.

Pour une granularité $g = 1$ (fig. 2.4a), $na_{max} = 78022$ composantes connexes, ce qui correspond à $\approx 7{,}44\%$ pour une image 1024×1024. Ce maximum est atteint pour une densité $d = 16{,}5\%$. Le nombre d'étiquettes temporaires est supérieur au nombre de composantes connexes du fait du caractère local de l'algorithme alors que le problème à traiter est global. Il atteint son maximum ($\approx 8{,}25\%$) pour une densité supérieure $d = 20{,}4\%$. La courbe *ns* représente le nombre d'étiquettes qui n'ont plus d'utilité à la fin de l'exécution de l'algorithme : c'est donc un indicateur de la surconsommation de ressources (mémoire et opérations) introduite par l'algorithme lui-même. Les algorithmes modernes présentés dans ce chapitre vont chacun à leur manière agir sur cet indicateur. Par exemple, la forme du masque va influer sur la valeur de *ns* tandis que la méthode de gestion des équivalences va influer sur son impact sur les opérations d'union. Le maximum $ns_{max} = 3{,}6\%$ est atteint pour $d \approx 41\%$.

Pour la granularité g=2 (fig. 2.4b), la forme des courbes diffère. Le maximum de la courbe *ne* se décale vers $d \approx 26\%$ et bien que la proportion d'étiquettes supplémentaires *ns* augmente significativement par rapport au nombre de composantes connexes de l'image, leur nombre diminue. Cette augmentation de la proportion et cette diminution globale s'expliquent par deux phénomènes conjugués. D'une part, l'augmentation de la taille des blocs divise le nombre de composantes connexes et donc le nombre de connexions réellement aléatoires par g^2 (la surface des blocs). D'autre part, cette même augmentation crée des configurations en *marches d'escalier* (sec. 1.5.3) supplémentaires (fig. 2.5). Malgré ces modifications, le maximum des étiquettes supplémentaires est toujours atteint pour $d \approx 41\%$.



(a) Marche d'escalier non créatrice d'étiquettes supplémentaires car à l'échelle du masque

(b) Marche d'escalier créatrice d'étiquettes supplémentaires car en dehors de la portée du masque

Fig. 2.5 – Création de nouvelles étiquettes supplémentaires lors de l'augmentation de la granularité

Toutes les courbes pour des granularités $g > 2$ sont des homothéties des courbes obtenues pour $g = 2$ (fig. 2.4c) car aucune nouvelle cause d'étiquettes supplémentaires n'apparaît et seule l'augmentation de la surface des blocs influe (g^2).

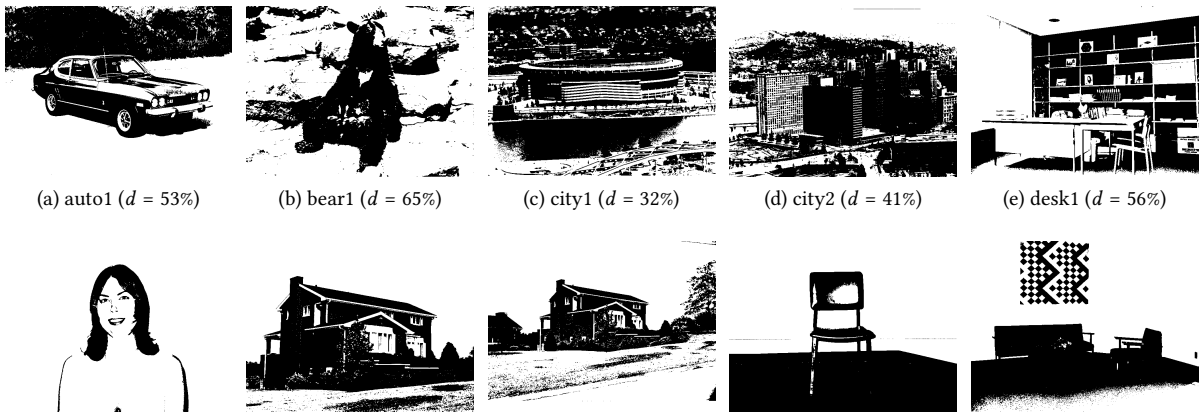
Quel que soit g, la courbe *ns* est maximale pour $d \approx 41\%$. Cette valeur charnière est le *seuil de percolation*. La théorie de la percolation de sites s'intéresse à la création d'amas (agglutination) dans un milieu aléatoire (à l'origine en science des matériaux). Au delà d'une certaine densité, les amas sont peu nombreux et de grande taille alors qu'en deçà, les amas sont nombreux et de petite taille. Dans [92], les auteurs mettent en évidence que ce seuil pour une maille carrée est de 40,77%. C'est autour de ce point de percolation que se situe la difficulté maximum pour les algorithmes. En effet, ce point conjugue à la fois des composantes connexes étendues et «fines» (peu de pixels comparativement à l'enveloppe de la composante connexe), ce qui favorise l'apparition de configurations en marche d'escalier et en V. Cette évolution rapide de la structure des composantes connexes autour du point de percolation est mise en évidence dans la figure 2.6.

(a) d = 35% (b) d = 40% (c) d = 41% (d) d = 42% (e) d = 47%

Fig. 2.6 – Mise en évidence de l'évolution de la taille des composantes connexes

2.3.2 Cas des images des bases de données SIDBA et SIDBA4

La base de données d'images SIDBA a été développée par l'université de Tokyo [88] et est utilisée par une majorité d'articles d'étiquetage depuis 2003. Les images en niveau de gris ont été binarisées en utilisant l'algorithme de Otsu [93]. Elles sont de taille 800×600. Toujours dans un souci de reproductibilité, le jeu d'images ainsi binarisé et son équivalent pour SIDBA4 sont disponibles en ligne [1]. La figure 2.8a représente la densité de composantes connexes et d'étiquettes supplémentaires par pixel de chacune des images de la base. On constate une forte variabilité entre les images, allant jusqu'à $na_{city1}/na_{face1} = 24{,}3$. SIDBA4 étant obtenue en remplaçant chaque pixel de SIDBA par un bloc de taille 4×4, le nombre de composantes connexes n'évolue pas et leur proportion par rapport au nombre de pixels de l'image est donc divisée par 16. Pour les raisons décrites dans la figure 2.5, la proportion d'étiquettes supplémentaires progresse par rapport au nombre de composantes connexes.



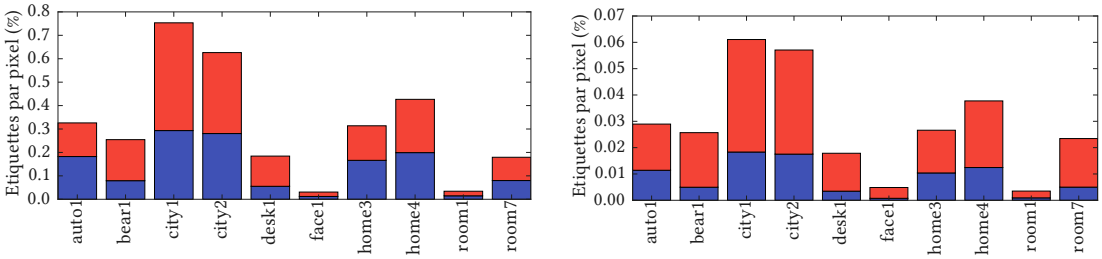
(a) auto1 (d = 53%) (b) bear1 (d = 65%) (c) city1 (d = 32%) (d) city2 (d = 41%) (e) desk1 (d = 56%)

(f) face1 (d = 92%) (g) home3 (d = 58%) (h) home4 (d = 74%) (i) room1 (d = 59%) (j) room7 (d = 63%)

Fig. 2.7 – Standard Image DataBAse

Pour faire le pont entre SIDBA et le jeu de données pseudo-aléatoire, il est intéressant de constater qu'une image réelle est la combinaison linéaire d'images de granularité et de densité différentes et comporte des zones très homogènes et d'autres très hétérogènes. Sur ce point, la différence entre *home3* et *room1* est frappante : bien que de densité très proche (58% contre 59%), le ratio entre leur nombre de composantes connexes est de 100 : 1. Avec l'algorithme de Rosenfeld, le temps de traitement de *home3* est ×1,38 plus long que celui de *room1*.

1. https://www.lri.fr/~cabaret/sidba.zip



(a) SIDBA (b) SIDBA4

Fig. 2.8 – Densité de composante connexe et d'étiquettes supplémentaires par pixel

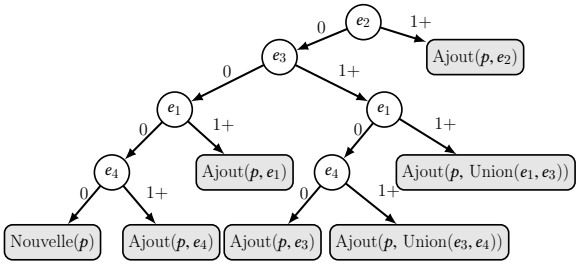
2.4 Améliorations algorithmiques

A la suite des pionniers, de nombreux algorithmes ont amélioré le domaine de l'étiquetage en composantes connexes. Nous allons présenter dans cette section les modifications algorithmiques notables.

2.4.1 Arbre de décision

2.4.1.1 Principe

L'arbre de décision est une procédure qui vise à réduire le nombre moyen de tests et d'accès à la mémoire par pixel lors du calcul du min^+. L'utilisation d'un arbre de décision permet en effet de ne procéder au chargement d'un pixel que s'il est nécessaire de tester sa valeur[59].



(a) Représentation en arbre, chaque cercle implique un chargement et un test

$e_1.e_2.e_3.e_4$	*Action*	N	*Ch*
x 1 x x	*Ajout*(p, e_2)	8	1
1 0 0 x	*Ajout*(p, e_1)	2	3
1 0 1 x	*Ajout*(p, *Union*(e_1,e_3))	2	3
0 0 1 1	*Ajout*(p, *Union*(e_3,e_4))	1	4
0 0 1 0	*Ajout*(p, e_3)	1	4
0 0 0 1	*Ajout*(p, e_4)	1	4
0 0 0 0	*Nouvelle*(p)	1	4

(b) Table de vérité de l'arbre de décision, avec N le nombre de cas et Ch le nombre de chargements

Fig. 2.9 – Arbre de décision, p est le pixel courant à étiqueter

La figure 2.9a représente l'arbre de décision sous forme graphique. Chaque cercle indique qu'il faut charger le pixel correspondant et tester sa valeur (étiquette nulle ou non). Selon le résultat de ce test, une action sur l'étiquette courante est prise ou une nouvelle étiquette est chargée. Selon la configuration du pixel et de son voisinage, plusieurs parcours sont possibles impliquant 3 longueurs de parcours différentes : 2, 3 ou 4 chargements (égal au nombre de tests). Si l'on étudie tous les cas en fonction du voisinage, on obtient le tableau 2.9b. Le nombre moyen de chargement par rapport au nombre de cas possibles dans un voisinage est donc de $\frac{1\times8+4\times3+4\times4}{16} = 2{,}25$.

2.4.1.2 Étude dans le contexte de notre jeu de données et portée réelle

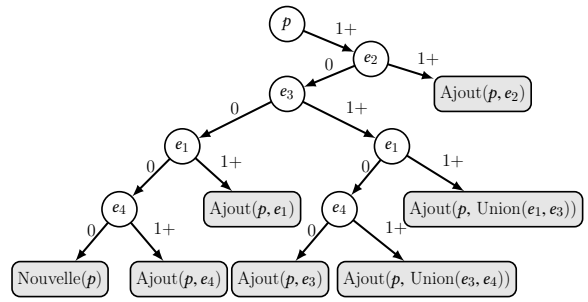
Dans la littérature, l'impact de l'arbre de décision est envisagé par rapport au nombre d'accès moyen théorique dans la phase de décision de l'étiquette à affecter au pixel courant. Ceci suppose de

considérer l'équiprobabilité des configurations. Or la configuration du voisinage d'un pixel est couplée à celle du pixel précédent et elle dépend de la densité de l'image. La valeur moyenne théorique est donc peu représentative. Notre jeu de données va nous permettre d'étudier le nombre d'accès moyen réel en fonction de la densité et de la granularité de l'image. Afin d'envisager l'impact de l'arbre de décision sur les performances, il est nécessaire de le compléter. En effet, l'arbre tel que décrit ci-dessus ne tient pas compte du chargement initial du pixel courant ni du test correspondant. La figure 2.10 présente la version incluant ce chargement (et ce test) ainsi que la table recensant les cas et le nombre de chargements correspondants (*Ch*).

La figure 2.11 représente l'évolution du nombre moyen de chargements en fonction de la densité de l'image (les mesures ont été répétées avec 10 graines différentes et l'écart-type maximal mesuré est de $\sigma = 1{,}5 \times 10^{-3}$).

Sans arbre de décision, ce nombre moyen suit une droite allant de 1 à 5 chargements quelle que soit la granularité. Cette forme est à rapprocher de celles de la figure 2.2. On constate qu'avec l'augmentation de la granularité et donc la diminution du nombre de composantes connexes, les courbes réelles tendent à s'approcher de la courbe de la figure 2.11a.

La présence de l'arbre de décision (figs. 2.11b, 2.11c et 2.11d) diminue bien le nombre moyen de chargements et de tests et ceci d'autant plus que la densité et la granularité augmentent. L'efficacité de l'arbre de décision dépend donc de la densité et de la granularité de l'image, dans le cas d'une granularité g=1, ce nombre moyen est m=1,91 et tend vers m=1,5 quand g augmente.

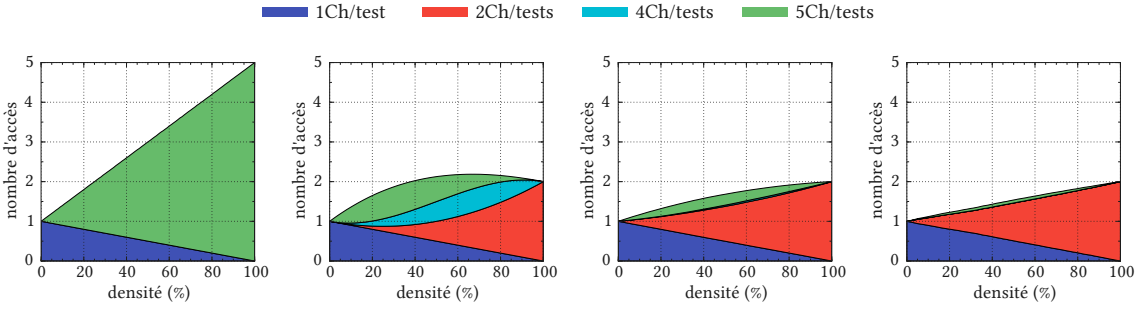


(a) Représentation en arbre, chaque cercle implique un chargement et un test

$p.e_1.e_2.e_3.e_4$	*Action*	N	*Ch*
0 x x x x	*Pixel suivant*	16	1
1 x 1 x x	*Ajout(p, e_2)*	8	2
1 1 0 0 x	*Ajout(p, e_1)*	2	4
1 1 0 1 x	*Ajout(p, Union(e_1,e_3))*	2	4
1 0 0 1 1	*Ajout(p, Union(e_3,e_4))*	1	5
1 0 0 1 0	*Ajout(p, e_3)*	1	5
1 0 0 0 1	*Ajout(p, e_4)*	1	5
1 0 0 0 0	*Nouvelle(p)*	1	5

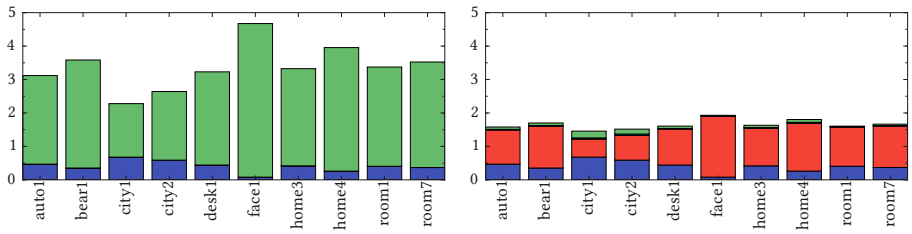
(b) Table de vérité de l'arbre de décision, avec *N* le nombre de cas et *Ch* le nombre de chargements

Fig. 2.10 – Arbre de décision avec prise en compte du pixel courant



(a) sans (b) g = 1, avec (c) g = 4, avec (d) g = 16, avec

Fig. 2.11 – Images aléatoires : Nombre moyen de chargements et tests par pixel avec le chargement initial sans et avec arbre de décision



(a) sans (b) avec

Fig. 2.12 – Base de données SIDBA : Nombre moyen de chargements et tests par pixel sans et avec arbre de décision

2.4.2 Gestion des équivalences : Suzuki

L'utilisation d'un arbre de décision permet de limiter le nombre de chargements et de tests pour déterminer quelle opération de gestion des équivalences doit être réalisée. Après cette étape, il est nécessaire de procéder aux opérations *Nouvelle*, *Ajout* et *Union* qui mettent en œuvre les mécanismes de gestion des équivalences. Dans la section 1.3.6, nous avons vu comment l'algorithme Union-Find réalisait ces opérations. Cependant lors du mécanisme d'union, la hauteur de l'arbre de chacune des deux composantes connexes peut conduire à un nombre indéterminé (mais fini) de chargements mémoires dispersés d'autant plus important que le nombre d'étiquettes temporaires est élevé par rapport aux nombre réel de composantes connexes.

Afin de ne plus être tributaire de ce comportement potentiellement coûteux, les auteurs de [58] ont proposé une nouvelle méthode de gestion des étiquettes dont l'objectif principal est de réaliser en permanence la fermeture transitive de la table d'équivalences.

Cet algorithme que nous nommerons algorithme de Suzuki dans la suite de ce document remplace la table T de l'algorithme Union-Find par un ensemble de trois tables :

- R : la table des racines qui contient en permanence la racine correspondant à chaque étiquette.
- N : la table des successeurs (`Next`) qui contient les listes chaînées (mais non ordonnées selon la valeur des étiquettes) reliant toutes les étiquettes d'une composante connexe.
- T : la table des fins de listes (`Tail`) qui indique l'étiquette finale de la liste qui correspond à chaque racine.

Algorithme 10 : Suzuki : recherche de la racine

Input : e une étiquette, R la table des racines
Result : r, la racine de e

```
1 r ← R[e]
2 return r
```

Algorithme 11 : Suzuki : mise à jour des tables

Input : u et v deux racines à unir, avec $u < v$

```
1 i ← v while i do
2 |   R[i] ← i
3 |   i ← N[i]
4 N[T[u]] ← v
5 T[u] ← T[v]
```

Lorsqu'un pixel isolé est rencontré, il y a création d'une nouvelle étiquette qui est utilisée comme nouvelle racine dans la table R et comme nouvelle fin dans la table T. La case correspondante de la table N est alors initialisée avec la valeur 0 qui indique qu'il n'y a pas d'autre étiquette dans la liste. Dans les autres cas, l'accès aux racines du voisinage est accéléré car la racine est obtenue par lecture directe de la table R (Algo. 10). Cependant, lorsqu'une *Union* doit être réalisée (Algo. 11), il faut mettre à jour l'intégralité des tables R, N et T pour faire pointer la composante ayant la plus grande racine vers la composante ayant la plus petite racine. Pour cela, il faut parcourir l'ensemble de la liste contenue dans N et, à chaque indice, faire correspondre la nouvelle valeur de la racine contenue dans R. Enfin, il faut remplacer l'étiquette de fin de la composante connexe de la nouvelle racine commune par celle de l'ancienne composante connexe de plus grande valeur. Cette procédure est plus coûteuse que celle de Union-Find, selon les cas, ce coût est compensé par le gain réalisé pour toutes les autres opérations, ce qui est d'autant plus vrai que le nombre d'étiquettes supplémentaires est faible.

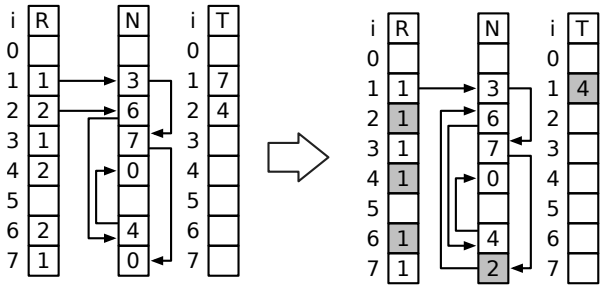


Fig. 2.13 – Principe de la gestion d'une *Union* entre la classe 1 et la classe 2 avec les tables de Suzuki : chaque étiquette contenue dans la classe 2 est mise à 1 (dans R), la liste des successeurs de la classe 2 est ajoutée à celle de la classe 1 et la nouvelle fin de la classe 1 est celle de la classe 2 (4)

2.4.3 Compression de chemin

La compression de chemin [94] est une modification de l'algorithme Union-Find qui minimise la hauteur de l'arbre au fur et à mesure de sa construction. Son utilisation répond au même souci que la gestion des équivalences de Suzuki : réduire le coût des opérations de remontée dans l'arbre. Dans l'article [95], l'auteur a prouvé que, munie de cette modification, la hauteur de l'arbre de connexité augmentait moins vite que l'inverse de la fonction d'Ackermann. Cette modification a cependant un coût, car la compression de chemin consiste à remonter à la racine d'un arbre puis à affecter à chaque étiquette rencontrée lors de cette remontée la valeur de la racine. Si une étiquette de la branche est de nouveau rencontrée dans la suite de l'étiquetage, le gain est immédiat mais sinon, ces opérations auront été superflues.

2.4.4 RCM

Les modifications précédentes avaient pour but d'agir sur le processus de décision ou la complexité de gestion des étiquettes. Une autre évolution majeure est la modification du voisinage pris en compte et donc de la forme du masque utilisé.

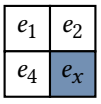


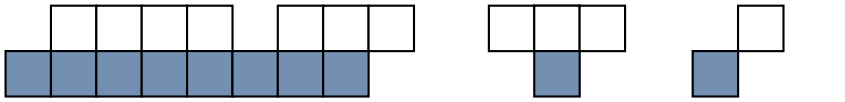
Fig. 2.14 – Masque spécifique de RCM

L'algorithme RCM pour Reduced Connectivity Mask [63] propose un masque réduit à quatre éléments (fig. 2.14) afin de limiter le nombre de chargements mémoire à effectuer. Le ratio étiquettes chargées / pixel étiqueté passe alors de 4 : 1 pour le masque de Rosenfeld à 3 : 1 pour RCM. Le nombre total de chargements (pixel + étiquettes) passe alors de 5 à 4. Les auteurs ont évalué les performances de ce masque avec les deux types de gestion des étiquettes Union-Find et Suzuki et en ont conclu que leur implémentation était plus rapide avec la gestion Suzuki. Dans l'article, les auteurs ne proposent pas d'utilisation d'arbre de décision ou l'emploi d'autres améliorations.

Cependant, cette baisse apparente du nombre de chargements cache en fait une augmentation du nombre moyen de chargements. L'étude proposée en annexe (section A.1.4) détaille ce phénomène et nous renforce dans notre démarche de test systématique des performances pour évaluer l'impact réel de telle ou telle évolution.

2.4.5 HCS : un algorithme à machine d'états

L'algorithme HCS décrit dans [61] est un algorithme pixel dont le comportement se rapproche de celui d'un algorithme segment. Dans la veine de l'utilisation d'un arbre de décision, le but de la proposition est de réduire le nombre de tests et de chargements en tenant compte du contexte du pixel. On distingue deux états que nous nommerons *fond* et *segment*.



(a) approche globale (b) Masque *fond* (c) Masque *segment*

Fig. 2.15 – Masque spécifique de HCS

Au début de la ligne l'état est *fond*, et on progresse pixel à pixel. Si le pixel est un pixel de fond, on passe au suivant, si c'est un pixel de premier plan, le voisinage supérieur (fig. 2.15b) est testé pour rechercher des *Ajouts* ou des *Unions* potentielles puis l'état passe à *segment*. Ensuite, tant que le pixel courant est de premier plan, c'est le masque de la figure 2.15c qui est utilisé car c'est la seule configuration qui puisse déclencher des *Unions*. Si le pixel courant est à 0, on repasse alors dans l'état *fond*.

Seul le premier pixel d'un segment nécessite 4 chargements (au maximum). Tous les autres pixels sont traités en 1 chargement (hors segment) ou 2 chargements (dans un segment). Du fait de ce comportement, HCS peut-être considéré comme un algorithme hybride pixel/segment.

2.4.6 HCS2

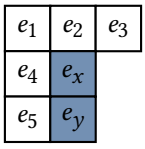


Fig. 2.16 – Masque spécifique de RCM

HCS2 est un algorithme «bloc» qui propose de traiter deux lignes à la fois. À l'aide d'un arbre de décision, ce masque est sensé traiter plus rapidement certaines configurations [62]. Le principal avantage est la diminution du ratio étiquettes chargées / pixel étiqueté qui passe alors de 4 : 1 à 5 : 2. Le nombre de chargements passe alors de 5 à 7 pour deux écritures. Un des inconvénients est la création d'un plus grand nombre d'étiquettes supplémentaires.

Dans l'article, les auteurs utilisent la gestion des étiquettes Suzuki. D'autres travaux ont modifié l'algorithme pour intégrer des méthodes de gestion alternatives.

2.4.7 ARemSP

Dans l'article [81], les auteurs ont étudié 35 variantes différentes de l'algorithme Union-Find et ont déterminé que la variation Rem combinée à une procédure nommée *Splicing* (ARemSP) était l'optimisation la plus efficace. L'idée est de ne pas remonter aux racines de chaque arbre mais de remonter dans les deux arbres alternativement selon que l'étiquette courante d'un arbre est supérieure ou non à l'étiquette courante de l'autre arbre. Si dans un des deux arbres on atteint la racine, il suffit de remonter à une étiquette immédiatement inférieure à cette racine dans le second arbre (fig. 2.17) et de réaliser l'union en ce point.

Algorithme 12 : ARemSP : Union(x, y)

```
   Input : x and y two labels to merge
1  while T[x] ≠ T[y] do
2      if T[x] > T[y] then
3          if x = T[x] then
4              T[x] = T[y]
5              return T[x]
6          z ← T[x], T[x] ← T[y], x ← z
7      else
8          if y = T[y] then
9              T[y] ← T[x]
10             return T[x]
11         z ← T[y], T[y] ← T[x], y ← z
12 return T[x]
```

Dans l'article [64], les auteurs ont montré qu'en remplaçant la gestion des équivalences *Suzuki* par la procédure ARemSP dans l'algorithme HCS2 original, ils obtenaient de meilleures performances.

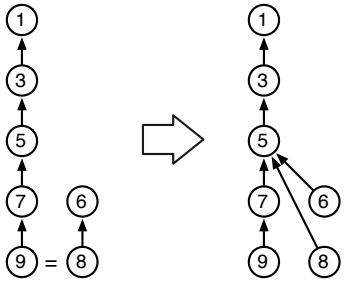


Fig. 2.17 – Union d'arbres avec ARemSP : lors de la rencontre de l'équivalence entre 8 et 9, au lieu de connecter 6 à 1, ARemSP se contente de connecter 6 et 8 à la première étiquette immédiatement inférieure à 6 (étiquette 5)

2.4.8 Grana

L'algorithme dit de Grana tel que décrit dans [60] s'appuie sur l'idée d'optimiser la procédure d'arbre de décision en agrandissant la taille du masque.

Chaque élément du masque de Rosenfeld se trouve remplacé par un macro-bloc de 2×2 pixels (fig. 2.18). Le point clef est qu'au lieu de considérer les pixels indépendamment, il commence par

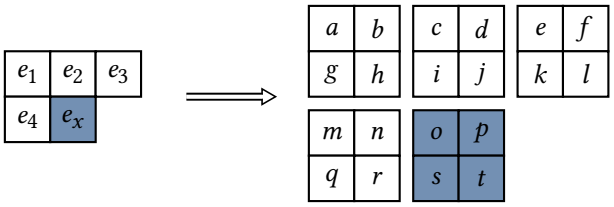


Fig. 2.18 – Construction du masque spécifique de Grana

étudier la connexité de blocs 2×2 pour réaliser un premier étiquetage partiel avec un nombre limité de chargements, suivi par une seconde passe qui se concentre sur la connectivité au sein des blocs. Il est important de noter pour la suite que la seconde passe n'est pas un simple réétiquetage.

2.4.9 LSL : Light Speed Labeling

2.4.9.1 Principe et notations

En 2000, Lionel Lacassagne a proposé la première implémentation de l'algorithme Segment LSL [53] dont l'objectif principal est d'adapter l'étiquetage aux spécificités des architectures RISC et donc de tenir compte des caches et de limiter les suspensions de pipeline dues aux tests.

Le mécanisme de LSL repose sur l'utilisation des segments à la manière de Ronse et d'une table locale proche de celle proposée par Lumia. Là où Ronse disposait d'un matériel spécifique pour extraire les segments, LSL génère lui-même les segments (codage RLC) à partir de l'image binaire (X). Au fur et à mesure de la génération des segments, ceux-ci sont étiquetés relativement à la ligne courante, puis leur connexité est évaluée relativement à la ligne supérieure.

C'est donc par l'ajout d'étapes locales que LSL propose une accélération globale de l'étiquetage.

Cet algorithme est considérablement plus complexe à expliciter que l'algorithme de Rosenfeld. Afin de bien appréhender son comportement et les modifications qui lui ont été apportées dans le cadre des travaux de thèse, il est nécessaire de présenter les étapes spécifiques et donc les notations correspondantes.

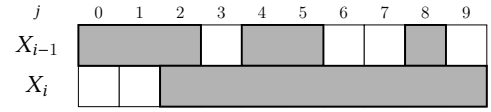
- er, une étiquette relative,
- ea, une étiquette absolue,
- a, la racine de l'arbre de ea,
- X_i la ligne courante de X, et X_{i-1} la ligne précédente de l'image $X[H][W]$
- EA, une image de taille $H \times W$ d'étiquettes absolues ea,
- ER_i, une table associative de taille W contenant les étiquettes relatives er pour la ligne courante X_i, ER_{i-1} l'équivalent pour la ligne précédente,
- ner, le nombre de segments de ER_i,
- RLC_i, la table contenant le codage RLC de la ligne courante X_i, RLC_{i-1} l'équivalent pour la ligne précédente,
- ERA_i, une table associative contenant les informations de correspondance entre er et ea : $ea = ERA_i[er]$, , ERA_{i-1} l'équivalent pour la ligne précédente,
- T, la table contenant les classes d'équivalences avant la fermeture transitive,
- RLC, une table 2D de taille $H \times 2W$ contenant tous les segments de l'image,
- LEA, une liste 2D d'étiquettes absolues de toutes les lignes, utilisée par la version LSL_{RLE},

Il existe plusieurs versions de LSL. Dans le cadre de ce manuscrit, nous nous limiterons à l'étude des deux principales :

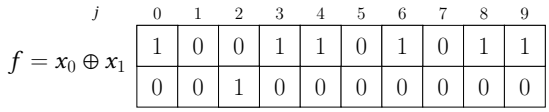
- LSL_{STD} : une version systématique conçue avec l'objectif d'être la plus indépendante possible des données (pas de structure `if-then-else`).
- LSL_{RLE} : Une version optimisée pour tirer parti des données les plus structurées en tirant profit du codage RLC.

2.4.9.2 Génération des segments et étiquetage relatif à la ligne

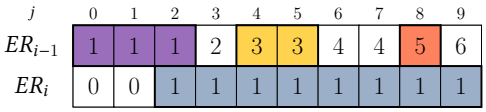
Les segments sont construits par une détection des transitions *fond* $\longrightarrow$ *segment*, *segment* $\longrightarrow$ *fond* que nous appellerons fronts. Considérons la figure 2.19a. Au début de la ligne X_i, l'étiquette relative est initialisée à 0 ($e_r = 0$), l'état est *fond* (f=0) et le registre x_1 contenant l'état du pixel précédent est mis à 0 ($x_1 \leftarrow 0$ - les bords extérieurs sont considérés comme du fond). Pour chaque pixel $x_0 = X_i[j]$ (*fond* comme *segment*), on affecte à $ER_i[j]$ la valeur de l'étiquette relative courante $ER_i[j] \leftarrow e_r$ et on recherche un front $f = x_0 \oplus x_1$. Si un front est détecté (f=1), e_r est incrémenté.



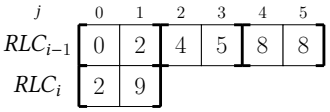
(a) X_i, la ligne courante de l'image binaire et X_{i-1} la ligne précédente



(b) f, vaut 1 à chaque transition entre le fond et les segments



(c) ER_i contient les étiquettes relatives de la ligne courante, ER_{i-1} celles de la ligne précédente



(d) RLC_i contient les coordonnées des segments de la ligne courante, RLC_{i-1} celles pour la ligne précédente

Fig. 2.19 – LSL : tables relatives et construction des segments

Au fur et à mesure de cette procédure, les coordonnées des segments sont enregistrées dans la table RLC_i. À chaque pixel pour LSL_{STD} (Algo. 13) et seulement lors des fronts pour LSL_{RLE} (test supplémentaire en ligne 6 algo. 14). LSL_{STD} est donc moins dépendant des données que LSL_{RLE}. Par construction, les fins de segments sont détectées avec un pixel de retard (fig. 2.19b). b joue donc le rôle de correcteur pour obtenir les bonnes coordonnées.

L'étiquetage relatif assigne une étiquette à chaque segment (*er* impaire) mais aussi à chaque zone de fond (*er* paire). Cette stratégie possède deux avantages : d'une part l'étiquetage relatif est très rapide car il s'agit d'une simple incrémentation à chaque front et il est réalisé dans la même passe que la création des segments (RLC_i), d'autre part la détermination de la nature d'un pixel donné est immédiate tout comme les opérations de recherche des segments connexes, suivants et précédents.

2.4.9.3 Construction des équivalences

Une fois ER_i et RLC_i obtenues, il est possible de construire les équivalences (Algo. 15). Pour cela, il faut aussi disposer de ER_{i-1} ainsi que de ERA_{i-1} et de la table d'équivalences T. À la fin de cette étape, T sera mise à jour.

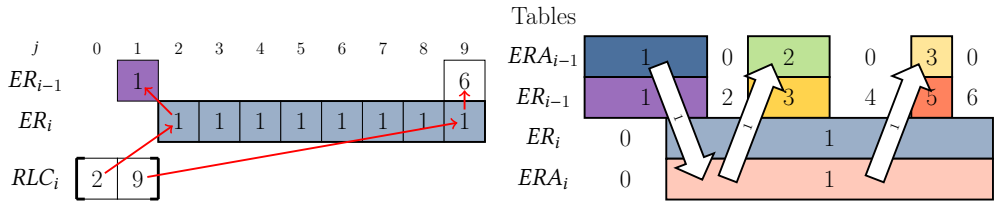
Pour chaque étiquette *er* impaire de la ligne courante (segments), on récupère les coordonnées (j_0, j_1) du segment correspondant par lecture de la table RLC_i ($j_0 \leftarrow RLC_i[er-1]$, $j_1 \leftarrow RLC_i[er]$). Pour déterminer la liste des étiquettes relatives de la ligne précédente qui lui sont connexes, il suffit de lire ER_{i-1} aux coordonnées correspondantes, soit $j_0 - 1$ et $j_1 + 1$ pour la 8C dans le cas général.

Si le début du segment (j_0) est aussi le début de la ligne, le segment connexe minimal est à rechercher en $er_0 \leftarrow ER_{i-1}(j_0)$. Si la fin du segment ($j_1$) est aussi la fin de la ligne, le segment connexe

Algorithme 13 : LSL : détection de segment pour la version STD

Input : X_i une ligne de largeur W
Result : ER_i, RLC_i et ner

```
1  x_1 ← 0 ; f ← 0
2  b ← 0 ; er ← 0
3  for j = 0 to W − 1 do
4      x_0 ← X_i[j]
5      f ← x_0 ⊕ x_1
6      RLC_i[e_r] ← j − b
7      b ← b ⊕ f
8      er ← er + f
9      ER_i[j] ← er
10     x_1 ← x_0
11 x_0 ← 0
12 f ← x_0 ⊕ x_1
13 RLC_i[er] ← w − b
14 er ← er + f
15 ner ← er
```



(a) Accès aux étiquettes connexes au segment courant (b) Union des étiquettes connexes au segment courant

Fig. 2.20 – LSL : construction des équivalences à partir de l'étiquetage relatif

maximal est à rechercher $er_1 \leftarrow ER_{i-1}(j_1)$. Enfin en 4C, les étiquettes des segments connexes sont à rechercher en $er_0 \leftarrow ER_{i-1}(j_0)$ et $er_1 \leftarrow ER_{i-1}(j_1)$. La différence entre 8C et 4C ne réside que dans cette étape pour les algorithmes LSL. Dans le cas où une de ces étiquettes serait paire (fond), il faudrait la remplacer par l'étiquette de segment immédiatement supérieure pour le bord gauche (j_0) et immédiatement inférieure pour le bord droit (j_1). Dans l'exemple (fig. 2.20a), $j_0 \leftarrow RLC_i[0] = 2$ et $j_1 \leftarrow RLC_i[1] = 9$, et donc $er_0 \leftarrow ERA_{i-1}[2-1] = 1$ et $er_1 \leftarrow ERA_{i-1}[9] = 6$ (9 au lieu de 10 car j_1 est au bord droit de la ligne). 6 étant une étiquette paire, er_1 est corrigée en $er_1 \leftarrow 5$.

L'étape suivante est d'étudier le voisinage ainsi déterminé. Selon les valeurs relatives des étiquettes er_0 et er_1, trois cas sont possibles :

- $er_1 < er_0$: il n'y a pas d'étiquette dans le voisinage ce qui entraîne création d'une nouvelle étiquette globale $ERA_i[er] \leftarrow ne{+}{+}$.
- $er_1 = er_0$: il n'y a qu'une étiquette dans le voisinage et l'étiquette globale correspondante est donc propagée au segment courant (global) $ERA_i[er] \leftarrow ERA_i[er_0]$.
- $er_1 > er_0$: il y a plusieurs étiquettes dans le voisinage et la procédure d'union des étiquettes est alors utilisée.

La procédure d'union des étiquettes consiste à affecter l'étiquette globale de la première étiquette du voisinage à l'étiquette courante globale $ERA_i[er] \leftarrow ERA_i[er_0]$, puis à comparer chaque étiquette globale des autres étiquettes du voisinage et à réaliser les opérations d'union de manière classique. Dans l'exemple (fig. 2.20b), l'étiquette 1 est propagée au segment courant et à tous les segments du

Algorithme 14 : LSL : détection de segment pour la version RLE

Input : X_i une ligne de largeur W
Result : ER_i, RLC_i et ner

```
1  x1 ← 0 ; f ← 0
2  b ← 0 ; er ← 0
3  for j = 0 to W − 1 do
4      x0 ← Xi[j]
5      f ← x0 ⊕ x1
6      if f ≠ 0 then
7          RLCi[er] ← j − b
8          b ← b ⊕ 1
9          er ← er + 1
10     ERi[j] ← er
11     x1 ← x0
12 x0 ← 0
13 f ← x0 ⊕ x1
14 RLCi[er] ← w − b
15 er ← er + f
16 ner ← er
```

voisinage (2 et 3).

Originellement [53, 54], LSL utilisait la gestion des équivalences dite de Selkow [96] [2]. Cette procédure régulièrement utilisée à l'ETCA consiste à remplacer la procédure Find du Union-Find par un simple accès à l'ancêtre de rang 2 d'une étiquette ($a = T[T[e]]$). [97] montre que la procédure n'est pas fiable pour les algorithmes *pixels* car l'arbre correspondant peut croître au-delà de cette hauteur dans certaines configurations de pixels. Durant nos travaux de thèse, nous avons mis en évidence que LSL était lui aussi vulnérable à cette faille aux travers de formes spécifiques. En effet, si au sein d'un segment, le comportement de Selkow est maîtrisé, ce n'est pas le cas pour une succession de segments. La section A.1.5 en annexe illustre ce phénomène.

La version actuelle de LSL est donc basée sur la procédure Union-Find classique. Dans le cadre de nos travaux, une version utilisant la procédure Suzuki a été utilisée afin d'illustrer le comportement de cette procédure dans un contexte parallèle.

2.4.9.4 Construction de l'image des étiquettes

C'est la dernière étape spécifique au LSL. L'image est parcourue entièrement et chaque étiquette $EA_i[j]$ est construite par le parcours de $ER_i[j]$ au travers de la table ERA_i (Algo.16)

2.4.9.5 Fermeture transitive et réétiquetage

Les deux dernières étapes sont identiques à l'algorithme de Rosenfeld. Après la fermeture transitive de la table d'équivalences, l'image est réétiquetée.

2. Merci a Mme Montanvert pour le travail d'archéologie documentaire

Algorithme 15 : LSL : construction des équivalences

Input : ER_{i-1}, RLC_i, T, ERA_{i-1}, ERA_i, ner
Result : nea le nombre actuel d'étiquettes absolues, T et $ERAi$ mises à jour

```
1  for er = 1 to ner step 2 do
2      j0 ← RLCi[er − 1]
3      j1 ← RLCi[er]
       ▷ 8C avec correction des bords
4      if j0 > 0 then j0 ← j0 − 1
5      if j1 < n − 1 then j1 ← j1 + 1
6      er0 ← ERi−1[j0]
7      er1 ← ERi−1[j1]
       ▷ Recherche des segments - étiquettes impaires
8      if er0 pair then er0 ← er0 + 1
9      if er1 pair then er1 ← er1 − 1
10     if er1 ≥ er0 then
11         ea ← ERAi−1[er0]
12         a ← T[ea]
           ▷ Recherche de la racine
13         while T[a] ≠ a do
14             a ← T[a]
           ▷ Recherche et propagation de l'étiquette minimale
15         for erk = er0 + 2 to er1 step 2 do
16             eak ← ERAi−1[erk]
17             ak ← T[eak]
               ▷ Recherche de la racine
18             while T[ak] ≠ ak do
19                 ak ← T[ak]
20             if a < ak then
21                 T[ak] ← a
22             if a > ak then
23                 T[a] ← ak
24                 a ← ak
25         ERAi[er] ← a ▷ Minimum global
26
27     else
           ▷ Création d'une nouvelle étiquette
28         nea ← nea + 1
29         ERAi[er] ← nea
```

Algorithme 16 : Construction de l'image des étiquettes

```
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          EAi[j] ← ERAi[ERi[j]]
```

2.5 Calcul des descripteurs

Selon la nature de l'algorithme (pixel ou segment), le mode de calcul des descripteurs diffère (cf. sec. 1.6). Mais il peut aussi différer selon le moment où il est calculé.

- *A posteriori* : dans un contexte classique où l'analyse en composantes connexes succède à l'étiquetage en composantes connexes, le calcul des descripteurs ne s'appuie que sur l'image finalisée des étiquettes. Il est alors nécessaire de réaliser une passe supplémentaire (une troisième passe) en perdant toute la synergie entre l'affectation des étiquettes et le calcul des descripteurs.
- *A la volée* : une autre solution est d'intégrer au cœur de l'algorithme, les tables de descripteurs et de mettre à jour celles-ci pour chaque opération *Nouvelle*, *Ajout* ou *Union*. C'est la solution retenue pour les algorithmes LSL dès leur création (Algo. 18).
- *Entrelacée* ou *par ligne* : enfin il est aussi possible de ne pas modifier le cœur de l'algorithme, mais de venir calculer les descripteurs entre chaque ligne traitée. C'est la solution retenue pour les algorithmes pixels (Algo. 17).

Algorithme 17 : Calcul par ligne des descripteurs pour les algorithmes pixels

Input : E L'image des étiquettes, *Desc* la structure de données contenant les descripteurs, i le numéro de la ligne courante, W la largeur de la ligne
Result : *Desc* mise à jour

```
1  for j = 0 to W − 1 do
2      e ← E[i][j]
3      if e ≠ 0 then
4          if j < Desc[e].X_min then Desc[e].X_min ← j
5          if j > Desc[e].X_max then Desc[e].X_max ← j
6          if i < Desc[e].Y_min then Desc[e].Y_min ← i
7          if i > Desc[e].Y_max then Desc[e].Y_max ← i
8          Desc[e].S ← Desc[e].S + 1
9          Desc[e].S_x ← Desc[e].S_x + j
10         Desc[e].S_y ← Desc[e].S_y + i
```

Dans le cas du calcul à la volée ou du calcul entrelacé, les équivalences entre étiquettes évoluent au fur et à mesure de la construction de l'image. Il est donc nécessaire de réaliser l'union des descripteurs en même temps que l'union des étiquettes (Algo. 19).

Algorithme 18 : Calcul pour chaque segment des descripteurs

Input : e l'étiquette du segment courant, *Desc* la structure de donnée contenant les descripteurs, i le numéro de la ligne courante, l_{min} le début du segment, l_{max} la fin du segment
Result : *Desc* mise à jour

if *e est une nouvelle étiquette* **then**
  $Desc[e].xmin \leftarrow l_{min}$; $Desc[e].xmax \leftarrow l_{max}$
  $Desc[e].ymin \leftarrow i$; $Desc[e].ymax \leftarrow i$
  $S \leftarrow l_{max} - l_{min} + 1$
  $Sx \leftarrow (l_{max} * (l_{max} + 1) - l_{min} * (l_{min} - 1))/2$
  $Sy \leftarrow i * S$
  $Desc[e].S \leftarrow S$
  $Desc[e].Sx \leftarrow Sx$
  $Desc[e].Sy \leftarrow Sy$
else
  $Desc[e].ymax \leftarrow i$
  if $l_{min} < Desc[e].xmin$ **then** $Desc[e].xmin \leftarrow l_{min}$
  if $l_{max} > Desc[e].xmax$ **then** $Desc[e].xmax \leftarrow l_{max}$
  $S \leftarrow l_{max} - l_{min} + 1$
  $S_x \leftarrow (l_{max} \times (l_{max} + 1) - l_{min} \times (l_{min} - 1))/2$
  $S_y \leftarrow i \times S$
  $Desc[e].S \leftarrow Desc[e].S + S$
  $Desc[e].S_x \leftarrow Desc[e].S_x + Sx$
  $Desc[e].S_y \leftarrow Desc[e].S_y + Sy$

Algorithme 19 : Union des descripteurs lors de l'union des racines

Input : r_1 et r_2, deux racines à unir, *Desc* la structure de données contenant les descripteurs
Result : *Desc* mise à jour

if $Desc[r_2].xmin < Desc[r_1].xmin$ **then** $Desc[r_1].xmin \leftarrow Desc[r_2].xmin$
if $Desc[r_2].xmax > Desc[r_1].xmax$ **then** $Desc[r_1].xmax \leftarrow Desc[r_2].xmax$
if $Desc[r_2].ymin < Desc[r_1].ymin$ **then** $Desc[r_1].ymin \leftarrow Desc[r_2].ymin$
if $Desc[r_2].ymax > Desc[r_1].ymax$ **then** $Desc[r_1].ymax \leftarrow Desc[r_2].ymax$
$Desc[r_1].S \leftarrow Desc[r_1].S + Desc[r_2].S$
$Desc[r_1].Sx \leftarrow Desc[r_1].Sx + Desc[r_2].Sy$
$Desc[r_1].Sx \leftarrow Desc[r_1].Sx + Desc[r_2].Sy$

2.6 Conclusion

Nous venons de mettre en évidence la variété du paysage de l'étiquetage en composantes connexes au travers des algorithmes et de leurs variations. Déterminer la performance d'un algorithme sans se référer à l'architecture sur laquelle il est implémenté peut conduire à des contresens car certaines optimisations algorithmiques peuvent être néfastes du point de vue implémentation. Dans la suite de nos travaux, nous allons sélectionner une série d'algorithmes représentatifs qui servira de socle pour l'analyse et pour la construction des versions parallèles.

Chapitre 3

La maturité de l'homme est d'avoir retrouvé le sérieux qu'on avait au jeu quand on était enfant.

–La Horde du contrevent, Alain Damasio

Performance des algorithmes séquentiels d'étiquetage et d'analyse en composantes connexes

3.1 Introduction

Le chapitre précédent a permis d'établir que le paysage de l'étiquetage en composantes connexes était composé d'un grand nombre d'algorithmes différents et qu'il existait des briques de base qui apportaient des améliorations potentielles mais dont le bien-fondé dépendait finalement des données. Pour comprendre et évaluer leur impact, il est nécessaire de soumettre ces algorithmes à l'épreuve de tests réels. Dans ce chapitre, nous allons tout d'abord faire une analyse comparative des résultats bruts des algorithmes selon le protocole de test décrit dans la section 2.2 puis, afin de comprendre plus finement les écarts observés, nous analyserons les performances intermédiaires des différentes parties des algorithmes que sont : la première passe, la fermeture de la table d'équivalences, la seconde passe. Dans un second temps, nous nous pencherons sur les performances des algorithmes d'analyse en composantes connexes dont l'importance grandit à mesure que les applications d'étiquetage en composantes connexes évoluent vers des systèmes de plus en plus intégrés où l'intervention humaine est inexistante ou réservée aux analyses de haut niveau.

Toutes les mesures ont été effectuées sur un cœur de i7-6700K (Skylake 4Ghz) avec 16 Go de DDR4 3466Mhz (PC4-27700) sur une Debian 8 en utilisant le compilateur Intel C Compiler 16.1 avec

le speedstep (changement dynamique de la fréquence d'horloge en fonction de la charge et de l'enveloppe thermique) désactivé et sans hyperthreading (création de deux processeurs logiques par cœur). Comme indiqué dans le paragraphe 2.2.3, les résultats seront exprimés en *cpp*. La moyenne des *cpp* sur les densités allant de 0% à 100% pour une granularité donnée sera nommée cpp_d et la moyenne des cpp_d sur les granularités allant de 1 à 16 sera nommée cpp_g.

3.2 Constitution d'un ensemble d'algorithmes de référence

La liste des algorithmes décrits au chapitre précédent est par nature incomplète. Il est d'une part possible de créer de nouveaux algorithmes en associant plusieurs mécanismes appartenant à tel ou tel algorithme. Et d'autre part, le domaine étant très actif, de nombreux algorithmes sont proposés chaque année et aucune liste aussi complète soit-elle ne peut le rester longtemps. Les algorithmes ont été choisis pour leur représentativité au sein de la communauté, ce qui permet à tout nouvel algorithme d'être positionné par rapport à au moins l'un d'entre eux.

Afin de couvrir tout de même au maximum de nos possibilités la diversité des algorithmes intermédiaires, nous avons généré de nombreuses variantes à partir des algorithmes originaux. Dans cette section, nous avons comparé ces variantes entre elles afin de sélectionner un représentant dans chaque famille. L'ensemble ainsi construit servira de base de comparaison dans les sections et chapitres suivants.

3.2.1 Variantes de la famille Rosenfeld

3.2.1.1 Présentation des variantes

L'algorithme original décrit dans [73] est un point de départ très légitime. En effet, parmi tous les algorithmes pionniers, c'est de cette famille (2 passes avec table d'équivalences) dont dérivent les algorithmes directs étudiés dans le chapitre 2 qui sont les plus adaptés aux architectures scalaires.

A partir de cet algorithme de base, nous avons construit les variantes suivantes :

- Rosenfeld classique,
- Rosenfeld + arbre de décision (DT),
- Rosenfeld + compression de chemin (PC),
- Rosenfeld + arbre de décision + compression de chemin (DT + PC),
- Rosenfeld + arbre de décision + ARemSP Union-Find (DT + ARemSP).

3.2.1.2 Résultats pour les images aléatoires

	granularité				
algorithmes	g=1	g=2	g=4	g=8	g=16
Rosenfeld DT PC	20,18	11,87	8,30	6,64	6,00
Rosenfeld DT ARemSP	19,81	12,03	8,56	6,98	6,19
Rosenfeld DT	20,19	12,21	8,72	7,08	6,26
Rosenfeld	31,06	19,70	13,16	10,06	8,69
Rosenfeld PC	35,05	22,88	14,70	10,73	8,87

TABLE 3.1 – Comparatif des variantes de la famille Rosenfeld pour les images aléatoires : exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1, 2, 4, 8, 16\}$ sur un cœur Skylake

Les variantes se classent en deux catégories (fig. 3.1) : celles qui utilisent un arbre de décision et les autres. Tous les algorithmes utilisant un arbre de décision sont très proches avec toutefois un avantage

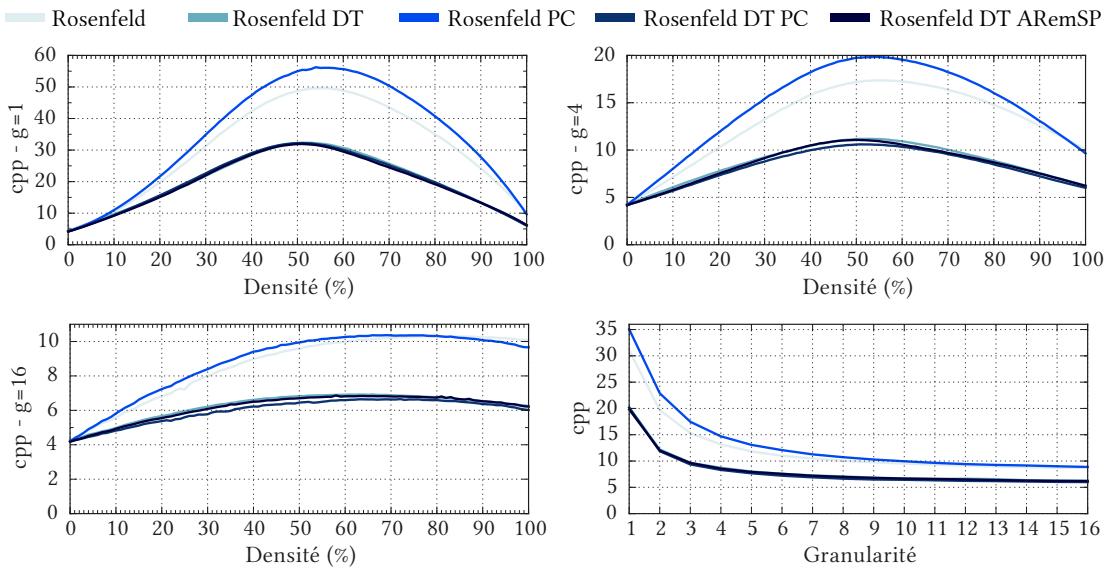


Fig. 3.1 – Variantes de la famille Rosenfeld : exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1, 4, 16\}$ et cpp_d en fonction de la granularité sur un cœur Skylake

constant pour les versions DT + PC et DT + ARemSP. La compression de chemin a un effet négatif sur les performances de la version classique et positif sur la version DT. Dans la version classique, l'opération de compression de chemin est répétée systématiquement pour tous les pixels du voisinage y compris ceux déjà compressés dans le voisinage courant et dans celui du pixel précédent. Cela induit des écritures inutiles dans la table d'équivalences qui n'existaient pas dans la version sans PC. Dans la version DT, l'opération de compression ne s'applique que pour les étiquettes chargées par l'arbre de décision dans les cas susceptibles de déclencher une procédure d'union.

Bien que meilleure du point de vue algorithmique, la compression de chemin n'apporte donc pas un gain systématique, la hauteur du graphe n'étant pas le seul élément coûteux de l'algorithme. Cela renforce notre conviction de la nécessité de soumettre toutes les améliorations à des mesures de performance pour évaluer leur pertinence.

3.2.1.3 Résultats pour les images de SIDBA

	t (ms)			*cpp*		
Algorithmes	min	moy	max	min	moy	max
Rosenfeld DT	0,54	0,71	0,87	4,53	5,96	7,28
Rosenfeld DT ARemSP	0,55	0,72	0,88	4,56	6,00	7,35
Rosenfeld DT PC	0,62	0,79	0,96	5,19	6,59	7,99
Rosenfeld	0,82	1,12	1,38	6,85	9,31	11,52
Rosenfeld PC	0,85	1,20	1,53	7,05	10,04	12,78

Table 3.2 – Comparatif des variantes de la famille Rosenfeld pour la base de données SIDBA : exprimées en *ms* et *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

Pour la base de données SIDBA, Rosenfeld DT est le plus rapide suivi par Rosenfeld DT ARemSP et Rosenfeld DT PC. Comme pour les images aléatoires, Rosenfeld classique et Rosenfeld PC sont bien plus lents (×0,63). L'utilisation de PC a un impact négatif sur les performances dans tous les cas.

3.2.1.4 Conclusion

L'inversion du rang de Rosenfeld DT et Rosenfeld DT PC entre les images aléatoires et SIDBA met en évidence l'intérêt de disposer des deux référentiels.

Afin de représenter la famille Rosenfeld, nous avons décidé d'utiliser Rosenfeld + DT + PC car il appartient au groupe le plus rapide et il permettra de mettre en évidence l'intérêt ou non de la compression de chemin dans les chapitres suivants lors de la parallélisation de ces algorithmes.

3.2.2 Variantes de la famille HCS_2

3.2.2.1 Présentation des variantes

La famille HCS_2 est composée de l'algorithme original et de la version ARemsP [64]. Afin de compléter les variantes autour du masque HCS_2 nous avons modifié l'algorithme original pour utiliser Union-Find avec un arbre de décision sans la modification ARemSP. Cet ajout nous permet d'évaluer l'impact de la modification ARemSP [64].

La comparaison pour cette famille a donc été faite pour :

- la version originale HCS_2, basée sur la gestion des équivalences de Suzuki et sur l'utilisation d'une table de vérité pour déterminer les équivalences,
- HCS_2 + Union-Find (UF) + arbre de décision (DT),
- HCS_2 + Union-Find (UF) + arbre de décision (DT) + ARemSP [64].

3.2.2.2 Résultats pour les images aléatoires

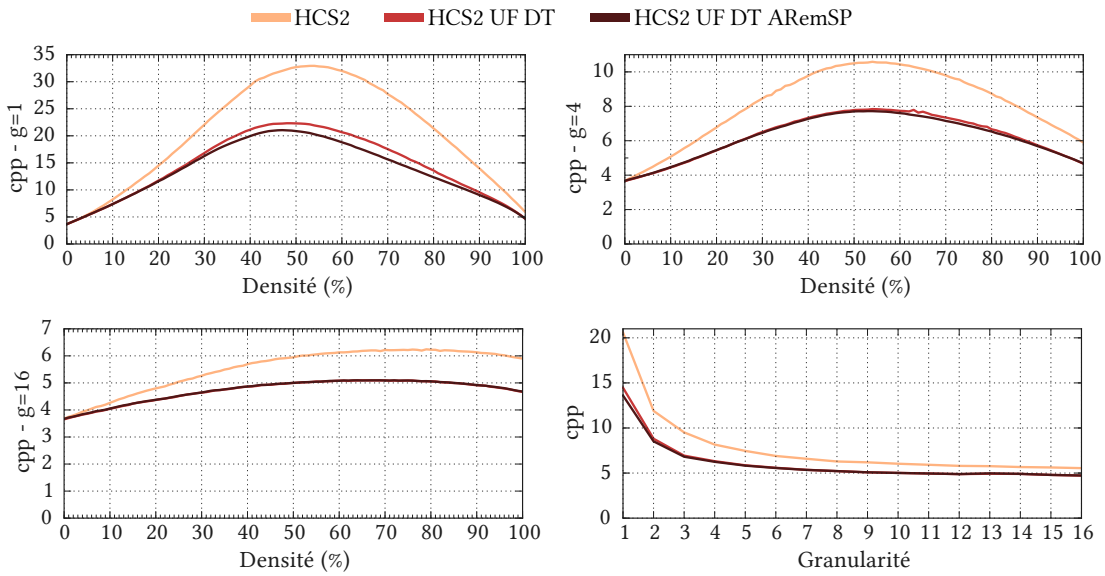


Fig. 3.2 – Variantes de la famille HCS_2 : performances exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1,4,16\}$ et cpp_den fonction de la granularité sur un cœur Skylake

Tout comme pour les variantes de la famille Rosenfeld, la modification la plus significative est l'arbre de décision (fig. 3.2). Si la version HCS_2 UF DT ARemSP a bien l'avantage sur la version HCS_2 UF DT pour les images peu structurées, celui-ci disparaît avec l'augmentation de la granularité.

	granularité				
Algorithmes	g=1	g=2	g=4	g=8	g=16
HCS_2 UF DT ARemSP	13,60	8,52	6,25	5,22	4,73
HCS_2 UF DT	14,47	8,81	6,32	5,22	4,72
HCS_2	20,60	11,89	8,16	6,29	5,55

TABLE 3.3 – Variantes de la famille HCS_2 : exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1, 2, 4, 8, 16\}$ sur un cœur Skylake

3.2.2.3 Résultats pour les images de SIDBA

Les résultats pour la base de données SIDBA confirment ceux des images aléatoires. HCS_2 UF DT ARemSP et HCS_2 UF DT ont des performances très proches. HCS_2 UF DT ARemSP est plus rapide que la version originale de HCS_2 d'un rapport $\times$1,33. La variabilité des résultats ($cpp_{max} - cpp_{min}$) est à l'avantage de la version HCS_2 UF DT ARemSP.

	t (ms)			*cpp*		
Algorithmes	min	moy	max	min	moy	max
HCS_2 UF DT ARemSP	0,53	0,63	0,72	4,40	5,24	6,01
HCS_2 UF DT	0,53	0,63	0,73	4,38	5,28	6,12
HCS_2	0,64	0,84	1,02	5,30	6,97	8,47

TABLE 3.4 – Comparatif des variantes de la famille HCS_2 pour la base de données SIDBA : exprimées en *ms* et *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

3.2.2.4 Conclusion

HCS_2 UF DT ARemSP et HCS_2 UF DT sont très similaires. La variation ARemSP diminue la variabilité des résultats selon les images et la version originale de HCS_2 est dans tous les cas la plus lente.

Afin de représenter la famille HCS_2, nous avons décidé d'utiliser HCS_2 UF DT ARemSP car elle permettra de mettre en évidence l'intérêt ou non de ARemSP dans les chapitres suivants lors de la parallélisation des algorithmes.

3.2.3 Variantes de la famille Suzuki

La gestion des équivalences de Suzuki a pris une place très importante dans la littérature. Du fait de l'impact de l'arbre de décision sur les algorithmes précédents, nous évaluerons :

- la version originale de Suzuki qui utilise le masque de Rosenfeld et la gestion des équivalences avec trois tables de Suzuki,
- la version précédente + arbre de décision (DT),

3.2.3.1 Résultats pour les images aléatoires

Une fois encore, l'arbre de décision améliore significativement les résultats (tab. 3.5 et fig. 3.3) et tend à symétriser les courbes autour de la valeur maximale. En considérant cpp_d, Suzuki DT est plus rapide que la version classique d'un rapport allant de $\times$1,5 à $\times$2 selon la granularité.

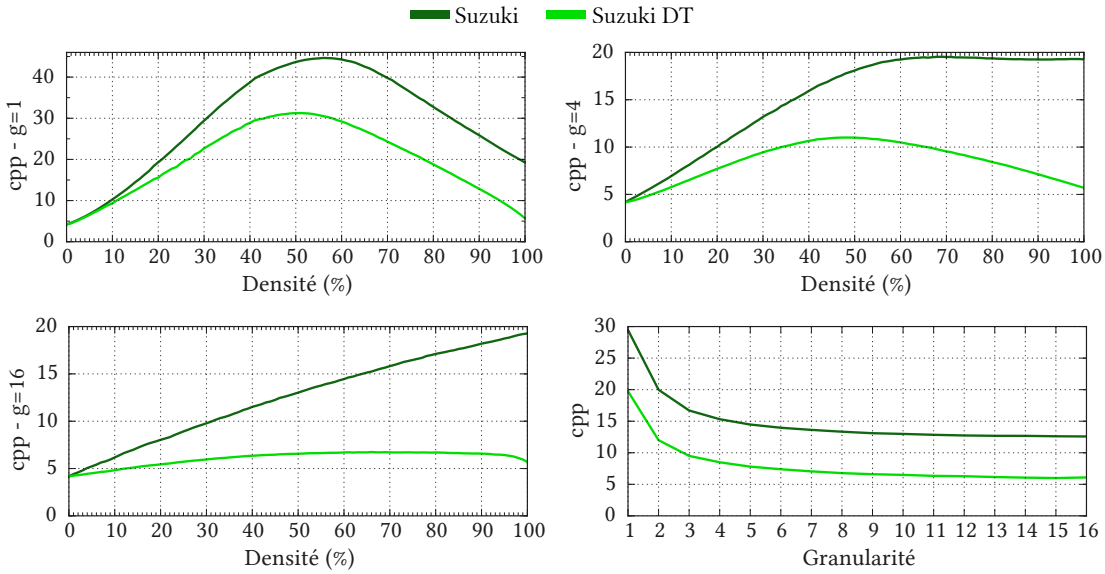


Fig. 3.3 – Variantes de la famille Suzuki : exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1, 4, 16\}$ et cpp_d en fonction de la granularité sur un cœur Skylake

	granularité				
Algorithmes	g=1	g=2	g=4	g=8	g=16
Suzuki DT	19,75	12,00	8,49	6,77	6,09
Suzuki	29,46	19,98	15,32	13,34	12,58

TABLE 3.5 – Variantes de la famille Suzuki : exprimées en *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1, 2, 4, 8, 16\}$ sur un cœur Skylake

3.2.3.2 Résultats pour les images de SIDBA

L'étude sur la base SIDBA renforce encore l'avantage de Suzuki DT qui devient en moyenne plus rapide d'un rapport ×2,5. L'intérêt de l'arbre de décision est encore plus flagrant que pour les familles HCS_2 et Rosenfeld.

	t (ms)			*cpp*		
Algorithmes	min	moy	max	min	moy	max
Suzuki	1,43	1,70	2,04	11,92	14,16	16,96
Suzuki DT	0,51	0,68	0,84	4,25	5,63	6,96

TABLE 3.6 – Comparatif des variantes de la famille Suzuki pour la base de données SIDBA : exprimées en *ms* et *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

3.2.3.3 Conclusion

L'arbre de décision s'utilise aussi bien avec le mécanismes Union-Find classique qu'avec les tables de Suzuki. Encore une fois, il vient diminuer le nombre d'accès inutiles à l'image des étiquettes ainsi qu'aux trois tables qui gèrent les équivalences. Dans la suite du manuscrit, la famille Suzuki sera représentée par Suzuki DT .

3.2.4 Algorithmes de références pour la suite des expérimentations

Suite à cette étude préalable, nous proposons le jeu d'algorithmes suivant que nous nommerons à partir de ce point algorithmes directs de référence.

- Rosenfeld : le masque de Rosenfeld [73] (5 pixels) avec la gestion des équivalences Union-Find (UF) améliorée avec l'arbre de décision (DT) et la compression de chemin (PC).
- Suzuki : le masque de Rosenfeld avec la gestion des équivalences Suzuki [58] améliorée avec l'arbre de décision (DT),
- Grana : le masque bloc de Grana (20 pixels) avec un arbre de décision à 128 étages [60] utilisant la gestion des équivalences de Suzuki,
- RCM : le masque réduit de RCM (4 pixels) [63] avec la gestion des équivalences Suzuki,
- HCS_2 UF DT ARemSP : le masque HCS_2 avec la gestion des équivalences Union-Find (UF) améliorée avec l'arbre de décision (DT) et l'optimisation Rem+Splicing (ARemSP) [64],
- HCS : algorithme à machine d'états hybride pixel/segment [61] à masque variable utilisant la gestion des équivalences Suzuki,
- LSL : algorithme segment avec la gestion des équivalences Union-Find (UF) (une variante utilisant la gestion des équivalences Suzuki sera utilisée à partir du chapitre 4 et 5). Deux variantes sont utilisées : LSL_{STD} conçue avec l'objectif d'être le plus systématique possible et LSL_{RLE} conçue pour tirer parti des segments les plus longs par une utilisation intensive du codage RLC.

3.3 Confrontation des algorithmes de référence au jeu de données

3.3.1 Comportement vis-à-vis de la densité

La figure 3.4 met en évidence le comportement des courbes représentant les algorithmes du jeu de données. Pour g=1, elles sont symétriques par rapport à leur valeur maximale. L'abscisse des maximum étant comprise entre [45%; 55%] selon l'algorithme.

L'augmentation du *cpp* pour les densités autour de 50% est due à l'action conjuguée de plusieurs phénomènes. D'une part, comme vu dans la section 1.5.3, les concavités et les marches d'escalier (fig. 1.26) entraînent la création d'étiquettes supplémentaires et par conséquent plus d'opérations d'union d'étiquettes. D'autre part, ces mêmes structures sont responsables de l'augmentation du nombre de tests dans l'arbre de décision car la création d'une étiquette est la conséquence d'un parcours de taille maximale de l'arbre (sec. 2.4.1).

3.3.2 Comportement vis-à-vis de la granularité

La table 3.7 et la figure 3.4 décrivent le comportement des algorithmes en fonction de la granularité des images. La tendance principale est que lorsque g croît, le *cpp* décroît. Rapidement pour $g \in \{1,2\}$, puis lentement pour $g \in [2:16]$.

On peut remarquer que LSL_{RLE} est le plus accéléré quand la granularité croît tandis que LSL_{STD} est le plus constant des algorithmes. C'est bien dans cet esprit qu'ils ont été conçus (cf. 2.4.9).

Au-dessus de g=4, LSL_{RLE} est le plus rapide de tous les algorithmes et LSL_{STD} est le plus régulier en *cpp*. Un point notable est que tous les algorithmes à l'exception de LSL_{RLE} tendent à se stabiliser en fonction de la granularité de l'image ($\times$1.1 entre g=8 et g=16) alors que LSL_{RLE} continue d'accélérer ($\times$1.3 entre g=8 et g=16).

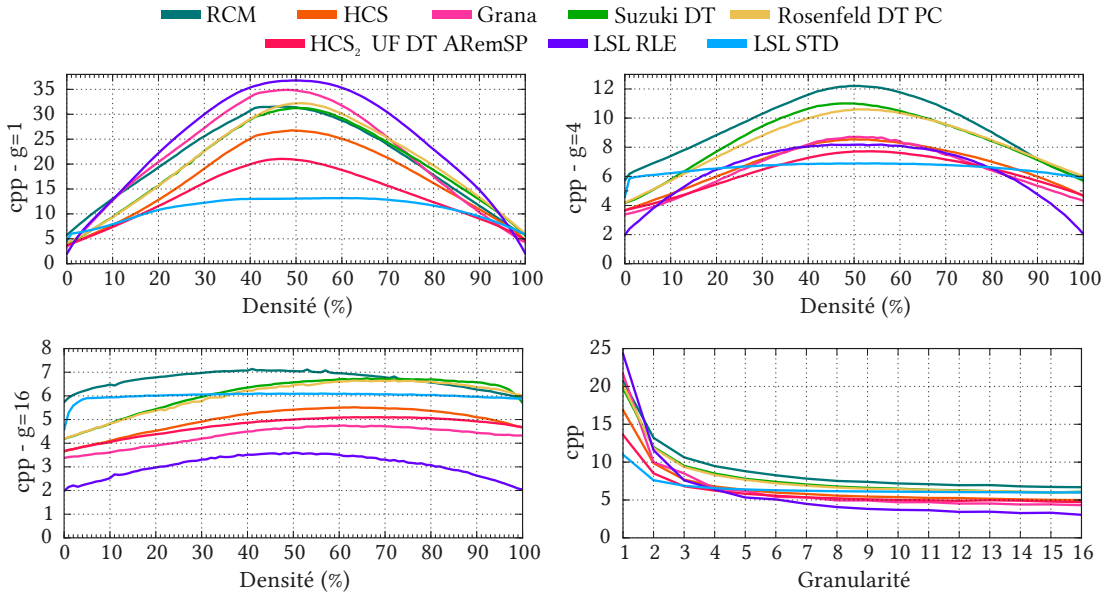


Fig. 3.4 – Algorithmes directs de référence : *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1,4,16\}$ et *cpp* moyen en fonction de la granularité sur un cœur Skylake

	granularité				
Algorithmes	g=1	g=2	g=4	g=8	g=16
LSL_{RLE}	24,31	11,58	6,40	4,07	3,05
Grana	21,66	9,99	6,53	4,94	4,32
HCS_2 UF DT ARemSP	13,60	8,52	6,25	5,22	4,73
HCS	16,88	9,86	6,78	5,57	4,98
LSL_{STD}	10,99	7,61	6,55	6,16	6,00
Rosenfeld DT PC	20,18	11,87	8,30	6,64	6,00
Suzuki DT	19,75	12,00	8,49	6,77	6,09
RCM	20,73	13,21	9,48	7,52	6,69

TABLE 3.7 – Algorithmes directs de référence : *cpp* moyen pour des images de taille 1024×1024 en fonction de la granularité sur un cœur Skylake

3.3.3 Résultats par rapport aux images de SIDBA

Les mesures sur la base de données d'images confirment les conclusions générales issues des images aléatoires mais introduisent des variations notables.

La table 3.8 donne les résultats en *ms* et en *cpp* (minimum, moyen et maximum). En moyenne, LSL_{RLE} est le plus rapide suivi par HCS_2 UF DT ARemSP, Suzuki DT, HCS, Grana, LSL_{STD} Rosenfeld DT PC et RCM. L' algorithme Suzuki DT est plus performant sur la base d'images que sur les images aléatoires.

Du point de vue de la stabilité, LSL_{STD} (fig. 3.5 et tab. 3.8) est le plus performant. Sa variation est de 1.00 *cpp* tandis que le second le plus stable (HCS_2 UF DT ARemSP) varie de 1,54 *cpp*.

	t (ms)			*cpp*		
Algorithmes	min	moy	max	min	moy	max
LSL_{RLE}	0,28	0,54	0,82	2,36	4,49	6,86
HCS_2 UF DT ARemSP	0,50	0,60	0,69	4,19	5,03	5,73
HCS	0,48	0,62	0,76	4,02	5,16	6,31
Suzuki DT	0,48	0,65	0,79	4,03	5,39	6,61
Grana	0,47	0,66	0,85	3,92	5,52	7,09
LSL_{STD}	0,66	0,73	0,78	5,52	6,05	6,52
Rosenfeld DT PC	0,59	0,77	0,94	4,94	6,40	7,84
RCM	0,63	0,77	0,93	5,26	6,46	7,77

TABLE 3.8 – Algorithmes directs de référence : résultats pour SIDBA exprimés en *ms* et *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

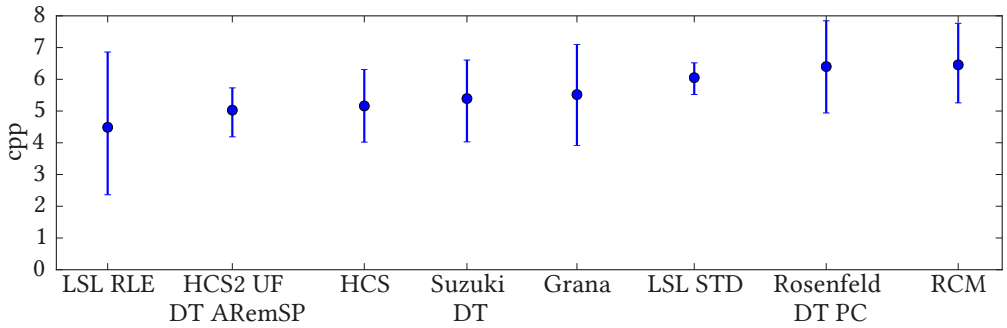


Fig. 3.5 – Algorithmes directs de référence : *cpp* moyen et variabilité (cpp_{max} et cpp_{min}) sur la base de données SIDBA sur un cœur Skylake

3.3.4 Conclusion sur le comportement général des algorithmes de référence vis-à-vis du jeu de données

Les objectifs fixés au jeu de données du point de vue de sa capacité à soumettre les algorithmes à des cas de figures simples comme complexes sont atteints.

- Les images aléatoires de granularité variable nous renseignent sur l'évolution des algorithmes en fonction de la densité et de la granularité et mettent en évidence le comportement des algorithmes dans des cas très complexes (g=1) comme dans des cas plus structurés (g=16).
- Les images de la base SIDBA complètent cette information en illustrant le comportement des algorithmes pour des images dont la granularité interne est variable et nous permettent de classer les algorithmes dans des conditions réalistes bien qu'elles ne puissent rendre compte de tous les cas de figures. Dans le cas d'une application spécialisée, une base de données métier sera utilisée pour valider ce classement.

On peut constater que les images de SIDBA sont globalement plus rapides à traiter que les images aléatoires (en *cpp*). Mais la granularité équivalente (cf. 2.2.6) des images de la base SIDBA est variable selon l'algorithme. Pour trouver un *cpp* identique entre les données aléatoires et SIDBA, il faut atteindre $g \approx 6$ pour LSL_{RLE} et Grana, $g \approx 9$ pour HCS_2 UF DT ARemSP , $g \approx 12$ pour HCS, LSL_{STD}, Rosenfeld DT PC et $g \geq 16$ pour Suzuki DT et RCM. Cette information ne renseigne pas sur la qualité des algorithmes, mais sur la nécessité d'utiliser les deux référentiels pour évaluer les performances d'un algorithme donné.

Ce que les différentes mesures confirment est :

- que dans le cas d'applications réelles (SIDBA ou $g > 8$), LSL_{RLE} est le plus rapide d'un facteur compris entre ×1,12 par rapport au second et ×1,43 par rapport au plus lent.
- qu'il est aussi très dépendant de la structure des images du fait de l'utilisation intensive du codage RLC,
- que LSL_{STD} est le plus stable de tous les algorithmes, ce qui peut-être utile lorsqu'il s'agit de limiter l'aléa sur le temps de traitement pour des applications embarquées.

Ces renseignements sur le comportement global cachent la disparité entre les différentes opérations. Pour comprendre les comportements de chacune des étapes, il faut décomposer le *cpp*. C'est l'objet de la section suivante.

3.4 Parts des étapes intermédiaires dans la composition de la performance globale de l'étiquetage en composantes connexes

La mesure brute des *cpp* ne permet pas de rendre compte du coût des différentes étapes que sont la première passe, la fermeture transitive et réétiquetage. Afin de mieux les comprendre et les évaluer, le code a été instrumenté pour obtenir le *cpp* correspondant à chaque partie. Le cas de Grana est spécifique car la seconde passe y réalise bien plus qu'un réétiquetage. Pour simplifier la lecture des résultats suivants et éviter les confusions, les deux passes de Grana ont été intégrées sous l'appellation première passe.

3.4.1 Résultats pour les images aléatoires

Pour des images aléatoires de granularité g=1 (fig. 3.6), la première passe représente la très grande majorité du temps de traitement global (> 90%) pour tous les algorithmes à l'exception de LSL_{RLE} (> 55%). Le coût de la gestion des étiquettes est négligeable. Le réétiquetage est constant et peu impactant pour tous les algorithmes à l'exception de LSL_{RLE}. Celui-ci doit décompresser tous les segments pour pouvoir générer l'image des étiquettes et cette opération est très dépendante de la taille et du nombre de segments. Or pour g=1, la table contenant le codage RLC ($[j_0, j_1]$) est plus grande que l'image.

Pour des images aléatoires de granularité plus élevée (g=4 fig. 3.7 et g=16 fig. 3.8), le coût de l'étape de réétiquetage augmente relativement à l'ensemble (à l'exception de LSL_{RLE}) car il reste constant en *cpp* et alors que la première passe s'accélère. Du fait de la baisse du nombre de composantes connexes et d'étiquettes supplémentaires il n'est plus possible de distinguer le coût de la gestion des étiquettes. Pour LSL_{RLE}, le coût de l'étape de réétiquetage ne fait que diminuer car il tire alors pleinement parti du codage RLC.

L'accélération progressive de la première passe, commune à tous les algorithmes en fonction de la granularité, s'explique par l'amélioration de la localité temporelle et spatiale des caches lorsque g augmente, conjuguée avec l'avantage d'un nombre réduit d'étiquettes et des parcours plus courts dans l'arbre de décision qui tendent à aplatir les courbes.

3.4.2 Résultats pour les images de SIDBA

Les résultats pour les images de la base de données (tab. 3.9) nous confirment trois informations : la première passe est l'opération la plus coûteuse de l'étiquetage, la fermeture transitive est une opération dont le temps est négligeable devant les autres opérations et le réétiquetage est une opération très stable en temps qui est identique pour tous les algorithmes à l'exception de LSL_{RLE}.

De plus, si l'on compare le *cpp* moyen de la première passe pour les images SIDBA avec celui pour les images aléatoires, il place celles-ci dans l'intervalle $g \in [8 \rightarrow 16]$ pour les algorithmes pixels et $g \in [12 \rightarrow 16+]$ pour les versions LSL.

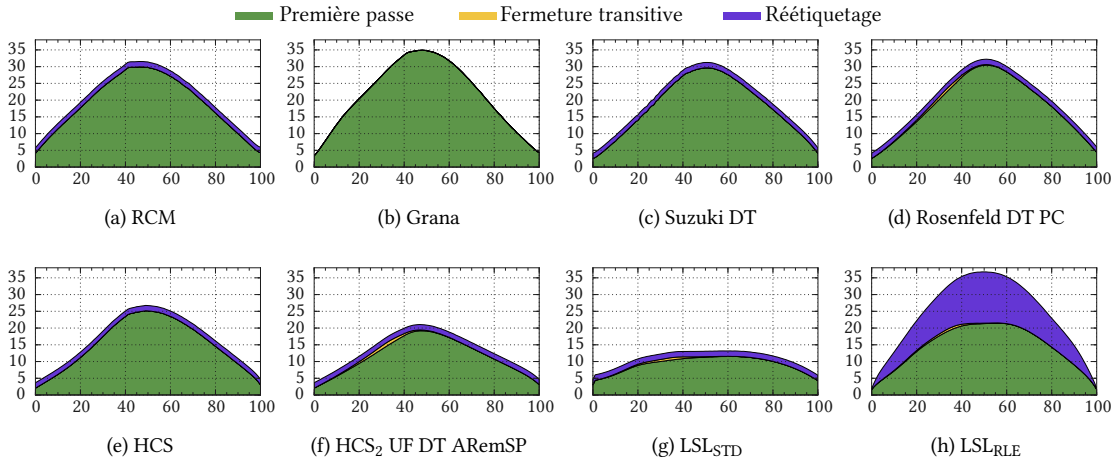


Fig. 3.6 – Composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 1$ sur un cœur Skylake

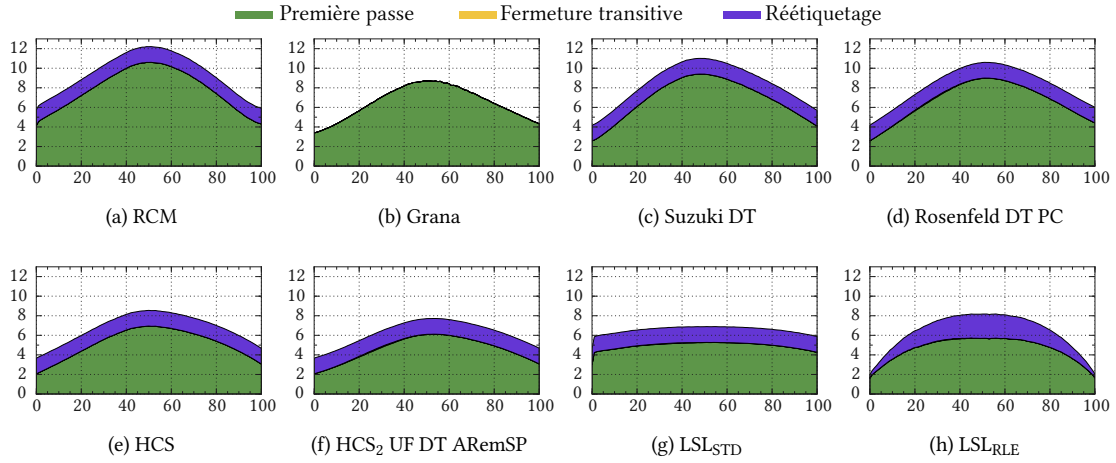


Fig. 3.7 – Composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 4$ sur un cœur Skylake

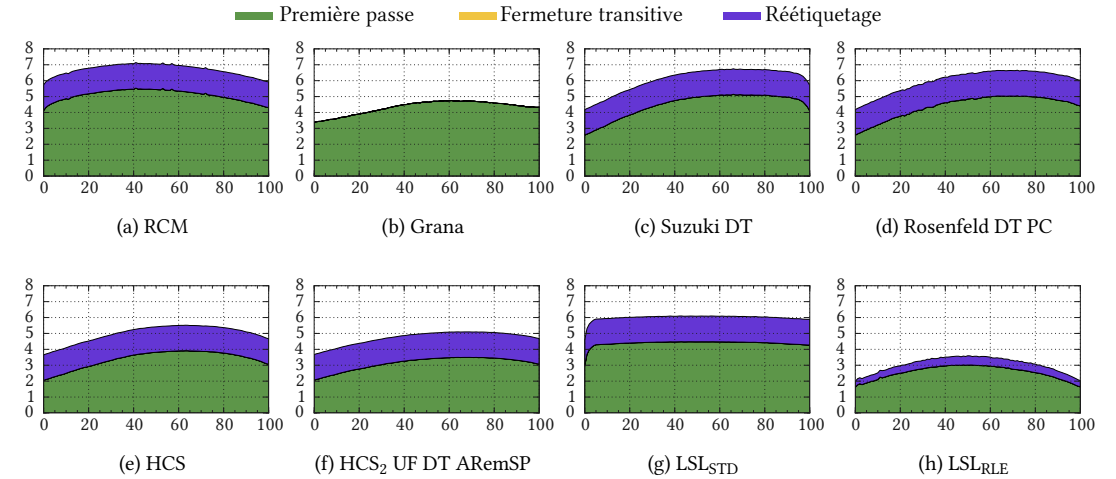


Fig. 3.8 – Composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 16$ sur un cœur Skylake

	Première passe			Fermeture transitive			Réétiquetage		
Algorithmes	min	moy	max	min	moy	max	min	moy	max
LSL_{RLE}	1,87	2,88	4,00	0,00	0,02	0,05	0,49	1,59	2,81
HCS	2,50	3,61	4,83		0[1]		1,52	1,64	1,72
HCS_2 UF DT ARemSP	2,66	3,45	4,13	0,00	0,03	0,08	1,52	1,62	1,72
LSL_{STD}	3,97	4,47	4,95	0,00	0,02	0,05	1,55	1,63	1,74
Suzuki DT	2,51	3,84	5,09		0[1]		1,52	1,64	1,72
Rosenfeld DT PC	3,41	4,71	6,02	0,00	0,03	0,08	1,52	1,62	1,72
RCM	3,75	4,93	6,27		0[1]		1,52	1,64	1,72
Grana	3,90	5,46	7,07[2]		0[1]			0[2]	

[1] Les algorithmes utilisant Suzuki n'ont pas besoin de la fermeture transitive.
[2] La seconde passe de Grana n'est pas un réétiquetage, elle a été comptée ici dans la première passe.

TABLE 3.9 – Algorithmes directs de référence : résultats sur SIDBA exprimés en *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

3.4.3 Conclusion pour l'étiquetage en composantes connexes

LSL_{RLE} confirme ses performances. L'utilisation des segments offre un avantage décisif. De son côté, la fermeture transitive du graphe après étiquetage ne représente pas un enjeu d'amélioration et seul le comportement de la méthode de gestion des étiquettes au sein de la première passe a un impact sur la performance globale. Dans la suite du chapitre, nous ne ferons donc plus apparaître le *cpp* correspondant à la fermeture transitive qui sera intégré à la première passe sauf précision contraire.

L'utilisateur final de l'étiquetage en composantes connexes est l'humain, en ce sens que le réétiquetage n'est pas utile à un algorithme qui dispose de toutes les informations nécessaires dans l'image des étiquettes et la table d'équivalences. Dans la suite du chapitre, nous étudierons la version dédiée aux chaînes de traitement intégrées : l'analyse en composantes connexes.

3.5 Analyse en composantes connexes

Les chaînes de traitement qui intègrent des opérations d'étiquetage doivent disposer ou construire les descripteurs des composantes connexes (cf. sec. 1.6). Dans l'étiquetage en composantes connexes, une fois la première passe terminée, il est nécessaire de procéder à la fermeture transitive puis au réétiquetage de l'image bien que cette dernière étape ne soit pas nécessaire aux calculs des descripteurs. En supprimant l'étape de réétiquetage, le temps de traitement peut diminuer en moyenne (plus pour LSL_{RLE} et rien pour Grana) de 1,6 cycles par pixel (cf. figs. 3.6, 3.7 et 3.8).

Une étape supplémentaire dans l'intégration du calcul des descripteurs est de réaliser les calculs en même temps que la première passe. Cette possibilité est intégrée par construction dans les algorithmes LSL qui de plus tirent parti du codage RLC pour accélérer ces opérations (comme détaillé dans [98] et [99]). Aucun article concernant les algorithmes de la base de référence n'a abordé le calcul des descripteurs et les algorithmes autres que LSL ne sont donc pas prévus pour une telle opération. Pour permettre de comparer tout de même les performances des algorithmes dans le cadre de l'analyse en composantes connexes, nous avons pourvu tous les algorithmes d'une opération de calcul des descripteurs par ligne (double ligne pour HCS_2 et Grana). Pour Grana, la seconde passe ne peut pas être retirée car elle réalise plus d'opérations qu'un simple réétiquetage et le calcul des descripteurs ne peut donc être réalisé qu'après chaque double ligne de la seconde passe.

Dans la suite du chapitre, la passe de réétiquetage est donc supprimée et celle du calcul des descripteurs ajoutée. Il n'y a donc plus qu'une passe comportant deux opérations distinctes. C'est donc un algorithme une passe au sens des catégories (cf : sec 1.2.4.1). Par souci de clarté, le terme première

passe est utilisé pour les opérations équivalentes de l'étiquetage en composantes connexes et le terme descripteurs est utilisé pour la phase de calcul des descripteurs. La valeur du *cpp* spécifique au calcul des descripteurs est obtenue par différence entre le traitement avec et sans calcul des descripteurs.

3.5.1 Résultats pour les images aléatoires

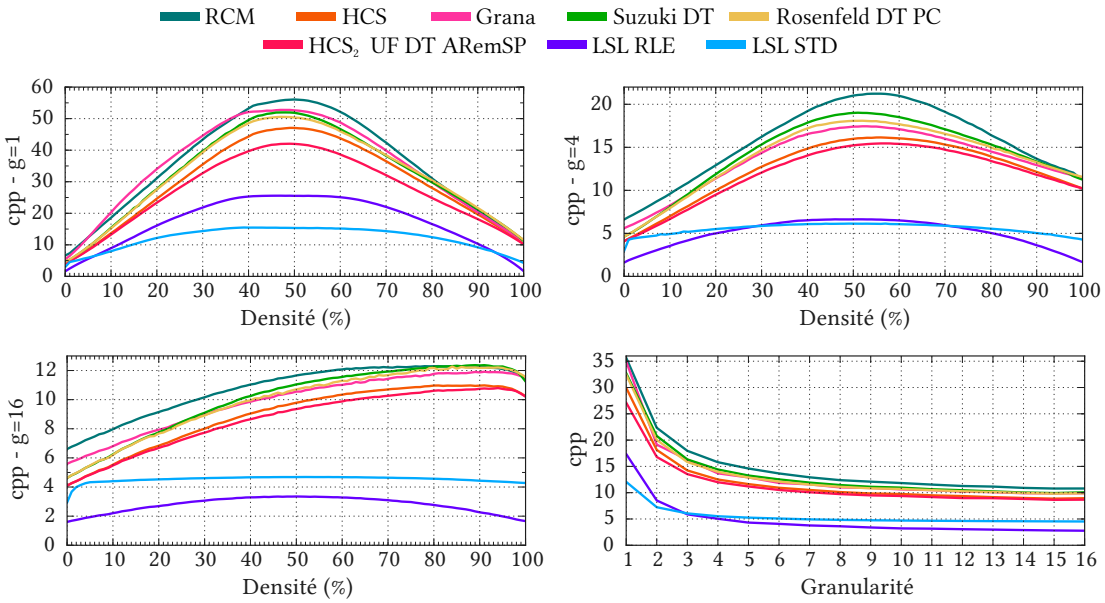


Fig. 3.9 – Analyse en composantes connexes : *cpp* pour des images de taille 1024×1024 et de granularité $g \in \{1,4,16\}$ et *cpp* moyen en fonction de la granularité sur un cœur Skylake

La mesure du *cpp* pour l'analyse en composantes connexes (fig. 3.9) met en évidence la supériorité intrinsèque des algorithmes segments par rapport aux algorithmes pixels pour le calcul des descripteurs. En effet, là où le nombre de cycles par pixel augmente pour les algorithmes pixels, il diminue fortement pour les algorithmes LSL qui tirent profit de la suppression du réétiquetage et du calcul des descripteurs à la volée.

Dès $g > 2$, LSL_{RLE} est plus rapide que LSL_{STD} et dans tous les cas, il est plus rapide que les algorithmes pixels. La figure 3.10 représente le ratio entre LSL_{RLE} et le plus rapide des algorithmes pixels. Le ratio moyen est de 1,66 pour g=1, de 2,49 pour g=4 et de 3,21 pour g=16.

	granularité				
Algorithmes	g=1	g=2	g=4	g=8	g=16
LSL_{RLE}	17,36	8,52	5,04	3,61	2,77
LSL_{STD}	12,06	7,24	5,52	4,83	4,52
HCS_2 UF DT ARemSP	27,12	16,79	11,99	9,73	8,67
HCS	29,95	18,13	12,50	10,12	8,95
Grana	34,96	19,14	13,68	11,00	9,89
Rosenfeld DT PC	32,46	20,04	13,97	11,12	9,90
Suzuki DT	32,72	20,79	14,41	11,45	10,07
RCM	35,66	22,34	15,80	12,38	10,80

TABLE 3.10 – Algorithmes directs de référence : *cpp* moyen pour des images de taille 1024 × 1024 en fonction de la granularité sur un cœur Skylake

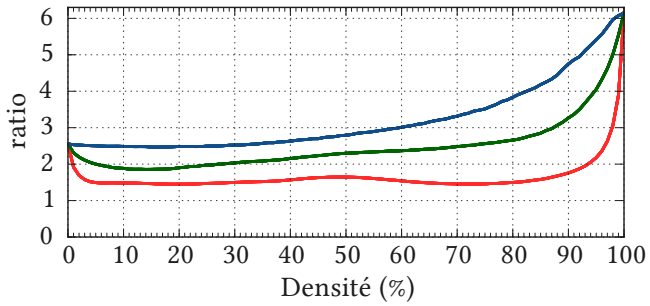


Fig. 3.10 – Analyse en composantes connexes : ratio entre le *cpp* de LSL_{RLE} et le *cpp* du meilleur algorithme pixels sur un cœur Skylake

3.5.2 Résultats pour les images de SIDBA

	t (ms)			*cpp*		
Algorithmes	min	moy	max	min	moy	max
LSL_{RLE}	0,24	0,38	0,56	1,97	3,20	4,65
LSL_{STD}	0,50	0,58	0,65	4,20	4,82	5,45
HCS_2 UF DT ARemSP	0,97	1,20	1,41	8,11	9,97	11,73
HCS	0,97	1,23	1,46	8,11	10,28	12,17
Rosenfeld DT PC	1,05	1,33	1,57	8,75	11,10	13,06
Grana	1,11	1,42	1,67	9,27	11,80	13,88
Suzuki DT	1,09	1,48	1,80	9,12	12,34	15,03
RCM	1,15	1,58	1,87	9,56	13,17	15,61

TABLE 3.11 – Algorithmes directs de référence : résultats sur SIDBA exprimés en *ms* et *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

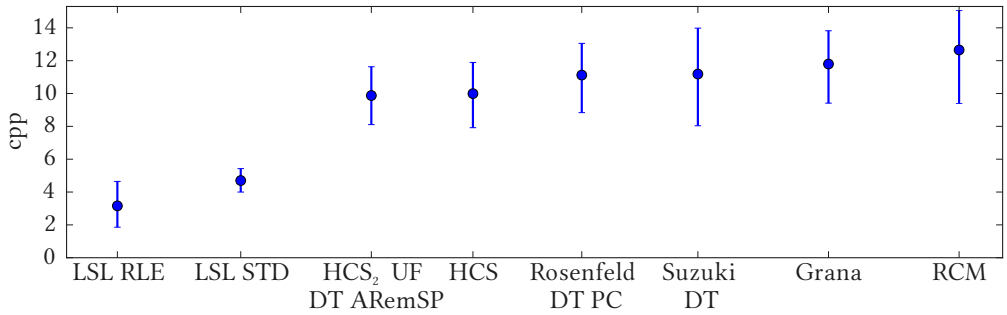


Fig. 3.11 – Analyse en composantes connexes : *cpp* moyen et variabilité (cpp_{max} et cpp_{min}) sur la base de données SIDBA sur un cœur Skylake

Les résultats pour la base de données SIDBA confirment ceux pour les images aléatoires. LSL_{RLE} est en moyenne plus rapide que le meilleur des algorithmes pixels (ici HCS_2 UF DT ARemSP) d'un facteur $\frac{cpp_{LSL}}{cpp_{\text{pixel}}} > 3.1$. Le *cpp* le plus élevé pour LSL_{RLE} et LSL_{STD} est toujours plus faible que le plus faible des *cpp* pour le meilleur des algorithmes pixels. L'algorithme le plus stable est LSL_{STD} ($cpp_{max} - cpp_{min} = 1{,}25$) suivi par LSL_{RLE} ($cpp_{max} - cpp_{min} = 2{,}68$).

3.5.3 Conclusion

Les algorithmes de la famille LSL sont très adaptés à l'analyse en composantes connexes du fait du calcul des descripteurs à la volée et du codage RLC. Le *cpp* pour le traitement total est très proche du *cpp* obtenu pour la première passe de l'étiquetage en composantes connexes. Les algorithmes pixels sont quant à eux ralentis d'un facteur proche de 2 par rapport à leur équivalent pour l'étiquetage. La conjugaison de ces deux phénomènes fait des algorithmes LSL le meilleur choix pour l'analyse en composantes connexes.

Comme pour l'étiquetage en composantes connexes, une connaissance fine du coût des étapes intermédiaires pourra nous renseigner sur les causes de ce constat.

3.6 Part des étapes intermédiaires dans la composition de la performance globale de l'analyse en composantes connexes

Afin de comprendre l'influence de la nouvelle composition *première passe + calcul des descripteurs* (la fermeture transitive ayant été intégrée dans la première passe) dans les variations des résultats globaux, nous avons réalisé l'étude du coût relatif des différentes phases de l'analyse en composantes connexes.

3.6.1 Résultats pour les images aléatoires

Pour g=1 (fig. 3.12), le calcul des descripteurs pour les algorithmes pixels représente une part importante du *cpp* total : entre 34% (Grana) et 47% (RCM) dans le pire cas. Pour les algorithmes LSL, cette part est de 27% pour LSL_{STD} et de 17% pour LSL_{RLE}. Bien que cette proportion soit similaire entre Grana et LSL_{STD}, il faut prendre en compte que LSL_{STD} est globalement plus rapide que Grana d'un facteur ×2,8 .

Pour les granularités supérieures (figs. 3.13 et 3.15), tous les algorithmes pixels tendent à avoir un $cpp_{\text{descripteurs}}$ identique et dont la part progresse du fait de l'accélération de la première passe jusqu'à atteindre 74% dans le pire cas pour g=16. Dans le même temps, du fait du codage RLC, cette part diminue et passe à 5% pour LSL_{STD} avec g=16 et 8% pour LSL_{RLE} avec g=16.

3.6.2 Résultats pour les images de SIDBA

Sur la base SIDBA, le calcul des descripteurs est l'opération la plus coûteuse pour les algorithmes pixels et représente en moyenne entre 52% et 64% du *cpp* total. Il ne représente que 3,9% pour LSL_{RLE} et 4,6% pour LSL_{STD}.

	Première passe			Descripteurs		
Algorithmes	min	moy	max	min	moy	max
LSL_{RLE}	1,97	3,08	4,29	0,00	0,12	0,35
LSL_{STD}	4,16	4,59	4,97	0,05	0,23	0,48
HCS_2 UF DT ARemSP	2,80	3,64	4,42	5,31	6,33	7,57
HCS	2,89	4,01	5,11	5,22	6,27	7,52
Rosenfeld DT PC	3,46	4,85	6,27	5,29	6,26	7,45
Suzuki DT	3,70	5,04	6,32	5,42	7,29	9,17
Grana	4,02	5,63	7,37	5,25	6,16	7,36
RCM	3,91	5,41	7,15	5,51	7,76	9,76

TABLE 3.12 – Analyse en composantes connexes : résultats sur SIDBA exprimés en *cpp* pour les valeurs minimale (min), moyenne (moy) et maximale (max) sur un cœur Skylake

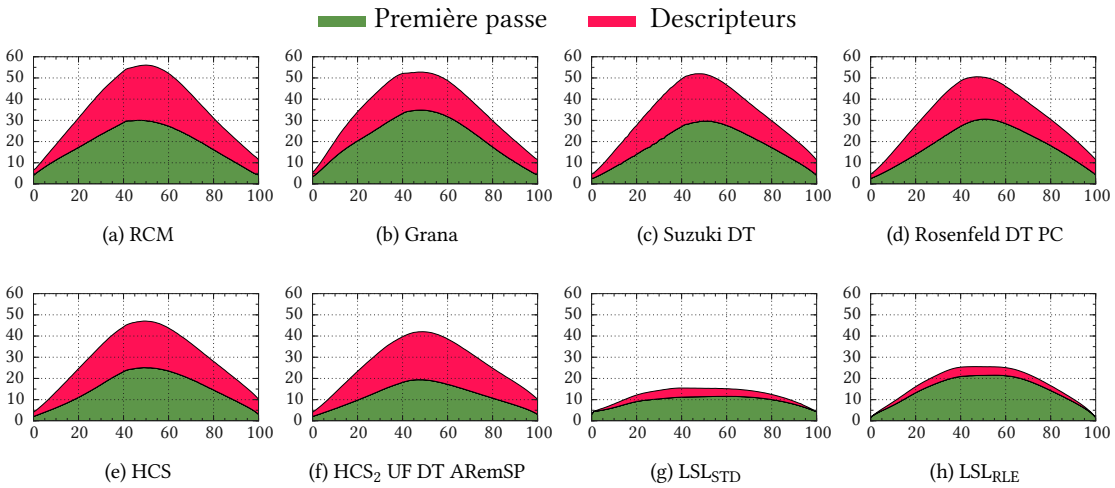


Fig. 3.12 – Analyse en composantes connexes : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 1$ sur un cœur Skylake

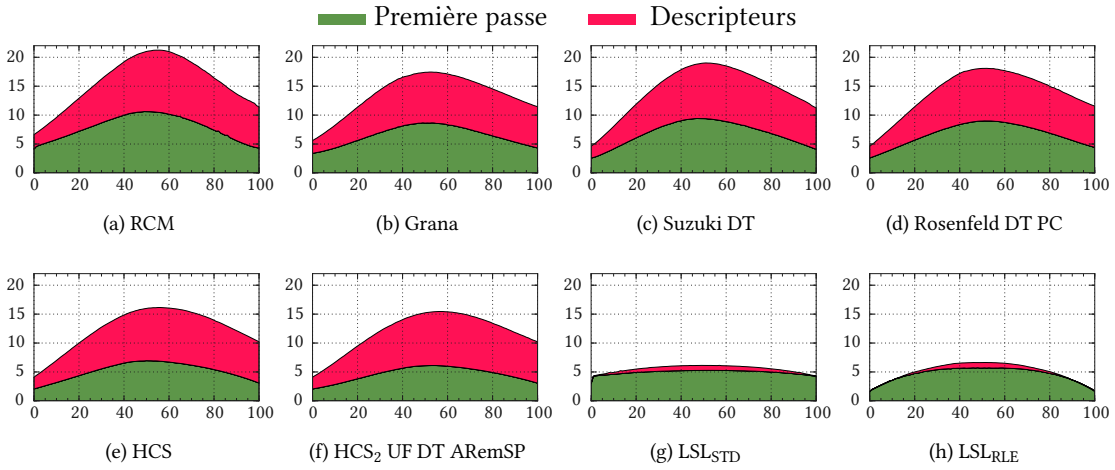


Fig. 3.13 – Analyse en composantes connexes : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 4$ sur un cœur Skylake

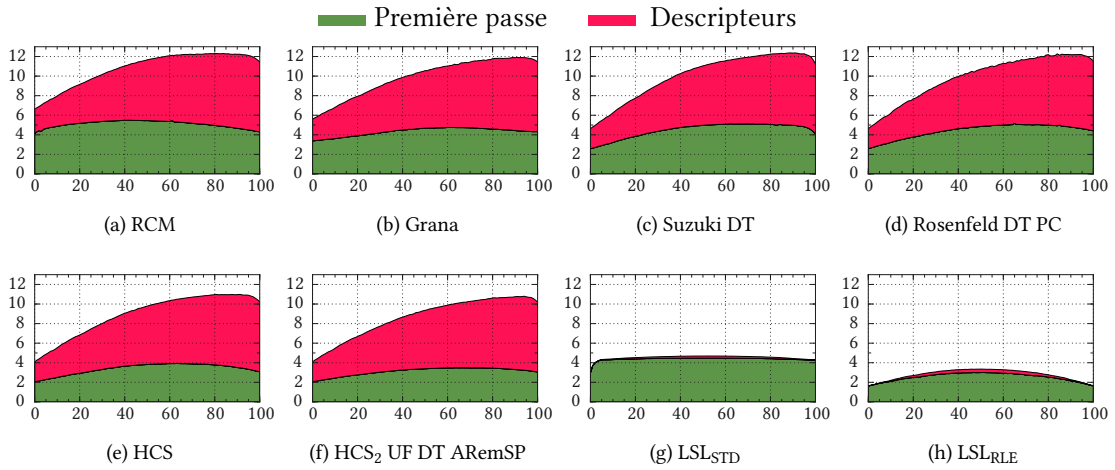


Fig. 3.14 – Analyse en composantes connexes : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 1024×1024 et $g = 16$ sur un cœur Skylake

3.6.3 Conclusion

L'analyse des *cpp* intermédiaires nous indique que le calcul des descripteurs pour les algorithmes LSL est une opération dont le coût est très faible. D'une part, les calculs sont masqués par les opérations de contrôle et d'autre part, l'utilisation du codage RLC permet de tirer parti de la structure des images et rend le calcul des descripteurs symétrique vis-à-vis de la densité 50%.

L'analyse en composantes connexes est bien plus rapide lorsqu'elle est basée sur les segments que sur les pixels. L'ensemble des algorithmes pixels présente des résultats très proches. La forme du masque, la méthode de gestion des étiquettes, et le nombre de passes, sont des éléments bien moins différenciants que l'utilisation des segments au lieu de pixels. Les algorithmes LSL sont donc sans conteste, les plus adaptés à l'analyse en composantes connexes.

3.7 Évolution des performances avec les générations d'architectures

Afin de comprendre les liens entre les algorithmes et les architectures, nous avons validé nos résultats sur d'autres familles de processeurs. Nous avons mené les tests d'analyse en composantes connexes sur plusieurs machines différentes (machine de bureau, portable et serveurs). Le tableau 3.13 récapitule les architectures testées et leurs caractéristiques. Chaque algorithme représente un choix d'équilibre entre le nombre d'accès à la mémoire, les opérations de contrôle et de calcul. D'un autre côté, chaque famille de processeur est aussi un équilibre entre des caractéristiques telles que : les capacités de calcul, la bande passante mémoire, la quantité de cache, les mécanismes de prédictions, etc. La figure 3.15 présente le *cpp* des différents algorithmes sur toutes les architectures sélectionnées pour les images de SIDBA. Les architectures récentes sont en moyenne plus efficaces avec des variations selon les algorithmes. L'évolution la plus marquante date de l'arrivée de la famille Nehalem qui coïncide avec la présence d'un cache L3 et l'augmentation significative de la bande passante mémoire due au remplacement du FSB (Front Side Bus) par le lien QPI (Quick Path Interconnect). LSL_{RLE} a progressé à chaque génération, il est plus rapide d'un facteur ×2,4 sur le SKL que sur le CNR. Dans le même temps LSL_{STD}, a progressé d'un rapport ×1,4 et les algorithmes pixels d'un rapport compris entre ×1,3 et ×1,6.

Alias	Famille	Identifiant	fréquence (GHz)	Mémoire Cache	Bande passante mémoire maximale	Date de lancement
CNR	Conroe	E6600	2,4	4Mo L2	8,4 Go/s	Q3'06
PNR	Penryn	SU9400	1,4	3Mo L2	6,4 Go/s	Q3'08
NHM	Nehalem	W3530	2,8	8Mo L3	25,6 Go/s	Q1'10
SNB	Sandy Bridge	i7-2600K	3,4	8Mo L3	21,0 Go/s	Q1'11
HSW	Haswell	i7-4770K	3,5	8Mo L3	25,6 Go/s	Q2'13
IVB	Ivy Bridge	E5-2695 v2	2,4	30Mo L3	59,7 Go/s	Q3'13
SKL	Skylake	i7-6700K	4,0	8Mo L3	34,1 Go/s	Q3'15

TABLE 3.13 – Familles de processeurs ayant servi de base pour évaluer l'évolution des performances de l'analyse en composantes connexes comparativement à l'évolution des processeurs

Dans le même temps, le nombre de cœurs des processeurs a augmenté (de 2 pour CNR à 12 pour IVB) mais les algorithmes de référence ne sont pas prévus pour travailler sur plusieurs cœurs et une adaptation est donc nécessaire. Ce sera l'objet du chapitre suivant.

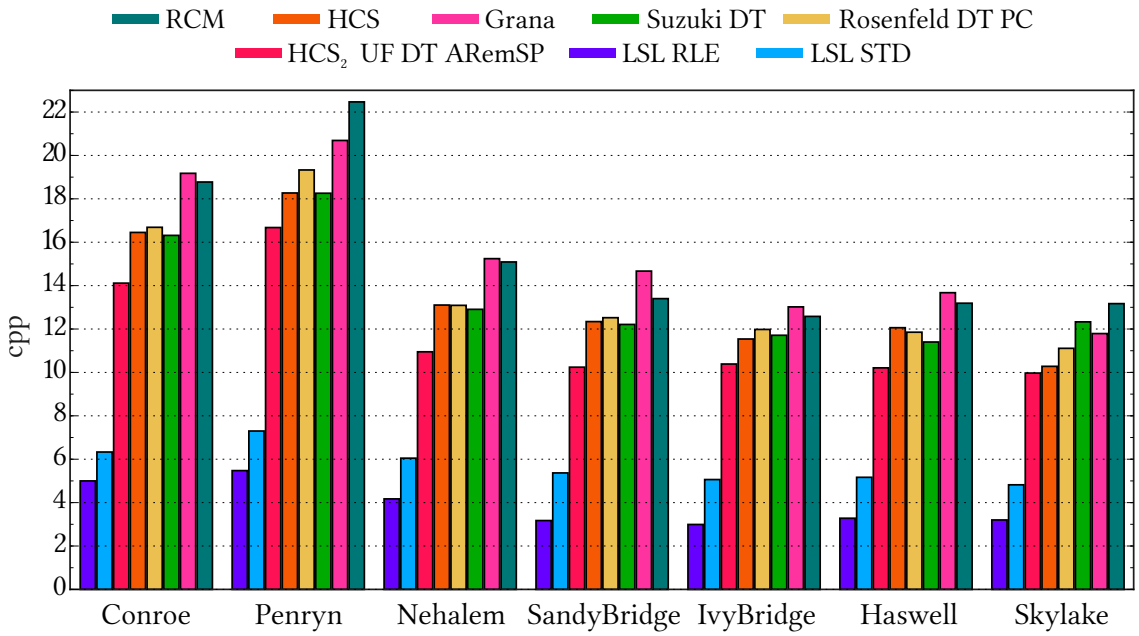


Fig. 3.15 – Analyse en composantes connexes : *cpp* moyen pour les algorithmes de référence sur la base de données SIDBA sur 7 architectures du Conroe (2006) au Skylake (2015)

3.8 Conclusion

Ce chapitre a permis de mettre en évidence le comportement des algorithmes directs dans un contexte séquentiel et de faire le point sur la hiérarchie des algorithmes modernes. Il apparaît que les algorithmes segments sont plus rapides pour l'étiquetage en composantes connexes et que cet avantage augmente dans le cadre d'applications réelles d'analyse en composantes connexes. LSL_{RLE} est donc le plus efficace des algorithmes d'étiquetage en composantes connexes et d'analyse en composantes connexes.

La problématique de l'étiquetage comme celle du traitement d'image est plus que jamais orienté vers la rapidité et vers l'autonomie par deux phénomènes conjugués.

- Le passage à un paradigme de données massives induit par la croissance continue de la quantité d'images nécessite de traiter les images rapidement et sans intervention humaine. Par exemple, la base de données ImageNet [24] qui regroupe des images et les données associées propose plus de 14 millions d'images dans 21841 catégories. Mais dans le même temps, le nombre de photos partagées par jour sur les réseaux sociaux dépasse les 500 millions (en 2013) et double tous les ans)
- L'avènement d'applications nécessitant un traitement temps réel d'images dont la résolution augmente d'année en année : robotique autonome, voiture intelligente, vidéosurveillance, drones, smartphones.

Dans le contexte d'autonomie de traitement vis-à-vis de l'humain, l'analyse en composantes connexes est à privilégier et sera donc mise en avant dans la suite du manuscrit. Cependant, son caractère séquentiel ne permet pas de tirer pleinement parti des performances des processeurs modernes. Pour faire progresser l'étiquetage il faut envisager la parallélisation des algorithmes.

Chapitre 4

Moins que d'autres, je ne savais si le but de notre vie avait un sens. Mais je savais, plus que quiconque, qu'elle avait une valeur. Par elle-même, directement, hors de toute réussite ou déroute. Cette valeur venait du combat.

–La Horde du contrevent, Alain Damasio

Étiquetage en composantes connexes pour les architectures multi-cœur

4.1 Introduction

Depuis les années 80 jusqu'au début des années 2000, les générations successives des technologies MOS puis CMOS ont permis une croissance exponentielle, de l'ordre de 25% par an, de la fréquence d'horloge des processeurs (fig. 4.1). La combinaison de cette augmentation des fréquences et des progrès microarchitecturaux (superscalaires dans l'ordre puis non ordonné, prédicteurs de branchement, hiérarchie de caches, nouvelles instructions, etc.) a permis des gains de performance de l'ordre de 50% par an pour les générations successives de processeurs, permettant de réduire les temps de traitement d'une application, ou de traiter des applications de plus grande taille. C'est à cette période qu'Herb Sutter fait référence sous l'appellation « free lunch » dans l'article [100]. Pour obtenir plus de performance, il suffisait d'attendre la prochaine génération de processeurs d'une famille donnée, éventuellement en recompilant le programme pour utiliser les nouvelles instructions. Le début des années 2000 a vu la fin de la croissance des fréquences d'horloge, limitée depuis à 4 GHz maximum pour limiter la densité de puissance dissipée à ce que peuvent supporter les boîtiers couramment utilisés. Ces problèmes thermiques, se combinant avec les limites du « parallélisme d'instructions » exploitable par un monoprocesseur (cœur) a conduit aux architectures multi-cœur, qui sont maintenant la norme dans toutes les gammes de processeurs, des processeurs des smartphones et tablettes aux superordinateurs en passant par les PC et les serveurs.

Dans le cadre de la thèse, nos travaux ont porté sur deux catégories de plateforme : les multi-cœur pour lesquels nous proposons une implémentation des algorithmes directs dans ce chapitre ainsi qu'une analyse des résultats dans le chapitre 5 et les many-core (Xeon Phi et GPU) pour lesquels nous proposons des algorithmes spécifiques dans les chapitres 6 et 7.

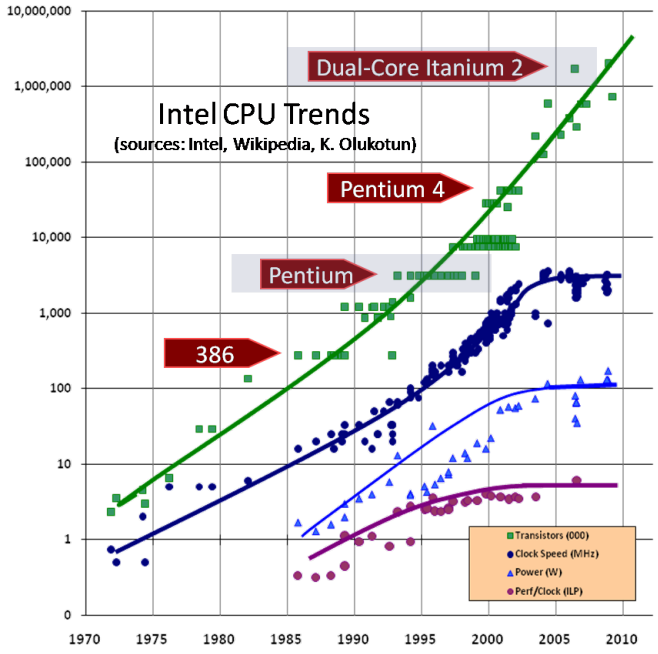


Fig. 4.1 – Courbes tendancielles issues de «The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software» [100] (mise à jour en 2009)

4.2 Découpage des données pour le multi-cœur

4.2.1 Principe

L'utilisation de plusieurs cœurs implique de réécrire les programmes pour implémenter les versions parallèles des algorithmes. Selon la nature des algorithmes, le travail à effectuer est très différent, allant du simple découpage des données à la refonte complète des algorithmes.

Il est tout à fait possible pour l'étiquetage comme pour l'analyse en composantes connexes de traiter en même temps différentes images sans modification algorithmique. Cela ne correspond pas à l'ensemble des applications d'étiquetage. Dans le cadre de nos travaux, nous avons fait le choix d'analyser la parallélisation du traitement d'une image sur plusieurs cœurs pour proposer un nouvel algorithme permettant à la fois de traiter plus de données mais aussi de produire plus rapidement l'analyse en composantes connexes d'une image, ce qui est indispensable par exemple dans le cas d'applications embarquées.

Une façon d'aborder l'adaptation d'un algorithme dans le cadre d'une utilisation multi-cœur est de suivre le schéma de la figure 4.2 :

- un découpage des structures de données (homogène ou non),
- un traitement en parallèle de ces données avec d'éventuelles barrières de synchronisation ainsi que des mécanismes de verrous sur les données pour gérer la concurrence d'accès,
- enfin une éventuelle étape de fusion / réconciliation des données.

Ce découpage peut se répéter et varier plusieurs fois au sein de l'algorithme, par exemple pour chaque passe.

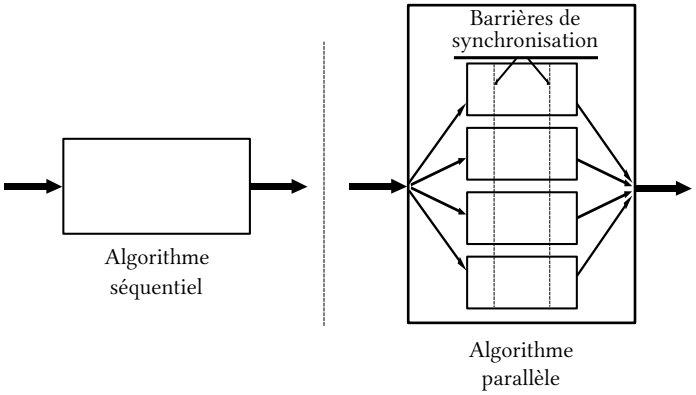
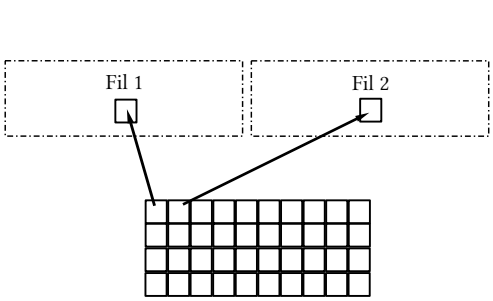


Fig. 4.2 – Parallélisation sur des cœurs

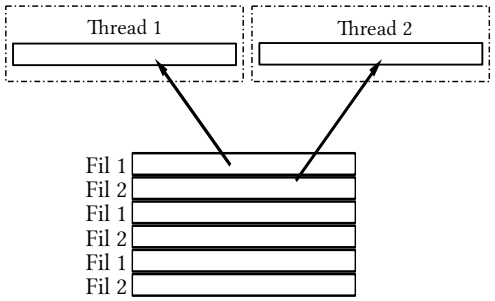
4.2.2 Structures de données

4.2.2.1 Granularité de la décomposition

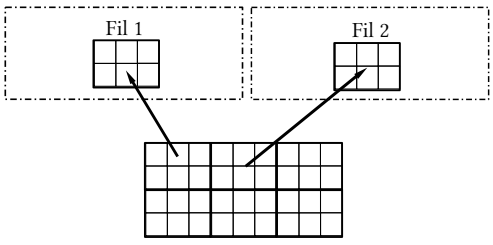
La décomposition efficace des structures de données pour un traitement parallèle est très dépendante de l'algorithme et résulte d'un compromis entre les dépendances de données internes et l'équilibrage désiré de la charge de calcul entre les processeurs. Nous présentons ici quelques décompositions et équilibrages qui seront discutés dans la section suivante en fonction de leur application potentielle dans le cadre de l'étiquetage et de l'analyse en composantes connexes.



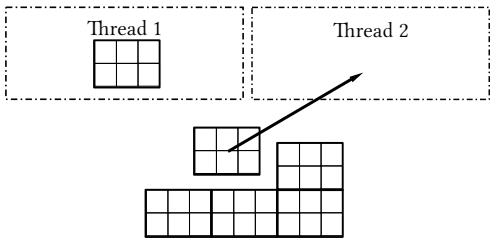
(a) Décomposition en pixels *a priori* : les pixels pairs sont traités par le thread 1 et les pixels impairs par le thread 2



(b) Décomposition en lignes *a priori* : les lignes paires sont traitées par le thread 1 et les lignes impaires par le thread 2



(c) Décomposition étendue en tuiles : les tuiles paires sont traitées par le thread 1 et les tuiles impaires par le thread 2



(d) Système de pile de tuiles (farming) : le thread traite la tuile suivante dans la pile des tuiles à traiter

Fig. 4.3 – Exemples de découpages en blocs élémentaires pour 2 threads : pixels, lignes, tuiles avec ordonnancement *a priori* ou par pile

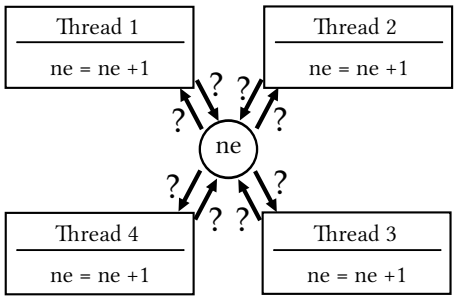
Le principe est de déterminer, d'une part la taille et la forme de l'élément de base qui sera distribué entre les différents threads (figs. 4.3a, 4.3b et 4.3c), et d'autre part l'ordonnancement de l'attribution

des données aux différents threads.

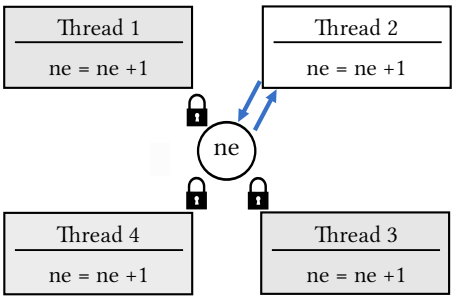
L'ordonnancement dépend de la variabilité du temps de traitement des éléments de base et de la dépendance de données entre ces éléments. En effet, dans l'exemple des figures 4.3a, 4.3b et 4.3c qui représentent une répartition *a priori* de la charge, un thread peut avoir traité l'ensemble des données prévues avant l'autre et se retrouver inactif. Cela représente une perte du point de vue de la répartition de la charge de travail. Une solution est donc que chaque thread récupère de nouvelles données dès qu'il est disponible (figs. 4.3d).

4.2.2.2 Verrous

Si dans le cadre d'un programme, plusieurs threads souhaitent accéder en même temps à une donnée (*ne*), pour par exemple l'incrémenter et enfin réécrire la nouvelle valeur en mémoire (fig. 4.4a), il est impossible de garantir la valeur de *ne* à la fin de l'opération. En effet, si toutes les lectures ont lieu avant la première écriture, il n'y aura qu'une incrémentation effective. Selon le décalage entre les threads le résultat, sera $ne + 1$, $ne + 2$, $ne + 3$ ou $ne + 4$. Afin d'éviter cette incertitude, le mécanisme de verrou vient interdire l'accès à *ne* à tous les autres threads lorsqu'une opération est en cours (fig. 4.4b). Dans tous les cas, le résultat sera $ne + 4$ mais les trois autres threads auront dans ce cas été bloqués durant le travail du premier.



(a) Accès simultanés par plusieurs threads



(b) Sérialisation des accès par verrou : aucun thread ne peut lire *ne* tant que le thread 2 n'a pas libéré la ressource. Ensuite, le thread suivant posera à son tour un verrou sur *ne*

Fig. 4.4 – Principe du verrou

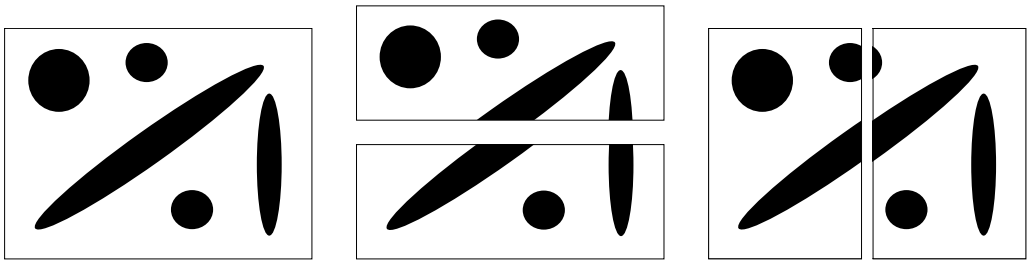
4.2.3 Cas de de l'étiquetage et de l'analyse en composantes connexes

Comme nous l'avons vu dans les chapitres précédents, plusieurs structures de données sont à considérer :

- L'image binaire à étiqueter.
- L'image des étiquettes.
- Le graphe de connexité (sous la forme d'une ou plusieurs tables d'équivalence).
- Les structures qui permettent de calculer les descripteurs.

4.2.3.1 Décomposition de l'image binaire à étiqueter

Le chargement de l'image binaire à étiqueter ne présente aucune dépendance de données et celle-ci peut donc être découpée selon tous les modes présentés dans la section 4.2.2.1. Il est donc aussi possible d'étendre la taille des tuiles au maximum pour obtenir des bandes verticales ou horizontales (fig. 4.5). Cela a l'avantage de découper l'image de base en sous-images et de se retrouver dans une configuration proche des algorithmes séquentiels.



(a) Image binaire

(b) Image binaire découpée en 2 bandes horizontales à traiter en parallèle

(c) Image binaire découpée en 2 bandes verticales à traiter en parallèle

Fig. 4.5 – Découpage de taille maximale pour deux threads

4.2.3.2 Décomposition de l'image des étiquettes

Dans le cas des algorithmes d'étiquetage en composantes connexes et d'analyse en composantes connexes, la dépendance de données due au masque et au balayage direct rend très coûteux la parallélisation par découpage en éléments d'un pixel ou d'une ligne. En effet, dans le cas du masque de Rosenfeld (fig. 4.6), le traitement de e_x dépend du pixel précédent e_4 et des trois pixels de la ligne précédente e_1, e_2, e_3. Dans le cas d'un découpage à l'échelle d'un pixel, aucun traitement ne pourrait débuter sans que ces 4 étiquettes n'aient été attribuées. Dans le cas d'un découpage en lignes, c'est toute la ligne précédente qui devrait avoir été traitée en plus du pixel précédent. De plus, plus le nombre d'éléments est élevé, plus le nombre de fusions à réaliser pour l'étape de réconciliation des données est élevé.

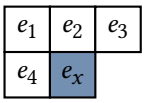


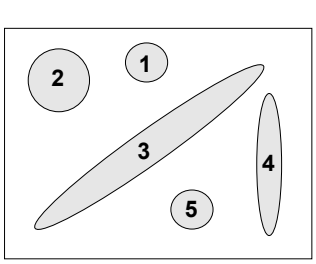
Fig. 4.6 – Masque de Rosenfeld

Le chapitre 6 explorera des mécanismes pour diminuer la dépendance entre les données. Dans ce chapitre, l'approche choisie est de limiter les évolutions par rapport aux algorithmes séquentiels. En suivant cette logique, la dépendance des données ne pouvant pas être supprimée, une solution est de maximiser la taille de l'élément de base et de minimiser le nombre de fusions. Le découpage en bandes répond à ces critères. Il a de plus l'avantage de respecter les accès successifs aux lignes de caches et d'éviter les interactions entre les threads *via* le cache (false-sharing).

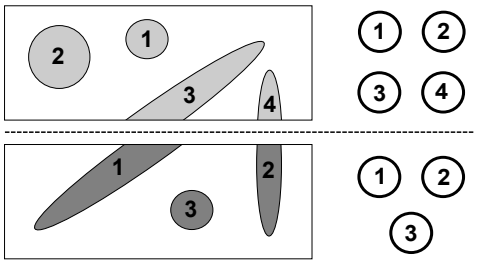
4.2.3.3 Décomposition du graphe des connexités

En considérant un découpage en bandes horizontales et en considérant chaque bande comme une image à part, plusieurs variantes sont envisageables.

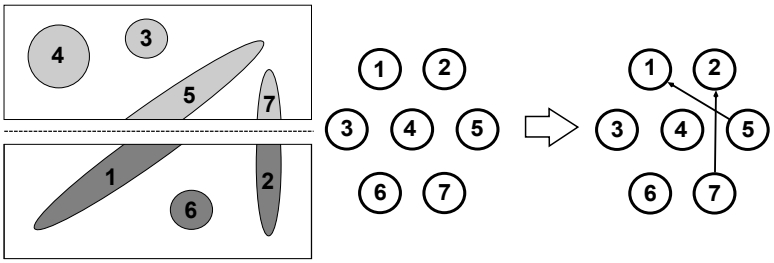
- Un version naïve utilisant deux structures séparées et indépendantes de table d'équivalences donnerait deux sous-images étiquetées et deux graphes indépendants. Lors de la fusion des bandes, les graphes ayant des étiquettes identiques, des erreurs d'étiquetage seraient générées. Dans l'exemple de la figure 4.7b, la composante 3 du graphe 1 n'est pas connexe avec celle du graphe 2, mais suite à la fusion, elle sera considérée comme telle. Cette solution n'est donc pas correcte et illustre que le découpage et la fusion correspondante doivent être envisagées en tenant compte de l'image des étiquettes et du graphe de connexité.



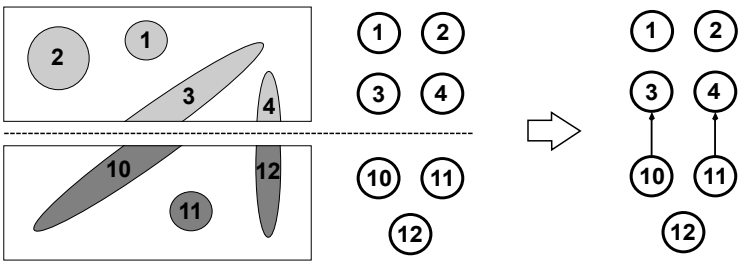
(a) Image des étiquettes



(b) Version naïve : l'image des étiquettes est découpée en 2 bandes horizontales avec deux graphes de connexité indépendants : l'étape de fusion provoquera des erreurs



(c) Image des étiquettes découpée en 2 bandes horizontales avec un graphe de connexité unique : l'étape de fusion sera correcte mais les accès au graphe seront concurrents pendant la phase d'étiquetage des bandes



(d) Image des étiquettes découpée en 2 bandes horizontales avec un graphe de connexité unique à plages d'étiquettes disjointes (ici 1 à 9 et 10 à 19) : l'étape de fusion sera correcte et les accès au graphe seront indépendants

Fig. 4.7 – Découpages en bandes horizontales de l'image des étiquettes et structures de graphes correspondantes

- En utilisant une structure unique de table d'équivalences avec un partage de l'accès au compteur des étiquettes (*ne*) entre les deux threads, on obtient bien un résultat cohérent après fusion des deux bandes. Mais un mécanisme de verrou est nécessaire sur *ne* pour s'assurer que la même étiquette n'est pas attribuée à 2 composantes connexes et que le compteur des étiquettes n'est pas incrémenté 2 fois (cf. 4.2.2.2).
- En utilisant une structure unique de table d'équivalences mais avec des plages de *ne* différentes et des compteurs d'étiquettes indépendants, on obtient bien un résultat cohérent après union des deux bandes sans pour autant avoir à partager l'accès entre les threads.

4.2.3.4 Mécanisme de fusion des bandes

Le découpage, quelle que soit la forme des éléments, impose un mécanisme supplémentaire pour obtenir un résultat correct. Les sous-images obtenues et leur graphe doivent être fusionnés en fonction de la connexité entre les pixels des bandes. Cette information est contenue aux frontières des bandes. Il est nécessaire de réaliser une lecture du voisinage des pixels de frontières et de réaliser les opérations

d'unions correspondantes (fig. 4.8). Cette étape doit être faite après le traitement des deux bandes à fusionner.

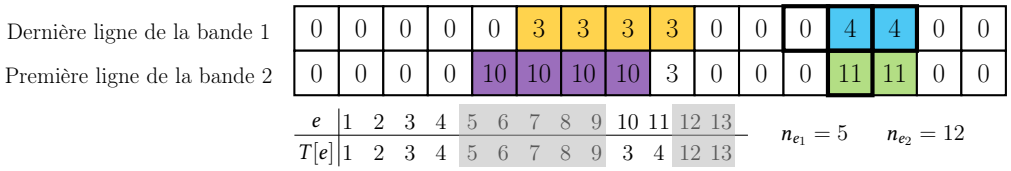
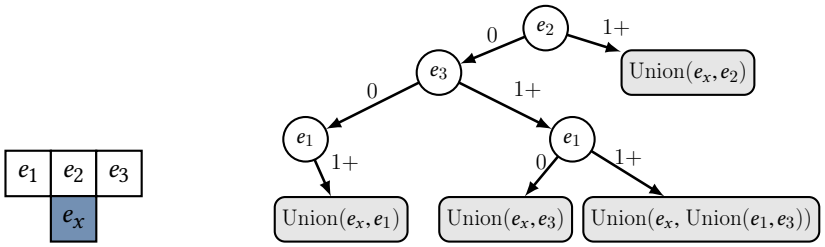


Fig. 4.8 – Fusion des deux bandes avec deux compteurs d'étiquettes, la séparation des plages d'étiquettes permet une union correcte (les étiquettes non utilisées sont grisées dans la table d'équivalences)

La figure 4.8 donne un exemple d'union basé sur le découpage de la figure 4.7d.



(a) Masque pixel de fusion des bandes

(b) Arbre de décision pour la fusion des bandes

Fig. 4.9 – Fusion des bandes : masque et arbre de décision correspondant

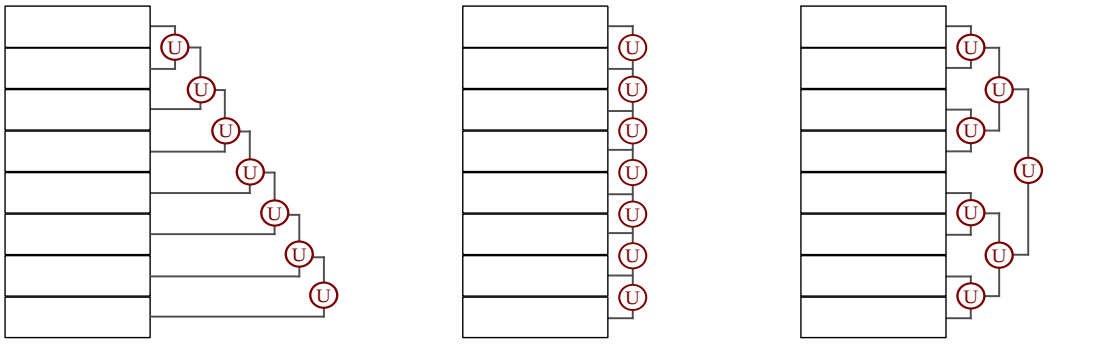
4.2.3.5 Ordonnancement des fusions

La figure 4.10 présente 3 mécanismes d'ordonnancement des fusions :

- Une méthode séquentielle ou *en cascade* (fig. 4.10a), qui par construction ne nécessite pas de mécanisme de verrou sur les données contenues dans les graphes. En effet, une fusion n'est possible que si la fusion précédente est terminée. Cette méthode réalise les n-1 étapes de fusion en n-1 unités de temps (variables selon les données).
- Une méthode parallèle (fig. 4.10b), où toute les fusions sont réalisées simultanément. Afin de rester valide, cette solution nécessite de mettre en place des verrous sur les données du graphe pour s'assurer que le graphe n'est pas cassé lors des mises à jour simultanées.
- Une méthode pyramidale (fig. 4.10c), où les bandes sont fusionnées deux à deux, puis les bandes produites sont fusionnées deux à deux jusqu'à obtenir toute l'image des étiquettes. Cette solution ne nécessite aucun verrou sur les données car les ensembles d'étiquettes sont deux à deux disjoints mais une fusion ne peut cependant démarrer que lorsque les deux bandes à fusionner sont correctement étiquetées ou fusionnées. Cette méthode réalise les n-1 étapes de fusion en $log_2(n)$ unité de temps (variables selon les données).

4.2.3.6 Calcul des descripteurs

Le calcul des descripteurs est réalisé dans les bandes et n'est donc pas affecté directement par le découpage en bandes. Cependant, lors de l'étape d'union des bandes, il est nécessaire d'utiliser le mécanisme d'union des descripteurs tel que décrit en section 2.5. Pour la validité des résultats, il est



(a) Unions en cascade : aucun risque d'accès concurrent aux données mais n-1 étapes successives

(b) Unions simultanées : une seule étape mais un accès concurrent aux données

(c) Unions en structure pyramidale : pas d'accès concurrent aux données et $log_2(n)$ étapes

Fig. 4.10 – Exemples de découpage d'ordonnancement des unions de bandes

nécessaire de s'assurer que l'union de deux étiquettes n'est réalisée qu'une seule fois. Considérons le masque de la figure 4.9a. Si $e_1 = 3$, $e_3 = 3$ et $e_x = 13$, la première union de 13 et 3 doit empêcher la seconde, sinon les descripteurs de l'étiquette 13 seront comptés deux fois. Une remontée systématique à la racine avant toute union permet d'éviter cette configuration.

4.2.3.7 Conclusion

Il existe plusieurs solutions viables pour le découpage des données mais toutes ne sont pas efficaces. Les solutions sans verrou sur les données seront plus indépendantes de l'ordonnancement réel des opérations sur le processeur. La section suivante fera le point sur les travaux antérieurs sur la parallélisation d'algorithmes directs existants dans la littérature et abordera les travaux réalisés sur des architectures non généralistes.

4.3 Travaux antérieurs de parallélisation de l'étiquetage en composantes connexes

Peu de travaux présentent des parallélisations d'algorithmes directs sur processeurs multi-cœur généralistes et aucun ne prend en compte le calcul des descripteurs. Nous présentons ici trois articles qui ont proposé des adaptations ainsi que des travaux sur des architectures spécialisées.

4.3.1 Travaux sur architectures modernes généralistes

4.3.1.1 Niknam *et al.*

[101] présente la parallélisation de l'algorithme Suzuki sur une machine 16 cœurs AMD Opteron 885. L'image est découpée en bandes verticales (fig. 4.11). Lorsque un thread termine une ligne, il permet au thread suivant de démarrer et peut continuer de son côté. Ainsi une fois la première ligne étiquetée, tous les cœurs sont actifs. Cette méthode introduit une dépendance à chaque ligne et pour chaque thread (sauf le dernier). Selon l'équilibre de l'image à traiter, les dépendances ralentiront ou non le processus.

L'accélération maximale atteinte est ×2,5 sur 4 threads pour des images 256 × 256. Avec le même nombre de threads, l'accélération tombe à ×1,3 pour des images de taille 512 × 512. Avec 16 threads, l'accélération est de ×1,2 pour des images 256 × 256 et ×1,25 pour des images 512 × 512. Pour les

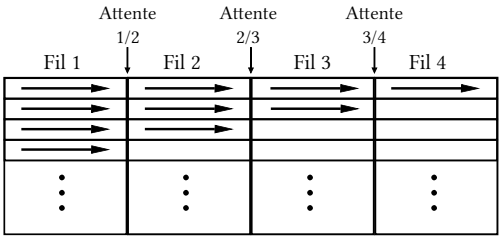


Fig. 4.11 – Découpage en bandes verticales de Niknam et al : le thread 2 attend la fin de chaque ligne du thread 1 (1/2), idem pour 2/3 et 3/4

auteurs, l'explication de cette contre performance est à imputer aux accès aléatoires non uniformes (NUMA) ainsi qu'aux nombreux défauts de cache pour les images de grande taille (512×512).

4.3.1.2 Chen *et al.*

[102] présente la parallélisation de l'algorithme Suzuki sur le processeur Tile64 (Tilera). Le découpage est une nouvelle fois vertical mais chaque bande empiète d'une colonne sur les bandes voisines. Les zones frontières sont donc étiquetées deux fois. Deux mécanismes d'union de frontières sont étudiés : en cascade ou de manière pyramidale. La meilleure accélération est d'environ $\times$11,4 avec 48 cœurs pour des images de taille 2000×1500. L'accélération maximale n'atteint que $\times$4,2 avec 16 cœurs pour des images 256×256 et $\times$5,1 avec 24 cœurs pour des images 512×512.

4.3.1.3 Gupta *et al.*

[64] met en œuvre la version parallèle de l'algorithme HCS_2 + PARemSP. PARemSP est la version parallèle de ARemSP. Le découpage se fait en bandes horizontales et l'étape de fusion des frontières se fait en parallèle avec des verrous OpenMP [103] pour assurer la validité des opérations en situation de concurrence. La machine est un 2×12 cœurs AMD Magnycours. La meilleure accélération est d'environ $\times$10 avec 24 threads pour les images issues d'une base de données de taille et de propriétés variables. Afin d'atteindre un niveau de performance plus élevé, les auteurs doivent utiliser une image 465,20 Mo (dimensions non renseignées) avec laquelle ils ont atteint une accélération de $\times$20,1 sur les 24 cœurs.

4.3.2 Travaux sur d'autres architectures

D'autres travaux traitent de la parallélisation de l'étiquetage pour des architectures spécialisées.

4.3.2.1 Connection Machine 5

Bader et Jaja [66] ont proposé une implémentation sur Connection Machine CM5, le découpage de l'image étant réalisé en tuiles avec un mécanisme pyramidal de gestion des unions. Dans l'article, les auteurs proposent un récapitulatif des algorithmes sur machines parallèles entre 1986 et 1994. La conversion en nombre de pixels traités par seconde des résultats de l'article indique $0{,}87 Mp/s$ pour le l'étiquetage en composantes connexes sur 32 processeurs élémentaires.

4.3.2.2 FPGA

Les FPGA sont très adaptés à la création d'architectures spécialisées. Les travaux de Bailey [68, 104] qui ciblent spécifiquement les FPGA et les caméras intelligentes approchent l'étiquetage par le biais des mécanismes spécifiques à ce genre de plateforme. Le but n'est pas améliorer de manière

statistique le nombre d'accès et de tests à effectuer, mais de minimiser le temps de traitement du pire cas afin de maximiser la fréquence de fonctionnement du processus global. Cela exclut les structures de longueur indéterminée manipulées par des algorithmes ayant des boucles de type `while()`. Si la structure Union-Find est bien utilisée pour maintenir les équivalences, cet algorithme utilise une table locale interne à chaque ligne (pile) afin d'éviter de recourir à l'implémentation classique et non déterministe de la fonction `FindRoot()`. L'étiquetage en composantes connexes qui nécessite un stockage complet de l'image des étiquettes afin de réaliser le réétiquetage est peu présent sur FPGA. Les algorithmes sont donc des algorithmes d'analyse en composantes connexes.

Les algorithmes de Bailey ont été optimisés et parallélisés par Klaiber*et al.* dans [69, 70]. Le débit obtenu est de $0{,}1Gp/s$ par canal. En utilisant une caméra spécialisée capable de fournir 32 flux parallèles correspondants aux différentes bandes de l'image, le débit maximal obtenu est de $3{,}2Gp/s$.

4.3.2.3 GPU

Des algorithmes ont été développés pour des GPU et seront étudiés dans le chapitre 6.

4.4 Parallel Light Speed Labeling : LSL adapté au multi-cœur

4.4.1 Principe général

La philosophie retenue pour PLSL[105, 106] est de réaliser un algorithme sans verrou sur les données avec un nombre limité de synchronisations tout en minimisant l'impact des modifications par rapport à l'algorithme séquentiel. Trois étapes sont nécessaires :

- Une étape de découpage en bandes qui génère et initialise les structures de données pour chaque bande. Si les images sont de taille constante, seule l'initialisation doit être répétée.
- L'étiquetage parallèle des bandes sur chaque cœur (parallélisme total).
- La fusion pyramidale des bandes (parallélisme partiel).

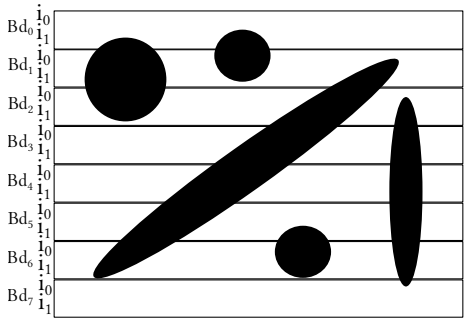
4.4.2 Un découpage en bandes

LSL est un algorithme fondamentalement pensé pour travailler sur des lignes et tirer parti de la longueur des segments internes aux lignes. Faire le choix de découper l'image binaire à étiqueter en bandes permet d'avoir un mécanisme d'union lui aussi pensé pour travailler sur des lignes. De plus, dans les structures internes de LSL, les lignes de fin et de début de bande ont déjà subi un étiquetage relatif et l'opération de fusion est donc très proche de l'opération classique d'étiquetage d'une ligne.

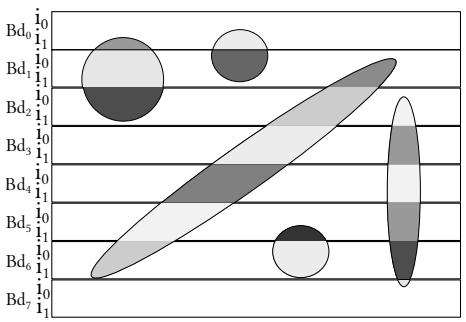
L'algorithme 20 présente la construction de la structure Bd qui décrit les bandes et le partitionnement de la table d'équivalences :

- $Bd.i_{0_k}$ qui représente l'index de début de la bande dans l'image,
- $Bd.i_{1_k}$ qui représente l'index de fin de la bande dans l'image,
- $Bd.e_{0_k}$ qui représente l'index de début de la bande dans la table d'équivalence,
- $Bd.e_{1_k}$ qui représente l'index de fin de la bande dans la table d'équivalence,
- $Bd.ne_k$ qui est le compteur d'étiquettes de la bande,

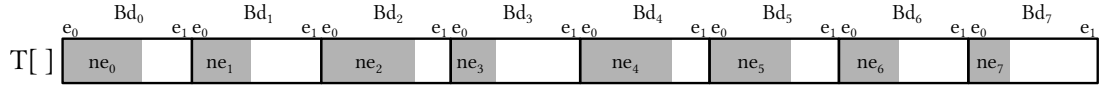
Le modèle de division des données retenu (fig. 4.12a) est le découpage de l'image $I[H][W]$ en autant de bandes de taille $I_k[h][W]$ (avec $h = H/P$) que de threads (P). Chaque bande sera étiquetée comme une image indépendante (fig. 4.12b) mais le graphe de connexité sera inscrit dans une table d'équivalences commune avec une plage d'étiquettes dédiée (fig. 4.12c). Dans chaque bande, le nombre maximal d'étiquettes est $h * W/4$ en 8C ($h * W/2$ en 4C), h étant la hauteur de la bande.



(a) Image binaire $I[H][W]$ découpée en 8 bandes



(b) Image des étiquettes $E[H][W]$ découpée en 8 bandes traitées en parallèle par 8 threads



(c) Table d'équivalences $T[H \times W/4]$ découpée en 8 bandes de taille h*W/4 correspondant à des plages d'étiquettes allant de $Bd.e_{0_k}$ à $Bd.e_{1_k}$. Chaque bande possède un compteur d'étiquettes indépendant $Bd.ne_k$

Fig. 4.12 – Découpage en bandes horizontales de l'image des étiquettes et structure de graphes correspondante

Algorithme 20 : DecoupageBandes

Input : H la hauteur totale de l'image à étiqueter, $P > 1$ le nombre de threads
Result : Bd_k, la structure de données décrivant chacune des bandes

1 $h \leftarrow E(H/(P))$ ▷ Avec E la partie entière
2 $z \leftarrow E(h \times W/4)$
3 **for** $k = 0$ **to** $P - 2$ **do**
4 **if** $k = 0$ **then**
5 $Bd_0.i_0 \leftarrow 0 \quad Bd_0.i_1 \leftarrow h$
   $Bd_0.e_0 \leftarrow 0 \quad Bd_0.e_1 \leftarrow z$
6 $Bd_0.ne \leftarrow 1$ ▷ 0 est l'étiquette du fond : le compteur commence en 1
7 **else**
8 $Bd_k.i_0 \leftarrow Bd_{k-1}.i_1 + 1 \quad Bd_k.i_1 \leftarrow Bd_k.i_0 + h - 1$
   $Bd_k.e_0 \leftarrow Bd_{k-1}.e_1 + 1 \quad Bd_k.e_1 \leftarrow Bd_k.e_0 + z - 1$
9 $Bd_k.ne \leftarrow Bd_k.e_0$
10 $Bd_{P-1}.i_0 \leftarrow Bd_{P-2}.i_1 + 1$
11 $Bd_{P-1}.i_{1_{P-1}} \leftarrow H - 1$ ▷ La dernière bande doit atteindre la fin de l'image
12 $h \leftarrow Bd_{P-1}.i_{1_{P-1}} - Bd_{P-1}.i_0$
13 $z \leftarrow E(h \times W/4)$
14 $Bd_{P-1}.e_0 \leftarrow Bd_{P-2}.e_1 + 1$
15 $Bd_{P-1}.e_1 \leftarrow Bd_{P-1}.e_0 + z$

4.4.3 Étiquetage d'une bande

Une fois le découpage réalisé, chaque bande est étiquetée par un numéro de thread. La spécificité par rapport à l'algorithme séquentiel est que chaque bande est constituée d'une première ligne qu'il faut traiter séparément. Cette procédure consiste à étiqueter la ligne en considérant que la ligne précédente ne contient aucun segment.

L'instruction `ParallelFor` indique qu'un mécanisme exécute l'étiquetage des P bandes simultanément.

Algorithme 21 : EtiquetageBandes

Input : I l'image binaire $H \times W$, Bd_k, Desc la structure qui contient les descripteurs
Result : E l'image étiquetée et T prêtes à être fusionnées par bande

```
1 ParallelFor k = 0 to P − 1 do
2     Bd_k.ne = EtiquetageLigneZero(I, E, T, W, Bd_k.i_0, Desc)
3     for i = Bd_k.i_0 + 1 to Bd_k.i_1 do
4         Bd_k.ne = EtiquetageLigneCourante(I, E, T, W, i, Bd_k.ne, Desc)
5     FermetureTransitive(T, Bd_k.e_0, Bd_k.ne)
```

4.4.4 Fusion pyramidale

Le principe de la fusion pyramidale a été présenté en section 4.2.3.5. Une adaptation est nécessaire pour les architectures dont le nombre de cœurs n'est pas une puissance de deux.

Algorithme 22 : FusionPyramidale

Input : P, Bd_k, E et T prêtes à être fusionnées par bande
Result : E l'image étiquetée

```
1  if Log_2(P) = E(log_2(P)) then
2      NbEtapes ← log_2(P)
3  else
4      NbEtapes ← log_2(P) + 1 ▷ si P n'est pas une puissance de 2
5  pas ← 1
6  debut ← 0
7  for lEtape = 1 to NbEtapes do
8      pas ← pas × 2
9      ParallelFor k = 0 to max(0, P/pas − 1) do
10         FusionBande(E, FinsBande[k × pas + debut])
11     debut ← pas − 1
```

4.4.5 FusionBande

Dans le cas de LSL, la fusion de bande est la même opération que l'étiquetage d'une ligne pour laquelle on a supprimé l'étape d'affectation globale pour la remplacer par la gestion des équivalences.

4.5 Implémentation de PLSL

4.5.1 Utilisation d'OpenMP et alternatives

L'implémentation d'un algorithme parallèle peut se faire à différents niveaux de granularité : en utilisant directement des pThreads [107], en utilisant une bibliothèque spécialisée (TBB [108]) ou des extensions de langages (OpenMP [103], Intel CilkPlus [109]). Notre choix s'est porté sur OpenMP car il est le plus répandu, il est pensé pour simplifier la parallélisation d'applications existantes et il correspond au besoin. Le découpage choisi et l'absence de verrous sur les données nous permettent d'utiliser OpenMP dans sa version 2. L'intérêt d'OpenMP est que la distribution des données et leur décomposition sont gérées automatiquement via les directives OpenMP. Il est de plus utilisable sur les processeurs X86 ainsi que le Xeon Phi et sur processeurs ARM et PowerPC. Les directives proposées par OpenMP permettent d'implémenter tous les mécanismes nécessaires à notre proposition de parallélisation des algorithmes directs. Dans l'exemple du code 4.1, la directive `#pragma omp parallel`

`for` indique que la boucle doit être répartie sur les différents threads. Le nombre de threads peut être fixé par la variable d'environnement `OMP_NUM_THREADS`.

```
1
2 #pragma omp parallel for
3 for(int k = 0; k < P; k++) {
4         ne = Bd[k].ne;
5         i0 = Bd[k].i0;
6         i1 = Bd[k].i1;
7         ne = BandLabeling(i0, i1, ne);
8     }
```

Code 4.1 – Etiquetage parallèle des bandes avec OpenMP

4.5.2 Descripteurs

Les descripteurs sont calculés de la même façon que pour l'algorithme séquentiel et nécessitent, lors de la fusion des étiquettes, de gérer les mises à jour en s'assurant de ne pas compter deux fois des pixels. Une fois encore, c'est le mécanisme pyramidal qui permet ceci car les ensembles d'étiquettes sont deux à deux disjoints.

4.6 Évaluation de la performance de PLSL

4.6.1 Un modèle unifié

Afin de pouvoir comparer la performance de PLSL par rapport aux algorithmes séquentiels mais aussi aux versions parallélisées de tous les algorithmes directs, nous avons adapté le modèle de parallélisation à tous les algorithmes directs. Pour cela, il a été nécessaire de :

- construire des bandes de taille paire pour les algorithmes HCS_2 et Grana,
- construire l'étiquetage de la première ligne sur deux lignes pour HCS_2 et Grana,
- écrire des algorithmes de fusion en accord avec chaque algorithme (avec ou sans DT, avec UF ou Suzuki).

Pour mieux évaluer l'impact de la gestion des équivalences de Suzuki, les algorithmes LSL ont été modifiés pour utiliser l'une ou l'autre des méthodes de gestion des équivalences.

Dans le chapitre suivant, le jeu d'algorithmes directs de référence sera donc :

- Rosenfeld : Le masque de Rosenfeld (5 pixels) avec la gestion des équivalences Union-Find (UF) amélioré avec l'arbre de décision (DT) et la compression de chemin (PC).
- Suzuki : Le masque de Rosenfeld avec la gestion des équivalences Suzuki amélioré avec l'arbre de décision (DT),
- Grana : Le masque bloc de Grana (20 pixels) avec un arbre de décision à 128 étages utilisant la gestion des équivalences de Suzuki,
- RCM : Le masque réduit de RCM (4 pixels) avec la gestion des équivalences Suzuki,
- HCS_2 UF DT ARemSP : Le masque HCS_2 avec la gestion des équivalences Union-Find (UF) amélioré avec l'arbre de décision (DT) et l'optimisation Rem+Splicing (ARemSP),
- HCS : algorithme à machine d'états hybride pixel/segment à masque variable utilisant la gestion des équivalences Suzuki,
- $LSL_{\text{RLE-Rosenfeld}}$: la version de LSL conçue pour tirer parti de segments les plus longs par une utilisation intensive du codage RLC basée sur Union-Find,

- $\text{LSL}_{\text{RLE-Suzuki}}$: la version de LSL conçue pour tirer parti de segments les plus longs par une utilisation intensive du codage RLC, basée sur la gestion des équivalences de Suzuki,
- $\text{LSL}_{\text{STD-Rosenfeld}}$: la version de LSL conçue avec l'objectif d'être le plus systématique possible basée sur Union-Find,
- $\text{LSL}_{\text{STD-Suzuki}}$: la version de LSL conçue avec l'objectif d'être le plus systématique possible, basée sur la gestion des équivalences de Suzuki.

4.6.2 Métriques

Trois métriques sont utilisées pour évaluer la performance des algorithmes ainsi parallélisés.

- Le *cpp*, qui nous donnera la performance brute. Tout comme dans le chapitre 3, la moyenne des *cpp* sur les densités allant de 0% à 100% pour une granularité donnée sera nommée cpp_d et la moyenne des cpp_d sur les granularités allant de 1 à 16 sera nommée cpp_g.
- L'accélération A qui sera le rapport entre le *cpp* de la version séquentielle et celui de la version parallélisée.
- La portion de code séquentiel τ.

La portion de code séquentiel, sera calculée à partir de l'accélération (eq. 4.2) en nous basant sur la loi d'Amdahl [110](eq. 4.1).

$$sp = \frac{1}{\tau + \frac{1-\tau}{\pi}} \tag{4.1}$$

$$\tau = \frac{\pi - sp}{sp \times (\pi - 1)} \tag{4.2}$$

avec :

- τ, la portion de code séquentiel (non parallélisé),
- π, le parallélisme d'exécution qui est P (le nombre de cœurs) dans notre cas,
- sp (speedup), l'accélération entre le programme séquentiel et le programme parallèle

4.7 Conclusion

Les deux contributions présentées dans ce chapitre, PLSL et l'infrastructure de parallélisation pour les algorithmes directs, ont été conçues pour éviter les conflits dus aux accès concurrents aux données (images et table d'équivalence) tout en maximisant l'utilisation de chaque cœur. Afin d'évaluer la pertinence de ces propositions selon différents degrés de parallélisme, nous les avons soumises à la procédure de test décrite au chapitre 2 sur plusieurs architectures. Le chapitre 5, présentera les résultats obtenus.

Chapitre 5

Nous n'étions pas forcément plus athlétiques qu'eux, mais nous étions un bloc, avec à chaque poste les meilleurs ou peu s'en fallait, en tout cas mentalement les plus solides, ...

–La Horde du contrevent, Alain Damasio

Performance des algorithmes parallèles d'analyse en composantes connexes sur architectures multi-cœur

5.1 Introduction

La parallélisation des algorithmes directs proposée dans le chapitre précédent est maintenant évaluée du point de vue de ses performances réelles. Dans ce chapitre, nous mesurerons ces performances sur trois classes de machines (tab. 7.1) : machine de bureau, station de travail et serveur de calculs. Les algorithmes n'utilisant pas les instructions SIMD, le seul parallélisme (π) considéré sera le nombre de cœurs disponibles.

Alias	Famille	Identifiant	Nombre de processeurs	Fréquence (GHz)	Mémoire Cache	Bande passante mémoire maximale	Parallélisme π
$SKL_{1\times4}$	Skylake	i7-6700K	1	4,0	8Mo L3	34,1 Go/s	$1 \times 4 = 4$
$IVB_{2\times12}$	Ivy Bridge	E5-2695 v2	2	2,4	2×30Mo L3	$2 \times 59,7$Go/s	$2 \times 12 = 24$
$IVB_{4\times15}$	Ivy Bridge	E7-8890 v2	4	2,8	$4 \times 37,5$Mo L3	$4 \times 85,0$Go/s	$4 \times 15 = 60$

Table 5.1 – Machines de mesure des performances des algorithmes parallèles

Ces trois classes de machines correspondent à des usages différents. Chaque machine est testée pour des images de tailles 2048×2048, 4096×4096 et 8192×8192, afin de mettre en évidence l'influence du nombre de cœurs mais aussi de la quantité de données à traiter. La base SIDBA4 est utilisée pour évaluer l'impact de la parallélisation pour des images structurées.

5.2 Machine de bureau - 4 cœurs

5.2.1 Résultats pour les images aléatoires

5.2.1.1 Images 2048 × 2048

Pour la machine $SKL_{1\times4}$ ($\pi = 4$), le nombre d'étages de l'étape de fusion des bords est de 2. Le comportement des algorithmes parallèles sur les 4 cœurs pour des images 2048 × 2048 (fig. 5.1) est similaire à celui des algorithmes séquentiels sur un seul cœur pour des images 1024×1024 (cf. chap. 3) accéléré d'un rapport proche de 4.

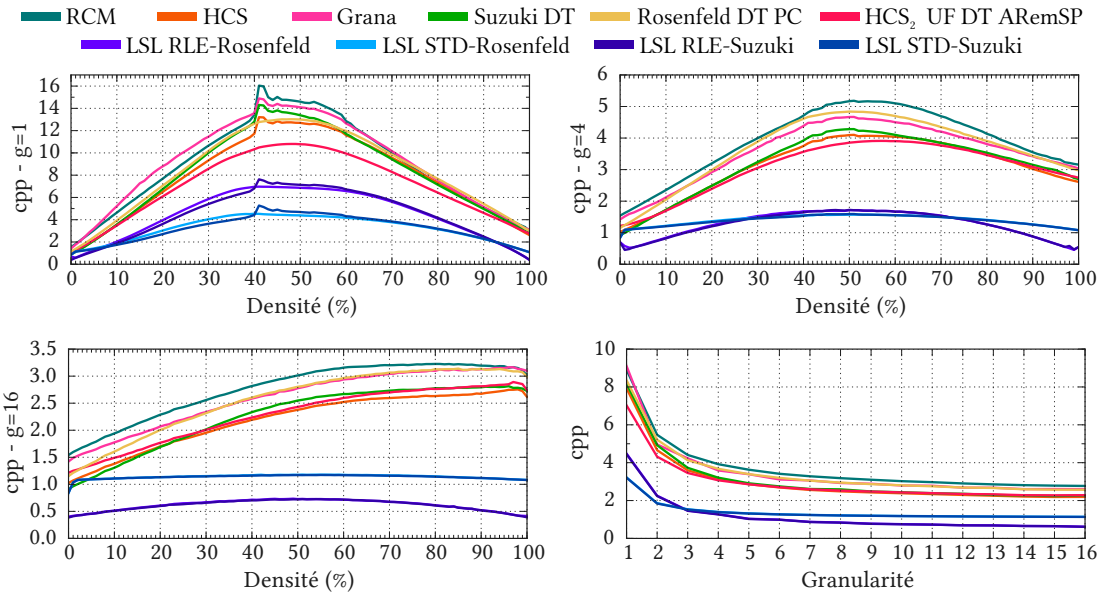


Fig. 5.1 – Parallélisation multi-cœur : *cpp* pour des images de taille 2048 × 2048 et de granularité $g \in \{1, 4, 16\}$ et cpp_d en fonction de la granularité sur les 4 cœurs de la machine $SKL_{1\times4}$

	Granularité					
Algorithmes	g=1	g=2	g=4	g=8	g=16	cpp_g
$LSL_{RLE\text{-}Rosenfeld}$	4,48	2,26	1,28	0,84	0,62	1,17
$LSL_{STD\text{-}Rosenfeld}$	3,22	1,86	1,40	1,22	1,14	1,40
$LSL_{RLE\text{-}Suzuki}$	4,45	2,24	1,27	0,83	0,61	1,17
$LSL_{STD\text{-}Suzuki}$	3,21	1,84	1,39	1,21	1,14	1,39
HCS_2 UF DT ARemSP	7,03	4,32	3,07	2,53	2,28	2,96
HCS	7,93	4,66	3,14	2,48	2,19	3,01
Suzuki DT	8,17	4,92	3,21	2,59	2,27	3,10
Rosenfeld DT PC	8,35	5,21	3,68	2,95	2,57	3,47
Grana	9,11	4,96	3,59	2,93	2,60	3,50
RCM	8,90	5,47	3,92	3,18	2,77	3,71

TABLE 5.2 – Parallélisation multi-cœur : cpp_d pour les granularités $g \in \{1, 2, 4, 8, 16\}$ et cpp_g pour des images 2048×2048 sur la machine $SKL_{1\times4}$

Un point remarquable est que les algorithmes basés sur la gestion des équivalences de Suzuki sont ralentis pour les densités proches de 41% pour g=1. Ce phénomène disparaît pour les images de granularité $g > 4$. L'implication directe de la gestion des équivalences de Suzuki est mise en évidence par la comparaison des algorithmes LSL dans leurs versions Rosenfeld ou Suzuki. Pour g=1, si pour

les densités d inférieures à 40% la version Suzuki est plus rapide que la version Rosenfeld, ses performances sont dégradées pour 40% < d < 60% et équivalentes pour d > 60%. La zone de dégradation correspond aux densités génératrices du plus grand nombre d'étiquettes supplémentaires. Pour les granularités $g > 1$, les deux versions (Rosenfeld ou Suzuki) sont équivalentes.

La table 5.2 présente le cpp_d (moyenne des cpp sur les densités allant de 0% à 100% pour une granularité donnée) pour les granularités $g \in \{1, 2, 4, 8, 16\}$ ainsi que cpp_g (moyenne des cpp_d sur les granularités allant de 1 à 16 - cf. 4.6.2). Le LSL_{RLE} est le plus rapide quelle que soit la version de l'algorithme de gestion des équivalences, Le LSL_{STD} est le deuxième, suivi par les algorithmes HCS_2 UF DT ARemSP, HCS, Suzuki DT, Rosenfeld DT PC, Grana et RCM.

5.2.1.2 Impact de la quantité de données

Afin d'évaluer l'impact de la quantité de données, des mesures équivalentes ont été menées pour des images de taille 4096 × 4096 et 8192 × 8192. La table 5.3 récapitule les cpp_g, l'accélération correspondante (comparées aux résultats obtenus avec l'algorithme séquentiel sur des images de même taille), ainsi que τ la portion de code séquentiel calculée *via* la loi d'Amdahl (sec. 4.6.2).

	cpp_g			sp			τ en %		
Algorithmes	2048	4096	8192	2048	4096	8192	2048	4096	8192
$\text{LSL}_{\text{RLE-Rosenfeld}}$	1,17	1,18	1,20	×3,75	×3,77	×3,76	2,20	2,04	2,12
$\text{LSL}_{\text{STD-Rosenfeld}}$	1,40	1,42	1,44	×4,00	×3,94	×3,89	0,04	0,52	0,95
$\text{LSL}_{\text{RLE-Suzuki}}$	1,17	1,17	1,19	×3,98	×3,99	×3,97	0,15	0,10	0,24
$\text{LSL}_{\text{STD-Suzuki}}$	1,39	1,41	1,43	×4,03	×3,96	×3,90	-0,25 [1]	0,30	0,87
HCS_2 UF DT ARemSP	2,96	2,95	2,94	×3,85	×3,88	×3,88	1,26	1,02	0,99
HCS	3,01	3,02	3,02	×3,93	×3,95	×3,96	0,58	0,44	0,38
Suzuki DT	3,10	3,12	3,14	×3,88	×3,90	×3,90	1,02	0,84	0,84
Rosenfeld DT PC	3,47	3,47	3,47	×3,76	×3,78	×3,78	2,11	1,97	1,91
Grana	3,50	3,48	3,51	×3,85	×3,89	×3,85	1,27	0,92	1,32
RCM	3,71	3,71	3,71	×3,84	×3,85	×3,86	1,41	1,26	1,19

TABLE 5.3 – Parallélisation multi-cœur : cpp_g, accélération moyenne sur les granularités de 1 à 16 et τ la portion de code séquentiel pour des tailles d'images 2048×2048, 4096×4096, 8192×8192 sur la machine $\text{SKL}_{1\times4}$

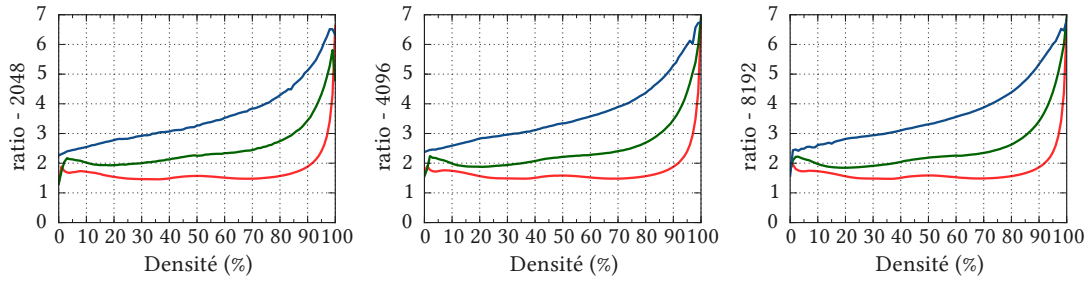


Fig. 5.2 – Analyse en composantes connexes : ratio entre le cpp de LSL_{RLE} et le minimum des cpp des algorithmes pixels sur la machine $\text{SKL}_{1\times4}$ pour les granularités g=1 (rouge), g=4 (vert) et g=16 (bleu)

Le cpp_g évolue très peu avec la taille des données, et le rang des algorithmes est donc conservé. L'accélération est proche de 4, le cas idéal sur cette architecture (tab. 7.1). Les versions Suzuki de LSL ayant des performances séquentielles en retrait par rapport aux versions Rosenfeld, l'accélération obtenue est meilleure pour les versions Suzuki alors que leur cpp est très proche pour les versions parallèles. Pour les images 2048 × 2048, la version $\text{LSL}_{\text{STD-Suzuki}}$ a même une accélération supérieure

1. Le τ négatif calculé correspond à l'accélération surlinéaire de $\text{LSL}_{\text{STD-Suzuki}}$ pour cette taille d'image

à 4 que l'on peut attribuer à la simplification de l'arbre de connexité par le découpage de l'image. En effet, chaque bande possède son sous-graphe de connexité dont la hauteur maximale est limité par la taille de la bande et qui est dans le cas présent le quart de la taille de l'image.

Le ratio entre le *cpp* minimum de l'ensemble des algorithmes pixels et le *cpp* de $\text{LSL}_{\text{RLE-Rosenfeld}}$ (fig. 5.2) est peu dépendant de la taille des images. Il confirme que $\text{LSL}_{\text{RLE-Rosenfeld}}$ profite plus de l'augmentation de la granularité que les algorithmes pixels et qu'il est plus rapide pour toutes les densités dès g=1.

5.2.2 Résultats pour les images de SIDBA4

Pour l'évaluation des algorithmes parallèles, nous avons utilisé les images de la base SIDBA4 qui représente plus fidèlement les résolutions usuelles modernes que la base SIDBA. En effet, c'est l'évolution de la taille des images qui pousse à l'utilisation du multi-cœur pour conserver des performances compatibles avec les applications temps réel.

La table 5.4 et la figure 5.3 illustrent les performances obtenues. Les algorithmes LSL_{RLE} sont les plus rapides d'un rapport ×4,7 par rapport au plus rapide des algorithmes pixels (HCS). Les algorithmes LSL_{STD} sont les plus stables.

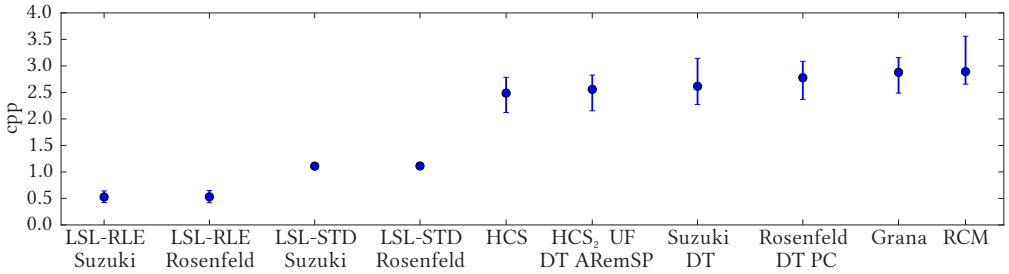


Fig. 5.3 – Parallélisation multi-cœur : *cpp* moyen et variabilité (cpp_{max} et cpp_{min}) sur la base de données SIDBA4 sur les 4 cœurs de la machine $\text{SKL}_{1\times4}$

Algorithmes	*cpp*	*sp*	τ	ratio
$\text{LSL}_{\text{RLE-Suzuki}}$	0,53	×3,68	2,95	×1.0
$\text{LSL}_{\text{RLE-Rosenfeld}}$	0,53	×3,52	4,59	×1.0
$\text{LSL}_{\text{STD-Suzuki}}$	1,11	×3,99	0,11	×2.1
$\text{LSL}_{\text{STD-Rosenfeld}}$	1,11	×3,98	0,17	×2.1
HCS	2,49	×3,37	6,19	×4.7
HCS_2 UF DT ARemSP	2,56	×3,36	6,30	×4.8
Suzuki DT	2,62	×3,36	6,34	×4.9
Rosenfeld DT PC	2,78	×3,39	6,02	×5.2
Grana	2,88	×3,43	5,56	×5.4
RCM	2,89	×3,63	3,43	×5.5

TABLE 5.4 – Parallélisation multi-cœur : *cpp* moyen, *sp* l'accélération par rapport à la version séquentielle, τ la portion de code séquentiel mesuré pour la base SIDBA4 et le ratio entre le *cpp* de l'algorithme et celui de $\text{LSL}_{\text{RLE-Rosenfeld}}$ sur la machine $\text{SKL}_{1\times4}$

À l'exception des LSL_{STD} qui atteignent une efficacité supérieure à 99%, l'accélération obtenue n'est pas aussi bonne que pour les images aléatoires. La portion de code séquentiel est donc elle aussi en augmentation et est comprise entre 2,95% et 6,34%. Une des explications est le caractère non homogène des images. En effet, par construction, les images aléatoires sont homogènes en granularité et en densité. Les images de la base peuvent présenter de grandes variations de ces paramètres dans

chaque bande. Si une bande est plus complexe à étiqueter que les autres, elle ralentit le processus global. La très bonne performance des algorithmes LSL_{STD} est due à leur régularité quelle que soit la densité et à la granularité qui les rendent moins sensibles au déséquilibre de charge entre les threads.

5.2.3 Parts des étapes intermédiaires

5.2.3.1 Images aléatoires

L'analyse des différentes parties des algorithmes : première passe, calcul des descripteurs, gestion des frontières, est représentée dans les figures 5.4, 5.5 et 5.6. La première passe et le calcul des descripteurs sont très proches de la version séquentielle accélérée d'un rapport 4. La gestion des frontières est responsable de la dégradation des performances dans la zone $40\% < d < 60\%$ qui a été observée pour g=1 pour les algorithmes utilisant la gestion des équivalences de Suzuki. L'algorithme RCM est le plus affecté (le *cpp* des fusions aux frontières représente 18% du *cpp* total) du fait d'un nombre plus important d'étiquettes supplémentaires que les autres algorithmes. Pour $g \geq 4$, le *cpp* des fusions aux frontières n'est plus perceptible pour les algorithmes pixels, et le *cpp* des descripteurs représente la majorité du *cpp* total. Alors que pour les algorithmes LSL, comme dans leur version séquentielle, la première passe représente la majorité du *cpp* total.

5.2.3.2 Images de la base SIDBA4

L'analyse des étapes pour les images de la base SIDBA4 (tab. 5.5) confirme trois points :

- pour les algorithmes pixels, le *cpp* des descripteurs est prédominant sur celui de la première passe là où ce calcul est dans tous les cas inférieur à 10% de celui de la première passe pour les algorithmes LSL.
- la gestion des frontières est négligeable devant les deux autres parties.
- l'accélération est comprise entre ×3,00 dans le pire cas (HCS_2 UF DT ARemSP) et ×4,04 $LSL_{STD\text{-}Suzuki}$.

	Première passe			Descripteurs			Frontières			*sp*		
Algorithmes	min	moy	max	min	moy	max	min	moy	max	min	moy	max
$LSL_{RLE\text{-}Suzuki}$	0,41	0,50	0,60	0,01	0,03	0,04	0,001	0,002	0,003	3,28	3,68	3,89
$LSL_{RLE\text{-}Rosenfeld}$	0,41	0,50	0,60	0,01	0,03	0,05	0,001	0,002	0,004	3,12	3,52	3,82
$LSL_{STD\text{-}Suzuki}$	1,08	1,09	1,11	0,00	0,02	0,04	0,001	0,002	0,003	3,91	3,99	4,04
$LSL_{STD\text{-}Rosenfeld}$	1,07	1,09	1,11	0,01	0,02	0,03	0,001	0,002	0,003	3,88	3,98	4,03
HCS	0,74	0,77	0,81	1,34	1,71	1,96	0,004	0,007	0,012	3,04	3,37	3,83
HCS_2 UF DT ARemSP	0,78	0,81	0,85	1,34	1,74	1,96	0,010	0,012	0,014	3,00	3,36	3,84
Suzuki DT	0,74	0,77	0,81	1,50	1,84	2,32	0,004	0,007	0,012	3,07	3,36	3,98
Rosenfeld DT PC	0,93	0,99	1,11	1,41	1,78	1,97	0,009	0,012	0,014	3,03	3,39	3,85
Grana	1,20	1,26	1,33	1,23	1,62	1,82	0,005	0,007	0,012	3,11	3,43	3,85
RCM	0,93	1,10	1,41	1,40	1,78	2,46	0,005	0,007	0,011	3,33	3,63	3,96

TABLE 5.5 – Parallélisation multi-cœur : *cpp* de la première passe, *cpp* du calcul des descripteurs, *cpp* des frontières et accélération globale par rapport à la version séquentielle sur la base de données SIDBA4 sur la machine $SKL_{1\times4}$

5.2.4 Conclusion pour la machine de bureau

Sur une machine quatre cœurs de dernière génération, les algorithmes parallèles LSL sont les plus rapides (d'un facteur ×4,7 sur SIDBA4, ×1,7 pour g=1, ×2,5 pour g=4 et ×3,6 pour g=16) et ceux qui

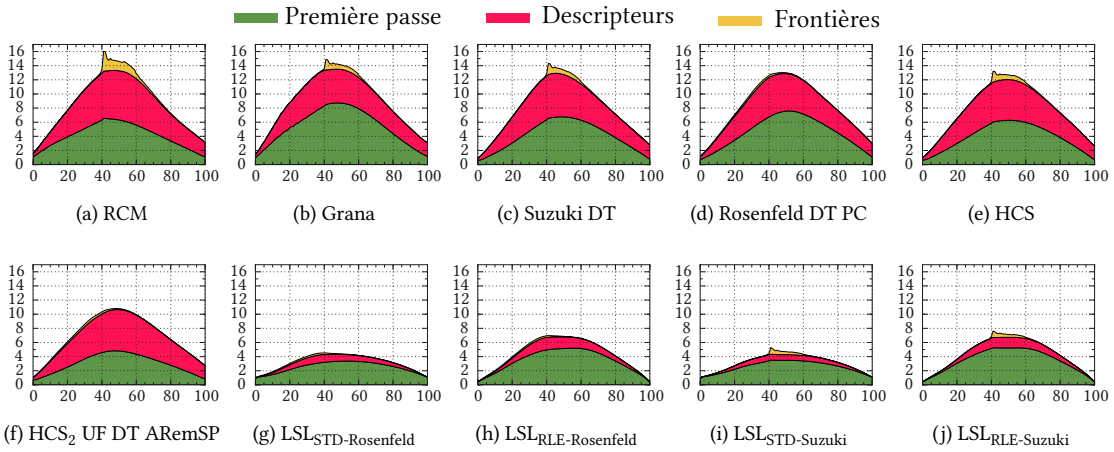


Fig. 5.4 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 1$ sur les 4 cœurs de la machine SKL$_{1\times4}$

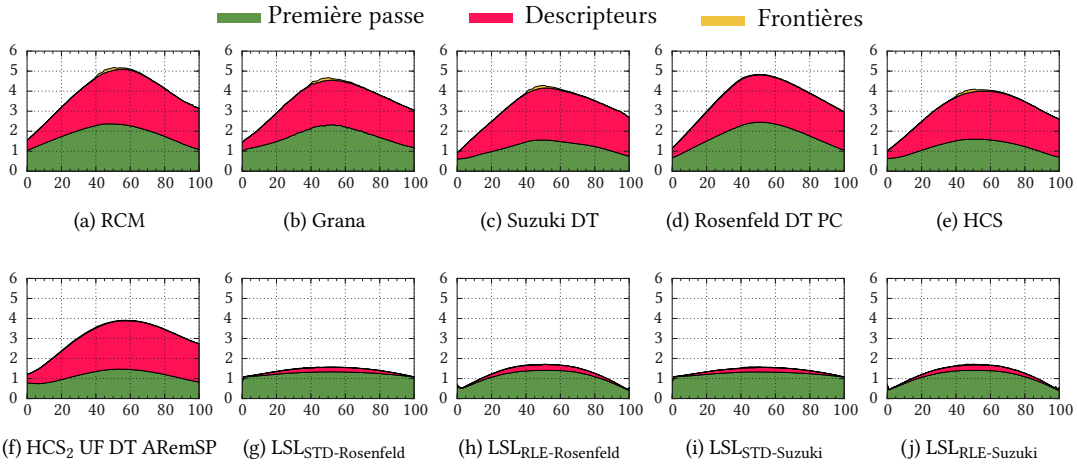


Fig. 5.5 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 4$ sur les 4 cœurs de la machine SKL$_{1\times4}$

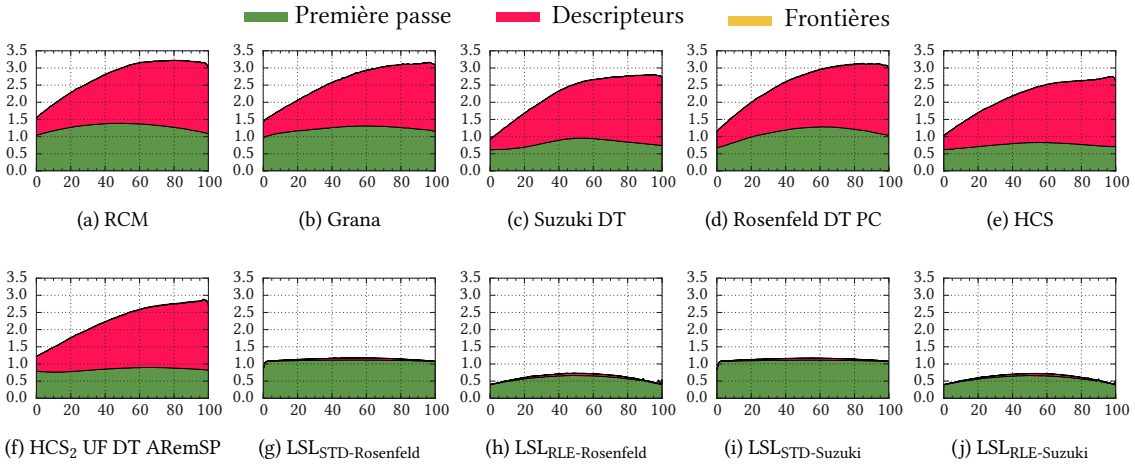


Fig. 5.6 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 16$ sur les 4 cœurs de la machine SKL$_{1\times4}$

présentent le moins de variations dans le temps de traitement. Sur ce type de machine et quelle que soit la taille de données, l'efficacité de la méthode de parallélisation est comprise entre 94,2% et 100,7% pour les images aléatoires et 75,0% et 100,7% pour les images de la base SIDBA4.

Dans le cadre d'une application de traitement d'images sur ce type de machine, la parallélisation apporte un gain important et la portion de code séquentiel est faible. L'augmentation du nombre d'étapes de fusion avec le nombre de cœurs peut dégrader cette performance. C'est l'objet des sections suivantes.

5.3 Station de travail - 2×12 cœurs

5.3.1 Résultats pour les images aléatoires

5.3.1.1 Images 2048 × 2048

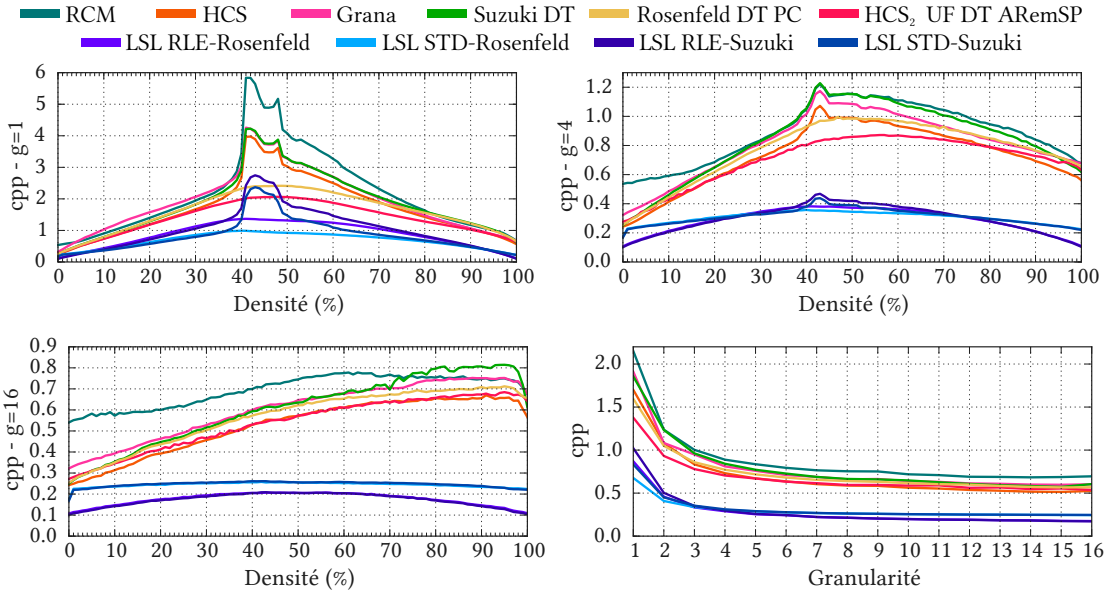


Fig. 5.7 – Parallélisation multi-cœur : *cpp* pour des images de taille 2048 × 2048 et de granularité $g \in \{1, 4, 16\}$ et *cpp* moyen en fonction de la granularité sur les 24 cœurs de la machine $\text{IVB}_{2\times12}$

Pour la machine $\text{IVB}_{2\times12}$ (π = 24), le nombre d'étages de l'étape de fusion des bords passe de 2 à 5. Le phénomène de dégradation des performances pour les algorithmes basés sur la gestion des équivalences de Suzuki s'amplifie et reste visible pour la granularité g=4 (fig. 5.7). Les algorithmes LSL dans les versions Rosenfeld sont plus rapides que ceux utilisant la gestion des équivalences de Suzuki. Pour cette taille d'image, les algorithmes LSL_{RLE} et les algorithmes LSL_{STD} sont en moyenne (cpp_g - tab. 5.6) très proches. Avec l'augmentation de g, LSL_{RLE} continue de progresser et est ×1,4 plus rapide que LSL_{STD} pour g=16.

5.3.1.2 Impact de la quantité de données

Sur la station de travail, l'effet de l'évolution de la quantité de données est très significatif de l'importance des échanges avec la mémoire dans l'étiquetage.

Pour LSL_{RLE}, que ce soit dans sa version Rosenfeld ou Suzuki, cpp_g diminue (×0,96) en passant d'une image 2048×2048 à une image 8192×8192. Tandis que tous les autres algorithmes voient leur

	Granularité					
Algorithmes	g=1	g=2	g=4	g=8	g=16	cpp_g
$LSL_{RLE\text{-}Rosenfeld}$	0,87	0,46	0,29	0,21	0,18	0,27
$LSL_{RLE\text{-}Suzuki}$	1,02	0,50	0,30	0,21	0,17	0,29
$LSL_{STD\text{-}Rosenfeld}$	0,68	0,41	0,30	0,26	0,24	0,30
$LSL_{STD\text{-}Suzuki}$	0,83	0,45	0,31	0,26	0,25	0,32
HCS_2 UF DT ARemSP	1,38	0,93	0,70	0,59	0,54	0,68
HCS	1,70	1,08	0,73	0,59	0,53	0,70
Rosenfeld DT PC	1,60	1,05	0,77	0,64	0,57	0,73
Grana	1,91	1,08	0,81	0,66	0,60	0,78
Suzuki DT	1,86	1,23	0,84	0,66	0,60	0,79
RCM	2,15	1,24	0,89	0,75	0,70	0,88

TABLE 5.6 – Parallélisation multi-cœur : cpp_d pour les granularités $g \in \{1, 2, 4, 8, 16\}$ et cpp_g sur la machine $IVB_{2\times12}$ pour des images 2048×2048

performance chuter avec une augmentation de cpp_g comprise entre ×1,56 et ×3,44.

L'accélération de l'algorithme LSL_{RLE} progresse avec la taille des images et atteint ×23,58 et ×20,56 respectivement pour $LSL_{RLE\text{-}Rosenfeld}$ et $LSL_{RLE\text{-}Suzuki}$. Pour les autres algorithmes, l'accélération chute avec la taille de l'image et est inférieure à ×12,03 pour les images 8192×8192.

	cpp_g			sp			τ en %		
Algorithmes	2048	4096	8192	2048	4096	8192	2048	4096	8192
$LSL_{RLE\text{-}Rosenfeld}$	0,27	0,25	0,26	×20,9	×23,1	×23,6	0,64	0,16	0,08
$LSL_{RLE\text{-}Suzuki}$	0,29	0,27	0,27	×18,1	×20,0	×20,6	1,41	0,87	0,73
$LSL_{STD\text{-}Rosenfeld}$	0,30	0,87	1,01	×21,2	×7,7	×6,3	0,58	9,12	12,18
$LSL_{STD\text{-}Suzuki}$	0,32	0,89	1,03	×20,6	×7,2	×6,2	0,72	10,40	12,55
HCS_2 UF DT ARemSP	0,68	0,93	1,28	×19,3	×14,1	×10,3	1,05	3,06	5,75
HCS	0,70	0,94	1,33	×20,1	×15,1	×10,3	0,84	2,54	5,77
Rosenfeld DT PC	0,73	0,94	1,32	×20,9	×16,1	×11,4	0,64	2,13	4,81
Suzuki DT	0,79	0,98	1,35	×19,7	×15,6	×11,2	0,94	2,34	4,95
RCM	0,88	1,12	1,38	×19,4	×15,0	×12,0	1,03	2,58	4,33
Grana	0,78	1,30	1,87	×20,2	×12,1	×8,7	0,81	4,24	7,68

TABLE 5.7 – Parallélisation multi-cœur : cpp moyen sur les granularités de 1 à 16, sp l'accélération moyenne sur les granularités de 1 à 16 et τ la portion de code séquentiel pour des tailles d'image 2048×2048, 4096×4096, 8192×8192 sur la machine $IVB_{2\times12}$

La figure 5.8 illustre ce phénomène. Dès g=5, les algorithmes pixels et les algorithmes LSL_{STD} atteignent leurs limites respectives. Pour les images de taille 8192×8192, Grana est le plus lent avec 1,8cpp, puis tous les autres algorithmes pixels se regroupent et stagnent à 1,25cpp suivis par les algorithmes LSL_{STD} avec 1,0cpp. Dans le même temps, les courbes des LSL_{RLE} sont identiques quelle que soit la taille des images. Le nombre d'accès à la mémoire (mimimal dans le cas de LSL_{RLE}) nécessaire à un algorithme est donc prépondérant sur toutes les autres variations. Pour les images de taille supérieure à 4096×4096, quelle que soit la granularité, le LSL_{RLE} est l'algorithme d'analyse en composantes connexes le plus rapide.

Le ratio entre le cpp minimum de l'ensemble des algorithmes pixels et le cpp de $LSL_{RLE\text{-}Rosenfeld}$ (fig. 5.9) est très sensible à la taille des images. Pour les images de taille 4096×4096 et 8192×8192, le cpp des algorithmes pixels est limité par la bande passante mémoire et le ratio augmente fortement atteignant en moyenne ×8,5 pour g =16 sur les images 8192×8192.

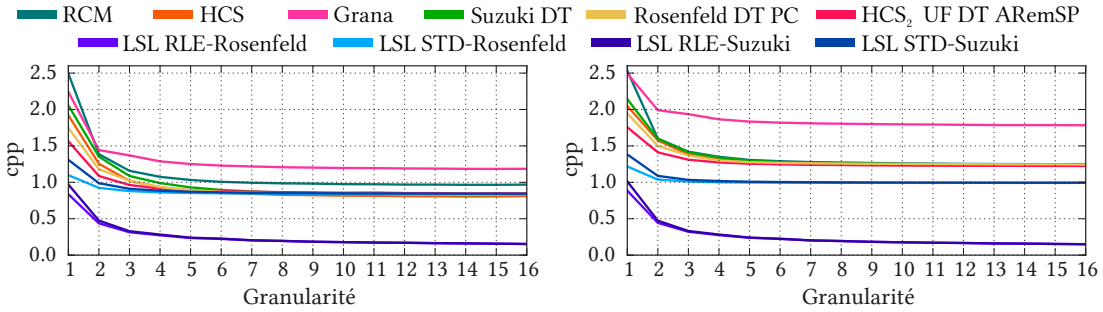


Fig. 5.8 – Parallélisation multi-cœur : cpp_d en fonction de la granularité sur les 24 cœurs de la machine $IVB_{2\times12}$ pour des images de taille 4096×4096 (gauche) et 8192×8192 (droite)

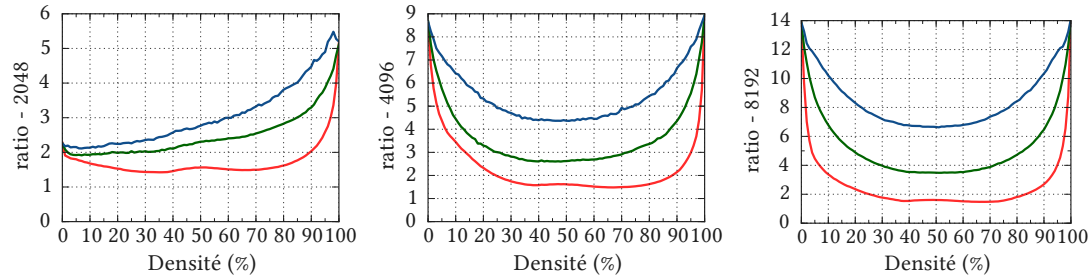


Fig. 5.9 – Analyse en composantes connexes : ratio entre le *cpp* de LSL_{RLE} et le minimum des *cpp* des algorithmes pixels sur la machine $IVB_{2\times12}$ pour les granularités g=1 (rouge), g=4 (vert) et g=16 (bleu)

5.3.2 Résultats pour les images de SIDBA4

Comme l'indiquent la figure 5.10 et la table 5.8, l'algorithme LSL_{RLE} est le plus rapide, d'un facteur ×3,7 pour HCS et ×4,7 pour RCM. L'écart s'est réduit par rapport à la machine de bureau. LSL_{STD} est le plus stable en temps de traitement, mais il est toujours plus lent que le LSL_{RLE}. Sur les images de la base, il n'y a pas de différence entre les versions Rosenfeld et Suzuki des algorithmes LSL.

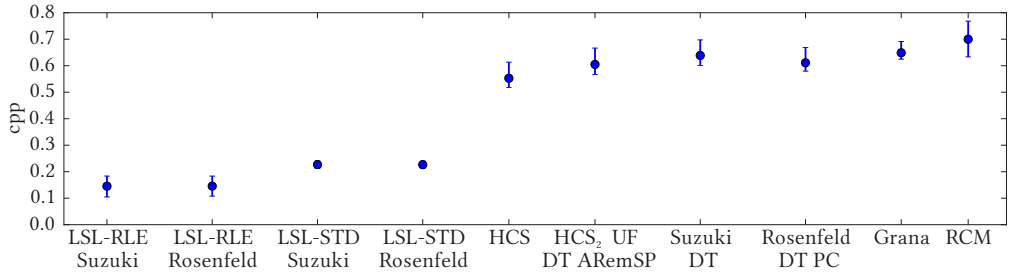


Fig. 5.10 – Parallélisation multi-cœur : *cpp* moyen et variabilité (cpp_{max} et cpp_{min}) sur la base de données SIDBA4 sur les 24 cœurs de la machine $IVB_{2\times12}$

L'accélération entre les versions séquentielles et les versions parallèles, est comprise entre ×14,03 (3,09% de code séquentiel) pour LSL_{RLE} et ×22,82 (0,23% de code séquentiel) pour LSL_{STD}. Les algorithmes pixels ont une accélération moyenne comprise entre ×16,75 (1,88% de code séquentiel) et ×18,40 (1,32% de code séquentiel). La portion de code séquentiel mesurée est plus faible que pour les 4 cœurs de la machine $SKL_{1\times4}$.

Algorithmes	*cpp*	*sp*	τ	ratio
$LSL_{RLE\text{-}Suzuki}$	0,15	×15,13	2,55	×1.0
$LSL_{RLE\text{-}Rosenfeld}$	0,15	×14,03	3,09	×1.0
$LSL_{STD\text{-}Suzuki}$	0,23	×22,82	0,23	×1.5
$LSL_{STD\text{-}Rosenfeld}$	0,23	×22,61	0,27	×1.5
HCS	0,55	×17,34	1,67	×3.7
HCS_2 UF DT ARemSP	0,60	×16,75	1,88	×4.0
Suzuki DT	0,61	×17,51	1,61	×4.1
Rosenfeld DT PC	0,64	×18,37	1,33	×4.3
Grana	0,65	×18,40	1,32	×4.3
RCM	0,70	×18,39	1,33	×4.7

TABLE 5.8 – Parallélisation multi-cœur : *cpp*, *sp* l'accélération par rapport à la version séquentielle, τ la portion de code séquentiel mesuré sur la base de données SIDBA4 et le ratio entre le *cpp* de l'algorithme et celui de $LSL_{RLE\text{-}Rosenfeld}$ sur les 24 cœurs de la machine $IVB_{2\times12}$

5.3.3 Parts des étapes intermédiaires

5.3.3.1 Images aléatoires 2048 × 2048

L'analyse des différentes parties des algorithmes : première passe, descripteurs, frontières, est représentée dans les figures 5.11, 5.12 et 5.13 pour les images de taille 2048×2048.

Pour d=41%, le *cpp* de la fusion des frontières des algorithmes basés sur la gestion des étiquettes de Suzuki est très élevée. L'algorithme RCM est encore une fois le plus affecté et le *cpp* des fusions aux frontières représente 60% du *cpp* total. Pour $g \geq 4$, il représente toujours plus de 20% du *cpp* total. Pour $g \geq 16$, il représente en moyenne de 7% du *cpp* total.

Pour les algorithmes RCM et LSL_{STD}, la forme de la première passe s'aplatit révélant un impact très faible du nombre d'étiquettes supplémentaires et donc une indépendance de l'algorithme vis-à-vis des données.

5.3.3.2 Images aléatoires : impact de la quantité de données

Pour simplifier la lecture du document, seuls RCM et les algorithmes LSL ont été représentés dans les figures (fig. 5.12 et 5.13). Les courbes pour tous les algorithmes ont cependant été reportées en annexes (fig. A.8, A.9 et A.10).

Pour les images de taille 4096×4096 et les algorithmes hors LSL_{RLE}, à partir de g=4 (fig. 5.12) la part de la première passe tend à devenir constante et la part du *cpp* des descripteurs diminue. Le *cpp* du calcul des descripteurs étant calculé par différence entre le *cpp* de la première passe avec et sans calcul des descripteurs, cela met en évidence que le *cpp* de la première passe converge vers une valeur qui ne dépend pas du calcul mais bien des échanges avec la mémoire. De plus, le *cpp* est constant quelle que soit la densité et illustre en fait que les algorithmes sont devenus indépendants du nombre d'étiquettes supplémentaires et donc des données.

Pour les images de taille 8192×8192, l'écart s'amplifie entre les algorithmes LSL_{RLE} et les autres. Du fait de l'usage de la compression RLC à toutes les étapes de l'algorithme LSL_{RLE}, la quantité de données à échanger entre les cœurs est minimale.

5.3.3.3 SIDBA4

Par rapport à $SKL_{1\times4}$, le point notable est l'augmentation d'un facteur ×4 du temps de fusion des frontières. C'est ce qui explique l'écart d'accélération entre les algorithmes. La première passe et le calcul des descripteurs étant très rapides pour les algorithmes LSL_{RLE}, l'impact de l'augmentation du

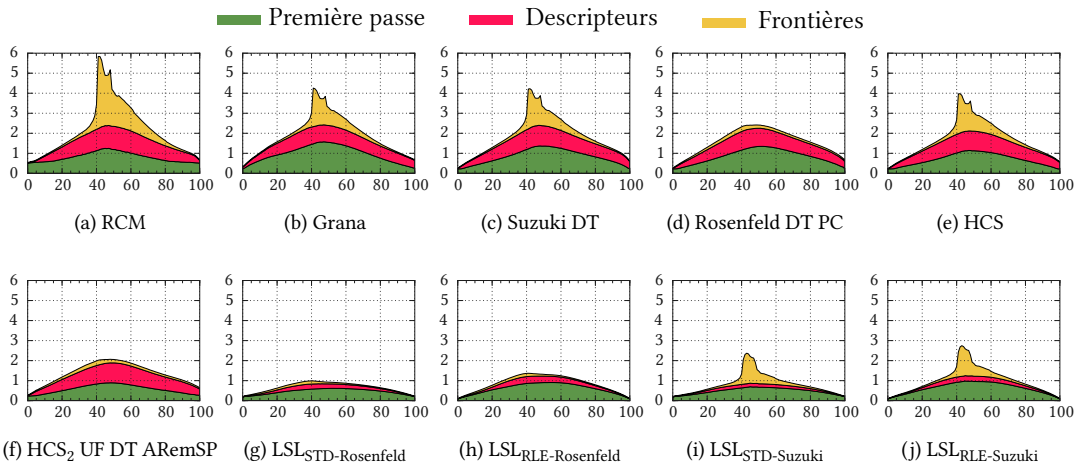


Fig. 5.11 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 1$ sur les 24 cœurs de la machine $IVB_{2\times12}$

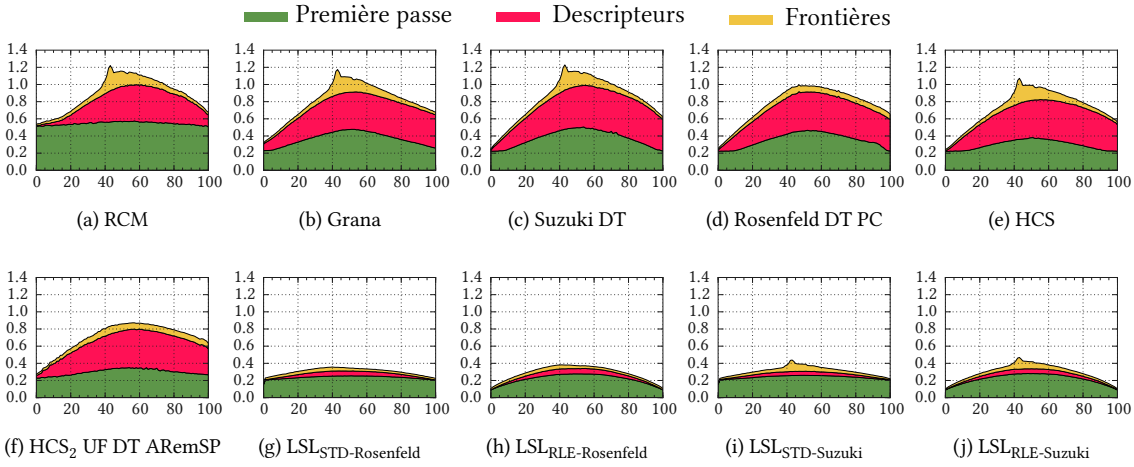


Fig. 5.12 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 4$ sur les 24 cœurs de la machine $IVB_{2\times12}$

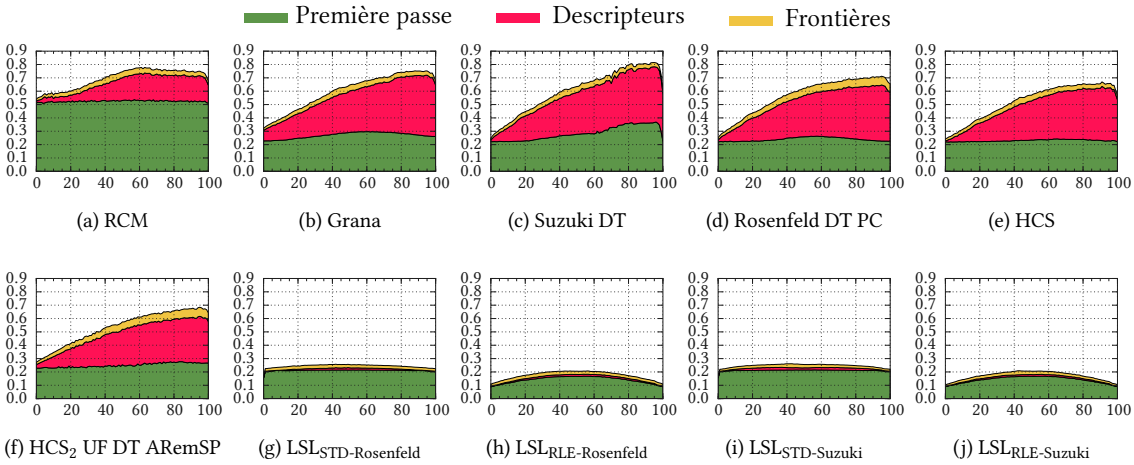


Fig. 5.13 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 16$ sur les 24 cœurs de la machine $IVB_{2\times12}$

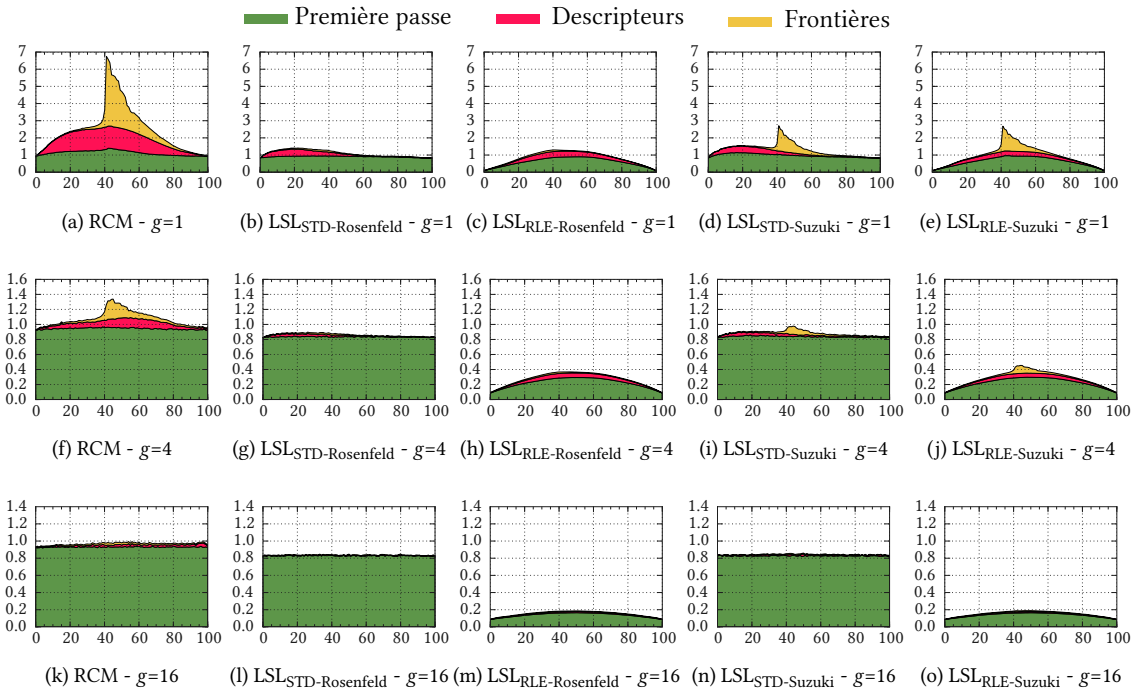


Fig. 5.14 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et g=1 (haut), g=4 (milieu) et g = 16 (bas) sur les 24 cœurs de la machine $IVB_{2\times12}$

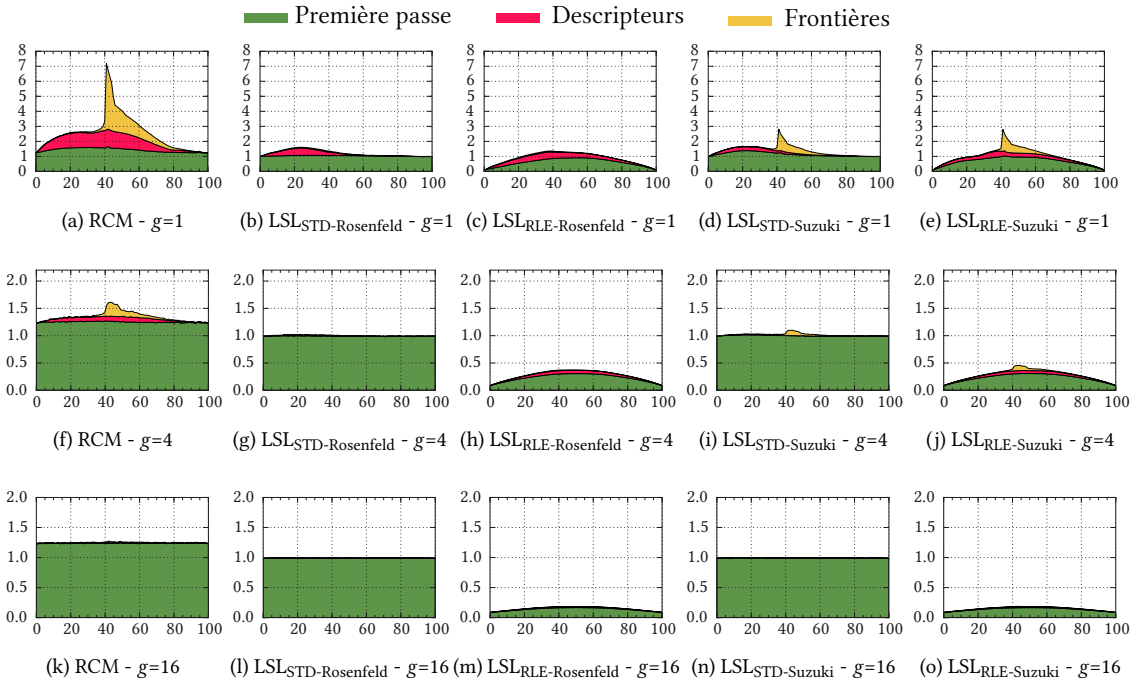


Fig. 5.15 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et g=1 (haut), g=4 (milieu) et g = 16 (bas) sur les 24 cœurs de la machine $IVB_{2\times12}$

temps de fusion des frontières est maximal. Comparativement, LSL_{STD} est moins sensible aux fusions car sa première passe est plus lente. Pour les algorithmes pixels, bien que le temps de fusion des frontières soit significativement plus long que pour les algorithmes segments, il l'est moins que la somme de la première passe et du calcul des descripteurs.

	Première passe			Descripteurs			Frontières			*sp*		
Algorithmes	min	moy	max	min	moy	max	min	moy	max	min	moy	max
$LSL_{RLE\text{-}Suzuki}$	0,09	0,13	0,16	0,00	0,01	0,01	0,008	0,012	0,016	11,85	15,13	16,60
$LSL_{RLE\text{-}Rosenfeld}$	0,09	0,13	0,16	0,00	0,01	0,02	0,010	0,011	0,013	11,04	14,03	15,51
$LSL_{STD\text{-}Suzuki}$	0,20	0,21	0,21	0,00	0,01	0,02	0,008	0,012	0,017	21,61	22,82	24,04
$LSL_{STD\text{-}Rosenfeld}$	0,20	0,21	0,21	0,00	0,01	0,02	0,010	0,011	0,012	21,26	22,61	23,49
HCS	0,24	0,26	0,28	0,24	0,27	0,34	0,018	0,028	0,036	14,72	17,34	21,20
HCS_2 UF DT ARemSP	0,27	0,30	0,34	0,21	0,26	0,31	0,036	0,045	0,053	14,79	16,75	19,80
Suzuki DT	0,24	0,25	0,26	0,33	0,36	0,41	0,019	0,028	0,036	15,31	18,37	22,91
Rosenfeld DT PC	0,23	0,25	0,28	0,30	0,32	0,36	0,033	0,041	0,048	15,53	17,51	21,15
Grana	0,25	0,26	0,28	0,33	0,36	0,38	0,018	0,028	0,035	16,13	18,40	21,87
RCM	0,50	0,51	0,52	0,09	0,16	0,23	0,019	0,028	0,036	16,01	18,39	22,30

TABLE 5.9 – Parallélisation multi-cœur : *cpp* de la première passe, *cpp* du calcul des descripteurs, *cpp* des frontières et accélération globale par rapport à la version séquentielle sur la base de données SIDBA4 sur la machine $IVB_{2\times12}$

5.3.4 Conclusion pour la station de travail

Sur une machine 24 cœurs, les algorithmes parallèles LSL sont les plus rapides (d'un facteur ×3,7 sur SIDBA4, ×1,7 pour g=1, ×2,5 pour g=4 et ×3,0 pour g=16 en 2048 × 2048) et ceux qui présentent le moins de variations dans le temps de traitement. Sur ce type de machine, l'efficacité de la méthode de parallélisation est comprise entre 75,5% et 88,3% pour les images aléatoires de taille 2048×2048 et 46,0% et 100,2% pour les images de la base SIDBA4. Avec l'augmentation de la taille des données, seul l'algorithme LSL_{RLE} tient la charge et l'efficacité de sa parallélisation augmente passant pour $LSL_{RLE\text{-}Rosenfeld}$ de 87,3% pour les images 2048 × 2048 à 96,4% pour les images 4096×4096 et 98,3% pour les images 8192×8192. Le ratio entre $LSL_{RLE\text{-}Rosenfeld}$ et les algorithmes pixels passe à ×2,7 pour g=1, ×5,3 pour g=4 et ×8,5 pour g=16 en 8192×8192.

5.4 Serveur de calculs - 4×15 cœurs

5.4.1 Résultats pour les images aléatoires

5.4.1.1 Images 2048 × 2048

Sur les 60 cœurs du serveur de calcul, le nombre d'étages de l'étape de fusion des bords passe de 5 à 6.

Pour g=1 (fig. 5.16), la dégradation des performances due à la gestion des équivalences de Suzuki prend des proportions considérables. Pour g=4, elle est encore très présente. Dès g=7, les courbes fusionnent et se répartissent en quatre groupes (fig. 5.16 et tab. 5.16) : RCM est le plus lent (cpp_d = 0,55 pour g=16), suivi par Grana et HCS_2 UF DT ARemSP ($cpp_d \approx 0{,}5$ pour g=16), suivis par HCS, Suzuki DT et Rosenfeld DT PC ($0{,}45 \leq cpp_d \leq 0{,}47$ pour g=16) et le dernier groupe qui comprend tous les algorithmes LSL ($0{,}14 \leq cpp_d \leq 0{,}18$ pour g=16).

Ces quatre groupes correspondent en fait à des formes de masque différentes :

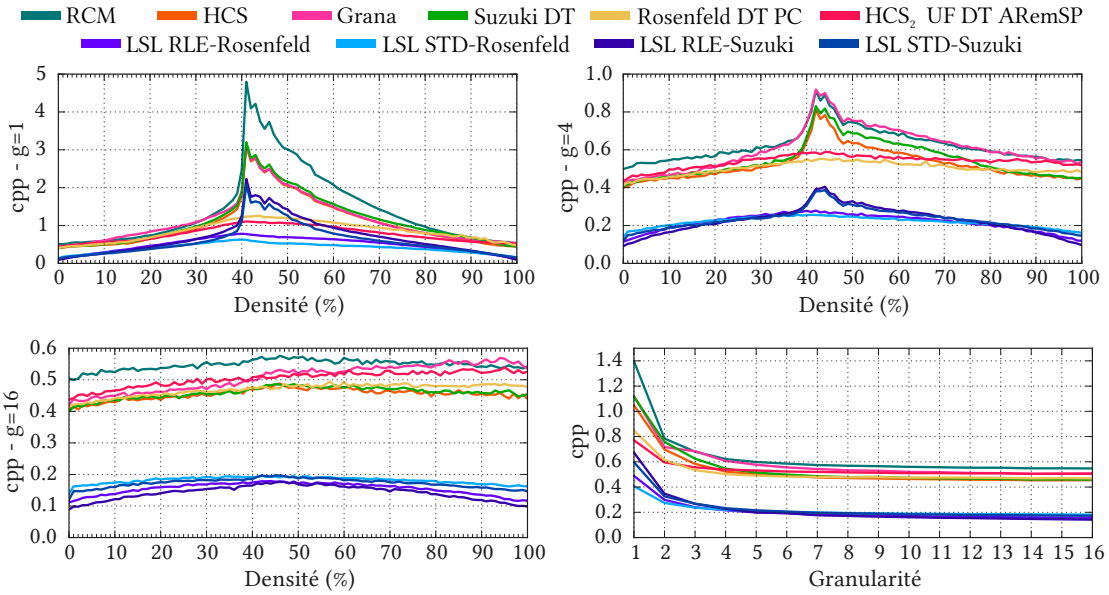


Fig. 5.16 – Parallélisation multi-cœur : *cpp* pour des images de taille 2048 × 2048 et de granularité $g \in \{1, 4, 16\}$ et cpp_d en fonction de la granularité sur les 60 cœurs $IVB_{4\times15}$

	Granularité					
Algorithmes	g=1	g=2	g=4	g=8	g=16	cpp_g
$LSL_{RLE\text{-}Rosenfeld}$	0,49	0,30	0,22	0,18	0,15	0,21
$LSL_{STD\text{-}Rosenfeld}$	0,41	0,27	0,22	0,19	0,18	0,21
$LSL_{RLE\text{-}Suzuki}$	0,67	0,35	0,23	0,17	0,14	0,22
$LSL_{STD\text{-}Suzuki}$	0,60	0,33	0,23	0,19	0,17	0,23
Rosenfeld DT PC	0,85	0,61	0,50	0,48	0,47	0,51
HCS	1,05	0,70	0,52	0,47	0,45	0,53
HCS_2 UF DT ARemSP	0,77	0,60	0,54	0,52	0,50	0,54
Suzuki DT	1,12	0,76	0,55	0,48	0,46	0,55
Grana	1,12	0,72	0,61	0,53	0,51	0,59
RCM	1,40	0,78	0,62	0,57	0,55	0,64

TABLE 5.10 – Parallélisation multi-cœur : cpp_d pour les granularités $g \in \{1, 2, 4, 8, 16\}$ et cpp_g pour des images 2048×2048 sur la machine $IVB_{4\times15}$

- RCM dispose d'un masque réduit mais qui impose un chargement du voisinage pour tous les pixels,
- Grana et HCS_2 UF DT ARemSP travaillent tous les deux sur plusieurs lignes à la fois,
- HCS, Suzuki DT et Rosenfeld DT PC ne travaillent que sur une ligne,
- enfin les algorithmes LSL travaillent sur des segments.

Un point remarquable est que l'algorithme $LSL_{STD\text{-}Rosenfeld}$, est plus rapide que tous les algorithmes pixels et ceci quelle que soit la granularité.

5.4.1.2 Impact de la quantité de données

Pour les images de taille 2048×2048, l'efficacité de la parallélisation est dans tous les cas inférieure à 50% (tab. 5.11). Le passage à des images de taille 4096×4096 est bénéfique à tous les algorithmes. L'écart se creuse entre les algorithmes pixels et les algorithmes LSL mais la distribution des résultats reste inchangée. Le passage à des images de taille 8192×8192 n'est bénéfique pour pour les versions RLE. Tous les autres algorithmes voient leur performance chuter. Grana est le plus lent avec ses deux passes sur l'image, suivi des LSL_{STD} qui diffèrent des algorithmes LSL_{RLE} par une écriture systématique en mémoire. HCS_2 UF DT ARemSP, Rosenfeld DT PC, HCS, Suzuki DT sont très proches. Le LSL_{RLE} est plus rapide que le plus rapide des algorithmes pixels d'un facteur ×6,6 et l'efficacité de la parallélisation passe à 91% avec une portion de code séquentiel τ = 0,16%.

	cpp_g			sp			τ en %		
Algorithmes	2048	4096	8192	2048	4096	8192	2048	4096	8192
$LSL_{RLE\text{-}Rosenfeld}$	0,21	0,13	0,11	×26,7	×44,5	×54,8	2,11	0,59	0,16
$LSL_{RLE\text{-}Suzuki}$	0,22	0,15	0,13	×23,2	×36,8	×46,0	2,68	1,07	0,51
$LSL_{STD\text{-}Rosenfeld}$	0,21	0,14	0,85	×29,1	×48,0	×9,8	1,79	0,42	8,71
$LSL_{STD\text{-}Suzuki}$	0,23	0,16	0,87	×28,2	×49,8	×7,6	1,91	0,35	11,71
HCS	0,53	0,51	0,73	×25,8	×29,7	×19,0	2,25	1,72	3,66
Rosenfeld DT PC	0,51	0,49	0,71	×29,3	×31,0	×21,3	1,78	1,58	3,08
HCS_2 UF DT ARemSP	0,54	0,51	0,73	×24,1	×26,3	×18,1	2,53	2,17	3,93
Suzuki DT	0,55	0,51	0,74	×27,8	×29,9	×20,7	1,96	1,70	3,22
RCM	0,64	0,59	0,76	×26,2	×28,8	×22,0	2,19	1,83	2,92
Grana	0,59	0,54	1,17	×26,1	×29,6	×13,7	2,19	1,73	5,75

TABLE 5.11 – Parallélisation multi-cœur : *cpp* moyen sur les granularités de 1 à 16, *sp* l'accélération moyenne sur les granularités de 1 à 16 et τ la portion de code séquentiel pour des tailles d'image 2048×2048, 4096×4096, 8192×8192 sur la machine $IVB_{4\times15}$

Sur la station de travail, la dégradation des performances pour les algorithmes hors LSL_{RLE} était visible dès 4096×4096. Sur le serveur de calculs (fig. 5.17), les performances sont toujours correctes pour 4096×4096 et bien que très dégradées en 8192×8192, elle restent supérieure à celles de la station de travail.

Tout comme pour la station de travail, le ratio entre le *cpp* minimum de l'ensemble des algorithmes pixels et le *cpp* de $LSL_{RLE\text{-}Rosenfeld}$ (fig. 5.18) est très sensible à la taille des images, passant pour g=16 d'un rapport minimal de ×2,5 pour les images 2048×2048 à un rapport ×8 pour les images 8192×8192.

5.4.2 SIDBA4

L'algorithme LSL_{RLE} est le plus rapide, d'un facteur ×4,2 par rapport à HCS et ×5,1 par rapport à RCM (fig. 5.19 et tab. 5.12). L'écart entre les algorithmes pixels et le LSL_{RLE} s'est donc accentué par rapport à la station de travail.

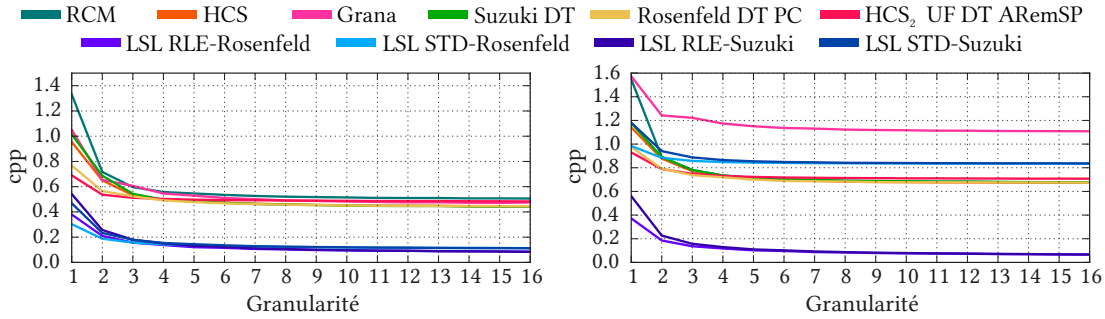


Fig. 5.17 – Parallélisation multi-cœur : cpp_d en fonction de la granularité sur les 60 cœurs de la machine $IVB_{4\times15}$ pour des images de taille 4096×4096 (gauche) et 8192×8192 (droite)

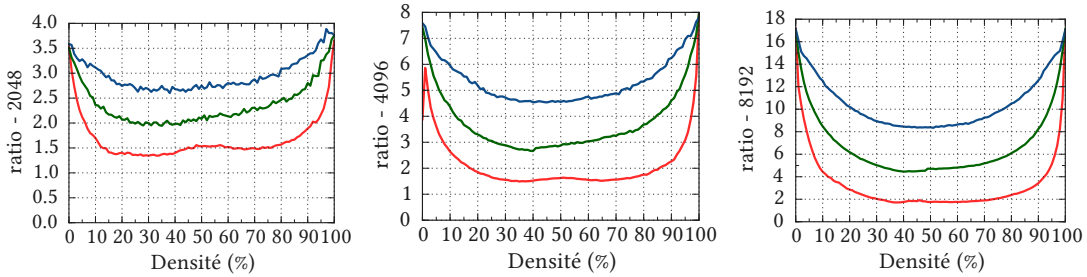


Fig. 5.18 – Analyse en composantes connexes : ratio entre le *cpp* de LSL_{RLE} et le minimum des *cpp* des algorithmes pixels sur la machine $IVB_{4\times15}$ pour les granularités g=1 (rouge), g=4 (vert) et g=16 (bleu)

LSL_{STD} est le plus stable en temps de traitement, mais il est toujours plus lent que le LSL_{RLE} ce qui est normal du fait de sa conception (une opération supplémentaire pour améliorer le déterminisme). Sur les images de la base, il n'y a pas de différence entre les versions Rosenfeld et Suzuki des algorithmes LSL.

L'accélération entre les versions séquentielles et les versions parallèles, est comprise entre ×20,4 (3,29% de code séquentiel) pour LSL_{RLE} et ×40,1 (0,84% de code séquentiel) pour LSL_{STD}. Les algorithmes pixels ont une accélération moyenne comprise entre ×21,5 (3,02% de code séquentiel) et ×27,6 (1,98% de code séquentiel).

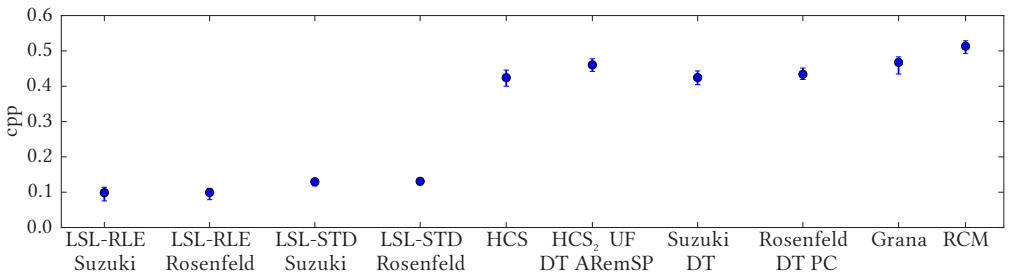


Fig. 5.19 – Parallélisation multi-cœur : *cpp* moyen et variabilité (cpp_{max} et cpp_{min}) sur la base de données SIDBA4 sur les 60 cœurs de la machine $IVB_{4\times15}$

Algorithmes	*cpp*	*sp*	τ	ratio
$LSL_{RLE\text{-}Suzuki}$	0,10	×22,1	2,90	×1,0
$LSL_{RLE\text{-}Rosenfeld}$	0,10	×20,4	3,29	×1,0
$LSL_{STD\text{-}Suzuki}$	0,13	×40,1	0,84	×1,3
$LSL_{STD\text{-}Rosenfeld}$	0,13	×39,4	0,88	×1,3
HCS	0,42	×22,6	2,80	×4,2
Suzuki DT	0,42	×27,6	1,98	×4,2
Rosenfeld DT PC	0,43	×24,6	2,43	×4,3
HCS_2 UF DT ARemSP	0,46	×21,5	3,02	×4,6
Grana	0,47	×25,1	2,35	×4,7
RCM	0,51	×24,9	2,38	×5,1

TABLE 5.12 – Parallélisation multi-cœur : *cpp* moyen, *sp* l'accélération par rapport à la version séquentielle, τ la portion de code séquentiel mesuré sur la base de données SIDBA4 et le ratio entre le *cpp* de l'algorithme et celui de $LSL_{RLE\text{-}Rosenfeld}$ sur la machine $IVB_{4\times15}$

5.4.3 Parts des étapes intermédiaires

5.4.3.1 Images aléatoires 2048×2048

Le *cpp* des fusions aux frontières a progressé (fig. 5.20, 5.21 et 5.22) et atteint dans le pire cas 80% du *cpp* total pour RCM et g=1. Pour les algorithmes basés sur Union-Find, il se limite à 30% dans le pire cas pour g=1. Dès g=4, le *cpp* du calcul des descripteurs devient faible pour tous les algorithmes en dehors de Grana qui doit les réaliser après chaque double ligne de sa seconde passe. En fait, le *cpp* global avec et sans descripteurs devient équivalent, illustrant le caractère prépondérant des échanges mémoires devant les calculs. Pour les algorithmes pixels, l'étape de fusion des frontières représente 30% du *cpp* total alors que pour les algorithmes LSL, elle avoisine les 50%. Cet écart explique la faible performance de la parallélisation pour le LSL pour cette taille d'image comparativement aux algorithmes pixels. Il n'y a plus assez de données à traiter.

Pour g=16, quelle que soit la densité, la première passe des algorithmes (hors LSL_{RLE}) est constante. Le *cpp* de la première passe est donc totalement indépendant des données et du nombre d'étiquettes et dépend donc uniquement des échanges avec la mémoire. La première passe de Suzuki DT, Rosenfeld DT PC et de HCS est strictement identique, malgré leurs différences algorithmiques.

5.4.3.2 Images aléatoires : impact de la quantité de données

Le passage à des images de taille 4096×4096, diminue l'influence relative de la gestion de frontières et permet donc d'accélérer le temps de traitement (fig. 5.23). Comme pour la station de travail, le passage à des images de taille 8192×8192 (fig. 5.24) ne profite qu'au LSL_{RLE}, la première passe de tous les autres algorithmes étant ralentie.

Dans tous les cas, les figures montrent que la première passe est devenue indépendante de la densité car limitée par la mémoire et le calcul des descripteurs n'est plus significatif dès que $g \geq 4$ pour tous les algorithmes hors LSL_{RLE}.

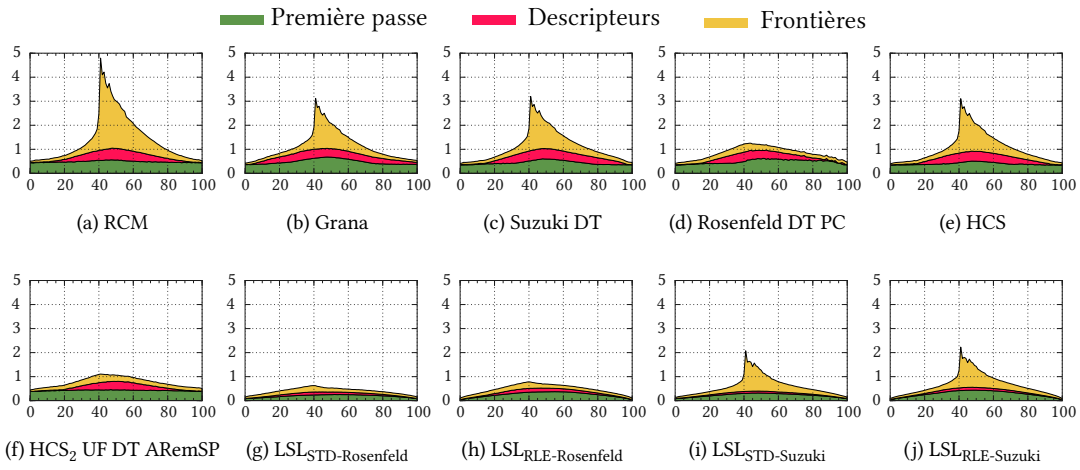


Fig. 5.20 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 1$ sur les 60 cœurs de la machine $IVB_{4\times15}$

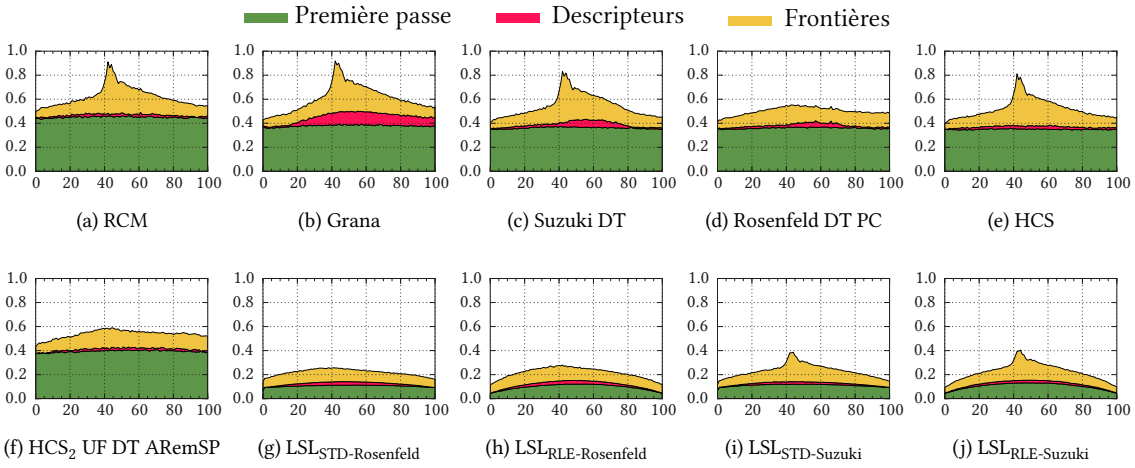


Fig. 5.21 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 4$ sur les 60 cœurs de la machine $IVB_{4\times15}$

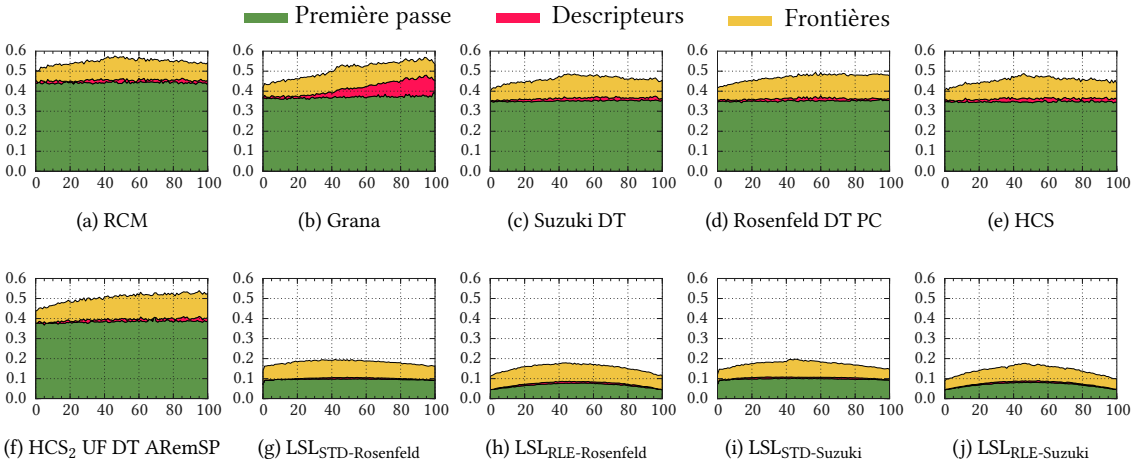
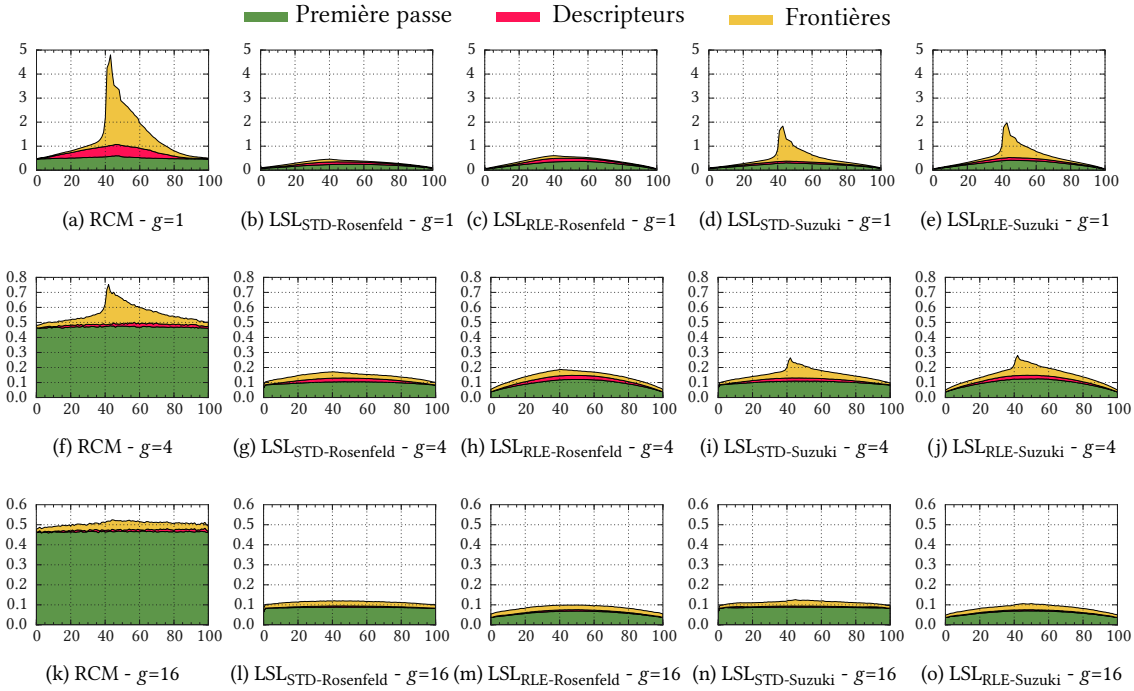


Fig. 5.22 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 2048×2048 et $g = 16$ sur les 60 cœurs de la machine $IVB_{4\times15}$

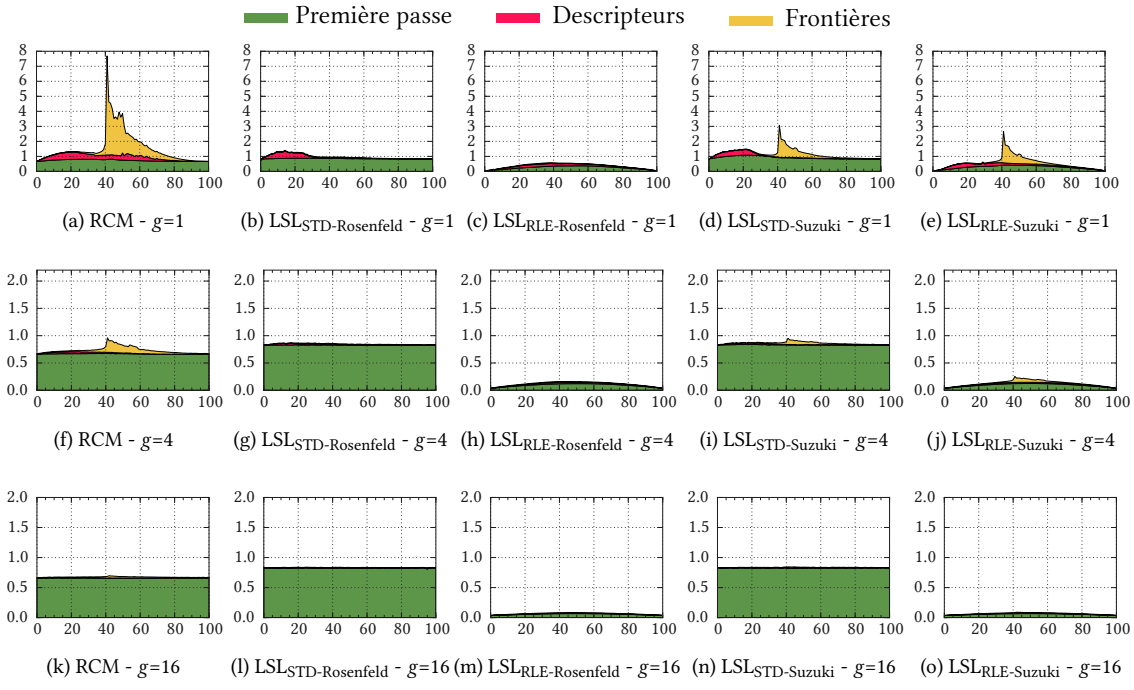


(a) RCM - g=1 (b) $LSL_{STD\text{-}Rosenfeld}$ - g=1 (c) $LSL_{RLE\text{-}Rosenfeld}$ - g=1 (d) $LSL_{STD\text{-}Suzuki}$ - g=1 (e) $LSL_{RLE\text{-}Suzuki}$ - g=1

(f) RCM - g=4 (g) $LSL_{STD\text{-}Rosenfeld}$ - g=4 (h) $LSL_{RLE\text{-}Rosenfeld}$ - g=4 (i) $LSL_{STD\text{-}Suzuki}$ - g=4 (j) $LSL_{RLE\text{-}Suzuki}$ - g=4

(k) RCM - g=16 (l) $LSL_{STD\text{-}Rosenfeld}$ - g=16 (m) $LSL_{RLE\text{-}Rosenfeld}$ - g=16 (n) $LSL_{STD\text{-}Suzuki}$ - g=16 (o) $LSL_{RLE\text{-}Suzuki}$ - g=16

Fig. 5.23 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et g=1 (haut), g=4 (milieu) et g = 16 (bas) sur les 60 cœurs de la machine $IVB_{4\times15}$



(a) RCM - g=1 (b) $LSL_{STD\text{-}Rosenfeld}$ - g=1 (c) $LSL_{RLE\text{-}Rosenfeld}$ - g=1 (d) $LSL_{STD\text{-}Suzuki}$ - g=1 (e) $LSL_{RLE\text{-}Suzuki}$ - g=1

(f) RCM - g=4 (g) $LSL_{STD\text{-}Rosenfeld}$ - g=4 (h) $LSL_{RLE\text{-}Rosenfeld}$ - g=4 (i) $LSL_{STD\text{-}Suzuki}$ - g=4 (j) $LSL_{RLE\text{-}Suzuki}$ - g=4

(k) RCM - g=16 (l) $LSL_{STD\text{-}Rosenfeld}$ - g=16 (m) $LSL_{RLE\text{-}Rosenfeld}$ - g=16 (n) $LSL_{STD\text{-}Suzuki}$ - g=16 (o) $LSL_{RLE\text{-}Suzuki}$ - g=16

Fig. 5.24 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et g=1 (haut), g=4 (milieu) et g = 16 (bas) sur les 60 cœurs de la machine $IVB_{4\times15}$

5.4.3.3 SIDBA4

Pour les algorithmes LSL_{RLE} le *cpp* des fusions aux frontières représente en moyenne 40% du *cpp* total, 30% pour LSL_{STD} et entre 13% et 18,5% pour les autres algorithmes (tab. 5.13).

	Première passe			Descripteurs			Frontières			*sp*		
Algorithmes	min	moy	max	min	moy	max	min	moy	max	min	moy	max
$LSL_{RLE\text{-}Suzuki}$	0,04	0,06	0,08	0,00	0,00	0,01	0,031	0,038	0,043	19,05	22,12	24,61
$LSL_{RLE\text{-}Rosenfeld}$	0,04	0,05	0,06	0,00	0,00	0,01	0,035	0,039	0,044	18,17	20,38	22,34
$LSL_{STD\text{-}Suzuki}$	0,09	0,09	0,09	0,00	0,00	0,01	0,032	0,039	0,045	38,31	39,93	42,46
$LSL_{STD\text{-}Rosenfeld}$	0,08	0,09	0,09	0,00	0,00	0,01	0,036	0,040	0,043	37,68	39,18	40,65
HCS	0,34	0,34	0,35	0,00	0,02	0,03	0,053	0,064	0,075	18,23	22,64	28,96
HCS_2 UF DT ARemSP	0,36	0,37	0,37	0,01	0,01	0,02	0,069	0,080	0,090	18,82	21,99	25,73
Suzuki DT	0,34	0,34	0,34	0,00	0,02	0,03	0,052	0,064	0,074	21,78	27,67	34,76
Rosenfeld DT PC	0,34	0,34	0,34	0,00	0,02	0,02	0,066	0,076	0,083	20,97	24,67	29,84
Grana	0,34	0,35	0,35	0,03	0,05	0,06	0,054	0,064	0,074	21,24	25,55	30,09
RCM	0,43	0,43	0,44	0,00	0,02	0,02	0,052	0,064	0,074	21,76	24,96	28,93

TABLE 5.13 – Parallélisation multi-cœur : *cpp* de la première passe, *cpp* du calcul des descripteurs, *cpp* des frontières et accélération globale par rapport à la version séquentielle sur la base de données SIDBA4 sur la machine $IVB_{4\times15}$

5.5 Influence conjuguée de la taille des données et du nombre de cœurs actifs

Les résultats des sections précédentes nous confirment que le $LSL_{RLE\text{-}Rosenfeld}$ est l'algorithme le plus adapté pour toutes les tailles d'images et pour toutes les architectures testées. Dans la suite du document, le terme LSL_{RLE} désignera uniquement la version $LSL_{RLE\text{-}Rosenfeld}$.

La table 5.14a synthétise les résultats de LSL_{RLE} sur les machines et images testées en *ms* et en *Gp/s*. À titre de comparaison, la table 5.14b présente les mêmes résultats pour HCS_2 UF DT ARemSP.

		Images aléatoires			SIDBA4
Machines		2048	4096	8192	
$SKL_{1\times4}$	t(*ms*)	0,65	2,6	10,2	1,0
	Gp/s	6,8	6,6	6,6	7,5
$IVB_{2\times12}$	t(*ms*)	0,31	1,05	4,2	0,48
	Gp/s	13,3	16,0	16,0	16,0
$IVB_{4\times15}$	t(*ms*)	0,24	0,50	1,6	0,27
	Gp/s	17,5	33,3	**42,4**	28,0

(a) LSL_{RLE}

		Images aléatoires			SIDBA4
Machines		2048	4096	8192	
$SKL_{1\times4}$	t(ms)	2,4	9,4	37,4	4,9
	Gp/s	1,8	1,8	1,8	1,6
$IVB_{2\times12}$	t(ms)	0,94	5,9	34,1	1,9
	Gp/s	4,4	2,8	2,0	4,0
$IVB_{4\times15}$	t(ms)	0,75	4,3	17,0	1,3
	Gp/s	5,6	**5,8**	3,9	6,1

(b) HCS_2 UF DT ARemSP

TABLE 5.14 – Parallélisation multi-cœur : temps d'exécution em *ms* et débit en *Gp/s* pour des images de taille 2048×2048, 4096×4096 et 8192×8192 pour g=16 et SIDBA4 (3200×2400)

Les résultats varient selon le nombre de processeurs :

- Pour la machine mono-processeur, la performance maximale est atteinte pour les images 2048×2048 et se maintient pour les tailles d'images supérieures. Toutes les machines permettent d'atteindre des performances temps réel ($t \leq 40$ ms pour une caméra délivrant 25 images/s) pour les tailles

d'image testées. le LSL_{RLE} est ×3,7 plus rapide que HCS_2 UF DT ARemSP. Le débit maintenu est de 6,6Gp/s pour LSL_{RLE} et de 1,8Gp/s pour HCS_2 UF DT ARemSP.

- Pour les machines multi-processeurs, les performances des deux algorithmes divergent. Elles progressent avec la taille de l'image pour LSL_{RLE} et régressent pour HCS_2 UF DT ARemSP. Sur $IVB_{2\times12}$, la performance maximale est atteinte pour les images 4096×4096 et se maintient pour les images 8192×8192 alors que pour HCS_2 UF DT ARemSP, elle ne fait que décroître. Sur $IVB_{4\times15}$, LSL_{RLE} atteint son maximum (42,4Gp/s) pour les images 8192×8192 alors que HCS_2 UF DT ARemSP progresse pour atteindre son maximum en 4096×4096 (5,8Gp/s) avant de régresser pour les images 8192×8192. Pour g=16 et des images 8192×8192, LSL_{RLE} est ×10,8 plus rapide que HCS_2 UF DT ARemSP.

	Image size		
machines	2048	4096	8192
$SKL_{1\times4}$	× 3,6	× 3,7	× 3,7
$IVB_{2\times12}$	× 3,0	× 5,7	× 8,1
$IVB_{4\times15}$	× 3,1	× 5,7	× **10,8**

TABLE 5.15 – Parallélisation multi-cœur : ratio du *cpp* entre LSL_{RLE} et HCS_2 UF DT ARemSP, pour g=16

Le rapport des performances (tab. 5.15) passe de ×3.7 sur les 4 cœurs de la machine $SKL_{1\times4}$ à ×10,8 sur les 60 cœurs de la machine $IVB_{4\times15}$. La raison est que comme tous les algorithmes pixels, HCS_2 UF DT ARemSP est limité par la mémoire. C'est ce qui explique que LSL_{RLE} est plus rapide avec 15 fois moins de cœurs sur $SKL_{1\times4}$ que HCS_2 UF DT ARemSP sur $IVB_{4\times15}$.

Tant que les données (image binaire, image des étiquettes, tables d'équivalences et structures propres à chaque algorithme) tiennent dans le cache, la performance peut progresser avec le nombre de cœurs. Mais quand ce n'est plus le cas, le nombre de défauts de cache augmente ce qui entraîne l'augmentation du *cpp* (sortie de cache). Afin de mettre ce phénomène en évidence, nous avons réalisé une étude des performances des deux algorithmes en fonction de la taille de l'image et sur un nombre variable de processeurs (1, 2, 3, 4 sur la machine $IVB_{4\times15}$). La figure 5.25 met en évidence cette sortie de cache.

Pour LSL_{RLE} :

- Avec g=1, on observe une très légère sortie de cache avec 15, 30 et 45 cœurs (1, 2, 3 sockets) qui se stabilise respectivement pour des images de taille 4096×4096, 6144×6144 et 7168×7168. Dans tous les cas, plus le nombre de cœurs augmente, plus le *cpp* diminue quellle que soit la taille de l'image.
- Avec g=4, seule la version 15 cœurs présente une sortie de cache qui se stabilise pour les images 6144×6144. Dans tous les cas, plus le nombre de cœurs augmente, plus le *cpp* diminue quelle que soit la taille de l'image.
- Avec g=16, aucune sortie de cache n'est décelable.

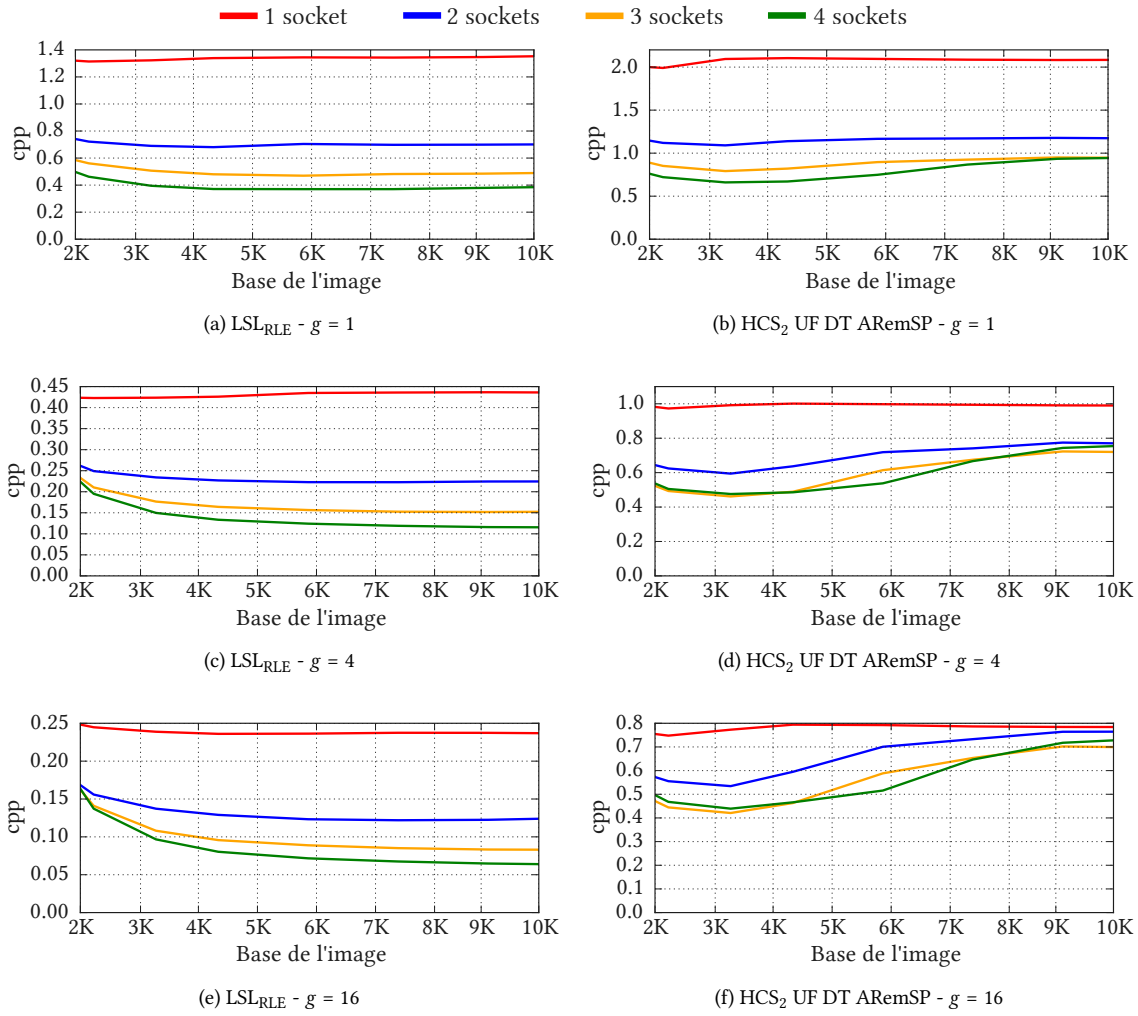
Pour HCS_2 UF DT ARemSP :

- Avec g=1, on observe une sortie de cache pour toutes les courbes. À partir des images de taille 8192×8192 les courbes pour 45 et 60 cœurs sont confondues.
- Avec g=4, les courbes pour 45 et 60 cœurs sont confondues pour toutes les tailles d'images observées. L'intérêt du quatrième processeur est dans ce cas très discutable. Les courbes pour 30, 45 et 60 cœurs convergent vers une limite commune ($\approx$ 0,75cpp) indiquant que pour les grandes

images, il n'y a aucun intérêt à utiliser un système de plus de deux processeurs. L'accélération obtenue en passant de 1 processeur à 4 n'est que de ×1,33.

- Avec g=16, l'écart entre les quatre courbes se réduit encore et l'accélération obtenue en passant de 1 processeur à 4 n'est que de ×1,11.

Pour HCS_2 UF DT ARemSP, l'utilisation d'un système multiprocesseur ne se justifie que pour des images qui tiennent dans les caches. Du fait de la compression RLC utilisée par LSL_{RLE}, il est le seul à tirer parti de ce type de systèmes.



(a) LSL_{RLE} - g = 1

(b) HCS_2 UF DT ARemSP - g = 1

(c) LSL_{RLE} - g = 4

(d) HCS_2 UF DT ARemSP - g = 4

(e) LSL_{RLE} - g = 16

(f) HCS_2 UF DT ARemSP - g = 16

Fig. 5.25 – *cpp* en fonction de la taille des images pour les granularités $g \in \{1, 4, 16\}$ sur la machine $IVB_{4\times15}$ avec 15, 30, 45 ou 60 cœurs actifs

5.6 Conclusion

La parallélisation proposée de LSL et des algorithmes de référence permet de tirer parti des architectures multi-cœur modernes. Lorsque le nombre de cœurs augmente et d'autant plus dans le cas clusters de multi-cœur, il est nécessaire d'augmenter la taille des données pour obtenir l'accélération optimale. Dans ces circonstances, les algorithmes pixels sont limités par la consommation de bande passante mémoire et ils ne passent donc pas à l'échelle. LSL_{RLE} est le seul algorithme direct adapté aux architectures multi socket.

La gestion des étiquettes de Suzuki, qui était compétitive pour les algorithmes séquentiels est très dégradée lors du passage au multi-cœur pour les images les moins structurées. La version recommandée pour LSL est donc la version $\text{LSL}_{\text{RLE-Rosenfeld}}$ basée sur la procédure Union-Find classique.

Chapitre 6

Le temps est en vous, comme l'eau dans une bouteille. Vous en buvez un peu chaque jour, à l'économie, et vous croyiez savoir ce qu'il est ?

–La Horde du contrevent, Alain Damasio

Algorithmes itératifs d'étiquetage en composantes connexes pour les architectures à très grand nombre de cœurs

6.1 Introduction

Le chapitre précédent a mis en évidence que seul LSL_{RLE} pouvait profiter pleinement de l'augmentation du nombre de cœurs et de sockets et ceci à condition de travailler sur des images de grande taille. Il est donc légitime de penser que le développement actuel d'architectures proposant toujours plus de cœurs (Xeon Phi, GPU, Tile64, MPPA de Kalray, TSAR ...) va nécessiter de concevoir différemment les algorithmes pour tirer parti efficacement du parallélisme matériel.

Dans ce cadre, les algorithmes itératifs, bien que très coûteux pour une architecture séquentielle, sont potentiellement plus adaptés aux machines à très grand nombre de cœurs car ils intègrent par construction des mécanismes de synchronisation. Déjà en 1981, Haralick & Shapiro écrivaient dans [82] à propos de leur proposition d'algorithme itératif : *«L'algorithme itératif n'utilise pas de structure externe de stockage des équivalences pour produire l'image étiquetée à partir de l'image binaire. Cela peut être utile dans des environnements où la quantité de mémoire est très limité ou sur les architectures SIMD*[1]*».*

Et en effet, l'exécution la plus rapide d'un étiquetage en composantes connexes à ce jour a été obtenue avec l'algorithme itératif massivement parallèle (MPAR EP cf. 6.2) sur la Maille Associative [20, 34]. Cette maille de processeurs élémentaires (PE), composée de PE synchrones et de PE asynchrones organisés en grille de 200×200 PE fonctionnant à 500MHz, étiquetait une image 200×200 pixels en $1\mu s$ atteignant ainsi un débit soutenu de $40Gp/s$ quelle que soit la structure de l'image pour une

1. Au sens de la taxinomie de Flynn [111].

consommation de $10\,W$. Seule la version parallèle LSL_{RLE}, pour des images de taille 8192×8192 avec g=16 sur une machine 4×15 cœurs consommant $620\,W$ a pu dépasser cette performance en 2015.

Dans ce chapitre, nous étudions le potentiel des algorithmes itératifs. Après une description des mécanismes et propriétés de l'algorithme itératif non-récursif (MPAR EP), nous proposons deux nouveaux algorithmes itératifs correspondants à deux approches complémentaires :

- MPAR FB + OMP + SIMD + AT : un algorithme proche de la proposition d'Haralick & Shapiro mais utilisant les instructions SIMD et OpenMP pour augmenter le parallélisme, ainsi qu'un découpage en tuile actives[83].
- WARP : une classe d'algorithmes hybrides basée à la fois sur un mécanisme itératif et un mécanisme de graphe pour réduire le nombre total d'itérations et le temps de traitement global. Cette version est la seule du manuscrit à s'adapter aux architectures de type GPU.

Il est important de noter que dans les cas des algorithmes itératifs, le résultat est par construction l'image complètement étiquetée et pas l'image analysée. L'étape d'analyse doit être ajoutée à la suite de l'étiquetage et ne permet pas d'accélérer le traitement de l'image. Nos travaux sur les algorithmes itératifs sont en cours et l'analyse en composantes connexes pour les algorithmes itératifs sera développée dans des travaux ultérieurs à la thèse. Dans le manuscrit, la comparaison entre algorithmes directs et algorithmes itératifs se fera donc sur la base de l'étiquetage en composantes connexes sans analyse et avec réétiquetage qui n'est pas le cas le plus favorable pour les algorithmes directs ni celui qui est utilisé dans les chaînes de traitements.

6.2 Algorithme itératif non récursif : MPAR EP

6.2.1 Principe

Pour les algorithmes itératifs, il est nécessaire de définir de nouvelles notations :

- B, l'image binaire à étiqueter,
- E, l'image des étiquettes obtenue à la fin de l'étiquetage,
- E_k, l'état de l'image des étiquettes correspondant à la k–ième itération de la propagation du minimum positif sur l'ensemble de l'image.

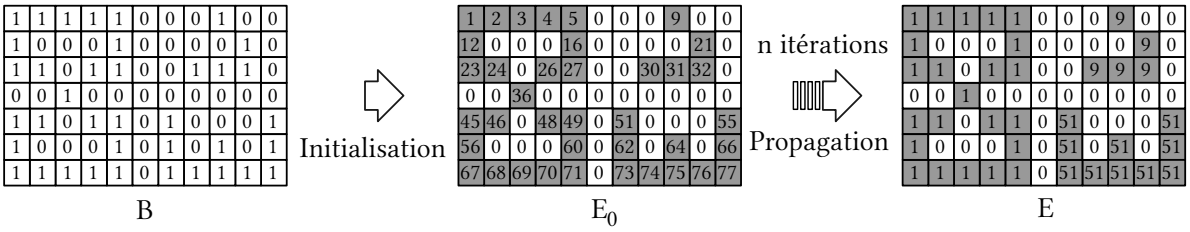


Fig. 6.1 – MPAR EP : l'image binaire (B) est initialisée avec des étiquettes uniques (E_0) puis la phase de propagation est réalisée un nombre indéterminé de fois, jusqu'à stabilisation de l'image (E)

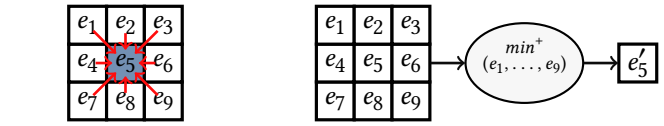
L'algorithme d'étiquetage en composantes connexes itératif MPAR EP (Embarrassingly Parallel), est constitué de deux phases :

- une phase d'initialisation (Algo. 23) qui fait correspondre à chaque pixel de l'image B une étiquette unique dans l'image E_0,

Algorithme 23 : MPAR EP : Initialisation de l'image des étiquettes

Input : $B[H][W]$ l'image binaire
Result : $E[H][W]$ l'image d'étiquettes

```
1 for i = 0 to H − 1 do
2     for j = 0 to W − 1 do
3         if B[i][j] ≠0 then
4             E[i][j] ← i × W + j + 1  ▷ unique et évite que le premier pixel ait une étiquette nulle (valeur du fond)
                                         si c'est un pixel de premier plan
```



(a) Sens de propagation : le pixel central reçoit le minimum du voisinage 3×3

(b) Point de vue producteur consommateur : 9 chargements pour une écriture

Fig. 6.2 – MPAR EP : masque de propagation du minimum $e_5 \leftarrow min^+(e_1, \ldots, e_9)$

- une phase de propagation qui réalise en tout pixel de E_k (Algo. 24) une propagation du minimum positif du voisinage (fig. 6.23) jusqu'à la stabilisation de l'image des étiquettes ($E_k = E_{k+1}$).

La phase d'initialisation garantit l'unicité des étiquettes dans E_0 (avant la propagation), la phase de propagation du minimum positif propage de proche en proche les plus petites étiquettes dans chaque composante connexe en un nombre fini mais indéterminé d'itérations.

6.2.2 Vitesse de propagation

Le principal inconvénient des algorithmes itératifs est la grande variabilité du nombre d'itérations nécessaires pour atteindre la stabilité en fonction de la structure des images. Dans le cas de la figure 6.3, la propagation n'est pas contrainte par la structure de l'image et peut s'effectuer dans toutes les directions d'un pixel par itération. 4 itérations sont nécessaires pour étiqueter l'image plus une pour s'assurer de la stabilité.

Dans le cas d'une spirale de taille 5×5 (fig. 6.4), 12 itérations sont nécessaires pour obtenir l'image finale, plus une pour s'assurer de la stabilité.

Algorithme 24 : MPAR EP : Propagation du min^+

Input : $E_k[H][W]$
Result : $E_{k+1}[H][W]$ mise à jour avec une propagation du min^+

```
1 for i = 0 to H − 1 do
2     for j = 0 to W − 1 do
3         si E_k[i][j] ≠ 0 alors
              e1 ← E_k[i − 1][j − 1]   e2 ← E_k[i − 1][j]   e3 ← E_k[i − 1][j + 1]
4             e4 ← E_k[i    ][j − 1]   e5 ← E_k[i    ][j]   e6 ← E_k[i    ][j + 1]
              e7 ← E_k[i + 1][j − 1]   e8 ← E_k[i + 1][j]   e9 ← E_k[i + 1][j + 1]
5             ε ← min+(e1, e2, e3, e4, e5, e6, e7, e8, e9)
6             E_{k+1}[i][j] ← ε
```

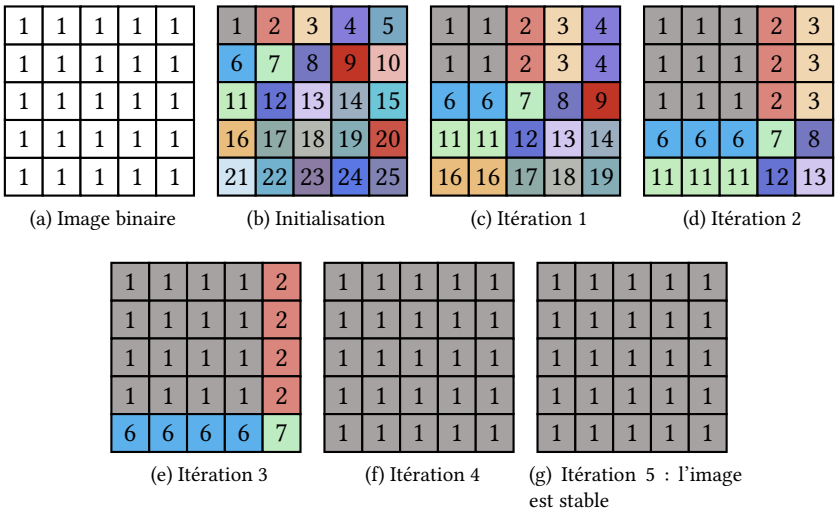


Fig. 6.3 – Vitesse de propagation : image 5×5 pleine, 5 itérations de propagation sont nécessaires pour s'assurer de la stabilité

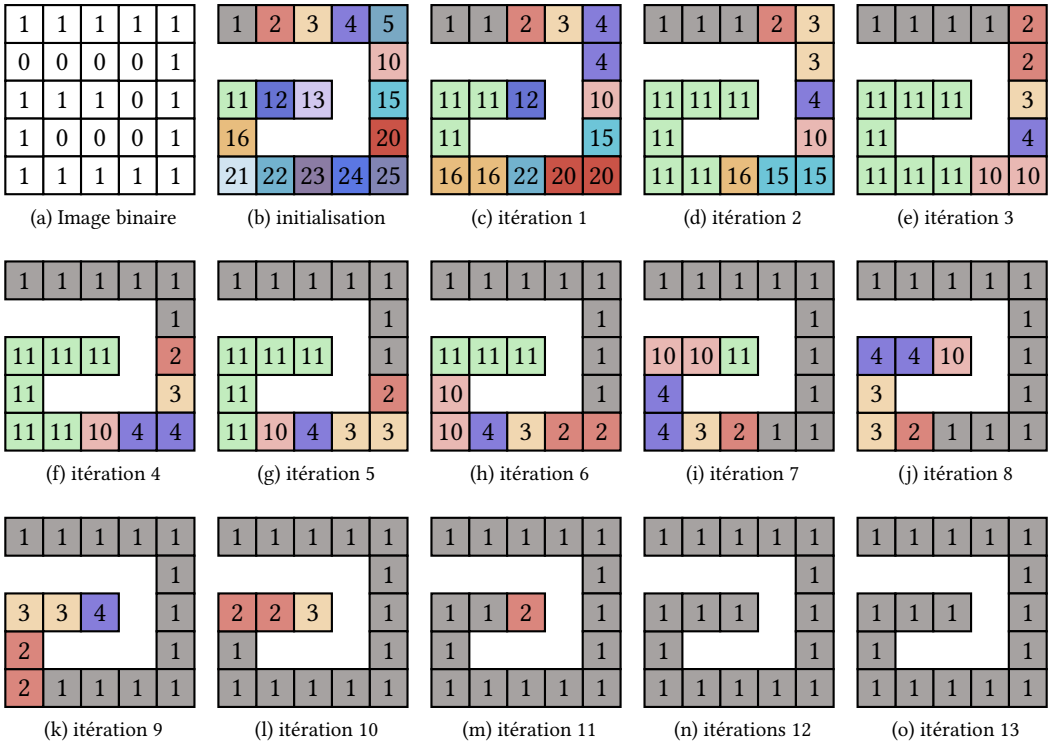


Fig. 6.4 – Vitesse de propagation : dans le cas d'une spirale 5×5, 13 itérations de propagation sont nécessaires pour s'assurer de la stabilité

Ces deux figures mettent en évidence un phénomène de propagation semblable à un front d'onde où chaque étiquette se comporte comme une source pour les étiquettes plus grandes.

Ce phénomène s'explique par le fait qu'à chaque itération, de par le voisinage 3×3, la distance maximale parcourue par une étiquette est de 1 pixel. La vitesse de propagation de l'onde «étiquette» sur le support «image» est donc constante et vaut $v = 1$ pixel par itération. Cette limitation «phy-

sique» de l'algorithme MPAR EP relie donc le nombre d'itérations avant stabilisation à la distance géodésique[112] maximale entre les pixels de l'image. En effet, la distance géodésique entre deux pixels est le plus court chemin qui les relie passant par des pixels de premier plan. La distance géodésique maximale, qui est le plus grand de ces chemins, est donc la borne supérieure du nombre d'itérations à réaliser.

Dans le cas des spirales (orientées comme dans la figure 6.4), la distance géodésique maximale (Dg_N) est particulièrement élevée et nécessite donc un grand nombre d'itérations. Pour une spirale de taille $N \times N$ avec N pair alors $Dg_N = N^2/2$ (si N est impair, $Dg_N = (N-1) \times (N+1)/2$). La table 6.1 donne un aperçu de l'évolution du nombre d'itérations nécessaires pour différentes tailles d'images de spirale. Pour une image de taille 2048×2048 il faut $nb_{\text{iter}} = Dg_{2048} + 1 \approx 2{,}1 \times 10^6$ itérations.

N	D_{gN}	Nombre d'itérations
5	12	13
7	24	25
9	40	41
100	5000	5001
1024	524288	524289
2048	$2{,}1 \times 10^6$	$2{,}1 \times 10^6$
8192	$33{,}6 \times 10^6$	$33{,}6 \times 10^6$

Table 6.1 – Évolution du nombre d'itérations en fonction de la largeur de la spirale

6.2.3 Conclusion

Pour accélérer l'algorithme itératif, il est nécessaire de dépasser la limite due à la vitesse de propagation. Une première solution pourrait être d'augmenter la taille du masque. Un masque $k \times k$ avec $k = 2r + 1$ aurait une vitesse de propagation $v = r$ (un masque 5×5 aurait une vitesse de propagation $v = 2$). Cette approche est peu efficace car plus le masque est grand, plus il contient de pixels à tester (k) et donc, plus il est lent.

Dans ce chapitre, nous allons proposer deux solutions pour augmenter la vitesse de propagation :

- travailler «en place» et faire varier le sens de parcours pour qu'au sein d'une même itération, l'information puisse se propager de plus d'un pixel (section 6.3),
- envisager l'algorithme itératif du point de vue des graphes afin de propager l'information de connexité au-delà de l'horizon du masque 3×3 (section 6.4).

Les performances de ces deux solutions seront évaluées dans le chapitre 7.

6.3 MPAR FB + SIMD + OMP + AT

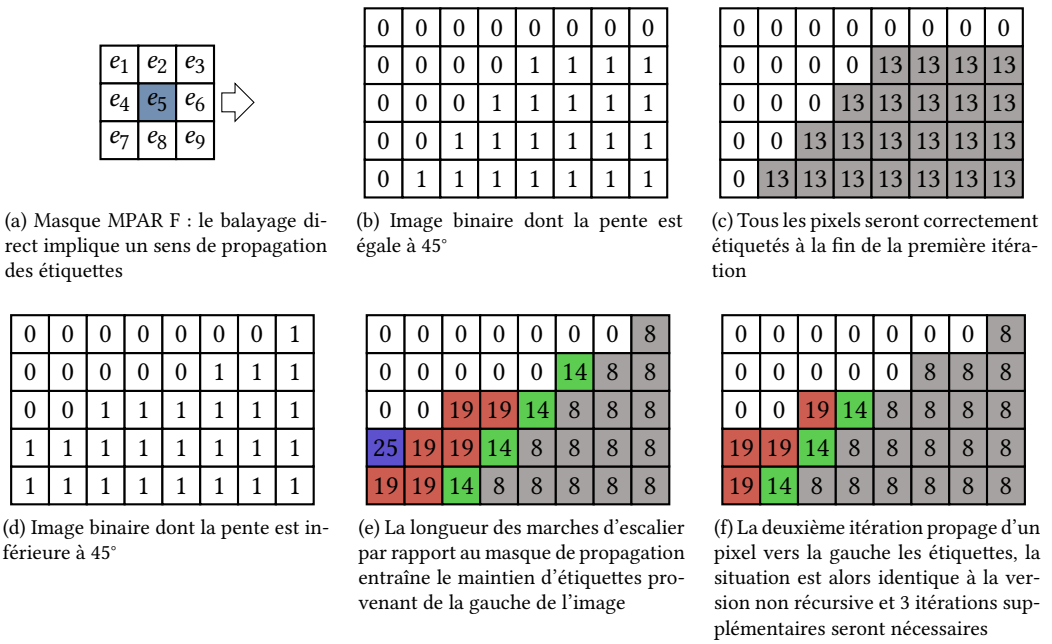
L'algorithme MPAR FB + SIMD + OMP + AT découle d'une succession d'améliorations apportées à MPAR EP afin :

- d'augmenter la vitesse de propagation (MPAR F et MPAR FB),
- d'utiliser les instructions SIMD (MPAR FB + SIMD),
- d'utiliser l'ensemble des cœurs (MPAR FB + SIMD + OMP),
- d'optimiser la charge sur chaque cœur en ne réalisant le traitement d'une tuile que lorsqu'il est nécessaire (MPAR FB + SIMD + OMP + AT),
- d'accélérer les opérations de propagation en modifiant le représentant de la composante connexe (MPAR FB + SIMD + OMP + AT + MAX).

6.3.1 MPAR F : algorithme récursif par balayage direct (Forward)

Pour augmenter la distance que peut parcourir une étiquette au cours d'une itération, une solution est d'utiliser une version récursive (cf. Haralick & Shapiro - sec 1.4.2). C'est-à-dire, d'utiliser une seule image des étiquettes et de travailler «en place».

Lors du balayage direct, tous les pixels connexes de premier plan sont correctement étiquetés en une seule passe si la forme de l'image le permet (fig. 6.5c). Pour des images plus complexes, avec par exemple des marches d'escalier de plus d'un pixel de long, certains pixels ne seront pas correctement étiquetés par le premier balayage (fig. 6.5e) et des itérations supplémentaires seront nécessaires (fig. 6.5f). Si les plus petites étiquettes proviennent de la gauche, la vitesse de propagation n'est pas limitée au sein d'une itération (propagation dans le sens du balayage) alors que si elles proviennent de la droite (propagation à contre sens) la vitesse de propagation reste $v = 1$ pixel par itération.



(a) Masque MPAR F : le balayage direct implique un sens de propagation des étiquettes

(b) Image binaire dont la pente est égale à 45°

(c) Tous les pixels seront correctement étiquetés à la fin de la première itération

(d) Image binaire dont la pente est inférieure à 45°

(e) La longueur des marches d'escalier par rapport au masque de propagation entraîne le maintien d'étiquettes provenant de la gauche de l'image

(f) La deuxième itération propage d'un pixel vers la gauche les étiquettes, la situation est alors identique à la version non récursive et 3 itérations supplémentaires seront nécessaires

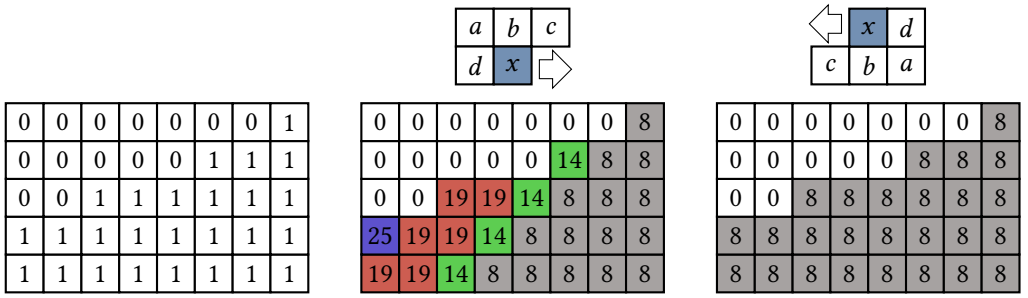
Fig. 6.5 – Dissymétrie de la vitesse de propagation due au sens de balayage : la vitesse de propagation des étiquettes dans le sens du balayage n'a pas de limite et vaut $v = 1$ dans le sens inverse

6.3.2 MPAR FB : algorithme récursif par balayage aller-retour (Forward Backward)

Comme vu dans la section 1.4.2, si la plus petite étiquette provient de la droite, seul un balayage inverse permet de la propager rapidement. Afin de réduire encore le nombre d'itérations, MPAR FB utilise alternativement un balayage direct et un balayage inverse (aller-retour). Le masque 3×3 n'est alors plus utile et il est remplacé par les deux masques (direct et inverse) proposés par Haralick & Shapiro.

Dans le cas de la figure précédente (fig. 6.5d), où les marches d'escalier impliquaient des itérations supplémentaires avec l'algorithme MPAR EP, l'algorithme MPAR FB permet d'étiqueter correctement l'image en un seul aller-retour (fig. 6.6). Pour des images plus complexes, le nombre d'itérations est réduit mais reste variable en fonction de la structure de l'image.

La figure 6.7 représente le nombre d'itérations nécessaires pour l'algorithme non récursif MPAR EP, pour l'algorithme itératif direct MPAR F et l'algorithme aller-retour MPAR FB en fonction de la densité pour des images de taille 128×128. Comme on peut l'observer pour g=1, la courbe représentant

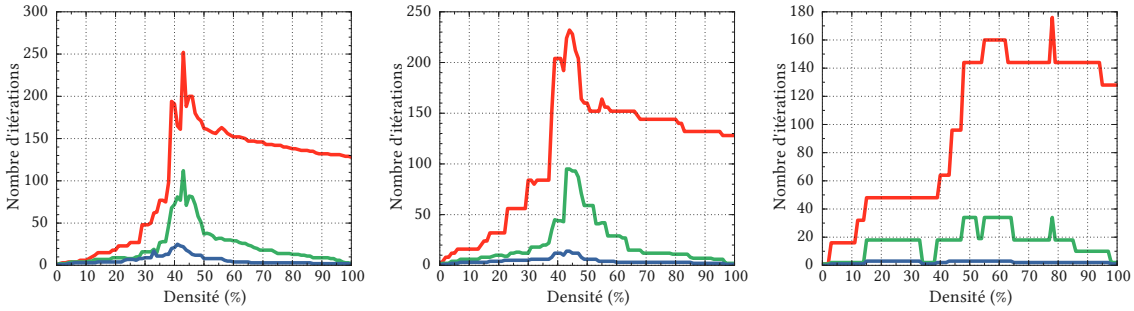


(a) Avec des marches d'escalier de plus d'un pixel de long, certains pixels (en rouge) ne seront pas correctement étiquetés à la fin de la première itération

(b) Phase F : le masque a le même effet que celui de MPAR F

(c) Phase B : le balayage inverse symétrise la vitesse de propagation

Fig. 6.6 – Propagation de l'étiquette aux pixels de premier plan (zone grise) dans une passe directe (gauche) et inverse (droite), les pixels rouges sont ceux qui n'auront pas la bonne étiquette à la fin de la passe



(a) g=1 (b) g=4 (c) g=16

Fig. 6.7 – Nombre d'itérations nécessaires à la stabilisation de l'image des étiquettes pour l'algorithme MPAR EP (rouge), l'algorithme MPAR F (vert) et l'algorithme MPAR FB pour des images de taille 128× 128 en fonction de la densité

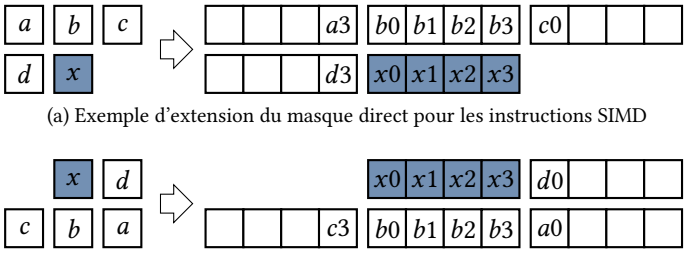
le nombre d'itérations pour l'algorithme MPAR EP est constituée de deux parties : jusqu'à la densité d=41%, le nombre d'itérations croît en variant autour d'une courbe exponentielle puis, après le pic pour d=42%, il décroît et tend vers n=128 (distance géodésique maximale dans l'image pleine). Pour l'algorithme MPAR F, la courbe se symétrise autour d'un pic correspondant à d =42%. Le nombre d'itérations est réduit pour toutes les densités et tend vers 1. Pour l'algorithme MPAR FB, le maximum est atteint pour d=40% et est inférieur d'un facteur 10 au maximum pour MPAR EP. Pour les granularités g=4 et g=16, la structuration de l'image diminue le nombre d'itérations. Pour MPAR FB avec g=16, l'image est étiquetée en 3 itérations dans le pire cas.

6.3.3 MPAR FB + SIMD : utilisation des instructions vectorielles

Avant même l'augmentation du nombre de cœurs, les processeurs ont été dotés d'unités SIMD permettant de réaliser la même instruction sur un ensemble de données. Ces unités sont accessibles *via* des instructions SIMD : MMX, SSE, AVX, AVX2, KNC pour la famille x86. Travaillant pour la plupart sur des données contiguës en mémoire, ces extensions sont inutilisables par les algorithmes directs. En effet, lors de la remontée à la racine, rien n'assure que les ancêtres soit contigus dans

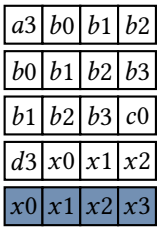
la table d'équivalence. C'est seulement avec l'utilisation d'extensions utilisant un adressage dispersé (gather-scatter) que des algorithmes directs SIMD pourront être créés.

Dans le cas des algorithmes itératifs, il est possible d'étendre le masque direct et inverse (fig. 6.8b) et de réaliser le minimum positif à l'aide de 5 registres (fig. 6.8c) pour les instructions 128 bits (CARD = 4 pour des données 32 bits), la dépendance de données ainsi générée étant résolue par itérations rapides dans les registres SIMD.



(a) Exemple d'extension du masque direct pour les instructions SIMD

(b) Exemple d'extension du masque inverse pour les instructions SIMD



(c) Extraction du minimum positif à l'aide de 5 registres SIMD

Fig. 6.8 – Adaptations des balayages direct et inverse pour des instructions SIMD de cardinal CARD=4 (128 bits)

6.3.4 MPAR FB + SIMD + OMP

Afin d'utiliser l'ensemble des cœurs, l'image est découpée en bandes sur le même principe que les algorithmes directs (chap. 4). Chaque cœur itère dans la tuile qui lui est assigné jusqu'à stabilisation totale de l'image.

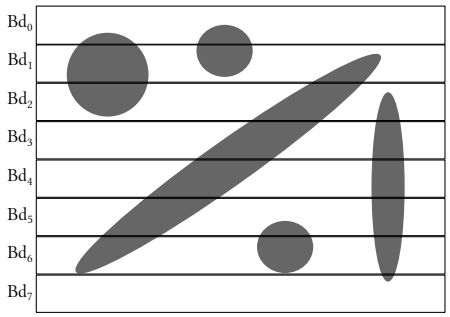


Fig. 6.9 – MPAR FB + SIMD + OMP : découpage en bandes

La contrepartie de ce découpage est qu'il diminue la vitesse de propagation des étiquettes en introduisant une barrière de synchronisation aux frontières des bandes. La propagation des étiquettes aux frontières étant réalisée lors du traitement de la première ligne (qui accède aux données de la bande précédente) pour la passe directe et de la dernière ligne (qui accède aux données de la bande suivante) lors de la passe inverse. De plus, selon les données contenues dans chaque bande, la charge peut-être déséquilibrée entre les cœurs (par exemple entre Bd_2 et Bd_7 dans la figure 6.9), alors que toutes les bandes doivent être traitées à chaque itération tant que l'image complète n'est pas stabilisée.

Cette version nous servira de référence en tant qu'algorithme utilisant à la fois le SIMD et le multi-cœur. Sa structure proche des algorithmes directs permettra aussi une comparaison rapide entre les deux classes d'algorithmes.

6.3.5 MPAR FB + SIMD + OMP + AT : Découpage en tuile et table d'activation

Afin de répartir la charge sur les différents cœurs, cette version découpe l'image en tuiles de largeur w_t et de hauteur h_t avec un ordonnancement en pile (cf. sec. 4.2.2.1). Le découpage en tuiles diminue encore plus la vitesse de propagation des étiquettes (et augmente ainsi le nombre d'itérations) que le découpage en bandes, en introduisant une barrière de synchronisation aux frontières de chaque tuile. Cependant, le procédé de tuiles actives permet de ne traiter une tuile que si elle ou ses voisines étaient instables à l'itération précédente. Cela entraîne, d'une part une diminution du nombre de pixels à traiter par itération, et d'autre part cela réduit les transferts en mémoire et diminue ainsi la dépendance de l'algorithme aux performances de la mémoire. Le coût de chaque itération est donc réduit.

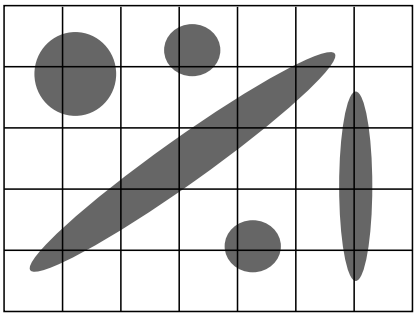


Fig. 6.10 – MPAR FB + SIMD + OMP + AT : découpage en tuiles actives

Un équilibre doit être trouvé dans la taille et la forme des tuiles, afin de maximiser la vitesse de propagation d'une part et d'obtenir une bonne répartition de la charge sur les différents cœurs d'autre part. L'impact de la taille et de la forme des tuiles sera évalué par la procédure d'étude des performances (chap.7).

Traiter chaque tuile jusqu'à sa stabilisation complète pourrait entraîner un déséquilibre de charge entre les cœurs. Afin d'éviter ce phénomène, le traitement de chaque tuile n'est composé que d'une passe directe et d'une passe inverse (Algo. 25). Si une modification de la tuile a eu lieu durant ces deux passes, alors la tuile est déclarée instable et devra être de nouveau traitée dans l'itération suivante.

Afin de gérer l'activation des tuiles, l'algorithme MPAR FB + SIMD + OMP + AT est doté d'une structure de données (matrice d'activation A) qui gère l'activation de la passe aller-retour (FB) dans chaque tuile. S'il n'est pas nécessaire de traiter une tuile, le cœur passe directement à la tuile suivante dans la pile. La matrice d'activation A de dimension $H/h_t \times W/w_t$ contient pour chaque tuile un indicateur de l'état de la tuile : 0 si la tuile est stable et un nombre strictement positif sinon. Afin de n'utiliser qu'une seule matrice tout en permettant la diffusion de l'information de l'activité entre tuiles voisines, nous utilisons deux bits pour encoder l'information de stabilité d'une tuile. Le premier pour la tuile elle-même et le second pour les tuiles voisines.

A la fin d'une passe de propagation (Algo. 26), il y a deux cas de figure :

- $(00)_b$: la tuile est stable,
- $(01)_b$: la tuile est instable.

Puis, l'information d'instabilité d'une tuile est propagée à ses voisines par dilatation morphologique (Algo. 27) amenant à 4 cas de figure :

- $(00)_b$: la tuile et ses voisines sont stables,

Algorithme 25 : Traitement d'une tuile $t(i_t,j_t)$ de coordonnées $[i_0,i_1] \times [j_0,j_1]$

```
1  flag ← (00)_b
2  for (i = i_0, i ≤ i_1 ; i++) do
3      for (j = j_0, j ≤ j_1 ; j++) do
4          a ← E(i − 1, j − 1), b ← E(i − 1, j)
5          c ← E(i − 1, j + 1), d ← E(i, j − 1)
6          x ← E(i, j)
7          x ← min+(a,b,c,d,x)
8          E(i,j) ← x
9          event ← x xor x ▹ Détection
10         flag ← flag or event ▹ Mémorisation
11 for (i = i_1, i ≥ i_0 ; i−) do
12     for (j = j_1, j ≥ j_0 ; j−) do
13         a ← E(i + 1, j + 1), b ← E(i + 1, j)
14         c ← E(i + 1, j − 1), d ← E(i, j + 1)
15         x ← E(i, j)
16         x ← min+(a,b,c,d,x)
17         E(i, j) ← x
18         event ← x xor x ▹ Détection
19         flag ← flag or event ▹ Mémorisation
20 return flag
```

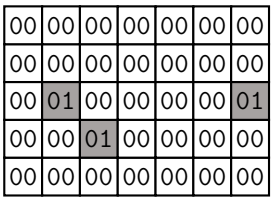


Fig. 6.11 – Exemple de matrice d'activation : trois tuiles ont été instables à l'itération précédente

- $(01)_b$: la tuile est instable et ses voisines sont stables,
- $(10)_b$: la tuile est stable et au moins une de ses voisines est instable,
- $(11)_b$: la tuile et au moins une de ses voisines sont instables.

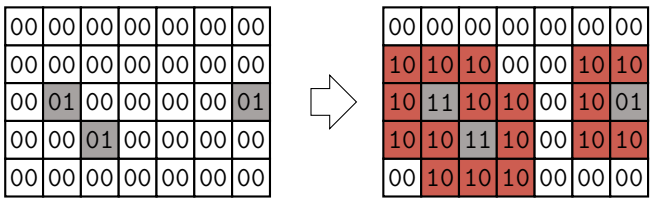


Fig. 6.12 – Diffusion de l'information d'instabilité aux tuiles voisines par dilatation morphologique

Dès qu'un des bits est à 1, la tuile doit être traitée. Au début de la procédure, toutes les tuiles sont actives et chaque case de A est donc à $(01)_b$. La procédure se poursuit jusqu'à stabilisation complète de l'image ($A = 0_{h_t \times w_t}$).

Algorithme 26 : Traitement des tuiles

```
1  foreach tile t(i_t,j_t) do
2      if A(i_t,j_t) ≠ (00)_b then
3          A(i_t,j_t) ← (00)_b
4          Traitement de t(i_t,j_t)
5          if t est instable then
6              A(i_t,j_t) ← (01)_b
```

Algorithme 27 : Diffusion de l'information d'instabilité

```
1  foreach cell A(i_t, j_t) do
2      if (A(i_t, j_t) and (01)_b) = (01)_b then
3          A(i_t − 1, j_t − 1) ← [A(i_t − 1, j_t − 1) or (10)_b]
4          A(i_t − 1, j_t + 0) ← [A(i_t − 1, j_t + 0) or (10)_b]
5          A(i_t − 1, j_t + 1) ← [A(i_t − 1, j_t + 1) or (10)_b]
6          A(i_t − 0, j_t − 1) ← [A(i_t − 0, j_t − 1) or (10)_b]
7          A(i_t − 0, j_t + 1) ← [A(i_t − 0, j_t + 1) or (10)_b]
8          A(i_t + 1, j_t − 1) ← [A(i_t + 1, j_t − 1) or (10)_b]
9          A(i_t + 1, j_t + 0) ← [A(i_t + 1, j_t + 0) or (10)_b]
10         A(i_t + 1, j_t + 1) ← [A(i_t + 1, j_t + 1) or (10)_b]
```

6.3.6 MPAR FB + SIMD + OMP + AT + MAX

Les algorithmes MPAR ont été construits en se basant sur la propagation du minimum positif comme pour les algorithmes directs. Cependant cette solution n'est pas unique. Le maximum du voisinage est lui aussi utilisable pour établir une relation d'ordre et il peut réduire le coût de l'opération de propagation. En effet, le calcul du minimum positif nécessite de tester chaque étiquette par rapport à 0 pour les intégrer dans le calcul du minimum, le calcul du maximum quant à lui ne nécessite aucun test supplémentaire car 0 est l'élément neutre pour le calcul du maximum (les étiquettes sont toutes positives). Pour les algorithmes directs, le remplacement du min^+ par le *max* n'apporte rien. En effet, l'arbre de décision nécessite de tester les étiquettes avant tout calcul de minimum (ou de maximum).

L'algorithme MPAR FB + SIMD + OMP + AT + MAX met en œuvre cette variante. Afin de garder un comportement proche de l'algorithme du calcul du min^+ il a été nécessaire d'inverser l'ordre d'initialisation ($E[i][j] \leftarrow H \times W + 1 - i * W - j$) afin que les étiquettes se propagent toujours dans le sens du balayage (gauche vers la droite et haut vers le bas pour le balayage direct et droite vers la gauche et bas vers le haut pour le balayage inverse). La version avec découpage en bandes MPAR FB + SIMD + OMP + MAX a aussi été testée et les résultats seront présentés au chapitre 7.

6.3.7 Implémentation

Les algorithmes ont été implémentés avec OpenMP. Pour les versions AT, OpenMP2 avec une boucle `#omp parallel for` a permis de paralléliser le traitement des tuiles en se basant sur une vue 1D de la matrice d'activation. Il est aussi possible d'utiliser OpenMP3 et les directives *tasking* ou OpenMP4 et les directives qui permettent d'activer les tâches en fonction de dépendances ou encore TBB qui implémente la structure *workpile* [113] qui correspond au modèle d'exécution.

Dans le chapitre 7, les performances de ces différentes versions seront évaluées.

6.4 Classe WARP

Les algorithmes MPAR apportent une réduction combinée du nombre d'itérations et du nombre de pixels à traiter. L'approche retenue pour les algorithmes de la classe WARP est de dépasser l'horizon de ce masque pour atteindre des vitesses de propagation plus élevées afin de réduire plus encore le nombre d'itérations.

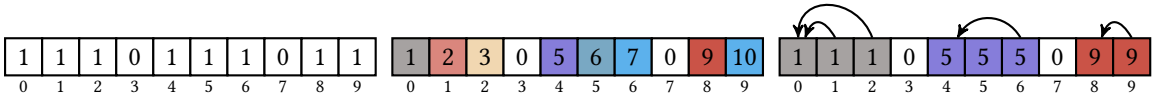
6.4.1 Principe

Les algorithmes de la classe WARP sont issus de l'analogie entre la propagation des étiquettes et celle d'une onde à partir d'une source (cf. sec. 6.2.2). Dans MPAR EP, les étiquettes se propagent en passant par les pixels du voisinage. Cette propagation est donc limitée par l'horizon du masque que ce soit pour la version MPAR EP ou la version MPAR FB. Dans le premier cas, cela limite la vitesse de propagation à 1 pixel par itération et dans le second cela implique une dépendance au sens de parcours et nécessite donc une passe en sens inverse et plusieurs itérations pour résoudre les cas plus complexes. Pour une spirale de largeur N, MPAR FB nécessite une itération par tour de spirale soit $nbiter = E(N/4) + 1$ soit 513 itérations pour une spirale 2048×2048. Par rapport à MPAR EP qui nécessite $2{,}1 \times 10^6$ itérations, le gain est spectaculaire. Cependant ce nombre reste trop élevé pour être rapide et compétitif avec les algorithmes directs.

La proposition des algorithmes de la classe WARP est d'étendre l'horizon de propagation au-delà du masque *via* des mécanismes complémentaires à l'utilisation du masque. Dans un premier temps, nous présenterons WARP dans sa version non récursive (basée sur MPAR EP) puis dans sa version récursive (basée sur MPAR F) et enfin nous présenterons une implémentation GPU dont les performances seront évaluées dans le chapitre 7.

6.4.2 Structure de graphe

L'initialisation utilisée pour MPAR (Algo. 23) permet de retrouver la position du pixel correspondant à une étiquette dans l'image des étiquettes (dans la suite l'appellation «indice du pixel» fera référence à cette position). Ainsi dans la figure 6.13, qui représente une image initialisée et la même image après étiquetage, il apparaît qu'il est possible d'atteindre l'étiquette source car la valeur d'une étiquette fait référence à l'indice du pixel unique qui contenait cette étiquette après l'initialisation. L'étiquette source 9 est la racine de la composante connexe 9.

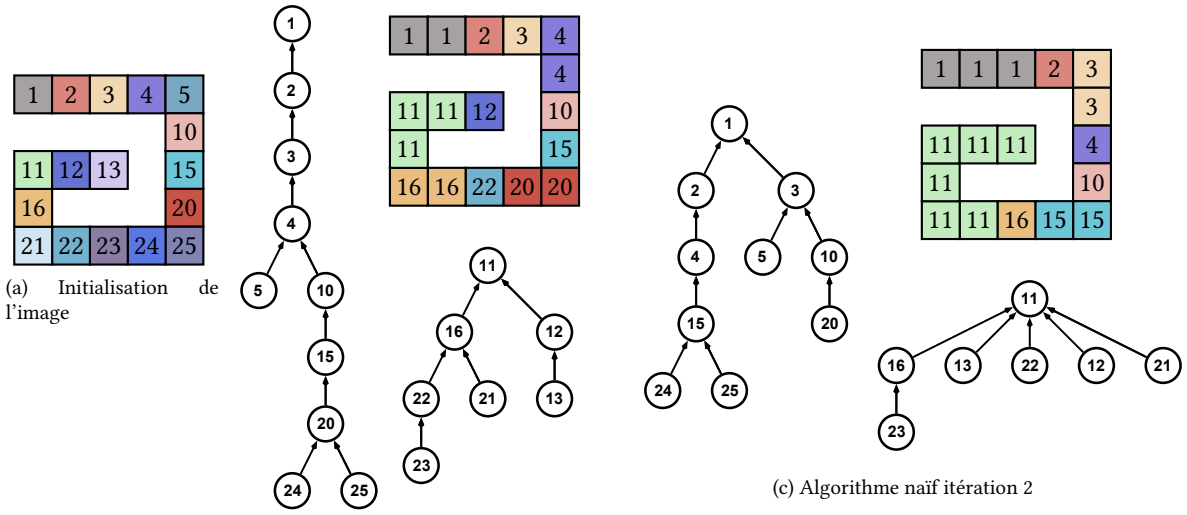


(a) Image binaire (b) Initialisation (c) Dualité image étiquetée / table d'équivalence

Fig. 6.13 – Accès au pixel d'origine d'une étiquette : l'étiquette d'un pixel après initialisation (gauche) informe sur sa position. L'image étiquetée est donc aussi une représentation du graphe de connexité.

Sans altérer la structure de l'image des étiquettes, cette représentation permet donc de doter l'image d'une structure de graphe comme le fait la table d'équivalence pour les algorithmes directs. Ici, l'image des étiquettes est confondue avec la table d'équivalences. Afin de clarifier les explications suivantes, nous ferons référence différemment à cet objet unique selon que nous souhaiterons décrire l'image des étiquettes $E[i][j]$ ou la table d'équivalence (indice du pixel racine) $I[i \times w + j + 1]$. Avec cette notation 1D, l'ancêtre de l'étiquette e est lu dans la case $e - 1$ de I [2] : $a \leftarrow I[e - 1]$.

2. Pour éviter de calculer $e - 1$ lors de chaque accès, il est possible de s'affranchir du décalage en modifiant le pointeur



(a) Initialisation de l'image

(b) Algorithme naïf itération 1

(c) Algorithme naïf itération 2

Fig. 6.14 – Décomposition de l'algorithme MPAR EP et représentation de sa structure de graphe associée, pour les 3 premières itérations

La figure 6.14 représente le déroulement de l'algorithme MPAR EP (identique à la figure 6.4) associé cette fois à la représentation de sa structure de graphe. On constate que l'algorithme itératif ne tire pas profit de l'information disponible dans l'image, ce qui limite sa vitesse de propagation. En effet, entre les itérations 1 et 2, le graphe se recompose, mais aucune exploitation de l'information contenue dans le graphe n'est réalisée.

6.4.3 Sous-composantes connexes et sous-graphes

Afin de simplifier les explications de cette section, nous définissons ici les termes que nous utiliserons dans la suite du manuscrit ainsi que leur cadre d'application.

- Les composantes connexes de l'image E ne sont connues qu'à la fin du processus itératif et I représente alors le graphe de connexité G des classes d'équivalence et chaque composante connexe possède une seule racine.
- Pendant le processus itératif, E_k représente l'état de connexion des sous-composantes connexes (que le processus itératif doit unir pour obtenir les composantes connexes de l'image). I_k représente alors un ensemble de sous-graphes temporaires de G correspondant chacun à une sous-composante connexe. Dans ce cadre, la racine d'un sous-graphe est la plus petite étiquette de la sous-composante connexe.
- Tant que l'étiquetage n'est pas terminé, le terme racine désignera la plus petite étiquette d'une sous-composante connexe.
- L'union de deux sous-composantes connexes se réalise en faisant pointer la racine d'une sous-composante vers la racine de l'autre.
- La fermeture transitive de E_k permet d'obtenir des sous-graphes de hauteur 2 en connectant chaque étiquette de la sous-composante à sa racine.

sur le début de l'image.

6.4.4 Algorithme WARP$_0$: fermeture transitive

WARP$_0$ est un algorithme itératif composé de trois étapes (sa portée est principalement pédagogique car il introduit le mécanisme de base de WARP : la fermeture transitive) :

- la même étape d'initialisation (Algo. 23) que pour les algorithmes MPAR,
- la même étape de propagation itérative du minimum positif dans E_k (Algo. 24) que pour les algorithmes MPAR,
- une étape supplémentaire de fermeture transitive de E_k (Algo. 28) réalisée entre chaque étape de propagation.

L'initialisation est faite une fois au début de l'algorithme puis les itérations se composent d'une passe de propagation suivie d'une passe de fermeture transitive.

Algorithme 28 : Fermeture transitive

Input : $E_k[H][W] \Leftrightarrow I_k[H \times W]$
Result : E_k mise à jour avec les racines des composantes connexes de chaque pixel

```
1 for i = 0 to H − 1 do
2     for j = 0 to W − 1 do
3         r ← E[i][j]
4         if r ≠0 then
5             while r ≠ I_k[r − 1] do
6                 r ← I_k[r − 1]
7             E[i][j] ← r
```

La passe de fermeture transitive de E permet d'exploiter l'information de connexité et d'atteindre alors une vitesse de propagation seulement limitée par la structure de l'image (notamment par la concurrence des sources/racines lors de la phase de propagation) et pas par celle du masque de voisinage.

La figure 6.15 représente le déroulement de l'algorithme WARP$_0$ pour la spirale 5×5. L'information contenue dans le graphe est exploitée par la fermeture transitive et permet d'atteindre en une itération un résultat équivalent à l'itération 7 de MPAR EP du point de vue de la propagation de l'étiquette 1. La vitesse de propagation n'est donc dans cette phase plus limitée et dépend uniquement de la structure de l'image et pas de celle du masque. C'est la concurrence des deux racines 1 et 11 due à l'agencement des pixels qui bloque la propagation. En effet, le point de E d'indice 23 bien que physiquement connecté aux deux racines ne peut contenir qu'une seule valeur et, dans son voisinage direct, c'est l'étiquette 22 (connectée à la racine 11) qui est le minimum positif.

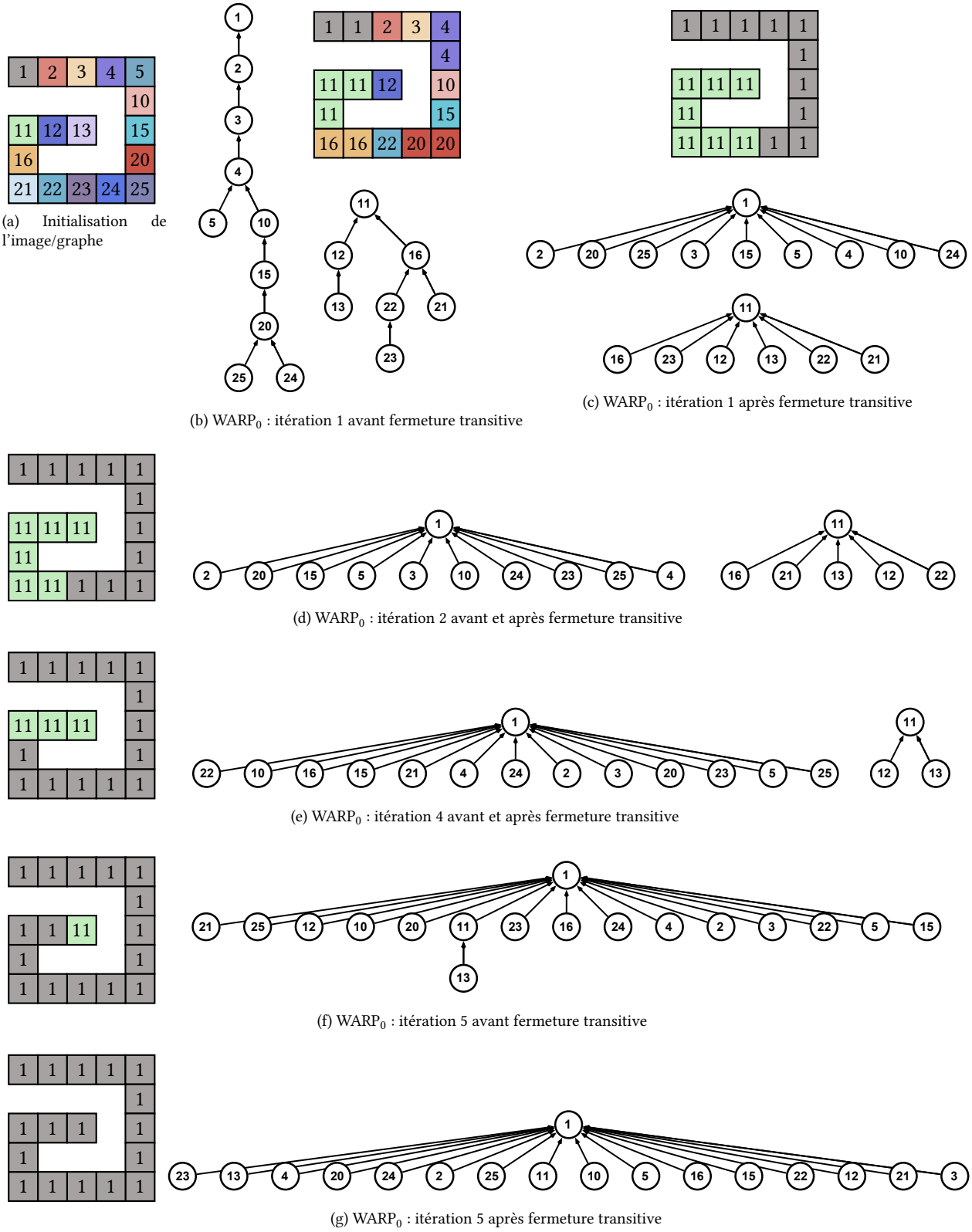
Après cette première itération, la vitesse de propagation retombe à 1 car les étiquettes rejoignent une à une la sous-composante connexe 1 (fig. 6.15d et fig. 6.15e) et la phase de fermeture transitive n'apporte plus rien. Seule l'itération 5 (fig. 6.15f) qui connecte la racine 11 à la racine 1 permet de propager l'étiquette 1 à tous les pixels liés à la racine 11 lors de fermeture transitive.

Avec ce mécanisme, le nombre d'itérations nécessaires à l'étiquetage de la spirale 5×5 passe à 6 pour WARP$_0$ contre 13 pour la version MPAR EP.

6.4.5 WARP : atteindre les sources

La section précédente a mis en évidence l'intérêt de la fermeture transitive ainsi que le fait qu'il est nécessaire d'écrire directement dans les sources pour unir les sous-graphes et ainsi profiter de cet avantage. C'est l'objet de l'algorithme WARP qui est un algorithme itératif composé de trois étapes :

- la même étape d'initialisation (Algo. 23) que pour les algorithmes MPAR,



(a) Initialisation de l'image/graphe

(b) $WARP_0$: itération 1 avant fermeture transitive

(c) $WARP_0$: itération 1 après fermeture transitive

(d) $WARP_0$: itération 2 avant et après fermeture transitive

(e) $WARP_0$: itération 4 avant et après fermeture transitive

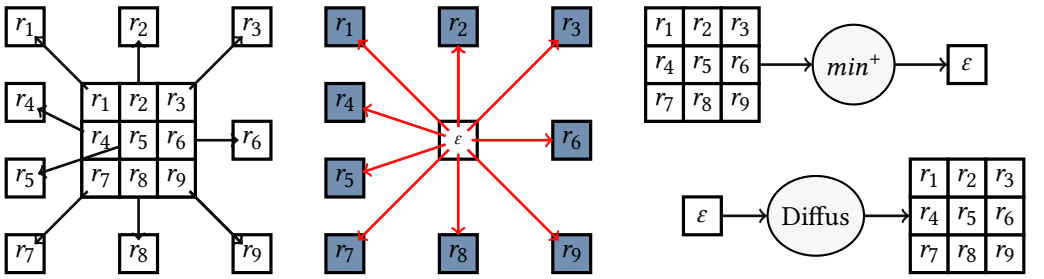
(f) $WARP_0$: itération 5 avant fermeture transitive

(g) $WARP_0$: itération 5 après fermeture transitive

Fig. 6.15 – Décomposition de l'algorithme $WARP_0$ et représentation du graphe associé (l'itération 3 a été omise car elle n'apporte pas d'information supplémentaire)

- une étape de diffusion itérative du minimum positif vers les racines des sous-composantes connexes présentes dans le voisinage (Algo. 30),
- la même étape de fermeture transitive que WARP_0 (Algo. 28).

Lors d'une phase de diffusion, les étiquettes présentes dans le voisinage ne sont pas nécessairement modifiées. En effet, une fois le minimum positif (ε) du voisinage calculé, il est affecté aux racines des sous-composantes connexes qui étaient présentes dans le voisinage. Cette étape repose entièrement sur la dualité entre E et I, les étiquettes contenues dans le voisinage étant utilisées une première fois pour leur valeur puis en tant qu'indice (adresse) des racines. Tout se passe comme si le masque de propagation avait été renversé (fig. 6.16 et algo. 30) pour obtenir le masque de diffusion.



(a) Masque du calcul du minimum ε, chaque étiquette du voisinage est aussi l'index de la racine de son sous-graphe

(b) Diffusion du minimum **vers** les racines des sous-graphes connectés au voisinage

(c) Point de vue producteur consommateur : le minimum positif utilise 9 chargements pour produire ε (haut), puis ε est diffusé aux 9 racines (maximum) des sous-composantes connexes (bas)

Fig. 6.16 – Masque algorithme WARP : on utilise le fait que les étiquettes du voisinage sont aussi les indices des racines de leurs sous-composantes connexes (gauche) pour diffuser le minimum local à ces racines (centre).

Algorithme 29 : SetRoot

Input : $I_{k+1}[H \times W]$, e une étiquette cible, ε la valeur à écrire
Result : $I[H \times W]$ avec le contenu de e mis à jour

```
1 v ← I_{k+1}[e − 1]
2 if v > ε then
3     I[e − 1] ← ε
```

C'est une procédure d'union des composantes connexes semblable à la procédure Union-Find des algorithmes directs. Tout comme pour la table d'équivalence des algorithmes directs, une seule connexion peut-être représentée dans I. Cependant plusieurs voisinages peuvent tenter de modifier la même racine successivement au sein d'une même itération et il est donc nécessaire de faire un choix et d'ignorer (temporairement) l'information de connexion entre deux sous-composantes connexes. Le choix fait dans WARP est le même que celui des algorithmes directs : afin de conserver la relation d'ordre, c'est la plus petite racine qui l'emporte (procédure *SetRoot* algo. 29).

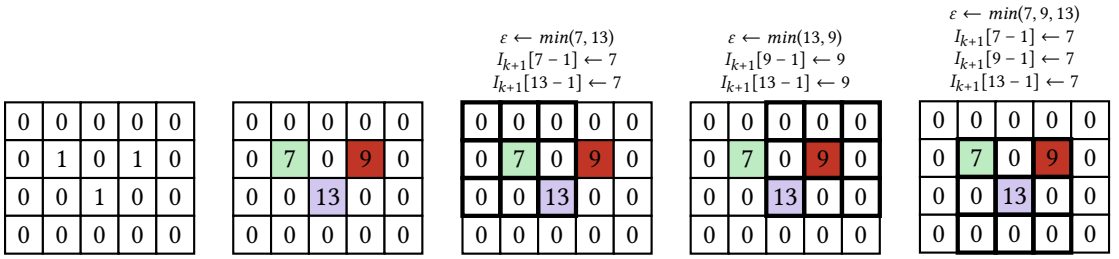
Dès l'itération suivante, lorsque les voisinages dont l'information a été ignorée seront de nouveau traités, l'information de connexion est de nouveau évaluée. Le processus itératif continuera jusqu'à ce que toutes les informations de connexion soient prises en compte et que les composantes connexes soient toutes complètement étiquetées.

L'exemple de la figure 6.17 est résolu en une itération mais il met en évidence le phénomène de choix qui est fait lors du traitement de chaque voisinage.

Algorithme 30 : Diffusion du min^+ aux racines

Input : $E_k[H][W] \Leftrightarrow I_k[H \times W]$
Result : $E_{k+1} \Leftrightarrow I_k$ avec les étiquettes connexes diffusées aux racines

```
1 for i = 0 to N − 1 do
2     for j = 0 to M − 1 do
          ▷ Chargement des pixels du voisinage
          r1 ← Ek[i − 1][j − 1]   r2 ← Ek[i][j − 1]   r3 ← Ek[i + 1][j − 1]
3         r4 ← Ek[i − 1][j]       r5 ← Ek[i][j]       r6 ← Ek[i + 1][j]
          r7 ← Ek[i − 1][j + 1]   r8 ← Ek[i][j + 1]   r9 ← Ek[i + 1][j + 1]
4         ε ← min+(r1, . . . , r9)
5         foreach rk do
6             if rk ≠ 0 then
7                 SetRoot(Ik+1, rk, ε)        ▷ La valeur ε est affectée au point de I désigné par l'étiquette rk
```



(a) B, l'image binaire (b) E_0, l'image initialisée (c) traitement de $E_0[1,1]$ (d) traitement de $E_0[1,3]$ (e) traitement de $E_0[2,2]$

Fig. 6.17 – Phase de diffusion : chaque voisinage tente d'écrire dans les racines issues de son voisinage. Ici les opérations du traitement de $E[1,3]$ (d) sont ignorées car la racine 7 l'emporte sur la racine 9 (9 et 7 sont en concurrence). Le traitement de $E[2,2]$ connecte 9 à 7 et regroupe les deux sous-graphes.

Par rapport au traitement de la même image avec un algorithme direct, WARP nécessite beaucoup plus d'instructions pour arriver au même résultat et ne semble pas efficace. C'est par rapport au mécanisme itératif qu'il faut considérer ses performances. En effet, dans un contexte massivement parallèle, où les dépendances de données vont ralentir les algorithmes directs, le mécanisme WARP, va permettre en ayant recours à des instructions `Atomic`, de réaliser la diffusion sans dépendance de données. Le mécanisme itératif permet alors de s'assurer par la répétition de ces opérations élémentaires d'obtenir une image correctement étiquetée.

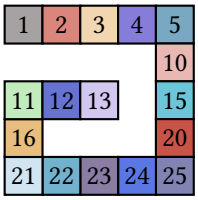
Ainsi, avec le mécanisme WARP, la spirale (fig. 6.18) est étiquetée en deux itérations (2 propagations + 2 remontées aux racines) à comparer avec les 6 itérations de WARP_0 et les 13 de MPAR EP. Pour une spirale de largeur 2048, le nombre d'itérations ne change pas et vaut toujours 2 à comparer aux $2{,}1 \times 10^6$ itérations nécessaires à MPAR EP et au 513 de MPAR FB.

Le nombre d'itérations nécessaires à l'algorithme WARP est tout de même dépendant de la structure de l'image. En effet, bien qu'affranchi de la distance géodésique, un autre phénomène devient prépondérant : la concurrence des sources.

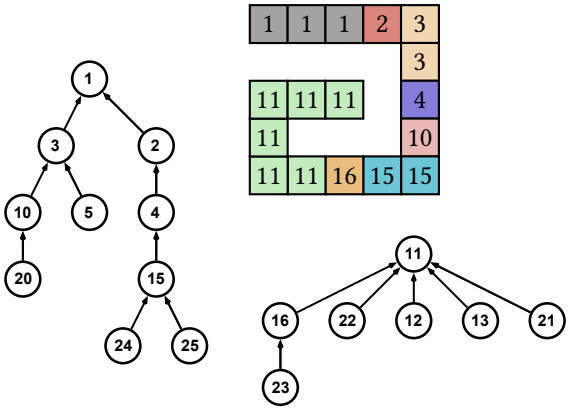
6.4.6 Concurrence des sources

La concurrence des sources peut survenir par la combinaison de plusieurs situations :

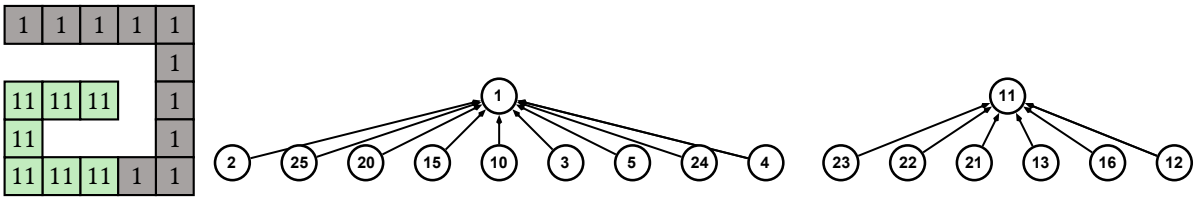
- si, trois sous-composantes connexes sont présentes dans un même voisinage,
- si, une sous-composante connexe est reliée à d'autres sous-composantes connexes par l'intermédiaire de plusieurs voisinages.



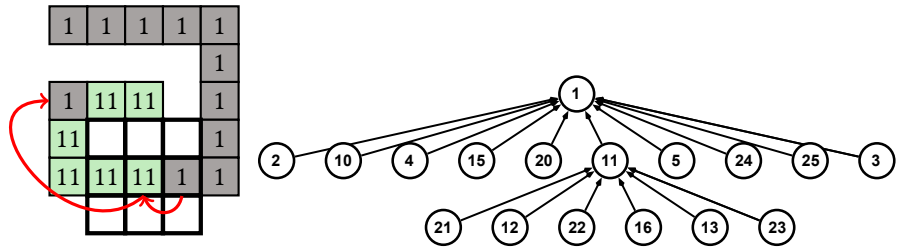
(a) Initialisation de l'image/graphe



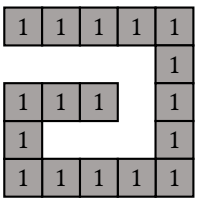
(b) Graphe algorithme WARP itération 1 avant la remontée à la racine



(c) Graphe algorithme WARP itération 1 après la remontée à la racine



(d) Graphe algorithme WARP itération 2 avant la remontée à la racine : dans le voisinage de $E_1[5,3]$, 11 et 1 sont connectés, la valeur 1 est donc écrite dans le point d'indice 11

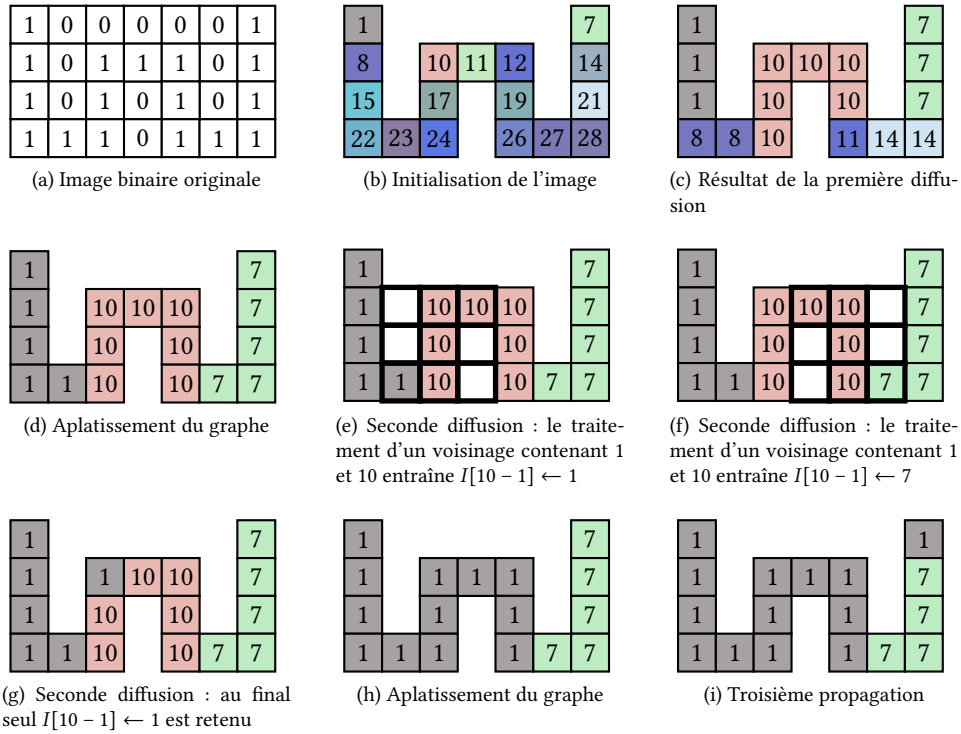


(e) Graphe algorithme WARP itération 2 après la remontée à la racine

Fig. 6.18 – Décomposition de l'algorithme Warp du point de vue graphe

Dans les deux cas, un point de E ne pouvant faire référence qu'à une étiquette à la fois, seule l'information de connexité vers la plus petite des racines est prise en compte et une itération supplémentaire est nécessaire pour prendre en compte la connexion avec les autres sous-composantes connexes. Le mécanisme d'union défini par *SetRoot* est donc incomplet par nature et n'est pas une implémentation correcte de la procédure Union-Find.

En effet, pour l'algorithme non récursif, les étiquettes étant lues dans E_k et écrites dans E_{k+1}, il n'est pas possible de tenir compte des connexions rencontrées précédemment par les autres voisinages avant le calcul du minimum positif et plusieurs itérations sont donc nécessaires. La figure A.17 présente une structure qui nécessite une itération de plus que la spirale (la version complète ainsi que le graphe associé sont reportés en annexe A.4). La racine 1 et la racine 7 sont en concurrence pour être la racine de l'étiquette 10 et seule la connexion à 1 est conservée (fig. 6.19g).



(a) Image binaire originale
(b) Initialisation de l'image
(c) Résultat de la première diffusion
(d) Aplatissement du graphe
(e) Seconde diffusion : le traitement d'un voisinage contenant 1 et 10 entraîne $I[10-1] \leftarrow 1$
(f) Seconde diffusion : le traitement d'un voisinage contenant 1 et 10 entraîne $I[10-1] \leftarrow 7$
(g) Seconde diffusion : au final seul $I[10-1] \leftarrow 1$ est retenu
(h) Aplatissement du graphe
(i) Troisième propagation

Fig. 6.19 – Exemple d'une structure à coupure de graphe

L'intérêt de considérer WARP du point de vue non récursif est que sa transposition à un contexte massivement parallèle est directe. En effet, si l'algorithme est valide en utilisant deux images distinctes pour les données de départ et les données d'arrivée, il le sera lorsque l'ordre des mises à jour par un grand nombre d'unités d'exécution sera inconnu.

Du fait de la concurrence des sources, bien que le nombre d'itérations diminue par rapport à WARP_0, le nombre d'accès en écriture qui est potentiellement multiplié par 9 du fait de l'inversion du masque entraîne une augmentation du temps de traitement de chaque itération. Il est donc nécessaire de corriger ce phénomène pour diminuer le nombre d'itérations et s'assurer que chaque écriture en mémoire sera utile. Un mécanisme d'union valide est donc nécessaire.

6.4.7 WARP Union : WARP + mécanisme d'union valide

L'algorithme WARP Union est un algorithme trois passes (non itératif), composé de :

- la même étape d'initialisation (Algo. 23) que pour les algorithmes MPAR,

- la procédure de diffusion de l'algorithme WARP améliorée avec un mécanisme récursif d'union des racines (Algo. 31),
- la même étape de fermeture transitive que WARP_0 (Algo. 28).

Dans le cas de WARP et du mécanisme *SetRoot*, si aucun correctif n'est envisageable avant le calcul du minimum positif, il est tout de même possible d'écrire un mécanisme d'union valide (*Union* - algo. 32) proche de l'algorithme *SetRoot*. Pour cela, il faut agir après le calcul du minimum positif lors de l'écriture dans E_{k+1}. En effet, lorsqu'à la suite du calcul du minimum positif, $Union(e_k, \varepsilon)$ est exécuté, quatre cas de figures sont possibles :

- le pixel e_k contient effectivement l'étiquette e_k, il s'agit donc bien d'une racine et on peut changer sa valeur par ε,
- le pixel e_k contient une étiquette e_r inférieure à e_k (par construction si la valeur est différente elle ne peut-être qu'inférieure) et cette valeur est supérieure à ε, il faut alors exécuter $Union(e_r, \varepsilon)$,
- le pixel e_k contient une étiquette e_r inférieure à e_k et cette valeur est inférieure à ε, il faut alors exécuter $Union(\varepsilon, e_r)$
- le pixel e_k contient une étiquette e_r inférieure à e_k et cette valeur est égale à ε, aucune action n'est alors nécessaire.

La fonction *Union* est donc récursive et permet d'empêcher les coupures de graphe. Bien que coûteux en nombre d'accès par pixels, l'algorithme WARP Union composé de l'algorithme WARP muni de la procédure *Union* en remplacement de la procédure *SetRoot* est un algorithme capable d'étiqueter toutes les images en deux passes : une passe de propagation et une passe de réétiquetage tout comme les algorithmes directs.

Algorithme 31 : Diffusion du min^+ aux racines

Input : $E_k[H][W] \Leftrightarrow I_k[H \times W]$
Result : $E_{k+1} \Leftrightarrow I_k$ avec les étiquettes connexes diffusées aux racines

```
1 for i = 0 to N − 1 do
2     for j = 0 to M − 1 do
          ▷ Chargement des pixels du voisinage
          r1 ← Ek[i − 1][j − 1]   r2 ← Ek[i][j − 1]   r3 ← Ek[i + 1][j − 1]
3         r4 ← Ek[i − 1][j]       r5 ← Ek[i][j]       r6 ← Ek[i + 1][j]
          r7 ← Ek[i − 1][j + 1]   r8 ← Ek[i][j + 1]   r9 ← Ek[i + 1][j + 1]
4         ε ← min+(r1, . . . , r9)
5         foreach rk do
6             if rk ≠ 0 then
7                 SetRoot(Ik+1, rk, ε)        ▷ La valeur ε est affectée au point de I désigné par l'étiquette rk
```

La figure 6.20 illustre le traitement de la spirale 5×5 en une seule itération composée de deux passes : l'étape de diffusion et l'étape de fermeture transitive qui tient lieu de réétiquetage.

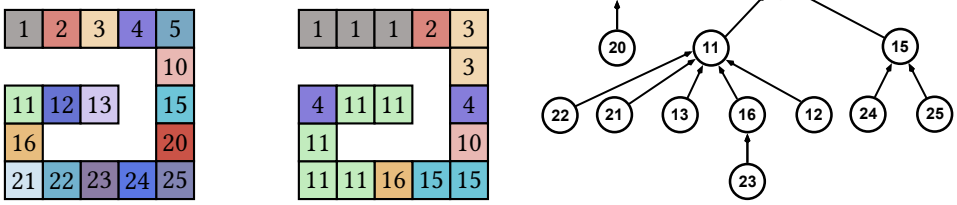
6.4.8 WARP CPU

Nous avons jusqu'ici envisagé la classe d'algorithmes WARP dans un cadre où l'image à l'étape k et et à l'étape $k + 1$ étaient dissociées. Le but de cette séparation est de pouvoir concevoir les mécanismes de concurrence sans avoir à prendre en compte l'ordre d'exécution des opérations d'étiquetage. Cette représentation a permis de créer une série de mécanismes qui seront utilisés dans le cadre des architectures massivement parallèles disposant d'instructions `Atomic`.

Algorithme 32 : Procédure récursive Union(e_k,ε)

Données : I_1 en cours de mise à jour, e_k et ε deux étiquettes valides telles que $e_k > \varepsilon$
Résultat : I_1 avec les composantes connexes de a et ε connectées sans rupture du graphe

```
1 e_r ← I[e_k − 1] ▷ e_k est il une racine ?
2 if e_r = e_k then
3 |   E[e_k − 1] ← ε
4 else
5 |   if e_r > ε then
6 |   |   Union(e_r,ε)
7 |   else
8 |   |   if e_r < ε then
9 |   |   |   Union(ε,e_r)
```



(a) Initialisation de l'image/-graphe

(b) E_1 et représentation du graphe correspondant pour l'algorithme WARP Union

Fig. 6.20 – Résolution de la spirale en une itération avec WARP Union : lors du traitement du pixel d'indice 23, l'étiquette 24 est reliée à l'étiquette 22, ce qui relie la racine 11 à l'étiquette 4 et complète le graphe

Il est tout à fait possible d'envisager les mécanismes WARP dans un cadre récursif (au sens d'Haralick & Shapiro - travail «en place» dans une seule image).

Dans ce cas, on distingue deux cas :

- le cas séquentiel : WARP CPU est alors très proche de l'algorithme de Rosenfeld avec la table d'équivalence remplacée par l'image elle-même. Le mécanisme SetRoot est alors suffisant s'il est précédé d'une remontée systématique aux racines. Dans la version WARP précédente, cette étape était réalisée par la fermeture transitive.
- le cas parallèle où le mécanisme Union tient le rôle de mécanisme de synchronisation.

L'intérêt envisageable de WARP CPU par rapport aux algorithmes directs classiques est qu'aucune table supplémentaire n'est requise et que le compteur d'étiquettes *ne* n'est plus nécessaire. Cependant la table d'équivalence confondue avec E est bien plus grande et les accès à celle-ci profiteront moins de la mémoire cache. Des travaux ultérieurs à la thèse établiront la pertinence d'une telle approche.

Un phénomène particulier à WARP CPU est la dualité fermeture transitive / réétiquetage. Ces deux opérations sont en fait identiques dans le cas de WARP du fait de la dualité E/I.

La parallélisation de WARP CPU peut s'envisager en deux parties sur le modèle du chapitre 4 :

- tout d'abord un étiquetage des bandes de l'image en parallèle. Dans ce cas, le découpage de

l'image réalise en même temps le découpage de la table d'équivalence et l'opération est transparente,
- puis, soit une union pyramidale des bandes utilisant *SetRoot*, soit une étape d'union parallèle des bandes en utilisant le mécanisme *Union*.

En utilisant le mécanisme *Union*, un découpage en tuiles est lui aussi possible.

Ces approches seront étudiées et comparées dans des travaux ultérieurs à la thèse. Il est déjà possible d'établir que, dans sa version récursive, WARP est un algorithme pixel et que même s'il peut potentiellement être plus efficace que les autres algorithmes pixels, il sera limité par le traitement pixel à pixel et ne remet pas en cause les travaux sur LSL_{RLE}. Son utilisation sera très indiquée dans un contexte où la mémoire est limitée ou encore sur des architectures *many-core* avec des mémoires séparées comme c'est le cas pour TSAR [114].

6.4.9 WARP GPU

Pensé pour les architectures parallèles, les mécanismes WARP peuvent être implémentés sur de nombreuses architectures. La première implémentation que nous avons développée est WARP GPU, car les GPU :

- donnent accès à une instruction *AtomicMin* performante, qui va permettre la réalisation des procédures SetRoot et Union,
- sont composés d'un grand nombre d'unités de calcul (2816 dans le cas de notre carte de référence - cf. 7),

De plus, le modèle de programmation des GPU qui traite les données en blocs au sein desquels les instructions sont exécutées par groupe de threads contigus (warp) ne sont pas adaptés aux algorithmes directs. Par exemple, la divergence entre threads d'un même warp n'étant pas possible, un code utilisant une structure `if-then-else` est sérialisé.

L'algorithme WARP a donc le potentiel pour être un algorithme d'étiquetage adapté aux architectures GPU.

6.4.9.1 Travaux antérieurs

Plusieurs travaux traitant de l'implémentation d'algorithmes d'étiquetage en composantes connexes sur GPU ont été présentés entre 2009 et 2011 :

- En 2009 dans [115], les auteurs ont présenté les performances de différents algorithmes impliqués dans la vidéo surveillance. Parmi ceux-ci, l'étiquetage en composantes connexes est réalisé par une méthode de type *Divide and conquer* programmée avec CUDA et exécutée sur une carte GTX280 (architecture Tesla - 240 cœurs). Pour une image 1600×1200 le temps d'étiquetage (sans prendre en compte les temps de chargement et de restitution de l'image) était de 2,649ms.
- En 2010 dans [116], les auteurs ont présenté deux algorithmes 4C : un algorithme itératif peu performant faisant alternativement la diffusion par colonne et par ligne ainsi qu'une version modifiée de l'algorithme «Label Equivalence» qui, après analyse, s'est révélé identique dans le principe à WARP_0. Les algorithmes étaient programmés avec CUDA et exécutés sur une carte TESLA C1060 (architecture Tesla et caractéristiques proches de la GTX280 - 240 cœurs). Pour une image aléatoire 2048×2048 de densité 60% et de granularité non renseignée, le temps d'étiquetage (sans prendre en compte les temps de chargement et de restitution de l'image) était de 39,34ms.

- En 2010 dans [117], les auteurs ont comparé deux algorithmes 4C : Label Equivalence et une version GPU de la procédure Union-Find qui comporte une procédure équivalente à la procédure Union de l'algorithme WARP Union. Sur les images de la base Berkeley Segmentation Dataset[118] de taille 481×321 en utilisant une carte GTX285 (architecture Tesla - 240 cœurs) le temps de traitement des images était de 173ms pour Label Equivalence et 185ms pour la version Union-Find. L'analyse du code source indique que les temps de transfert ont été pris en compte dans ces résultats. La très petite taille des images étudiées semble peu adaptée à des architectures de type GPU et ne permet pas de rendre compte des performances de l'algorithme.
- En 2011 dans [119], les auteurs ont présenté un algorithme 8C itératif pyramidal en quatre phases : étiquetage itératif d'une tuile, mise à jour des étiquettes de bords de tuiles, fusion des tuiles par blocs de 4×4 de manière pyramidale en utilisant une procédure équivalente à *SetRoot* et enfin réétiquetage de l'image. Sur une GTX 480 (architecture Fermi - 480 cœurs) avec des images de taille 2048×2048 et de densité moyenne mais inconnue, le temps d'étiquetage (sans prendre en compte les temps de chargement et de restitution de l'image) était de 5,7ms (735 Mp/s) et pour une densité très faible (et inconnue) 2,4ms (1740Mp/s).

Pour toutes ces contributions, l'absence de données reproductibles rend difficile la comparaison entre algorithmes. Dans le chapitre 7, nous confronterons l'algorithme WARP GPU au jeu de données présenté au chapitre 2 pour proposer une référence permettant une comparaison directe avec des algorithmes développés ultérieurement.

Ces articles ont tous comparé leurs résultats à une version séquentielle et non optimisée d'un algorithme direct, ce qui les a amené à conclure que les versions GPU étaient plus rapides. Cependant les temps présentés pour les algorithmes directs étaient très élevés comparativement à ceux obtenus dans nos travaux et ne tenaient pas compte de la possibilité de paralléliser les algorithmes directs. De plus, la plupart des articles ont exclu les temps de chargement et de récupération de l'image rendant la comparaison caduque.

Pour comparer les performances entre architectures CPU et GPU, les temps de chargement et de récupération de l'image vers et depuis la mémoire du GPU doivent être pris en compte. Cependant, s'il est nécessaire d'étiqueter à la suite plusieurs images, il est possible de réaliser les calculs pendant les phases de transfert et de masquer ainsi le coût des communications ou celui des calculs selon la longueur de chacun. Dans le cadre de nos travaux, nous n'avons pas réalisé cette optimisation car nous nous plaçons dans une démarche différente qui consiste à proposer un algorithme dédié à l'utilisation sur GPU. En effet, comme indiqué dans le chapitre 1, l'étiquetage en composantes connexes est un élément d'un chaîne et il est probable que dans le cadre de cette chaîne, l'image soit déjà sur le GPU pour la phase de segmentation et que les données de sortie du GPU soient des descripteurs (cf. 3). L'étude des performances sans tenir compte des temps de chargement et de récupération prennent alors un sens.

6.4.9.2 Déroulement général de WARP GPU

En se basant sur les différents mécanismes décrits dans les sections précédentes, beaucoup de variantes d'algorithmes sont possibles. Nous présentons ici celle qui a donné les meilleurs résultats.

Notre proposition est d'étiqueter l'image en 4 séquences de traitement (*kernel*) sur le GPU :

- kernel Diffusion : étiqueter l'image par morceaux indépendants (tuiles) dont la taille est un paramètre de l'algorithme,
- kernels Union S et Union E : unir toutes les tuiles par leurs bords (Sud et Est),
- kernel Réétiquetage : réétiqueter l'image.

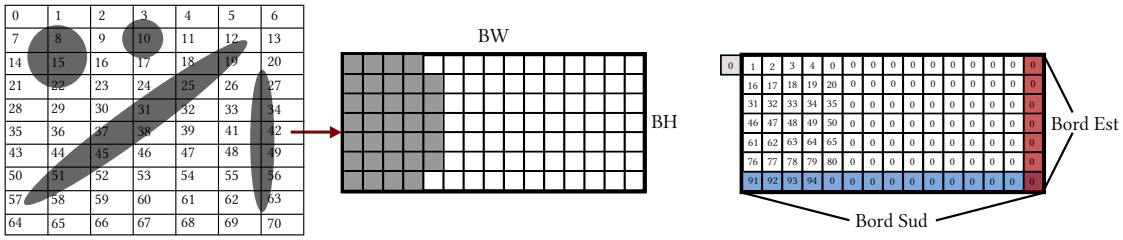
Chaque kernel est exécuté successivement. Ce découpage permet d'optimiser la configuration indépendamment pour chaque partie. Ainsi, la taille des tuiles, le nombre de points de bords traités

par bloc de calcul et la taille des blocs de calcul pour le réétiquetage ont chacun fait l'objet d'une recherche d'optimum.

6.4.9.3 Kernel Diffusion : étiquetage des tuiles

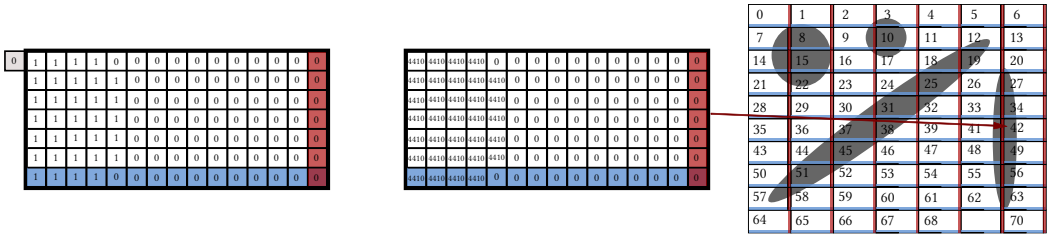
Le découpage en tuiles est régulièrement utilisé dans le cadre des applications de traitement d'images sur GPU du fait de leur structure de calcul en blocs et de la disponibilité d'une mémoire partagée rapide (mémoire *shared*).

Découpage L'image est découpée en tuiles de largeur BW et de hauteur BH. Pour chaque pixel (x,y) non nul de l'image à étiqueter, le pixel (i,j) correspondant dans la tuile se voit attribuer l'index $i \times w + j + 1$ en mémoire *shared*. Chaque tuile est donc dotée d'un graphe local qui permet l'étiquetage autonome de la tuile. À la fin du processus d'étiquetage, il est nécessaire de traduire ces coordonnées locales en coordonnées globales.



(a) L'image binaire $B[H][W]$ est découpée en tuiles de hauteur BH et de largeur BY (ici la tuile 42). La taille optimale des tuiles devra être établie par une recherche exhaustive.

(b) Chaque tuile se voit attribuer une portion de mémoire *shared* de taille $BH \times BW + 1$, la première case représente l'étiquette 0



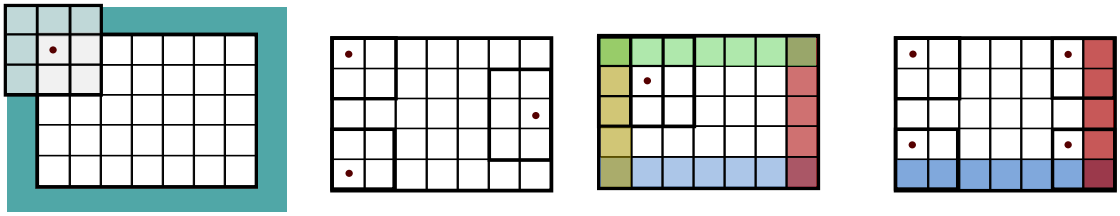
(c) La tuile est étiquetée localement en mémoire *shared*

(d) Une fois la tuile étiquetée, les étiquettes locales sont converties en étiquettes globales et transférées dans l'image globale des étiquettes

Fig. 6.21 – Découpage WARP GPU : pour profiter des performances de la mémoire partagée rapide (*shared*), l'image binaire est découpée en tuiles puis les tuiles étiquetées sont écrites en mémoire globale

Voisinage 2×2 Le voisinage 3×3 s'applique en tout point du premier plan et permet de propager les étiquettes dans toutes les directions de l'espace mais présente deux inconvénients dans le cas de l'utilisation sur GPU :

- il nécessite d'accéder au contenu des tuiles voisines, ce qui introduit une dépendance de données (fig. 6.22a),
- à défaut, il faut changer la taille du masque pour la gestion des bords et des coins, ce qui est très coûteux sur un GPU pour lequel il est plus efficace de minimiser la divergence d'exécution entre threads (fig. 6.22b),
- à défaut, il faut charger plus d'étiquettes que de threads ou désactiver les threads correspondants aux bords (Nord, Est, Sud, Ouest) (fig. 6.22c).



(a) Si tous les points de la tuile sont traités en tant que point central du masque, il est nécessaire d'accéder aux étiquettes en dehors de la tuile (mémoire globale)

(b) A défaut, si tous les points de la tuile sont traités en tant que point central du masque, il est nécessaire de faire varier la taille et la forme du masque pour gérer les bords

(c) Si on accepte de ne pas traiter tous les points comme point central du masque, alors un nombre de PE égal au périmètre de la tuile sera inutilisé

(d) Avec un masque 2×2, seuls les bords Est et Sud ne sont pas traités comme point «central» du masque et le nombre de PE en attente est égal à la moitié du périmètre

Fig. 6.22 – WARP GPU masque : le voisinage 2×2 génère moins de cas particuliers ou d'opérations supplémentaires que le voisinage 3×3

Pour l'application GPU, nous avons décidé d'utiliser un voisinage 2×2 qui ne nécessite que la prise en compte spécifique des bords Sud et Est afin de ne pas accéder aux tuiles voisines (fig. 6.22d). Cette solution a aussi l'avantage de réduire le nombre de chargements et d'écritures en mémoire qui passe de 9 points chargés et 9 points écrits (au maximum) à 4 points chargés et 4 points écrits (au maximum).

Fig. 6.23 – WARP GPU : masque 2×2

La gestion des bords Est et Sud est réalisée en désactivant les threads en charge de ces bords. C'est le mécanisme de propagation aux racines voisines qui se charge d'étiqueter les bords lors du traitement des lignes et colonnes précédentes. Cela signifie qu'il faut réaliser l'opération de diffusion du minimum aussi bien pour les points du premier plan que pour ceux de l'arrière plan. Cette condition est de toute façon nécessaire pour le masque 2×2 en 8C (cf. RCM chap. 2).

Étapes d'étiquetage Bien que la procédure «WARP Union» permette d'obtenir la tuile étiquetée en une itération, la procédure récursive *Union* nécessite un nombre indéterminé de chargements aléatoires et n'est pas efficace pour le premier étiquetage où toutes les étiquettes sont des racines de sous-composantes connexes. Il est donc pertinent de le réserver aux phases avancées du traitement. Les meilleurs résultats ont été obtenus en réalisant une première passe de l'algorithme WARP (utilisant SetRoot) suivi d'une fermeture transitive et d'une passe de l'algorithme WARP Union. La première passe et la fermeture transitive associée ayant diminué le nombre de sous-composantes connexes présentes dans la tuile, WARP Union permet d'obtenir efficacement la tuile étiquetée en une seule passe supplémentaire. La procédure appliquée pour WARP GPU est donc :

- chargement de la portion de l'image binaire correspondant à la tuile et initialisation dans le référentiel local (Algo. 33),
- premier étiquetage par l'algorihme WARP 2×2 (Algo. 38),
- fermeture transitive (Algo. 35),
- second étiquetage par WARP Union 2×2 (Algo. 37),
- fermeture transitive.

Algorithme 33 : WARP GPU - Initialisation de la tuile

Données : B l'image binaire en mémoire globale (en adressage 1D), (i, j) l'index du thread courant dans le bloc, (BW, BH) les dimensions de la tuile, $(BId.x, BId.y)$ l'index du bloc courant, la table $tuile[BW \times BH + 1]$ en mémoire *shared* accessible à tout le bloc

Résultat : La table $tuile[BW \times BH + 1]$ initialisée

$l \leftarrow i \times BW + j + 1$ ▷ indice local dans le référentiel de la tuile
$g \leftarrow (BId.x \times BH + i) \times W + (BId.y \times BW + j + 1)$ ▷ indice global dans le référentiel de l'image
if $l = 1$ **then**
    $tuile[0] \leftarrow 0$ ▷ La case 0 de tuile[] doit pointer vers 0 - Le thread 0 de chaque bloc est chargé de l'initialisation
$a \leftarrow B[g]$
if a **then**
    $tuile[l] \leftarrow l$
else
    $tuile[l] \leftarrow 0$

Algorithme 34 : WARP GPU - Passe WARP

Données : la table $tuile[BW \times BH + 1]$ en mémoire *shared* accessible à tout le bloc, (i, j) l'index du thread courant dans le bloc, (BW, BH) les dimensions de la tuile

Résultat :

▷ Calcul des indices des pixels du voisinage
$l_a \leftarrow i \times BW + j + 1$
$l_b \leftarrow l_a + 1$
$l_c \leftarrow l_a + BW$
$l_d \leftarrow l_a + BW + 1$
▷ Chargement du voisinage : *tuile* est la table représentant la tuile en mémoire shared
$a \leftarrow tuile[l_a]$
$b \leftarrow tuile[l_b]$
$c \leftarrow tuile[l_c]$
$d \leftarrow tuile[l_d]$
▷ Calcul du minimum du voisinage
$\varepsilon \leftarrow min^+(a,b,c,d)$
▷ Diffusion du minimum aux racines des sous-composantes connexes présentes dans la tuile
if a **and** $(a > \varepsilon)$ **then** $SetRoot(\&tuile[l_a], \varepsilon)$
if b **and** $(b > \varepsilon)$ **then** $SetRoot(\&tuile[l_b], \varepsilon)$
if c **and** $(c > \varepsilon)$ **then** $SetRoot(\&tuile[l_c], \varepsilon)$
if d **and** $(d > \varepsilon)$ **then** $SetRoot(\&tuile[l_d], \varepsilon)$

Algorithme 35 : WARP GPU - Fermeture transitive de la tuile

Données : (i, j) l'index du thread courant dans le bloc, la table $tuile[BW \times BH + 1]$ en mémoire *shared* accessible à tout le bloc

Résultat : La table $tuile[BW \times BH + 1]$ mise à jour

$l \leftarrow i \times BW + j + 1$ ▷ indice local dans le référentiel de la tuile
$a \leftarrow tuile[l]$
if a **then**
    while $a \neq tuile[a]$ **do**
        $a \leftarrow tuile[a]$
    $tuile[l] \leftarrow a$

Algorithme 36 : WARP GPU - Procédure SetRoot

Données : *addr*, *valeur*
Résultat : Le contenu de la case de mémoire *shared* d'adresse *addr* est remplacé par valeur s'il est supérieur à valeur

1 *atomicMin*(*addr*, *valeur*)

Algorithme 37 : WARP GPU - Second étiquetage utilisant WARP Union

Données :
Résultat :

1 **if** $(i < BH - 1)$ **and** $(j < BW - 1)$ ▹ Les bords Sud et Est ne sont pas traités
2 **then**
    ▹ Calcul des indices des pixels du voisinage
3 $l_a \leftarrow i \times BW + j + 1$
4 $l_b \leftarrow l_a + 1$
5 $l_c \leftarrow l_a + BW$
6 $l_d \leftarrow l_a + BW + 1$
    ▹ Chargement du voisinage : *tuile* a subi une fermeture transitive
7 $a \leftarrow tuile[l_a]$
8 $b \leftarrow tuile[l_b]$
9 $c \leftarrow tuile[l_c]$
10 $d \leftarrow tuile[l_d]$
    ▹ Du fait de la procédure Union, la remontée supplémentaire aux racines n'est pas strictement nécessaire mais elle améliore les performances ainsi le recours au caractère récursif d'Union est minimal
11 **if** a **then**
12 **while** $a \neq tuile[a]$ **do**
13 $a \leftarrow tuile[a]$
14 **if** b **then**
15 **while** $b \neq tuile[b]$ **do**
16 $b \leftarrow tuile[b]$
17 **if** c **then**
18 **while** $c \neq tuile[c]$ **do**
19 $c \leftarrow tuile[c]$
20 **if** d **then**
21 **while** $d \neq tuile[d]$ **do**
22 $d \leftarrow tuile[d]$
    ▹ Calcul du minimum du voisinage
23 $\varepsilon \leftarrow min^+(a,b,c,d)$
    ▹ Diffusion du minimum aux racines des sous-composantes connexes présentes dans la tuile avec le mécanisme Union qui garantit la validité de l'union des racines
24 **if** a **and** $(a > \varepsilon)$ **then** $Union(tuile, a, \varepsilon)$
25 **if** b **and** $(b > \varepsilon)$ **then** $Union(tuile, b, \varepsilon)$
26 **if** c **and** $(c > \varepsilon)$ **then** $Union(tuile, c, \varepsilon)$
27 **if** d **and** $(d > \varepsilon)$ **then** $Union(tuile, d, \varepsilon)$

Algorithme 38 : UnionGPU

Données : *tuile*, *dest* l'indice de la racine (présumée) à mettre à jour, *valeur* le minimum des racines du voisinage à diffuser
Résultat : *tuile* mise à jour pour rendre compte de l'union des sous-graphes

```
1 if valeur < dest then
      ▷ La fonction atomicMin renvoie la valeur contenue dans la case avant son exécution
2     minResult = atomicMin(&tuile[dest], valeur)
3     if minResult < valeur then
4         UnionGPU(tuile, valeur, minResult)
5     else
6         if minResult < dest then
7             UnionGPU(tuile, minResult, valeur)
```

Transfert dans le référentiel de l'image L'étiquetage ayant été réalisé dans le référentiel lié à la tuile, il est nécessaire de convertir les étiquettes lors de l'écriture dans la mémoire globale à la fin du traitement de la tuile. Pour cela il est nécessaire de connaître pour chaque pixel :

- la valeur de l'étiquette contenue dans le pixel exprimée dans le référentiel tuile :

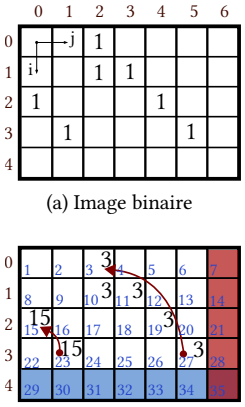
$$indexLocal \leftarrow I[i \times w_t + j + 1]$$

- le numéro dans la tuile de la ligne (*nligne*) correspondant à *indexLocal*,

$$nligne \leftarrow \lfloor indexLocal / w_t \rfloor$$

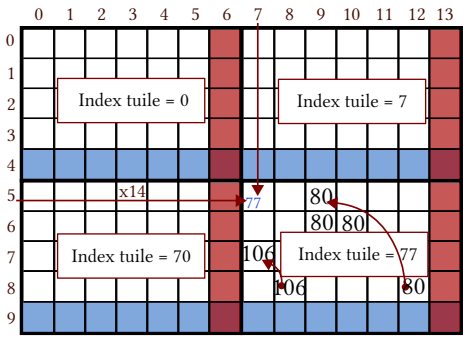
- les coordonnées du premier pixel de la tuile dans le repère image $(B0_X, B0_Y)$ pour en déduire l'index correspondant

$$indexTuile \leftarrow B0_Y \times W + B0_X$$



(a) Image binaire

(b) Étiquetage dans le jeu de coordonnées local à la tuile



(c) Dans l'image globale, ici composée de 4 tuiles, l'index global est obtenu en ajoutant l'index local, l'index de la tuile et un modificateur qui correspond à la différence entre la largeur de l'image (W) et la largeur de la tuile (BW)

Fig. 6.24 – WARP GPU transfert : l'étape de transposition locale ⟶ globale permet de conserver la dualité image / graphe à l'échelle de l'image globale

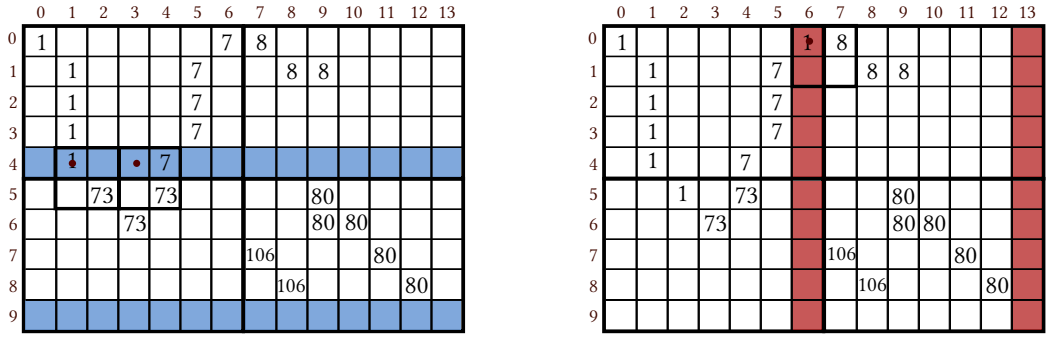
L'index dans l'image I est alors calculé en ajoutant l'index local à l'index de la tuile ainsi qu'un correctif de $W - BW$ par ligne dans la tuile.

$$indexGlobal \leftarrow indexTuile + indexLocal + nligne \times (W - BW)$$

6.4.9.4 Kernel Union S et Union E : fusion des frontières

Une fois les tuiles étiquetées et transférées, le résultat est une image étiquetée par morceaux. Afin de connecter toutes les sous-composantes connexes, il est nécessaire de parcourir tous les points des bords des tuiles à la recherche de connexions avec les tuiles voisines. Il n'est nécessaire de traiter que deux frontières orthogonales parmi les 4 (Nord, Est, Sud, Ouest). Nous avons choisi les bords Sud (kernel Union S - figure 6.25a) et Est (kernel Union E - figure 6.25b). Dans cette étape, c'est la procédure WARP Union qui est utilisée et qui permet de ne traiter qu'une seule fois chaque point des différents bords.

Cette étape, réalisée en mémoire globale, est similaire à l'union des bandes étudiée dans le chapitre 4.



(a) Fusion des bords Sud

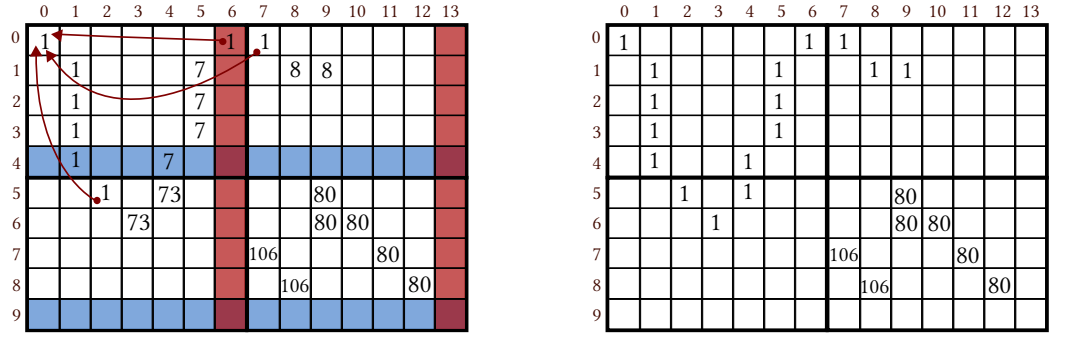
(b) Fusion des bords Est

Fig. 6.25 – WARP GPU Union S et E : les bords des tuiles sont fusionnés en traitant successivement les bords Sud et Est

6.4.9.5 Kernel Réétiquetage

Une fois toutes les tuiles fusionnées, les composantes connexes sont complètes. Cependant, tous les points de l'image ne pointent pas directement vers leur racine et une étape de ré-étiquetage est nécessaire. Ainsi, pour chaque point de l'image, une remontée à la racine est nécessaire. Si pour les algorithmes directs, le réétiquetage nécessitait la fermeture transitive de la table d'équivalence puis une passe de réécriture de l'image, ici ces deux opérations sont confondues.

Du point de vue des performances, l'opération de fermeture transitive est potentiellement coûteuse. En effet, la longueur de la remontée à la racine est indéterminée et nécessite des chargements non coalescents en mémoire globale (la plus lente). La performance du réétiquetage repose donc principalement sur celle des caches des cartes GPU et sur l'ordre de traitement des points de l'image qui profite ou non des remontées précédentes.



(a) Dans chaque tuile, les racines des sous-composantes connexes sont bien liées à la racine globale

(b) La phase de réétiquetage met à jour toutes les étiquettes

Fig. 6.26 – WARP GPU Réétiquetage

6.5 Conclusion

Dans ce chapitre, nous avons proposé deux classes d'algorithmes itératifs : MPAR et WARP. Chacune permet de réduire le nombre d'itérations et le nombre de pixels traités. Comme nous l'avons vu dans les chapitres précédents il est nécessaire de soumettre ces algorithmes à la procédure de test pour déterminer leurs performances. MPAR sera testé sur les architectures multi et many cœurs et WARP sur GPU. Des travaux sont en cours pour proposer des implémentations de WARP sur multi et many cœurs (version récursive) ainsi que sur l'architecture TSAR. Dans la suite de nos travaux de recherches, nous porterons MPAR sur GPU.

Chapitre 7

La folie n'est plus folle, dès qu'elle est collective.

–La Horde du contrevent, Alain Damasio

Performances des algorithmes itératifs parallèles

7.1 Introduction

Ce chapitre présente les résultats préliminaires de nos travaux sur les algorithmes itératifs des classes MPAR et WARP. Ayant été conçus pour des architectures différentes, ils seront tous les deux traités dans des sections séparées.

7.2 Algorithmes MPAR et architectures à grand nombre de cœurs

7.2.1 Infrastructure de mesure

L'algorithme MPAR FB + SIMD + OMP + AT, présenté au chapitre précédent, repose sur l'utilisation conjuguée de la parallélisation sur plusieurs cœurs avec OpenMP et de l'utilisation des instructions SIMD. Afin d'évaluer l'impact de ces différentes parallélisations et optimisations algorithmiques, nous avons utilisé 4 machines de mesures (tab. 7.1) avec un nombre de cœurs allant de 8 à 57 et trois générations d'extensions SIMD : SSE4.2 sur les machines $\text{NHM}_{2\times4}$ et $\text{IVB}_{2\times12}$, AVX2 sur une machine $\text{HSW}_{2\times14}$ et KNC sur le $\text{KNC}_{1\times57}$. La comparaison entre $\text{NHM}_{2\times4}$ et $\text{IVB}_{2\times12}$ renseignera donc sur l'impact du nombre de cœurs alors que la comparaison entre $\text{IVB}_{2\times12}$ et $\text{HSW}_{2\times14}$ renseignera sur l'impact du passage de SSE4.2 (CARD = 4) à AVX2 (CARD = 8). La machine $\text{KNC}_{1\times57}$ est une carte accélératrice dont chaque cœur est doté de KNC (CARD = 16) qui est un sous-ensemble du futur AVX512. Tous les programmes ont été compilés avec `icc` 15.

La table 7.2 récapitule les caractéristiques des machines utilisées pour les mesures :

- La performance crête théorique (en giga opérations par seconde).
- La bande passante crête (en gigaoctets par seconde).

Alias	Famille	Identifiant	Cœurs	Fréq (GHz)	Mémoire Cache	Bande passante mémoire maximale	SIMD	π
$NHM_{2\times4}$	Nehalem EP	X5555	2×8	2,67	2×8Mo	$2 \times 32{,}0$ Go/s	SSE 4.2	$8 \times 4 = 32$
$IVB_{2\times12}$	IvyBridge EP	E5-2695 v2	2×12	2,4	2×30Mo	$2 \times 59{,}7$ Go/s	AVX 1	$24 \times 4 = 96$
$HSW_{2\times14}$	Haswell-EP	E5-2697 v3	2×14	2,6	2×35Mo	$2 \times 68{,}0$ Go/s	AVX 2	$28 \times 8 = 224$
$KNC_{1\times57}$	Xeon Phi	3120A	1×57	1,1	1×28,5Mo	$1 \times 240{,}0$ Go/s	KNC	57×16 = 912

TABLE 7.1 – Machines de mesure des performances des algorithmes parallèles

- Le niveau de parallélisme π, qui est le produit du nombre de cœurs par le cardinal du SIMD (ici le nombre de mots de 32 bits dans un registre SIMD).
- Le ratio C/BW qui est la performance crête divisée par la bande passante crête, renseigne sur le comportement de la machine vis-à-vis des algorithmes limités par la mémoire. Plus ce ratio est élevé, plus les algorithmes sont limités par la mémoire.

Alias	Performance crête	Bande passante mémoire maximale	ratio C/BW
$NHM_{2\times4}$	85,1 Gops/s	64,0 Go/s	1,3
$IVB_{2\times12}$	255,4 Gops/s	119,4 Go/s	2,1
$HSW_{2\times14}$	582,4 Gops/s	136,0 Go/s	4,3
$KNC_{1\times57}$	1003,2 Gops/s	240,0 Go/s	4,2

TABLE 7.2 – Performances synthétiques des machines utilisées

Du point de vue du ratio C/BW, on peut distinguer d'une part les machines $NHM_{2\times4}$ et $IVB_{2\times12}$ et d'autre part les machines $HSW_{2\times14}$ et $KNC_{1\times57}$. Sur les premières, les algorithmes seront peu limités par la bande passante mémoire alors que sur les secondes, les algorithmes atteindront plus rapidement leurs limites.

7.2.2 Procédure de test

L'objectif de ces mesures est de confronter les algorithmes itératifs à l'augmentation du nombre de cœurs et de la taille des registres SIMD afin de déterminer s'ils ont le potentiel pour devenir plus rapides que les algorithmes directs sur cette classe d'architecture. À cette fin, nous utilisons les images aléatoires de la procédure de test établie au chapitre 2. Les résultats sont exprimés en *cpp* avec des images de taille 2048×2048 et 4096×4096 pour les granularités $g \in \{1, 4, 16\}$ ainsi que la moyenne des résultats pour ces trois granularités. De plus, les performances des algorithmes LSL_{RLE} et HCS_2 (la version ARemSP DT) sont aussi reportées pour permettre une comparaison directe. Les algorithmes itératifs, réalisant par construction l'étiquetage complet et non l'analyse en composantes connexes, les algorithmes directs sont eux aussi utilisés sans l'étape d'analyse et avec le réétiquetage.

Les algorithmes comparés sont :

- MPAR FB : la version scalaire séquentielle qui réalise l'étiquetage en alternant les passes directes et inverses.
- MPAR FB SIMD : amélioration de MPAR FB utilisant les instructions SIMD dont le niveau de parallélisme dépend de la machine.
- MPAR FB OMP : amélioration de MPAR FB avec un simple découpage en tuiles réparties sur les différents cœurs.
- MPAR FB SIMD + OMP : amélioration de MPAR FB SIMD avec un découpage en bandes réparties sur les différents cœurs.

- MPAR FB SIMD + OMP + AT : amélioration de MPAR FB SIMD + OMP utilisant un découpage en tuiles et une matrice d'activité pour gérer l'activation des itérations dans chaque tuile.
- LSL_{RLE} : la version séquentielle de LSL conçue pour tirer parti des segments les plus longs par une utilisation intensive du codage RLC.
- LSL_{RLE} + OMP : la version précédente utilisant la parallélisation présentée au chapitre 5 avec un découpage en bandes.
- HCS_2 : le masque HCS_2 avec la gestion des équivalences Union-Find (UF) améliorée avec l'arbre de décision (DT) et l'optimisation Rem+Splicing (ARemSP).
- HCS_2 + OMP : la version précédente utilisant la parallélisation présentée au chapitre 5 avec un découpage en bandes.

La taille et la forme des tuiles étant un paramètre de l'algorithme, nous avons fait varier celles-ci entre 2^4 et 2^{11}. Le *cpp* retenu pour les tables est celui correspondant à la meilleure taille de tuile (*cpp* minimal).

7.2.3 Résultats pour les machines à faible ratio C/BW

Les tables 7.3 et 7.4 présentent les performances des différents algorithmes pour les machines à faible ratio C/BW que sont $NHM_{2\times4}$ (1,3) et $IVB_{2\times12}$ (2,1).

7.2.3.1 $NHM_{2\times4}$

	Granularité			
	g = 1	g = 4	g = 16	cpp_g
cpp des algorithmes itératifs				
MPAR FB	1028	294,0	105,7	475,90
MPAR FB + SIMD	243,3	137,0	67,56	149,3
MPAR FB + OMP	169,2	78,92	55,89	101,3
MPAR FB + SIMD + OMP	44,46	33,15	26,89	34,83
MPAR FB + SIMD + OMP + AT	27,06	19,29	**15,96**	20,77
cpp des algorithmes directs				
LSL_{RLE}	13,16	5,233	3,559	7,320
HCS_2	13,800	7,644	6,260	9,230
LSL_{RLE}+OMP	2,157	1,139	**0,969**	1,420
HCS_2+OMP	3,080	2,342	**2,242**	2,55

TABLE 7.3 – Performance des algorithmes sur la machine $NHM_{2\times4}$ pour les images 2048 × 2048

Comme pour les algorithmes directs, les images de granularité g=1 sont plus complexes à étiqueter que celles de granularité g=16. Pour MPAR FB, le rapport $cpp_{d(g=1)}/cpp_{d(g=16)}$ vaut ×9,7. L'apport du SIMD (MPAR FB + SIMD) est très bénéfique pour g=1 avec une accélération de ×4,2 et se réduit à ×1,6 pour g=16. Ces valeurs sont à comparer avec le gain théorique lié au cardinal du SIMD (ici $CARD = 4$). L'apport du découpage en tuiles avec OpenMP (MPAR FB + OMP) est bénéfique pour g=1 avec une accélération de ×6,1 qui se réduit à ×1,9 pour g=16, à comparer avec le gain théorique lié aux 8 cœurs. La combinaison SIMD + OMP est très exigeante pour la mémoire. Cependant, elle se révèle tout de même utile dans le cas de l'étiquetage en composantes connexes avec une accélération globale pour MPAR FB + SIMD + OMP par rapport à MPAR FB de ×23.1 pour g=1 et ×3.93 pour g=16, à comparer avec le gain théorique $\pi = 32$.

Plus g augmente, plus le nombre d'itérations nécessaires à MPAR FB pour l'étiquetage complet de l'image diminue et donc plus l'étiquetage est rapide. Le découpage en bandes ou en tuiles vient

diminuer la vitesse de propagation en introduisant des barrières de synchronisation aux frontières et profite donc moins de l'augmentation de la granularité, ce qui explique que l'accélération bien que toujours supérieure à 1 diminue avec g.

La version MPAR FB + SIMD + OMP représente l'adaptation de MPAR FB aux possibilités de parallélisation de l'architecture. L'ajout du principe de tuiles actives permet d'optimiser l'utilisation des ressources en fonction de la structure de l'image et d'atteindre une accélération par rapport à MPAR FB de ×38,0 pour g=1 et ×6,6 pour g=16 à comparer avec le gain théorique $\pi = 32$. Au final, la variabilité du *cpp* en fonction de la granularité diminue et passe d'un rapport ×9,7 pour MPAR FB à un rapport ×1,7 pour MPAR FB + SIMD + OMP + AT.

Dans le même temps, l'accélération du LSL_{RLE} est de ×6,1 pour g=1 et ×3,7 pour g=16 et celle de HCS_2 est de ×4,4 pour g=1 et ×2,8 pour g=16 à comparer avec le gain théorique lié aux 8 cœurs. Les algorithmes directs sont plus rapides que les algorithmes itératifs sur cette machine, d'un rapport moyen ×14,6 pour LSL_{RLE} et ×8,14 pour HCS_2.

7.2.3.2 $\mathbf{IVB}_{2\times12}$

Sur cette machine, nous avons utilisé les instructions SSE4.2 car AVX ne propose pas d'instructions de calcul sur les entiers. Cela nous permettra donc d'évaluer directement l'influence du nombre de cœurs par comparaison avec la machine $NHM_{2\times4}$.

L'apport du découpage en bandes avec OpenMP (MPAR FB + OMP) est encore une fois bénéfique pour g=1 avec une accélération ×13,9 et se réduit à ×6,7 pour g=16 à comparer avec le gain théorique lié aux 24 cœurs. La version MPAR FB + SIMD + OMP permet d'atteindre une accélération globale par rapport à MPAR FB de ×46,6 pour g=1 et ×10,3 pour g=16 à comparer avec le gain théorique $\pi = 96$. L'ajout des tuiles actives permet de passer à une accélération par rapport à MPAR FB de ×83,0 pour g=1 et ×18,8 pour g=16. La variabilité du *cpp* en fonction de la granularité diminue et passe d'un rapport 7,4 pour MPAR FB entre g=1 et g=16, à un rapport ×1,7 pour MPAR FB + SIMD + OMP + AT entre g=1 et g=16.

	Granularité			
	g = 1	g = 4	g = 16	cpp_g
cpp des algorithmes itératifs				
MPAR FB	1317	446,6	176,8	646,8
MPAR FB + SIMD	348,4	175,3	87,11	203,6
MPAR FB + OMP	95,01	40,09	26,25	53,78
MPAR FB + SIMD + OMP	28,28	21,14	17,12	22,18
MPAR FB + SIMD + OMP + AT	15,86	11,40	**9,390**	12,22
cpp des algorithmes directs				
LSL_{RLE}	13,81	5,430	3,190	7,480
HCS_2	14,09	7,570	6,170	9,280
LSL_{RLE}+OMP	1,670	0,995	**0,854**	1,170
HCS_2+OMP	3,430	2,312	**2,096**	2,610

TABLE 7.4 – Performance des algorithmes sur la machine $IVB_{2\times12}$ pour les images 4096 × 4096

L'accélération du LSL_{RLE} est de ×8,3 pour g=1 et ×3,7 pour g=16 et celle de HCS_2 est de ×4,1 pour g=1 et ×2,9 pour g=16 à comparer avec le gain théorique lié aux 24 cœurs. Les algorithmes directs sont plus rapides que les algorithmes itératifs sur cette machine, d'un rapport ×10,4 pour LSL_{RLE} et ×4,7 pour HCS_2. L'écart entre les algorithmes directs et itératifs se réduit avec l'augmentation du nombre de cœurs.

7.2.4 Résultats pour les machine à fort ratio C/BW

7.2.4.1 $HSW_{2\times14}$

La table 7.5 présente les résultats pour la machine $HSW_{2\times14}$. Ces résultats nous permettent d'étudier l'impact de l'évolution des instructions SIMD par rapport à la machine $IVB_{2\times12}$ dont le nombre de cœurs est proche.

L'ajout du SIMD (MPAR FB + SIMD) apporte une accélération de ×6.2 pour g=1 et de ×2.8 pour g=16 à comparer avec le gain théorique lié au cardinal du SIMD ici *CARD* = 8. La version MPAR FB + SIMD + OMP permet d'atteindre une accélération globale par rapport à MPAR FB de ×18.7 pour g=1 et ×4.8 pour g=16 à comparer avec le gain théorique π = 224. Ces performances sont très en retrait par rapport à la machine $IVB_{2\times12}$ et illustrent la dépendance aux performances de la mémoire. En effet, le passage de SSE4.2 à AVX2 permet de doubler le cardinal des registres SIMD et augmente donc la bande passante nécessaire au traitement de chaque tuile. L'ajout des tuiles actives permet de passer à une accélération par rapport à MPAR FB de ×151.7 pour g=1 et ×35.6 pour g=16. La diminution les accès inutiles à la mémoire, permet de mieux utiliser la bande passante et de profiter ainsi de l'augmentation du cardinal des instructions SIMD apportée par AVX2 (×2). Avec les tuiles actives, les performances sont très élevées comparativement à $IVB_{2\times12}$.

	g = 1	g = 4	g = 16	moyenne
cpp des algorithmes itératifs				
MPAR FB	1180,2	360,5	140,2	560,3
MPAR FB + SIMD	190,4	360,5	50,67	200,5
MPAR FB + OMP	78,73	44,24	45,79	56,25
MPAR FB + SIMD + OMP	63,24	32,32	29,13	41,56
MPAR FB + SIMD + OMP + AT	7,780	4,500	**3,940**	5,410
cpp des algorithmes directs				
LSL_{RLE}	12,840	4,492	2,645	6,66
HCS_2	13,195	6,695	5,620	8,50
LSL_{RLE}+OMP	1,317	0,357	**0,249**	0,64
HCS_2+OMP	2,524	2,249	**1,711**	2,16

TABLE 7.5 – Performance des algorithmes sur la machine Haswell $HSW_{2\times14}$ pour les images 4096×4096

7.2.4.2 $KNC_{1\times57}$

Sur la carte Xeon Phi, les versions sans OpenMP n'ont pas de sens car elles sont contraires au principe même de fonctionnement de l'accélérateur.

L'augmentation du nombre de cœurs et du cardinal des instructions SIMD (CARD = 16) permet d'atteindre un *cpp* proche de celui du LSL_{RLE} (pour g=16, le ratio des cpp_d est de 1,96). De son côté, HCS_2 voit ses performances chuter comme dans le chapitre 5 et l'on retrouve la limitation due à la mémoire : pour toutes les granularités, le *cpp* est ≈ 52. La version itérative avec (et sans) tuiles actives sur la machine $KNC_{1\times57}$ est plus rapide que le plus rapide des algorithmes pixels.

7.2.5 Efficacité de la propagation de Max par rapport au min

Sur toutes les machines, la propagation du maximum pour l'algorithme avec tuiles actives est plus efficace que la propagation du min positif (tab. 7.7). Le gain est maximal pour la machine $IVB_{2\times12}$ et minimal pour la machine $KNC_{1\times57}$. Pour chaque voisinage, il n'est plus nécessaire de réaliser des tests spécifiques pour exclure les pixels nuls du calcul de l'étiquette représentative du voisinage, ce qui

	g = 1	g = 4	g = 16	moyenne
cpp des algorithmes itératifs				
MPAR FB + OMP	170,4	141,5	126,0	145,97
MPAR FB + SIMD + OMP	26,87	26,43	25,76	26,35
MPAR FB + SIMD + OMP + AT	6,04	4,08	**2,86**	4,33
cpp des algorithmes directs				
LSL_{RLE}+OMP	2,816	1,721	**1,460**	2,00
HCS_2+OMP	52,640	52,070	**51,971**	52,23

TABLE 7.6 – Performance des algorithmes sur l'accélérateur Knight Corner $KNC_{1\times57}$ pour les images 4096×4096

conduit à un code simplifié. Bien que significative, cette amélioration ne permet pas aux algorithmes itératifs de dépasser l'algorithme LSL_{RLE}+OMP.

	g = 1	g = 4	g = 16	Moyenne	gain *Min* → *Max*
$NHM_{2\times4}$					
MPAR FB + SIMD + OMP + AT (MIN)	27,06	19,29	15,96	20,77	×1,32
MPAR FB + SIMD + OMP + AT (MAX)	21,84	13,74	**11,91**	15,83	
LSL_{RLE}+OMP	2,157	1,139	0,969	1,42	
$IVB_{2\times12}$					
MPAR FB + SIMD + OMP + AT (MIN)	15,86	11,40	9,39	12,22	×1,67
MPAR FB + SIMD + OMP + AT (MAX)	10,16	6,36	**5,46**	7,33	
LSL_{RLE}+OMP	1,67	0,995	0,854	1,17	
$HSW_{2\times14}$					
MPAR FB + SIMD + OMP + AT (MIN)	7,78	4,50	3,94	5,41	×1,29
MPAR FB + SIMD + OMP + AT (MAX)	5,42	3,95	**3,21**	4,19	
LSL_{RLE}+OMP	1,317	0,357	0,249	0,64	
$KNC_{1\times57}$					
MPAR FB + SIMD + OMP + AT (MIN)	6,04	4,08	2,86	4,33	×1,18
MPAR FB + SIMD + OMP + AT (MAX)	5,23	3,30	**2,46**	3,66	
LSL_{RLE}+OMP	2,816	1,721	1,460	2,00	

TABLE 7.7 – Performance (*cpp*) de l'algorithme MPAR FB + SIMD + OMP + AT (MAX) comparativement à la version MIN et à LSL_{RLE}+OMP

7.2.6 Comportement vis-à-vis de la densité

Comme pour les algorithmes directs, le *cpp* dépend de la densité. La figure 7.2 représente le *cpp* pour la version MPAR FB + SIMD + OMP (découpage en bandes) et MPAR FB + SIMD + OMP + AT (tuiles actives). La densité d=41% est ici aussi un point particulier de chaque courbe et fait augmenter très sensiblement le *cpp*. Pour g=1 et $d < 41\%$, les deux algorithmes sont très proches. Au-delà, les communications et le traitement systématique pour chaque bande impose un *cpp* élevé pour MPAR FB + SIMD + OMP alors que MPAR FB + SIMD + OMP + AT retrouve un niveau similaire à $d < 41\%$. Pour d=70%, le ratio entre les deux algorithmes varie entre ×15,0 (g=1) et ×12,6 (g=16) sur la machine $HSW_{2\times14}$ et entre ×16,4 (g=1) et ×16,6 (g=16) sur la machine $KNC_{1\times57}$.

7.2.7 Comportement vis-à-vis du découpage en tuiles

Les versions MPAR FB + SIMD + OMP + AT et MPAR FB + SIMD + OMP + AT (MAX) se basent sur un découpage en tuiles. Dans les tables précédentes, le *cpp* retenu était celui correspondant à la

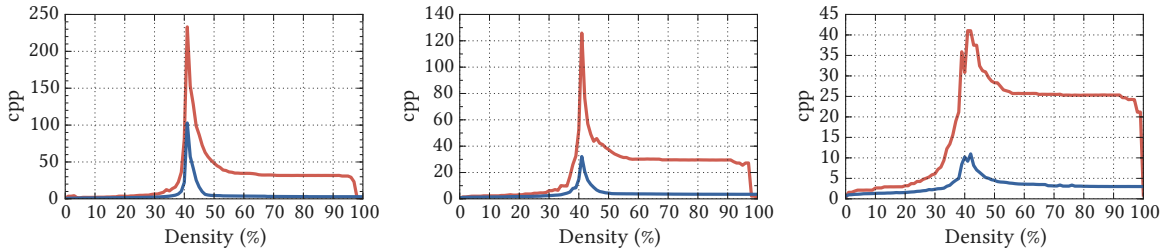


Fig. 7.1 – *cpp* pour les versions SIMD + OMP avec (bleu) et sans (rouge) le mécanisme de tuiles actives pour $g = 1$ (gauche), $g = 4$ (milieu) et $g = 16$ (droite) sur la machine $\text{HSW}_{2\times14}$

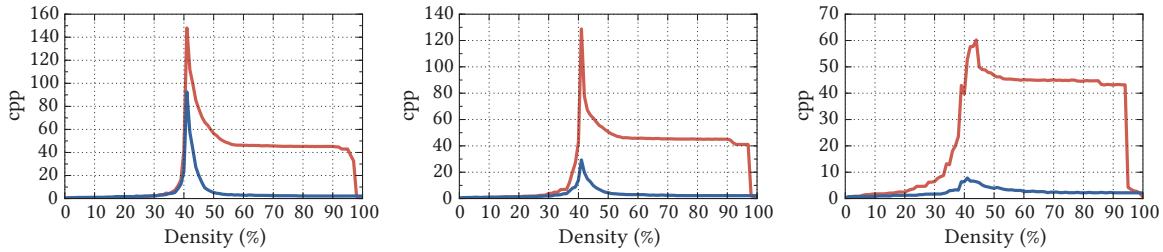


Fig. 7.2 – *cpp* pour les versions SIMD + OMP avec (bleu) et sans (rouge) le mécanisme de tuiles actives pour $g = 1$ (gauche), $g = 4$ (milieu) et $g = 16$ (droite) sur la machine $\text{KNC}_{1\times57}$

taille de tuiles permettant d'obtenir le *cpp* le plus faible. La figure 7.3 représente les *cpp* pour des tuiles correspondant aux combinaisons $\{32 \ldots 1024\} \times \{64 \ldots 1024\}$ sur les machines $\text{HSW}_{2\times14}$ et $\text{KNC}_{1\times57}$.

Largeur de la tuile

Hauteur de la tuile	32	64	128	256	512	1024
64	10,9	10,2	10,2	10,4	10,5	10,6
128	9,3	9,2	9,7	10,0	10,4	9,6
256	5,4	5,4	6,2	7,0	6,6	7,6
512	4,4	3,9	4,7	5,9	7,6	8,6
1024	5,2	4,4	6,2	8,6	9,2	8,0

Largeur de la tuile

Hauteur de la tuile	32	64	128	256	512	1024
64	531,8	272,1	140,6	67,5	24,9	12,1
128	128,2	64,6	35,1	11,0	8,0	8,1
256	34,8	17,1	10,1	5,1	5,0	5,9
512	23,7	9,4	4,9	4,0	3,9	6,9
1024	12,8	6,3	4,2	3,3	5,1	9,5

Fig. 7.3 – Cartographie des *cpp* en fonction de la forme des tuiles pour la version MPAR FB + SIMD + OMP + AT (MAX) pour les machines $\text{HSW}_{2\times14}$ (gauche) et $\text{KNC}_{1\times57}$ (droite)

Si la version MPAR FB + SIMD + OMP + AT (MAX) est toujours plus efficace que la version MPAR FB + SIMD + OMP pour la machine $\text{HSW}_{2\times14}$ (fig. 7.4), ce n'est pas le cas pour la machine $\text{KNC}_{1\times57}$ où un surdécoupage de l'image (tuiles trop petites) rend inefficace la version avec tuiles. Pour chaque machine testée, l'optimum correspond à la même taille de tuile qui dépend donc des caractéristiques de la machine plus que de celles des images. Pour un système en production, une séquence de tests sur des images représentatives des applications permettra de définir l'optimum.

Le découpage en tuiles plus petites que les bandes permet de mieux utiliser les caches et permet à la seconde itération (inverse) de se dérouler dans de meilleures conditions.

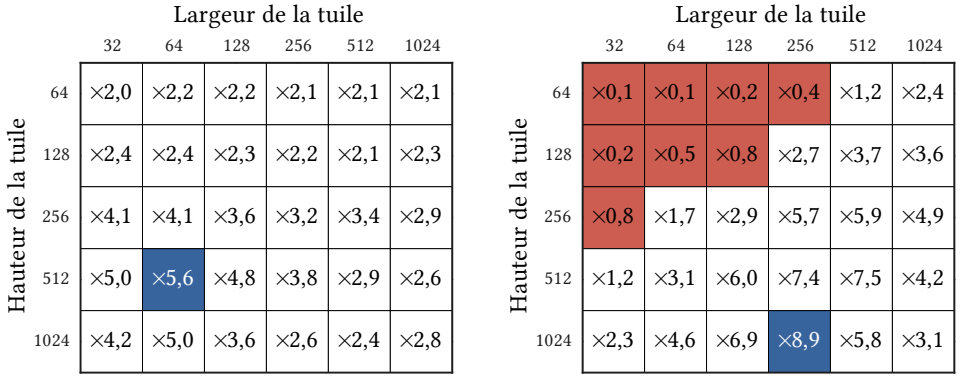


Fig. 7.4 – Cartographie des ratios entre la version MPAR FB + SIMD + OMP + AT (MAX) et MPAR FB + SIMD + OMP en fonction de la forme des tuiles pour la version MPAR FB + SIMD + OMP + AT (MAX) pour les machines $HSW_{2\times14}$ (gauche) et $KNC_{1\times57}$ (droite)

7.2.8 Perspectives d'adaptation aux machines à très grand nombre de cœurs

La table 7.8 représente la portion moyenne de code séquentiel calculée (cf. section 4.6.2) sur les machines $NHM_{2\times4}$, $IVB_{2\times12}$ et $HSW_{2\times14}$. τ reflètent la scalabilité des algorithmes et peut mettre en évidence l'impact de leur empreinte mémoire.

MPAR FB + SIMD + OMP + AT est l'algorithme avec le plus faible τ et pour lequel τ diminue avec le nombre de cœurs et l'augmentation du cardinal du SIMD. Cela permet de penser qu'une augmentation du nombre de cœurs lui sera profitable.

À l'opposé, HCS_2, le représentant des algorithmes pixels, est très pénalisé par ses besoins en mémoire et ne tire que très peu parti de l'augmentation du nombre de cœurs.

Machine	$NHM_{2\times4}$	$IVB_{2\times12}$	$HSW_{2\times14}$
LSL_{RLE} + OMP	7,9 %	12 %	6,3 %
HCS_2 + OMP	17,3%	25,0 %	22,6 %
MPAR FB + SIMD + OMP	4,3 %	2,4 %	7 %
MPAR FB + SIMD + OMP + AT	1,3 %	0,9 %	0,5 %

TABLE 7.8 – τ, la portion de code séquentiel calculé selon la loi d'Amdahl à partir des *cpp* moyens (le plus faible est le meilleur)

7.2.9 Conclusion pour les algorithmes MPAR

Dans le cas de l'étiquetage en composantes connexes, les algorithmes directs pixels ne passent pas à l'échelle sur des machines modernes. Ceci est dû à leurs besoins trop élevés en bande passante et au faible parallélisme de la procédure de fusion pyramidale. La compression de données apportée par *LSL* est une solution à ce problème, mais lorsque le nombre de lignes à traiter par tuile se rapproche du nombre de tuiles à fusionner, la part de la fusion pyramidale devient non négligeable et nécessite de travailler sur de plus grandes images. Les algorithmes itératifs avec découpage en tuiles actives permettent de minimiser le nombre de pixels à traiter et de réduire ainsi les besoins en bande passante.

7.3 Algorithmes WARP sur GPU

7.3.1 Infrastructure de mesure

Comme l'ensemble des algorithmes GPU antérieurs présentés dans le chapitre précédent, WARP-GPU a été initialement développé et testé pour des GPU de la société NVIDIA en utilisant les outils CUDA (version 7.5). La carte NVIDIA GTX 980_{Ti} (Maxwell), que nous avons choisie comme référence, est une carte GPU grand public hauts de gamme et se positionne sur le même segment de marché que les CPU haut de gamme de machine de bureau (mono socket). C'est donc à une machine de bureau que ses performances doivent être comparées. Son utilisation permet d'établir un étalon des performances attendues au premier semestre 2016 sur les algorithmes d'étiquetage en composantes connexes qui pourra servir de référence à des travaux ultérieurs. Afin d'envisager les performances de l'algorithme du point de vue de l'évolution de l'architecture GPU, et d'évaluer ainsi les possibilités d'accélération de l'algorithme sur les générations futures, trois générations de GPU (Fermi, Kepler, Maxwell) ont été testées (tab. 7.9).

Alias	Famille	Année de sortie	Cœurs	Fréquence de base	Mémoire	Bande passante mémoire maximale
NVIDIA GTX 480	Fermi	2010	480	700 Mhz	1,5Go	177,4 Go/s
NVIDIA GTX 780_{Ti}	Kepler	2013	2880	1110 Mhz	3Go	336,0 Go/s
NVIDIA GTX 980_{Ti}	Maxwell	2015	2816	1306 Mhz	6Go	336,5 Go/s

Table 7.9 – Cartes GPU utilisées pour évaluer les performances de WARP GPU

7.3.2 Procédure de test

Tout comme les algorithmes directs, WARP GPU est évalué à l'aide de la procédure définie dans le chapitre 2, la taille des images aléatoires est identique à celle utilisée pour les machines de bureau pour l'algorithme MPAR (2048×2048). Une étude de l'influence de la taille des images est aussi menée pour des images de taille 128×128 à 8192×8192. La base SIDBA4 permet d'évaluer le comportement de l'algorithme dans des conditions plus réalistes.

L'objectif de ces tests est multiple :

- permettre de mieux connaître le comportement de WARP GPU vis-à-vis de la densité et de la granularité ainsi que dans des conditions réelles (SIDBA4),
- évaluer le comportement de WARP GPU vis-à-vis des différentes générations de GPU pour évaluer l'impact des améliorations sur son fonctionnement,
- identifier les freins à l'augmentation des performances pour proposer des pistes de développement de nouveaux algorithmes.

Les performances seront exprimées en ms et en Gp/s (indépendant de la taille des images) :

- t représente le temps de traitement d'une image de densité d et de granularité g données,
- t_d est la moyenne de tous les t pour d allant de 0 à 100% pour une granularité g donnée,
- t_g est la moyenne de tous les t_d pour g allant de 1 à 16,
- D représente le débit en Gp/s pour une image de densité d et de granularité g données,
- D_d est la moyenne de tous les D pour d allant de 0 à 100% pour une granularité g donnée,
- D_g est la moyenne de tous les D_d pour g allant de 1 à 16.

Pour les implémentations GPU, la taille optimale des tuiles (w_t, h_t) est fonction de l'adéquation entre les caractéristiques de la carte et celles de l'algorithme. Afin d'obtenir la forme optimale des

tuiles, une procédure exhaustive de mesure des performances [120] en fonction de w_t et h_t a été utilisée.

7.3.3 WARP GPU et génération Maxwell

La génération Maxwell était la plus évoluée disponible lors de nos travaux. Même si WARP GPU est exécutable sur toutes les générations précédentes, il a été conçu en tenant compte des qualités spécifiques de cette génération :

- des fonctions `Atomic` performantes pour les opérations dans la mémoire `shared`,
- une gestion améliorée des différents niveaux de cache.

La performance maximale sur cette carte a été observée pour une découpage en tuiles de taille $(w_t,h_t) = (84, 8)$ (kernel Diffusion), ce qui correspond à 672 pixels, soit 672 threads par tuile permettant de placer 3 tuiles par SMM et de se placer ainsi juste en-dessous (2016 threads) de la limite des 2048 threads par SMM.

7.3.4 Images aléatoires

Les courbes de la figure 7.5 présentent le temps de traitement des images en fonction de la densité pour les granularités $g \in \{1, 4, 16\}$ ainsi que l'évolution du temps moyen de traitement en *ms* en fonction de la granularité. Les courbes sont similaires aux courbes des algorithmes directs sans arbre de décision (cf. chap. 3) à l'exception d'un pic pour la densité d=41% pour $g = 1$ et qui s'atténue avec l'augmentation de g. Plus la granularité augmente, plus la courbe représentant le temps de traitement se rapproche d'une droite. Le temps de traitement devient alors proportionnel à la densité de l'image. Ainsi la courbe de t_d est globalement décroissante en fonction de g.

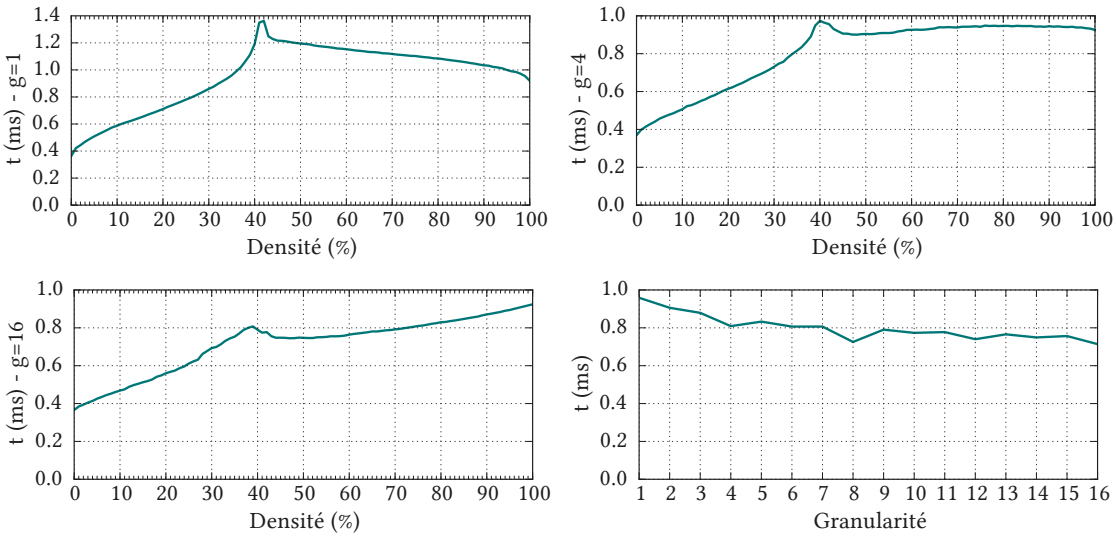


Fig. 7.5 – WARP GPU : temps de traitement t en *ms* pour des images de taille 2048×2048 et de granularité $g \in \{1, 4, 16\}$ et t_d en fonction de la granularité sur une carte GTX 980_{Ti}

La table 7.10 présente la moyenne du temps de traitement (en ms) et du débit (en Gps) par rapport à la densité pour $g \in \{1, 2, 4, 8, 16\}$ ainsi que la moyenne de ces valeurs sur l'ensemble des granularités testées. Les performances augmentent avec la granularité (×1,35 entre g=1 et g=16) mais dans une mesure bien moindre que pour les algorithmes directs (×8,35).

	Granularité					
GTX 980_{Ti}	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	moy_g
temps (ms)	0,96	0,91	0,81	0,73	0,71	0,80
Débit (Gp/s)	4,37	4,63	5,19	5,78	5,87	5,25

TABLE 7.10 – WARP GPU : temps de traitement moyen t_d en ms pour des images de taille 2048×2048 et de granularité $g \in \{1, 2, 4, 8, 16\}$ et t_g en fonction de la granularité sur une carte GTX 980_{Ti}

7.3.5 Parts des différents kernels dans la composition de la performance globale

La taille des blocs pour chaque kernel a été optimisée par analyse exhaustive. En effet, chaque kernel a des propriétés différentes et dépend des kernels précédents. Cependant la taille la plus décisive est celle du kernel Diffusion. Le maximum de performance du traitement global est obtenu pour la combinaison :

- kernel Diffusion : $(w_t, h_t) = (84, 8)$
- kernel Union S : $(w_t, h_t) = (112, 1)$
- kernel Union E : $(w_t, h_t) = (96, 1)$
- kernel Réétiquetage : $(w_t, h_t) = (64, 2)$

Le kernel Diffusion traite par morceaux tous les pixels de l'image à étiqueter et génère des tuiles complètement étiquetées. Le nombre de pixels à traiter par les kernels Union S (N_S) et Union E (N_E) dépend de la taille et de la forme des tuiles du kernel Diffusion. Pour les tuiles de taille $(84, 8)$ la quantité de pixels à traiter par kernel est :

- kernel Union S : $N_S = (2048 \times 2048)/h_t = 512Kpixels$ (pour $h_t = 8$),
- kernel Union E : $N_E = (2048 \times 2048)/w_t = 48{,}7Kpixels$ (pour $w_t = 84$).

Le kernel Union E qui unit les bords Est des tuiles réalise des accès verticaux non coalescents (cf. sec.6.4.9.4) aux pixels qui sont bien plus coûteux que les accès horizontaux réalisés par le kernel Union S qui unit les bords Sud. C'est un des facteurs justifiant la forme des tuiles ayant les performances optimales.

La fusion des tuiles a été décomposée en deux kernels (Union S et Union E) pour des raisons d'efficacité de l'implémentation mais sera présenté de manière globale car elle ne forme qu'une seule opération.

Le kernel Réétiquetage réalise un nombre indéterminé d'accès en lecture à l'image globale pour atteindre la racine de la composante connexe liée au pixel courant, ainsi qu'un accès en écriture pour mettre à jour l'étiquette. Dans le cas d'une image structurée, les racines de composantes connexes sont les plus demandées et la performance de ce kernel dépend donc fortement de la performance des caches.

La figure 7.6 représente les temps de traitement cumulés des kernels Diffusion (vert), Union S et E (rouge) et Réétiquetage (jaune). Le kernel Diffusion consomme la majorité du temps de traitement et présente un maximum pour d=60% et g=1. Il ne présente pas de dépendance à la densité d=41%. Pour les granularités supérieures, la courbe se rapproche de la droite qui relie les résultats pour d=0% et d=100%. L'ensemble formé par les kernels Union Sud et Est croît avec la densité jusqu'à la densité d=40%, présente un maximum pour d autour de 40% puis diminue jusqu'à d=50% et se stabilise au-delà. Le kernel Réétiquetage est très similaire à l'ensemble formé par les kernels Union Sud et Est alors qu'il traite l'ensemble de l'image.

La décomposition du temps de traitement global selon les trois ensembles de kernels (tab. 7.11) met en évidence que les kernels Union Sud et Est et le kernel Réétiquetage sont peu sensibles à l'évolution

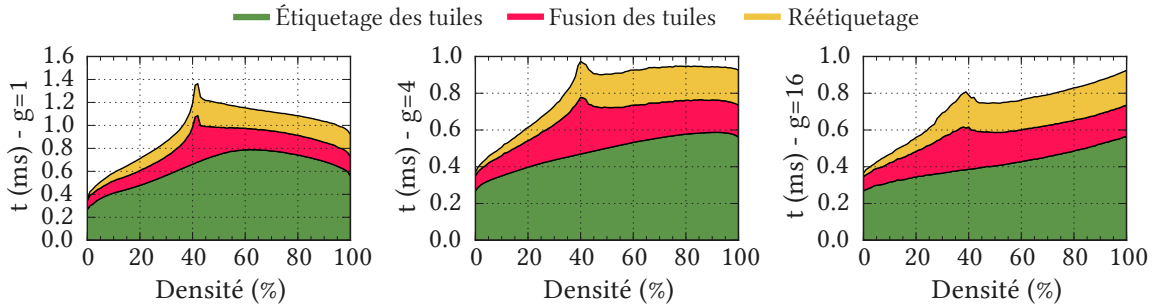


Fig. 7.6 – WARP GPU : temps de traitement t pour chaque kernel exprimés en ms pour des images de taille 2048×2048 et de granularité $g \in \{1, 4, 16\}$ sur une carte GTX 980_{Ti}

	Granularité					
GTX 980_{Ti}	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	t_g
kernel Diffusion	0,63	0,57	0,48	0,41	0,41	0,48
kernels Union Sud et Est	0,15	0,15	0,14	0,14	0,14	0,14
kernel Réétiquetage	0,18	0,18	0,18	0,18	0,16	0,18
Total	0,96	0,90	0,80	0,73	0,71	0,80

TABLE 7.11 – Décomposition du temps de traitement t_d des kernels Diffusion, Union Sud et Est, Réétiquetage pour les images 2048×2048 de granularité $g \in \{1, 2, 4, 8, 16\}$ et t_g en fonction de la granularité sur une carte GTX 980_{Ti}

de la granularité. Seul le kernel Diffusion profite de la structuration des données et passe de $0{,}63ms$ pour g=1 à $0{,}41ms$ pour g=16. En moyenne, le kernel Diffusion représente 60,0% du temps total, les kernels Union Sud et Est 17,5% et le kernel Réétiquetage 22,5%.

Il est intéressant de comparer le nombre de cycles par pixel par PE cpp_{PE} de chaque ensemble :

- kernel Diffusion : cpp_{PE} = 421 cycles par pixel pour 4Mpx à traiter
- kernels Union Sud et Est : cpp_{PE} = 897 cycles par pixel pour 0,57Mpx à traiter
- kernel Réétiquetage : cpp_{PE} = 158 cycles par pixel pour 4Mpx à traiter

Cela met en évidence que la procédure d'union des tuiles qui réalise des accès très dispersés est plus coûteuse (par pixel) d'un rapport ×2,13 que la procédure d'étiquetage des tuiles.

7.3.6 SIDBA4

Les images de la base SIDBA4 confrontent WARP GPU à des images non homogènes et illustrent son comportement du point de vue de l'équilibrage de charge. La table 7.12 représente les temps extrêmes (min et max) ainsi que la moyenne de traitement des images de la base SIDBA. Les proportions sont similaires à celles des images aléatoires. L'évolution de l'étape de fusion des tuiles est significative : pour l'image la plus simple à étiqueter (correspondant au temps minimal) les kernels Union Sud et Est ne concourent qu'à hauteur de 13% dans le temps total alors que ce chiffre atteint 20% pour l'image la plus compliquée (correspondant au temps maximal).

7.3.7 Influence de la génération de GPU

Les performances obtenues par WARP GPU dépendent de la génération de GPU. En effet d'une génération à l'autre, plusieurs paramètres évoluent tels que le nombre de cœurs, leur fréquence de fonctionnement, la bande passante mémoire et la latence des instructions. Une grande partie des évolutions sont dédiées à l'accélération du traitement des nombres flottants (en demi (F16) simple (F32)

	$t(ms)$				Pourcentage du temps total		
	Diffusion	Union Sud et Est	Réétiquetage	Total	Diffusion	Union Sud et Est	Réétiquetage
min	0,71	0,14	0,23	1,08	65,7%	13,0%	21,3%
moy	0,83	0,23	0,26	1,32	62,9%	17,4%	19,7%
max	0,98	0,32	0,30	1,60	61,3%	20,0%	18,8%

TABLE 7.12 – Temps et proportion min, moy et max de chaque étape pour les images de la base de données SIDBA4

ou double (F64) précision). WARP GPU opère sur les nombres entiers et ne tire donc pas partie des développement sur les nombres flottants. Les améliorations sont donc à trouver du côté de l'augmentation de la bande passante à tous les niveaux (caches, mémoire *shared*, mémoire globale) ainsi que de l'ajout d'instructions améliorant spécifiquement le temps d'accès lors d'accès concurrents (écriture ou lecture) à une donnée.

La table 7.13 représente les temps de traitement en *ms* pour les trois générations de GPU testées et la table 7.14 le débit correspondant. En moyenne (t_g) les performances de WARP GPU ont progressé d'un facteur ×3,24 entre la carte GTX 480 et la carte GTX 780_{Ti} et d'un facteur ×1,54 entre la carte GTX 780_{Ti} et la carte GTX 980_{Ti}.

	t_d					
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	t_g
GTX 480	4,83	4,73	3,83	3,43	3,39	3,98
GTX 780_{Ti}	1,56	1,50	1,27	1,09	1,05	1,23
GTX 980_{Ti}	0,96	0,91	0,81	0,73	0,71	0,80

TABLE 7.13 – WARP GPU : temps de traitement moyen t_d en *ms* pour des images de taille 2048×2048 et de granularité $g \in \{1, 2, 4, 8, 16\}$ et t_g en fonction de la granularité sur les cartes de tests

L'augmentation du nombre de cœurs (×6), de la fréquence de base (×1,6) et de la bande passante (×1,9) entre la carte GTX 480 et la carte GTX 780_{Ti} justifie la forte progression. Dans le cas de la transition GTX $780_{Ti} \longrightarrow$ GTX 980_{Ti}, seule la fréquence évolue significativement (×1,2). La justification de l'accélération obtenue est donc à rechercher dans l'amélioration des instructions Atomic notamment en mémoire *Shared* ainsi que dans la gestion améliorée des caches.

	D_d					
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	D_g
GTX 480 (Gp/s)	0,87	0,89	1,10	1,22	1,24	1,05
GTX 780_{Ti} (Gp/s)	2,68	2,80	3,30	3,85	4,01	3,42
GTX 980_{Ti} (Gp/s)	4,37	4,63	5,19	5,78	5,87	5,25

TABLE 7.14 – WARP GPU : Débit moyen D_d en Gp/s pour des images de taille 2048×2048 et de granularité $g \in \{1, 2, 4, 8, 16\}$ et D_g en fonction de la granularité sur les cartes de tests

7.3.8 Dépendance à la taille de l'image

Tout comme les algorithmes directs, WARP GPU est sensible à la taille des images. Contrairement aux algorithmes pixels qui ne passaient pas à l'échelle et dont la performance diminuait avec l'augmentation de la taille des images, WARP GPU accélère avec celle-ci comme LSL_{RLE}. La figure 7.6 représente le débit D_d pour $g \in \{1, 4, 16\}$ en fonction de la taille des images. Pour les tailles inférieures à 512×512, la granularité n'est pas un facteur significatif. Au-delà, les trois courbes progressent séparément et atteignent leur maximum respectif pour des images de taille 4096×4096. Pour les images

de taille 2048×2048 qui ont été utilisées pour les mesures précédentes, WARP GPU atteint 93% de la performance maximale.

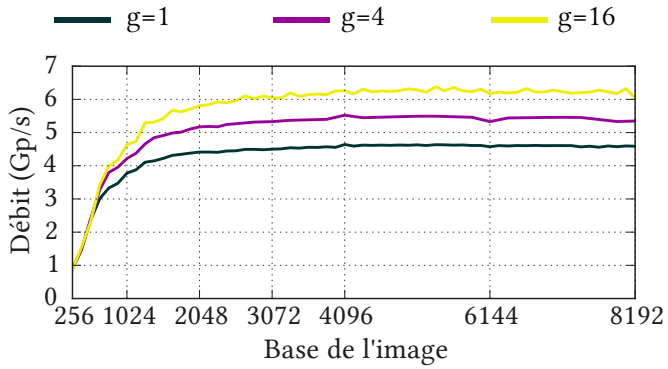


Fig. 7.7 – WARP GPU : Débit D en Gp/s pour des images de taille allant de 256×256 à 8192×8192 et de granularité $g \in \{1, 4, 16\}$ sur une carte GTX 980_{Ti}

Le débit maximal D_∞ pour chaque granularité ainsi que les taille $N_{0,5}$, $N_{0,90}$ et $N_{0,95}$ des images respectivement pour 50%, 90% et 95% de la performance sont présentés dans la table 7.15. On constate que comparativement aux algorithmes directs sur les architectures multi-cœur, WARP GPU est déjà efficace pour des images de petite taille et atteint son plein potentiel plus tôt.

Granularité	D_∞ (Gp/s)	$N_{0,5}$	$N_{0,9}$	$N_{0,95}$
$g = 1$	4,6	640×640	1536×1536	2048×2048
$g = 4$	5,4	768×768	1664×1664	2304×2304
$g = 16$	6,2	768×768	1792×1792	2688×2688

Table 7.15 – WARP GPU : Taille des images permettant d'atteindre 50%, 90% et 95% de la performance maximale D_∞

7.3.9 Prise en compte des temps de transferts

Les transferts de données entre la mémoire de l'hôte et celle du GPU passent par le bus PCIe (ici en version 3.0 dont le débit en 16X est $\approx 16 Go/s$). Ces transferts introduisent une latence non-négligeable et doivent être pris en compte dans le temps total d'une application sur GPU [31]. Deux approches sont pourtant possibles :

- considérer l'étiquetage en composantes connexes comme une des étapes d'une chaîne de traitement d'images embarquées sur GPU, ce qui suppose que l'image est présente dans la mémoire du GPU avant l'étiquetage et qu'un autre code GPU exploitera le résultat. C'est l'approche retenue par [119], [116] et [115],
- considérer l'étiquetage en composantes connexes comme un traitement autonome. Dans ce cas, les communications font partie du temps d'étiquetage.

La table 7.16 indique les temps avec et sans communications et la table 7.17 les débits correspondants. Il est possible d'optimiser le recouvrement des calculs et des communications pour diminuer le temps total. L'approche que nous privilégierons par la suite sera de développer le calcul des descripteurs sur GPU afin de ne transférer au CPU que les informations réellement utiles.

	t_d					
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	t_g
Sans transfert	0,96	0,91	0,81	0,73	0,71	0,80
Avec transferts	2,59	2,54	2,44	2,36	2,34	2,43

TABLE 7.16 – WARP GPU : temps t_d en ms avec et sans prise en compte des communications pour des images de taille 2048×2048 et de granularité $g \in \{1, 2, 4, 8, 16\}$ et t_g en fonction de la granularité sur les cartes de tests

	D_d					
	$g = 1$	$g = 2$	$g = 4$	$g = 8$	$g = 16$	D_g
Sans transfert	4,37	4,63	5,19	5,78	5,87	5,25
Avec transferts	1,62	1,65	1,71	1,77	1,79	1,72

TABLE 7.17 – WARP GPU : Débit D_d en Gp/s avec et sans prise en compte des communications pour des images de taille 2048×2048 et de granularité $g \in \{1, 2, 4, 8, 16\}$ et D_g en fonction de la granularité sur les cartes de tests

7.3.10 Conclusion pour les algorithmes de la classe WARP GPU

La confrontation au jeu de données a montré que WARP GPU est une proposition efficace d'algorithme d'étiquetage en composantes connexes dédié aux architectures GPU. Il est très adapté dans le cadre d'une chaîne de traitement intégrée dans la carte GPU. Dans le cas d'une application où le GPU est utilisé comme accélérateur pour déporter le traitement de l'étiquetage, bien qu'il soit plus performant que les algorithmes MPAR, ils n'est pas encore comparable aux algorithmes directs. En effet, il ne produit que l'image étiquetée et pas les descripteurs. Dans des travaux ultérieurs à la thèse nous développerons un algorithme de calcul des descripteurs spécifique au GPU.

7.4 Conclusion

Les algorithmes de la famille MPAR et de la classe WARP ont fait la preuve qu'ils avaient le potentiel pour mieux passer à l'échelle que les algorithmes directs avec l'augmentation continue du nombre de cœurs. Cependant, il reste du travail à accomplir pour développer l'analyse en composantes connexes MPAR ou WARP et s'intégrer dans des applications réelles. Dans le même temps l'arrivée prochaine du jeu d'extension AVX 512 va permettre d'accélérer les algorithmes directs à l'aide des lectures et écritures dispersées (gather-scatter).

La 34 – au bout !

–La Horde du contrevent, Alain Damasio

Conclusion

Le propos de cette thèse était de **comprendre** les algorithmes d'étiquetage en composantes connexes modernes et leurs spécificités, **d'évaluer** les apports de chacun de ces algorithmes pour finalement **contribuer** au domaine par l'apport de nouveaux algorithmes plus performants.

État des lieux des algorithmes séquentiels

Le point de départ de nos travaux était notre volonté de confronter l'algorithme LSL, créé en 2000 par Lionel Lacassagne, à l'évolution des architectures et de l'état de l'art. Pour cela, nous avons procédé dans les trois premiers chapitres à un état des lieux des algorithmes et de leurs performances. Afin d'être le plus exhaustif et le plus juste possible, nous avons proposé et publié une méthodologie de test des performances basée sur :

- un jeu d'images aléatoires reproductible dont les paramètres essentiels (densité, granularité et taille) permettent de proposer des niveaux de difficultés variables,
- un jeu d'images (SIDBA), déjà utilisé dans de nombreux articles qui donne un aperçu rapide des performances des algorithmes dans un contexte réaliste.

Au cours de nos travaux, nous avons mis en évidence la variété du paysage de l'étiquetage en composantes connexes au travers des différentes variantes d'algorithmes. Nous en avons extrait une base représentative, qui illustre le comportement des différentes améliorations proposées par la littérature depuis 2000. Le chapitre 3 a mis en évidence les performances générales des algorithmes d'étiquetage en composantes connexes, mais aussi les performances des différentes phases qui les constituent.

L'analyse en composantes connexes, qui dérive de l'étiquetage en composantes connexes, correspond à l'utilisation de l'étiquetage par une application car elle calcule des descripteurs sur les composantes connexes directement exploitables par d'autres algorithmes. Les performances des algorithmes ont été mesurées et présentées tant pour l'étiquetage que pour l'analyse en composantes connexes. Il en ressort que :

- dans les deux cas, les algorithmes LSL et en particulier LSL_{RLE} sont les algorithmes directs les plus rapides dans un contexte séquentiel,
- pour les images de la base SIDBA, sur une architecture Skylake (la plus récente au moment de nos travaux), LSL_{RLE} est plus rapide d'un rapport $\times 3{,}1$ que tous les algorithmes pixels,
- le calcul des descripteurs doit être réalisé pendant l'étiquetage pour éviter de réaliser inutilement l'étape de réétiquetage. Dans un contexte de systèmes autonomes qui devient le contexte majoritaire de la vision par ordinateur, l'analyse en composantes connexes est la classe d'algorithme à évaluer et à développer.

Parallélisation

Une des conclusions du chapitre 3, est que la performance des algorithmes dans un contexte séquentiel ne progresse plus avec les nouvelles générations de processeurs. En effet, les versions séquentielles n'exploitent pas les progrès liés à l'augmentation du nombre de cœurs. Dans le chapitre 4,

nous avons décrit nos contributions dans ce domaine. Nous avons proposé une version parallèle de l'algorithme LSL ainsi qu'une infrastructure logicielle de parallélisation de l'ensemble des algorithmes directs basées sur un découpage en bandes et une fusion pyramidale des bandes, qui permet de s'affranchir des problèmes liés à la concurrence tout en maximisant l'exploitation de chaque cœur.

La méthodologie d'évaluation construite au chapitre 2, a été appliquée à ces nouveaux algorithmes et a mis en évidence plusieurs points remarquables :

- l'algorithme LSL_{RLE} est l'algorithme d'étiquetage en composantes connexes le plus rapide dans un contexte parallèle,
- la parallélisation de tous les algorithmes sur les machines à petit nombre de cœurs est très efficace,
- sur les machines ayant plus de cœurs et de sockets, aucun algorithme pixels ne passe réellement à l'échelle et seule la version parallèle de LSL_{RLE} tire parti de l'augmentation du nombre de cœurs et du nombre de sockets,
- pour les images de grande taille, seule la version parallèle de LSL_{RLE} permet un traitement rapide en exploitant un grand nombre de cœurs,
- La gestion des étiquettes de Suzuki, qui était compétitive pour les algorithmes séquentiels est très dégradée lors du passage au multi-cœur pour les images les moins structurées.

Une conséquence des points précédents est que sur une machine 4 cœurs de dernière génération, LSL_{RLE} est plus rapide que le plus rapide des algorithmes pixels sur l'ensemble des machines de tests, y compris sur un serveur de calcul à 4 sockets de 15 cœurs chacun. LSL_{RLE} est le seul algorithme direct adapté aux architectures multi-socket.

Algorithmes itératifs

Les premiers chapitres ont mis en évidence les points forts et les points faibles des algorithmes directs. LSL_{RLE} est le plus rapide des algorithmes d'étiquetage et d'analyse en composantes connexes. Cependant, la nécessité d'augmenter la taille des données pour utiliser le plein potentiel des architectures proposant un grand nombre de cœurs nous a encouragés à ouvrir de nouvelles voies d'étude, notamment les algorithmes itératifs qui nous semblaient mieux armés pour gérer l'augmentation du nombre de cœurs.

Dans le chapitre 6, nous avons donc proposé un algorithme itératif (MPAR FB + SIMD + OMP + AT) ainsi qu'une nouvelle classe d'algorithmes itératifs (WARP) dont une déclinaison réalise l'étiquetage en deux passes. Enfin, nous avons proposé une implémentation GPU des algorithmes de la classe WARP (WARP GPU). Ces algorithmes, très différents des algorithmes directs, ne réalisent pas l'analyse en composantes connexes de manière native et les comparaisons avec les algorithmes directs doivent en tenir compte.

Pour MPAR FB + SIMD + OMP + AT (qui utilise les instructions SIMD, le multi-cœur ainsi qu'un mécanisme de tuiles actives), les résultats présentés au chapitre 7 sont très encourageants. En effet, l'augmentation du nombre de cœurs est très favorable aux performances de l'algorithme itératif. Bien que moins rapide que les algorithmes directs, son potentiel de progression avec l'augmentation du nombre de cœurs ainsi qu'avec l'augmentation de la largeur du SIMD est supérieur.

Pour WARP GPU, les performances atteintes sont supérieures à l'état de l'art et supérieure à MPAR FB + SIMD + OMP + AT sur le Xeon Phi.

Perspectives de recherche

Au-delà des résultats, les travaux présentés dans ce manuscrit ont permis d'ouvrir de nouveaux axes de recherche dans le champ de l'étiquetage en composantes connexes.

- Les algorithmes itératifs sont prometteurs pour l'étiquetage et il est nécessaire de les doter de mécanismes efficaces de calcul des descripteurs.
- La méthode d'union récursive, utilisant des instructions *atomic*, peut s'appliquer aux algorithmes directs et peut potentiellement diminuer le temps des fusions de bandes. Il serait intéressant d'évaluer la performance de ce mécanisme sur les architectures classiques (GPP).
- Des travaux sont déjà engagés pour étudier la portabilité des algorithmes directs et itératifs sur l'architecture many-core TSAR (TeraScale Architecture) du LIP6 [121–123] qui possède des mécanismes materiels et logiciels qui permettent/favorisent un passage à l'échelle d'un grand nombre d'algorithmes.
- L'arrivée prévue en 2017 de processeurs généralistes supportant les instructions SIMD AVX 512 (Xeon Skylake) va permettre d'utiliser les instructions d'adressage dispersé (*gather scatter*). Nous travaillons déjà à la création d'algorithmes directs utilisant cette propriété.

L'expérience acquise sur l'étiquetage pourrait aussi profiter à d'autres domaines et nous étudions les possibilités d'en étendre les applications.

Bibliographie

[1] Stuart J. Russell et Peter Norvig. *Artificial Intelligence : A Modern Approach*. 3e éd. Pearson Education, 2009. isbn : 0-13-604259-7.

[2] Chris Urmson et al. « Autonomous Driving in Urban Environments : Boss and the Urban Challenge ». In : *The DARPA Urban Challenge : Autonomous Vehicles in City Traffic*. Sous la dir. de Martin Buehler, Karl Iagnemma et Sanjiv Singh. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, p. 1–59. isbn : 978-3-642-03991-1. doi : 10.1007/978-3-642-03991-1_1.

[3] S. Wybo, D. Tsishkou, C. Vestri, F. Abad, S. Bougnoux et R. Bendahan. « Monocular vision obstacles detection for autonomous navigation ». In : *International Conference on Intelligent Robots and Systems, Acropolis Convention Center, Nice, France*. Sept. 2008, p. 4190. doi : 10.1109/IROS.2008.4651245.

[4] Paweł Gora et Inga Rüb. « Traffic Models for Self-driving Connected Cars ». In : *Transportation Research Procedia* 14 (2016), p. 2207–2216. doi : 10.1016/j.trpro.2016.05.236.

[5] Jeamin Koo, Jungsuk Kwac, Wendy Ju, Martin Steinert, Larry Leifer et Clifford Nass. « Why did my car just do that ? Explaining semi-autonomous driving actions to improve driver understanding, trust, and performance ». In : *International Journal on Interactive Design and Manufacturing (IJIDeM)* 9.4 (2015), p. 269–275. issn : 1955-2505. doi : 10.1007/s12008-014-0227-2.

[6] Michael Wagner et Philip Koopman. « A Philosophy for Developing Trust in Self-driving Cars ». In : *Road Vehicle Automation 2*. Sous la dir. de Gereon Meyer et Sven Beiker. Cham : Springer International Publishing, 2015, p. 163–171. isbn : 978-3-319-19078-5. doi : 10.1007/978-3-319-19078-5_14.

[7] Caitlin Lustig, Katie Pine, Bonnie Nardi, Lilly Irani, Min Kyung Lee, Dawn Nafus et Christian Sandvig. « Algorithmic Authority : The Ethics, Politics, and Economics of Algorithms That Interpret, Decide, and Manage ». In : *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA '16. Santa Clara, California, USA : ACM, 2016, p. 1057–1062. doi : 10.1145/2851581.2886426.

[8] Kunio Doi. « Computer-Aided Diagnosis in Medical Imaging : Historical Review, Current Status and Future Potential ». In : *Computerized medical imaging and graphics : the official journal of the Computerized Medical Imaging Society* (2017). doi : 10.1016/j.compmedimag.2007.02.002.

[9] Laszlo Marak, Jean Cousty, Laurent Najman et Hugues Talbot. « 4D Morphological segmentation and the miccai LV-segmentation grand challenge ». In : *MICCAI 2009 Workshop on Cardiac MR Left Ventricle Segmentation Challenge*. MIDAS Journal 1. France : MIDAS, nov. 2009, p. 1–8.

[10] Elias N. Malamas, Euripides G. M. Petrakis, Michalis Zervakis, Laurent Petit et Jean-didier Legat. « A survey on industrial vision systems, applications and tools, Image and Vision Computing ». In : *Image and Vision Computing* 21 (2003), p. 171–188.

[11] Hugues Talbot, Dominique Jeulin et Daniel Hanton. « Image analysis of insulation mineral fibres ». In : *Microscopy Microanalysis Microstructures* 7.5-6 (1996), p. 361–368.

[12] Ray Smith. « An Overview of the Tesseract OCR Engine ». In : *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*. 2007, p. 629–633.

[13] Sandip Rakshit et Subhadip Basu. « Recognition of Handwritten Roman Script Using Tesseract Open source OCR Engine ». In : *CoRR* abs/1003.5891 (2010).

[14] Ana Rebelo, Ichiro Fujinaga, Filipe Paszkiewicz, Andre R. S. Marcal, Carlos Guedes et Jaime S. Cardoso. « Optical music recognition : state-of-the-art and open issues ». In : *International Journal of Multimedia Information Retrieval* 1.3 (2012), p. 173–190. issn : 2192-662X. doi : 10.1007/s13735-012-0004-6.

[15] Larry Matthies et al. « Computer Vision on Mars ». In : *International Journal of Computer Vision* (2007).

[16] Alexey Kurakin, Zhengyou Zhang et Zicheng Liu. « A real time system for dynamic hand gesture recognition with a depth sensor ». In : *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European*. IEEE. 2012, p. 1975–1979.

[17] Hui Chen et Bir Bhanu. « Human Ear Detection from Side Face Range Images ». In : *Proceedings of the 17th International Conference on Pattern Recognition*. ICPR'04. 2004.

[18] A. Romero, M. Gouiffès et L. Lacassagne. « Covariance Descriptor Multiple Object Tracking and Re-Identification with Colorspace Evaluation ». In : *IEEE ACCV - Workshop on Detection and Tracking in Challenging Environnements*. 2012.

[19] F. Laguzet, A. Romero, M. Gouiffès, L. Lacassagne et D. Etiemble. « Color tracking with contextual switching : Real-time implementation on CPU ». In : *Journal of Real-Time Image Processing* 10.2 (2015), p. 403–422. issn : 1861-8219. doi : 10.1007/s11554-013-0358-x.

[20] L. Lacassagne, A. Manzanera, J. Denoulet et A. Mérigot. « High Performance Motion Detection : Some trends toward new embedded architectures for vision systems ». In : *Journal of Real Time Image Processing* (oct. 2008), p. 127–148. doi : 10.1007/s11554-008-0096-7.

[21] L. Lacassagne, A. Manzanera et A. Dupret. « Motion detection : fast and robust algorithms for embedded systems ». In : *IEEE International Conference on Image Analysis and Processing (ICIP)*. 2009, p. 3265–3268.

[22] V. Kantorov et I. Laptev. « Efficient feature extraction, encoding and classification for action recognition ». In : *IEEE Conference on Computer Vision and Pattern Recognition*. 2014.

[23] Marie-Odile Berger et Gilles Simon. « Réalité augmentée : entre mythes et réalités ». In : *Interstices* (mar. 2016).

[24] Olga Russakovsky et al. « ImageNet Large Scale Visual Recognition Challenge ». In : *International Journal of Computer Vision (IJCV)* 115.3 (2015), p. 211–252. doi : 10.1007/s11263-015-0816-y.

[25] Cour des comptes. *Rapport public annuel 2002 : Les aéroports de Paris*. 2003.

[26] Aéroport de paris. *Document de référence et rapport financier annuel 2015*. 2016.

[27] Laurent Cabaret. *Entretien avec le service Presse Communication financière, Immobilier et Filiales du groupe ADP*. 2016.

[28] L. Lacassagne. « Détection de mouvement et suivi d'objets en temps réel ». Thèse de doct. University of Paris 6, 2000.

[29] J. Denoulet, G. Mostafaoui, L. Lacassagne et A. Mérigot. « Implementing motion Markov detection on General Purpose Processor and Associative Mesh ». In : *Computer Architecture and Machine Perception*. IEEE. 2005.

[30] James Fung et Steve Mann. « Using graphics devices in reverse : GPU-based image processing and computer vision ». In : *2008 IEEE international conference on multimedia and expo*. IEEE. 2008, p. 9–12.

[31] Victor W. Lee et al. « Debunking the 100X GPU vs. CPU Myth : An Evaluation of Throughput Computing on CPU and GPU ». In : *SIGARCH Comput. Archit. News* 38.3 (juin 2010), p. 451–460. issn : 0163-5964. doi : 10.1145/1816038.1816021.

[32] Hussein M. Alnuweiri et V. K. Prasanna Kumar. « Fast Image Labeling Using Local Operators on Mesh-Connected Computers ». In : *IEEE Trans. Pattern Anal. Mach. Intell.* 13.2 (fév. 1991), p. 202–207. issn : 0162-8828. doi : 10.1109/34.67649.

[33] Alain Mérigot. « Associative Nets : A Graph-Based Parallel Computing Model ». In : *IEEE Trans. Comput.* 46.5 (mai 1997), p. 558–571. issn : 0018-9340. doi : 10.1109/12.589222.

[34] Julien Denoulet. « Architectures massivement paralleles de systemes sur circuits (SoC) pour le traitement de flux vidéos ». Thèse de doct. University Paris Sud, 2004.

[35] D. Wentzlaff et al. « On-Chip Interconnection Architecture of the Tile Processor ». In : *IEEE Micro* 27.5 (sept. 2007), p. 15–31. issn : 0272-1732. doi : 10.1109/MM.2007.4378780.

[36] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns et al. « Introduction to the cell multiprocessor ». In : *IBM journal of Research and Development* 49.4/5 (2005), p. 589.

[37] Ghassan Almaless et Franck Wajsburt. « On the scalability of image and signal processing parallel applications on emerging cc-NUMA many-cores ». In : *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*. IEEE. 2012, p. 1–8.

[38] Ghassan Almaless et Franck Wajsburt. « Does shared-memory, highly multi-threaded, single-application scale on many-cores ». In : *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*. 2012.

[39] Ghassan Almaless. « Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core ». Thèse de doct. Paris 6, 2014.

[40] T. Saidani, J. Falcou, C. Tadonki, L. Lacassagne et Daniel Etiemble. « Algorithmic Skeletons within an Embedded Domain Specific Language for the Cell Processor ». In : *Parallel Architectures and Compilation Techniques, PACT*. 2009, p. 67–76.

[41] F. Dias, F. Berry, J. Serot et F. Marmoiton. « Hardware, Design and Implementation Issues on a Fpga-Based Smart Camera ». In : *2007 First ACM/IEEE International Conference on Distributed Smart Cameras*. Sept. 2007, p. 20–26. doi : 10.1109/ICDSC.2007.4357501.

[42] D. Demigny, L. Kessal, R. Bourguiba et N. Boudouani. « How to use high speed reconfigurable FPGA for real time image processing ? » In : *Computer Architectures for Machine Perception, 2000. Proceedings. Fifth IEEE International Workshop on*. 2000, p. 240–246. doi : 10.1109/CAMP.2000.875983.

[43] Sébastien Pillement, Raphaël David et Olivier Sentieys. « Architectures reconfigurables : opportunités pour la faible consommation ». In : *papier invité aux journées Faible Tension Faible Consommation (FTFC)* (2003).

[44] Michal Fularz, Marek Kraft, Adam Schmidt et Andrzej Kasinski. « A high-performance FPGA-based image feature detector and matcher based on the fast and brief algorithms ». In : *International Journal of Advanced Robotic Systems* 12 (2015).

[45] H. Ye, L. Lacassagne, D. Etiemble, L. Cabaret, J. Falcou et O. Florent. « Impact of High Level Transforms on High Level Synthesis for motion detection algorithm ». In : *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2012, p. 1–8.

[46] H. Ye, L. Lacassagne, J. Falcou, D. Etiemble, L. Cabaret et O. Florent. « High Level Transforms to reduce energy consumption of signal and image processing operators ». In : *IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2013, p. 247–254.

[47] L. Lacassagne, D. Etiemble, A Hassan-Zahraee, A. Dominguez et P. Vezolle. « High Level Transforms for SIMD and low-level computer vision algorithms ». In : *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*. 2014, p. 49–56.

[48] É Grossiord, H. Talbot, N. Passat, M. Meignan, P. Tervé et L. Najman. « Hierarchies and shape-space for pet image segmentation ». In : *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*. Avr. 2015, p. 1118–1121. doi : 10.1109/ISBI.2015.7164068.

[49] Michael J. Klaiber, Zhe Wang et Sven Simon. « A Real-Time Process Analysis System for the Simultaneous Acquisition of Spray Characteristics ». In : *Process-Spray : Functional Particles Produced in Spray Processes*. Sous la dir. d'Udo Fritsching. Cham : Springer International Publishing, 2016, p. 265–305. isbn : 978-3-319-32370-1. doi : 10.1007/978-3-319-32370-1_8.

[50] Florian Wende, Guido Laubender et Thomas Steinke. « Integration of Intel Xeon Phi Servers into the HLRN-III Complex : Experiences, Performance and Lessons Learned ». In : *CUG2014 Proceedings*. 2014.

[51] Xun Wang, Jie Sun et Hao-Yu Peng. « Foreground object detecting algorithm based on mixture of gaussian and kalman filter in video surveillance ». In : *Journal of Computers* 8.3 (2013), p. 693–700.

[52] K. Aneja, F. Laguzet, L. Lacassagne et A. Merigot. « Video rate image segmentation by means of region splitting and merging ». In : *IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*. 2009.

[53] L. Lacassagne, M. Milgram et J. Devars. « Light Speed Labeling : un nouvel algorithme d'étiquetage en composantes connexes ». In : *Workshop Adéquation Algorithme Architecture*. 2000.

[54] L. Lacassagne et A. B. Zavidovique. « Light Speed Labeling for RISC architectures ». In : *IEEE International Conference on Image Analysis and Processing (ICIP)*. 2009.

[55] Y. Yang et D. Zhang. « A novel line scan clustering algorithm for identifying connected components in digital images ». In : *Image and Vision Computing* (2003). doi : 10.1016/S0662-8856(03)00015-5.

[56] K. Suzuki, I. Horiba et N. Sugie. « Linear-time connected component labeling based on sequential local operations ». In : *Computer Vision and Image Understanding* 89.1 (jan. 2003), p. 1–23. doi : http://dx.doi.org/10.1016/S1077-3142(02)00030-9.

[57] F. Chang et C. Chen. « A linear-time component-labeling algorithm using contour tracing technique ». In : *Computer Vision and Image Understanding* 93 (2004), p. 206–220.

[58] L. He, Y. Chao et K. Suzuki. « A run-based two-scan labeling algorithm ». In : *ICIAR*. LNCS 4633. 2007, p. 131–142.

[59] K. Wu, E. Otoo et A. Shoshani. « Optimizing connected component labeling algorithms ». In : *Pattern Analysis and Applications* (2008). doi : 10.1007/s10044-008-0109-y.

[60] C. Grana, D.Borghesani et R. Cucchiara. « Fast Block Based Connected Components Labeling ». In : *ICIP*. IEEE. 2009, p. 4061–4064.

[61] L. He, Y. Chao et K. Suzuki. « An efficient first-scan method for label-equivalence-based labeling algorithms ». In : *Pattern Recognition Letters* 31.1 (2010), p. 28–35.

[62] L. He, Y. Chao et K. Suzuki. « A New Two-Scan Algorithm for Labeling Connected Components in binary Images ». In : *Proceedings of the World Congress on Engineering*. Sous la dir. de World Congress. T. 2. 2012, p1141–1146.

[63] U.H. Hernandez-Belmonte, V. Ayala-Ramirez et R.E. Sanchez-Yanez. « Enhancing CCL algorithms by using a reduced connectivity mask ». In : *Mexican Conference on Pattern Recognition*. Sous la dir. de Springer. 2013, p. 195–203.

[64] S. Gupta, D. Palsetia, M.M. Ali Patwary, A. Agrawal et A. Choudhary. « A New Parallel Algorithm for Two-Pass Connected Component Labeling ». In : *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2014, p. 1355–1362.

[65] H. M. Alnuweiti et V. K. Prasanna. « Parallel architectures and algorithms for image component labeling ». In : *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.10 (oct. 1992), p. 1014–1034. issn : 0162-8828. doi : 10.1109/34.159904.

[66] David A. Bader et Joseph JáJá. « Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study (Extended Abstract) ». In : PPOPP '95 (1995), p. 123–133. doi : 10.1145/209936.209950.

[67] Robert Walczyk, Alistair Armitage et David Binnie. « Comparative study on connected component labeling algorithms for embedded video processing systems. » In : *IPCV'10*. Sous la dir. d'Hamid Deligiannidis. T. 2. Las Vegas, USA : CSREA Press, 2010.

[68] D. Bailey et C. Johnston. « Single Pass Connected Component Analysis ». In : *Image and Vision New Zeland (IVNZ)*. 2007, p. 282–287.

[69] M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud et S. Simon. « A Memory-Efficient Parallel Single Pass Architecture for Connected Component Labeling of Streamed Images ». In : *International Conference on Field Programmable Technology (FPT)*. IEEE. 2012, p. 159–165.

[70] M. Klaiber, D. Bailey, S. Ahmed, Y. Baroud et S. Simon. « A High-Throughput FPGA Architecture for Parallel Connected Components Analysis Based on Label Reuse ». In : *International Conference on Field Programmable Technology (FPT)*. IEEE. 2013, p. 302–305.

[71] M. Jablonski et M. Gorgon. « Handel-C implementation of classical component labelling algorithm ». In : *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*. Août 2004, p. 387–393. doi : 10.1109/DSD.2004.1333301.

[72] E. Mozef, S. Weber, J. Jaber et E. Tisserand. « Parallel architecture dedicated to connected component analysis ». In : *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*. T. 4. Août 1996, 699–703 vol.4. doi : 10.1109/ICPR.1996.547655.

[73] A. Rosenfeld et J.L. Platz. « Sequential operator in digital pictures processing ». In : *Journal of ACM* 13,4 (1966), p. 471–494.

[74] J.M. Chassery et A. Montanvert. *Géometrie discrète en analyse d'image*. Traité des Nouvelles technologies, Hermes, 1991, p. 200–214. isbn : 2-86601-271-2.

[75] R.M. Haralick et L.G. Shapiro. *Computer and Robot Vision*. T. 1. Addison-Wesley, 1992.

[76] Hanan Samet. « Connected Component Labeling Using Quadtrees ». In : 28.3 (juil. 1981), p. 487–501. issn : 0004-5411 (print), 1557-735X (electronic).

[77] Michael B. Dillencourt, Hanan Samet et Markku Tamminen. « A General Approach to Connected-Component Labelling for Arbitrary Image Representations ». In : 39.2 (avr. 1992), p. 253–280. issn : 0004-5411 (print), 1557-735X (electronic).

[78] R. Kadowaki, K. Motomura, S. Ohkura et K. Aizawa. « Graphs representing quadtree structures using eight edges ». In : *Communications, Control and Signal Processing (ISCCSP), 2010 4th International Symposium on*. Mar. 2010, p. 1–5. doi : 10.1109/ISCCSP.2010.5463412.

[79] Vikrant KHANNA, Phalguni GUPTA et C. Jinshong HWANG. « Maintenance of Connected Components in Quadtree-based Image Representation. » In : *ITCC*. IEEE Computer Society, 2001, p. 647–651. ISBN : 0-7695-1062-0.

[80] Zvi GALIL et Giuseppe F. ITALIANO. « Data Structures and Algorithms for Disjoint Set Union Problems ». In : *ACM Comput. Surv.* 23.3 (sept. 1991), p. 319–344. ISSN : 0360-0300. DOI : 10.1145/116873.116878.

[81] M.A. PATWARY, J.R. BLAIR et F. MANNE. « Experiments on Union-Find algorithms for the disjoint-set data structure ». In : *International symposium on experimental algorithms (SEA)*. Sous la dir. de LNCS 6049 SPRINGER. 2010, p. 411–423.

[82] R.M. HARALICK. « Some neighborhood operations ». In : *Real-Time Parallel Computing Image Analysis*. Plenum Press. 1981, p. 11–35.

[83] Lionel LACASSAGNE, Laurent CABARET, Daniel ETIEMBLE, Farouk HEBACHE et Andrea PETRETO. « A New SIMD Iterative Connected Component Labeling Algorithm ». In : *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. WPMVP '16. Barcelona, Spain : ACM, 2016, 1 :1–1 :8. ISBN : 978-1-4503-4060-1. DOI : 10.1145/2870650.2870652.

[84] Stephen WARSHALL. « A Theorem on Boolean Matrices ». In : *J. ACM* 9.1 (jan. 1962), p. 11–12. ISSN : 0004-5411. DOI : 10.1145/321105.321107.

[85] Bernard A. GALLER et Michael J. FISCHER. « An improved equivalence algorithm ». In : *Commun. ACM* 7.5 (1964), p. 301–303. DOI : 10.1145/364099.364331.

[86] R. LUMIA, L. SHAPIRO et O. ZUNGIA. « A new connected components algorithms for virtual memory computers ». In : *Computer Vision, Graphics and Image Processing* 22-2 (1983), p. 287–300.

[87] C. RONSE et P.A. DEJVIJVER. « Connected components in binary images : the detection problems ». In : *Research Studies Press*. 1984.

[88] M SAKAUCHI, Y OHSAWA, M SONE et M ONOE. « Management of the standard image database for image processing researches (SIDBA) ». In : *ITEJ Technial Report* 8.38 (1984), p. 7–12.

[89] Brian LEININGER et al. *Autonomous real-time ground ubiquitous surveillance-imaging system (ARGUS-IS)*. 2008. DOI : 10.1117/12.784724.

[90] P. VANDEWALLE, J. KOVACEVIC et M. VETTERLI. « Reproducible research in signal processing ». In : *Signal Processing Magazine* 26,3 (2009), p. 37–47.

[91] M. MATSUMOTO et T. NISHIMURA. « Mersenne twister : A 623-dimensionally equidistributed uniform pseudorandom number generator ». In : *Transactions on Modeling and Computer simulation* 8.1 (1998), p. 3–30.

[92] Zorica V DJORDJEVIC, H Eugene STANLEY et Alla MARGOLINA. « Site percolation threshold for honeycomb and square lattices ». In : *Journal of Physics A : Mathematical and General* 15.8 (1982), p. L405.

[93] N. OTSU. « A threshold selection method from gray-level histograms ». In : *Transactions on System, Man and Cybernetics* 9 (1979), p. 62–66.

[94] John E. HOPCROFT et Jeffrey D. ULLMAN. « Set Merging Algorithms ». In : *SIAM J. Comput.* 2.4 (1973), p. 294–303. DOI : 10.1137/0202024.

[95] R.E. TARJAN. « Efficiency of good but not linear set union algorithm ». In : *Journal of ACM* 22,2 (1975), p. 215–225.

[96] F. VEILLON. « One pass computation of morphological and geometrical properties of objects in digital pictures ». In : *Signal Processing* 1,3 (1979), p. 175–179.

[97] L. Lacassagne et B. Zavidovique. « Light Speed Labeling : Efficient connected component labeling on RISC architectures ». In : *Journal of Real-Time Image Processing* 6.2 (2011), p. 117–135.

[98] L. Cabaret et L. Lacassagne. « A Review of World's Fastest Connected Component Labeling Algorithms : Speed and Energy Estimation ». In : *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2014, p. 1–8.

[99] L. Cabaret et L. Lacassagne. « What is the world's fastest Connected Component Labeling Algorithm ? » In : *IEEE International Workshop on Signal Processing Systems (SiPS)*. 2014, p. 97–102.

[100] Herb Sutter. « The Free Lunch Is Over : A Fundamental Turn Toward Concurrency in Software ». In : *Dr. Dobb's Journal* 30.3 (mar. 2005).

[101] M. Niknam, P. Thulasiraman et S. Camorlinga. « A Parallel Algorithm for Connected Component Labeling of Gray-scale Images on Homogeneous Multicore Architectures ». In : *Journal of Physics - High Performance Computing Symposium (HPCS)* (2010). doi : doi:10.1088/1742-6596/256/1/012010.

[102] C.-W. Chen, Y.-T. Wu, S.-Y. Tseng et W.-S. Wang. « Parallelization of Connected-Component Labeling on TILE64 Many-Core Platform ». In : *Journal of Signal Processing Systems* 75,2 (2013), p. 169–183.

[103] Leonardo Dagum et Ramesh Menon. « OpenMP : An Industry-Standard API for Shared-Memory Programming ». In : *IEEE Comput. Sci. Eng.* 5.1 (jan. 1998), p. 46–55. issn : 1070-9924. doi : 10.1109/99.660313.

[104] N. Ma, D. Bailey et C. Johnston. « Optimised Single Pass Connected Component Analysis ». In : *International Conference on Field Programmable Technology (FPT)*. IEEE. 2008, p. 185–192.

[105] L. Cabaret, L. Lacassagne et D. Etiemble. « Parallel Light Speed Labeling : an efficient connected component labeling algorithm for multi-core processors ». In : *IEEE International Conference on Image Processing (ICIP)*. 2015, p. 1–4.

[106] L. Cabaret, L. Lacassagne et D. Etiemble. « Parallel Light Speed Labeling : an efficient connected component algorithm for labeling and analysis on multi-core processors ». In : *Journal of Real Time Image Processing* (2016), p. 1–18.

[107] Daniel Robbins. « POSIX threads explained A simple and nimble tool for memory sharing ». In : *IBM developerWorks* (). url : http://www.ibm.com/developerworks/library/l-posix1/l-posix1-pdf.pdf.

[108] Chuck Pheatt. « Intel&Reg ; Threading Building Blocks ». In : *J. Comput. Sci. Coll.* 23.4 (avr. 2008), p. 298–298. issn : 1937-4771.

[109] CilkPlus. *Intel Cilk Plus homepage*. Consultée le 31 mars, 2016.

[110] Gene M. Amdahl. « Validity of the single processor approach to achieving large scale computing capabilities. » In : *AFIPS Spring Joint Computing Conference*. T. 30. AFIPS Conference Proceedings. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967, p. 483–485.

[111] Michael J. Flynn. « Some Computer Organizations and Their Effectiveness ». In : *IEEE Trans. Comput.* 21.9 (sept. 1972), p. 948–960. issn : 0018-9340. doi : 10.1109/TC.1972.5009071.

[112] Azriel Rosenfeld. « Geodesics in Digital Pictures ». In : *Information and Control* 36.1 (1978), p. 74–84. doi : 10.1016/S0019-9958(78)90237-1.

[113] M. McCool, A.D. Robison et J. Reinders. *Structured Parallel Programming : patterns for efficient computation*. Morgan Kaufmann, 2012.

[114] TeraScale ARchitecture. *Projet TSAR*. Consultée le 19 mai, 2016.

[115] Sanyam Mehta, Arindam Misra, Ayush Singhal, Praveen Kumar, Ankush Mittal et Kannappan Palaniappan. « Parallel implementation of video surveillance algorithms on GPU architectures using CUDA ». In : *17th IEEE Int. Conf. Advanced Computing and Communications (ADCOM)*. 2009.

[116] Oleksandr Kalentev, Abha Rai, Stefan Kemnitz et Ralf Schneider. « Connected component labeling on a 2D grid using CUDA ». In : *Journal of Parallel and Distributed Computing* 71.4 (2011), p. 615–620.

[117] VMA Oliveira et RA Lotufo. « A study on connected components labeling algorithms using GPUs ». In : *Workshop of Undergraduate Works, XXIII Sibgrapi, Conference on Graphics, Patterns and Images*. Sept. 2010.

[118] D. Martin, C. Fowlkes, D. Tal et J. Malik. « A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics ». In : *Proc. 8th Int'l Conf. Computer Vision*. T. 2. Juil. 2001, p. 416–423.

[119] O Stava et B Benes. « Connected component labeling in CUDA ». In : *Hwu., WW (Ed.), GPU Computing Gems* (2010).

[120] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi et Andreas Moshovos. « Demystifying GPU microarchitecture through microbenchmarking ». In : *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, White Plains, NY, USA*. Mar. 2010, p. 235–246. doi : 10.1109/ISPASS.2010.5452013.

[121] Quentin Meunier, Frédéric Pétrot et Jean-Louis Roch. « Hardware/software support for adaptive work-stealing in on-chip multiprocessor ». In : *Journal of Systems Architecture* 56.8 (2010). Special Issue on HW/SW Co-Design : Tools and Applications, p. 392–406. issn : 1383-7621. doi : http://dx.doi.org/10.1016/j.sysarc.2010.06.007.

[122] Hao Liu, Clément Dévigne, Lucas Garcia, Quentin Meunier, Franck Wajsbürt et Alain Greiner. « RWT : Suppressing Write-Through Cost When Coherence is Not Needed ». In : *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE. 2015, p. 434–439.

[123] Mohamed Lamine Karaoui, Quentin L. Meunier, Franck Wajsbürt et Alain Greiner. « GECOS : Mécanisme de synchronisation passant à l'échelle à plusieurs lecteurs et un écrivain pour structures chaînées ». In : *Technique et Science Informatiques* 34 (mai 2015), p. 53–78. doi : 10.3166/tsi.34.53-78.

Annexe A

Annexes

Seule l'erreur est créatrice.

–La Horde du contrevent, Alain Damasio

A.1 Algorithmes

A.1.1 Rosenfeld & Pfalz

Algorithme 39 : Rosenfeld 4C avec Union-Find

```
   Input : I[H][W]
   Result : E[H][W], T
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          if I[i][j] ≠ 0 then
4              e2 ← E[i-1][j]
5              e4 ← E[i][j-1]
6              if e2 = e4 = 0 then
7                  ex ← ne++
8              else
9                  r2 = Find(T,e2)
10                 r4 = Find(T,e4)
11                 ε ← min+(r2,r4)
12                 if (r2 ≠ 0 and r2 ≠ ε) then Union(T, e2, ε)
13                 if (r4 ≠ 0 and r4 ≠ ε) then Union(T, e4, ε)
14                 ex ← ε
15         else
16             ex ← 0
17         E[i][j] ← ex
```

A.1.2 Haralick & Shapiro

Algorithme 40 : Haralick - propagation - première passe descendante 4C

Input : I une image binaire de taille $H \times W$ (0 pour le fond, 1 pour le premier plan)
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          e_x ← I[i][j]
4          if e_x ≠ 0 then
5              e_2 ← E[i-1][j]    e_4 ← E[i][j-1]
6              if e_2 = e_4 = 0 then
7                  ε ← ne++
8              else
9                  ε = min+(e_2, e_4)
10             E[i][j] ← ε
```

Algorithme 41 : Haralick - propagation - passe ascendante 4C

Input : E une image partiellement étiquetée de taille $H \times W$ résultant d'une phase directe
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = H − 1 to 0 do
2      for j = W − 1 to 0 do
3          e_x ← E[i][j]
4          if e_x ≠ 0 then
5              e_4 ← E[i][j+1]    e_2 ← E[i+1][j]
6              if e_2 = e_4 = 0 then
7                  ε ← ne++
8              else
9                  ε = min+(e_x, e_2, e_4)
10             E[i][j] ← ε
```

Algorithme 42 : Haralick - propagation - passe n descendante 4C

Input : E une image partiellement étiquetée de taille $H \times W$ résultant d'une phase inverse
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = 0 to H − 1 do
2      for j = 0 to W − 1 do
3          e_x ← E[i][j]
4          if e_x ≠ 0 then
5              e_2 ← E[i-1][j]    e_4 ← E[i][j-1]
6              ε = min+(e_x, e_2, e_4)
7              I[i][j] ← ε
```

(a) Masque direct (b) Masque inverse

Fig. A.1 – Masques spécifiques d'Haralick 4C

A.1.3 Lumia & Shapiro & Zuniga

Algorithme 43 : Lumia - propagation - passe n descendante 8C

Input : I une image binaire de taille $H \times W$ (0 pour le fond, 1 pour le premier plan)
Result : E l'image partiellement étiquetée de taille $H \times W$

```
1  for i = 0 to H − 1 do
       ▹ Création de la table d'équivalence vide pour la ligne courante
2      EQTABLE ←CREATE( );
3      for j = 0 to W − 1 do
           ▹ Mise à 0 de toutes les étiquettes de la ligne
4          E[i][j] ← 0;
5      for j = 0 to W − 1 do
           ▹ examine each pixel P in line L
6          if I[i][j] ≠ 1 then
7              NEXT endif
8          e1 ← E[i-1][j-1]   e2 ← E[i-1][j]   e3 ← E[i-1][j+1]   e4 ← E[i][j-1]
9          if e1 = e2 = e3 = e4 = 0 then
10             ex ← ne++
11         else
12             ε ← min+(e1, e2, e3, e4)
13             if e1 ≠ 0 and e1 ≠ ε then T[e1] ← M
14             if e2 ≠ 0 and e2 ≠ ε and e2 ≠ e1 then T[e2] ← M
15             if e3 ≠ 0 and e3 ≠ ε and e3 ≠ e2 and e3 ≠ e1 then T[e3] ← M
16             if e4 ≠ 0 and e4 ≠ ε and e4 ≠ e3 and e4 ≠ e2 and e4 ≠ e1 then T[e4] ← M
17             ex ← ε
18         I[i][j] ← ex
19     for j = 0 to M − 1/2 − 1 do
20         aplatissement de EQTABLE
21     for j = 0 to M − 1 − 1 do
22         e ← I[i][j]
23         if e ≠ 0 then
24             I[i][j] ← EQTABLE[ex]
```

Algorithme 44 : Lumia - propagation - passe n ascendante 8C

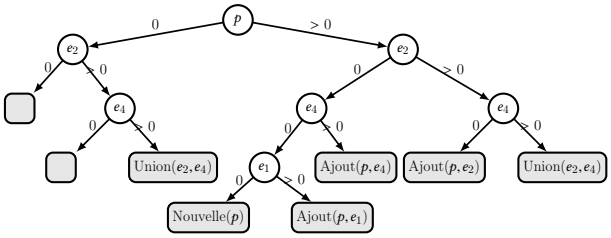
Input : I une image ($N \times M$) résultat d'une phase descendante, E
Result : I' l'image étiquetée de taille $N \times M$

```
1  for i = N − 1 to 0 do
2      EQTABLE ←CREATE( ) ; ▹ create empty equivalence table for line L
3      for j = 0 to M − 1 do
           ▹ examine each pixel P in line L
4          if I[i][j] ≠ 1 then
5              NEXT endif
6          e1 ← I[i + 1][j − 1] e2 ← I[i + 1][j] e3 ← I[i + 1][j + 1]
7          e4 ← I[i][j − 1] ex ← I[i][j] e6 ← I[i][j + 1]
8          if e1 ≠ 0 and e1 ≠ ex then T[e1] ← ex
9          if e2 ≠ 0 and e2 ≠ ex then T[e2] ← ex
10         if e3 ≠ 0 and e3 ≠ ex then T[e3] ← ex
11         if e4 ≠ 0 and e4 ≠ ex then T[e4] ← ex
12         if e6 ≠ 0 and e6 ≠ ex then T[e6] ← ex
13         ex ← ε
14         I[i][j] ← ex
15     for j = 0 to M − 1/2 − 1 do
16         aplatissement de EQTABLE
17     for j = 0 to M − 1 − 1 do
18         e ← I[i][j]
19         if e ≠ 0 then
20             I[i][j] ← EQTABLE[ex]
```

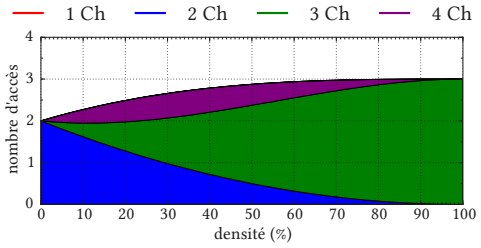
A.1.4 RCM : une fausse bonne idée

Alors que l'idée de diminuer la taille du masque semble légitime, les résultats sont moins bons que ceux obtenus pour Rosenfeld avec et sans arbre de décision. Le nombre plus important d'étiquettes supplémentaires (dû à la taille du masque) n'est pas la seule raison. Alors que le masque de Rosenfeld réalise 5 chargements et tests (au maximum), le masque RCM n'en nécessite que 4 au maximum. Cependant, lorsque l'image est peu dense, ce nombre diminue pour l'algorithme de Rosenfeld jusqu'à tendre vers 1 pour une image vide. De son côté, l'algorithme RCM doit tester aussi les pixels de fond pour trouver les connexions éventuelles entre e_2 et e_4. Il nécessite donc 4 chargements pour tous les pixels de l'image.

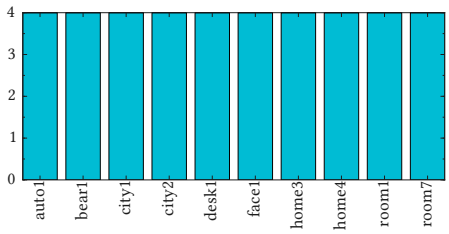
En nous basant sur le modèle d'arbre de décision étudié en 2.4.1, nous proposons une amélioration significative (non proposée par les auteurs de RCM) mais non suffisante. La figure A.2a présente l'arbre de décision pour RCM qui comporte une branche supplémentaire par rapport à l'arbre de décision usuel, qui correspond au traitement des pixels de fond. La figure A.2b montre que même avec cette amélioration, le nombre moyen de chargements est plus élevé que dans la version étudiée en 2.4.1. Les résultats pour la base de données SIDBA (fig. A.2c et A.2c) renforcent cette conclusion.



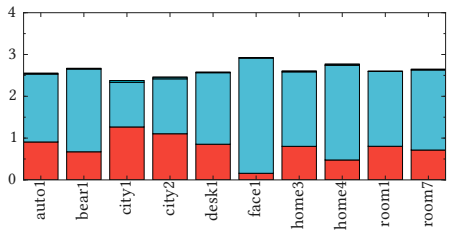
(a) Arbre de décision pour le masque RCM



(b) Nombre de chargements en fonction de la densité



(c) Nombre moyen de chargements pour la base de données SIDBA, sans arbre de décision



(d) Nombre moyen de chargements pour la base de données SIDBA, avec arbre de décision

Fig. A.2 – Influence du masque RCM sur le nombre moyen de chargements / tests

A.1.5 Selkow

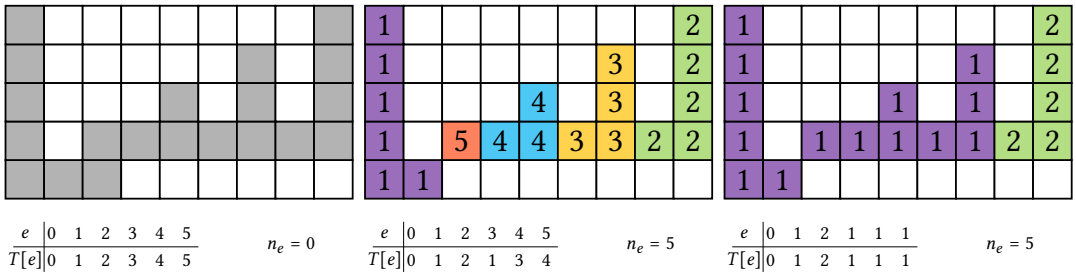


Fig. A.3 – Mise en évidence des lacunes de l'algorithme de Selkow pour les algorithmes pixels

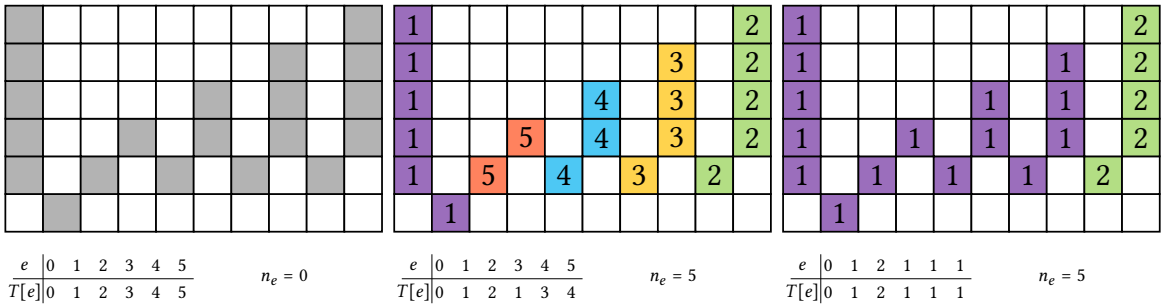


Fig. A.4 – Mise en évidence des lacunes de l'algorithme de Selkow pour les algorithmes segments

A.2 Performance des algorithmes parallèles sur $IVB_{2\times12}$

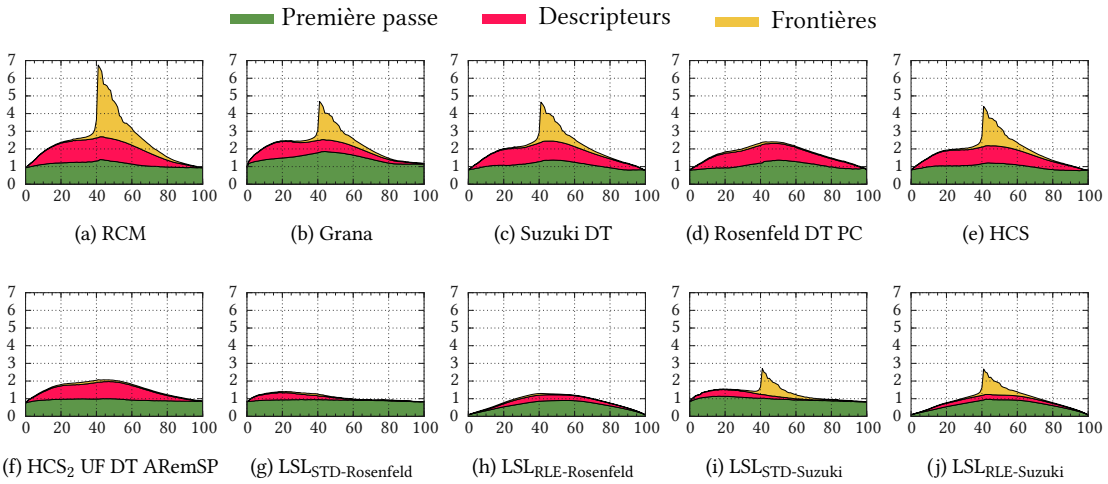


Fig. A.5 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 1$ sur 24 cœurs de la machine $IVB_2 \times 12$

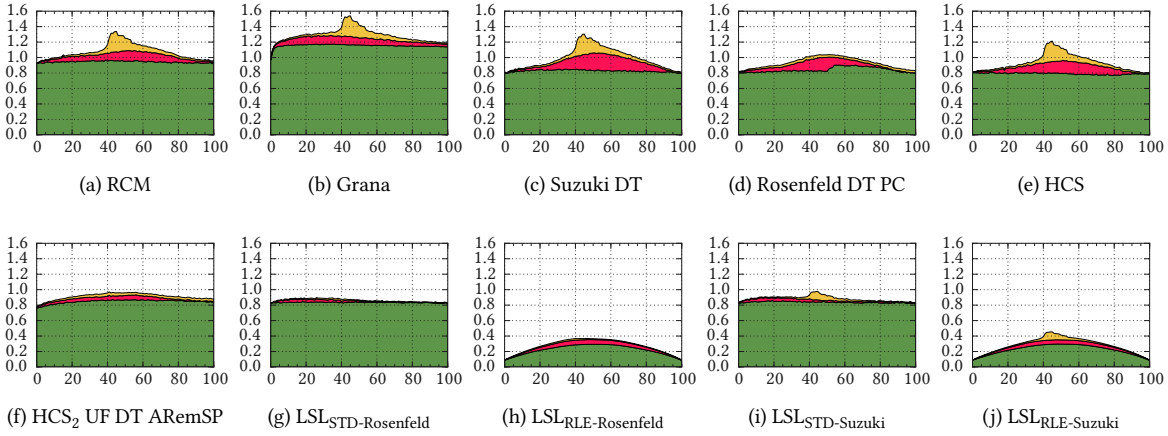


Fig. A.6 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 4$ sur 24 cœurs de la machine $IVB_2 \times 12$

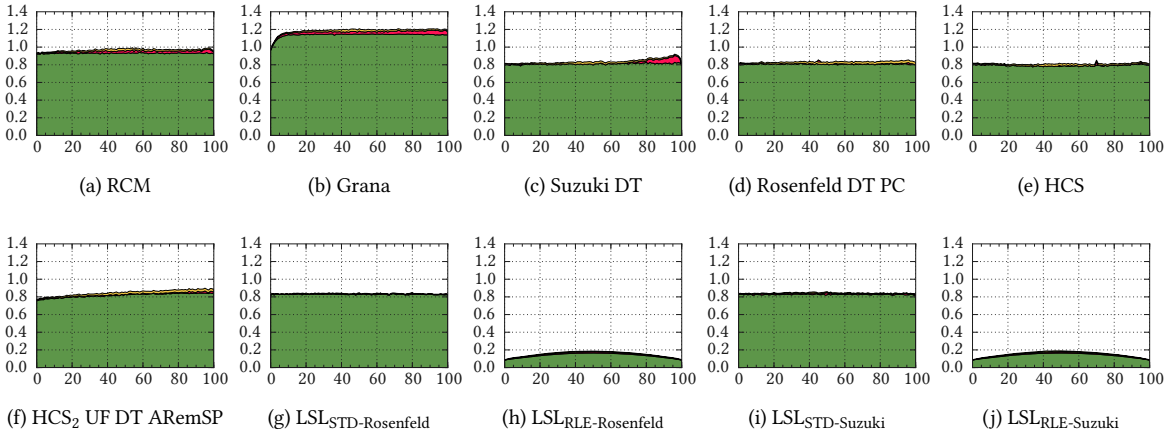


Fig. A.7 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 16$ sur 24 cœurs de la machine $IVB_2 \times 12$

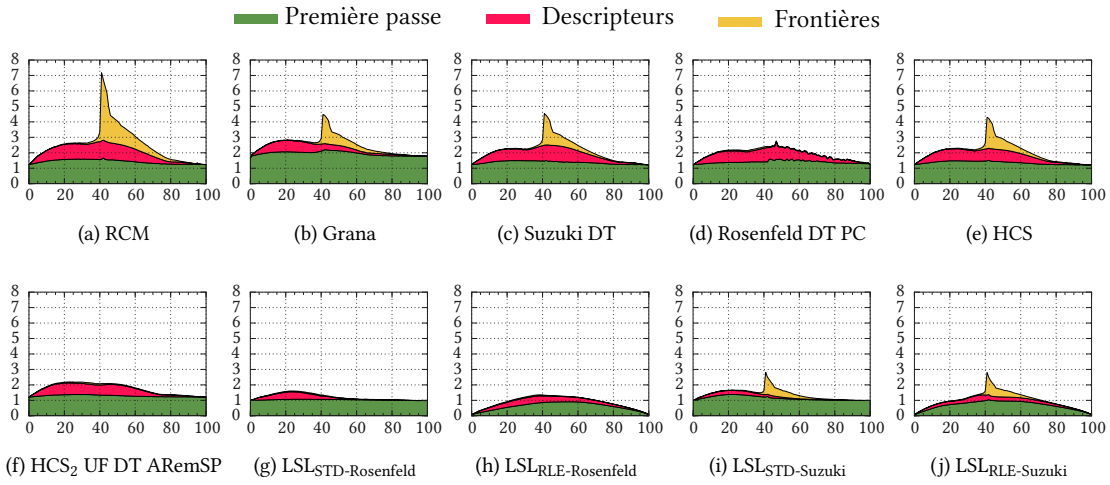


Fig. A.8 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et $g = 1$ sur 24 cœurs de la machine IVB$_2$ × 12

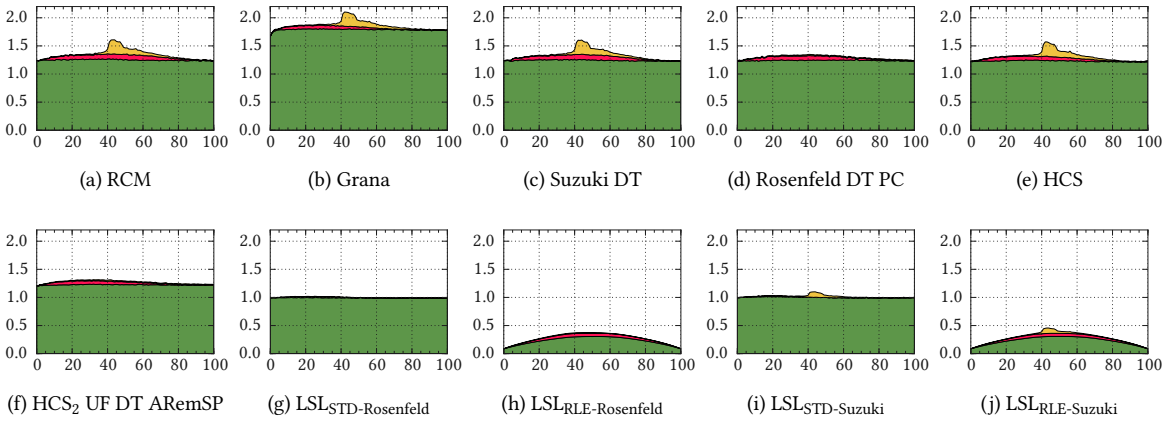


Fig. A.9 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et $g = 4$ sur 24 cœurs de la machine IVB$_2$ × 12

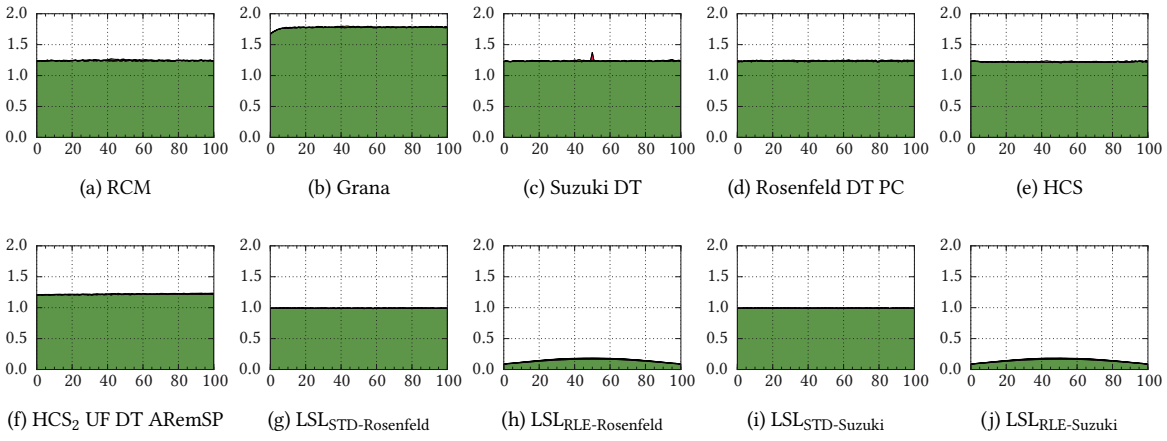


Fig. A.10 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et $g = 16$ sur 24 cœurs de la machine IVB$_2$ × 12

A.3 Performance des algorithmes parallèles sur $IVB_{4\times15}$

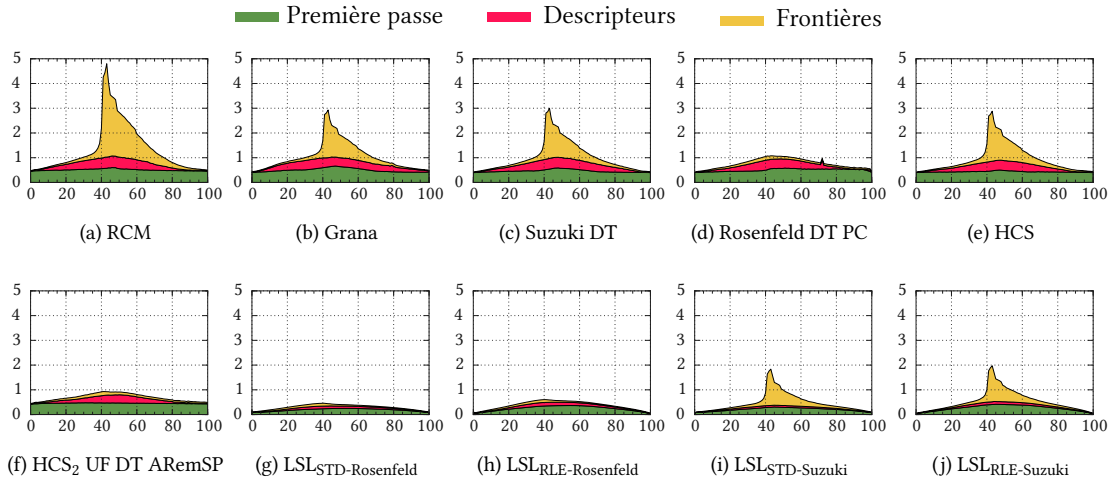


Fig. A.11 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 4$ sur 60 cœurs de la machine $IVB_{4\times15}$

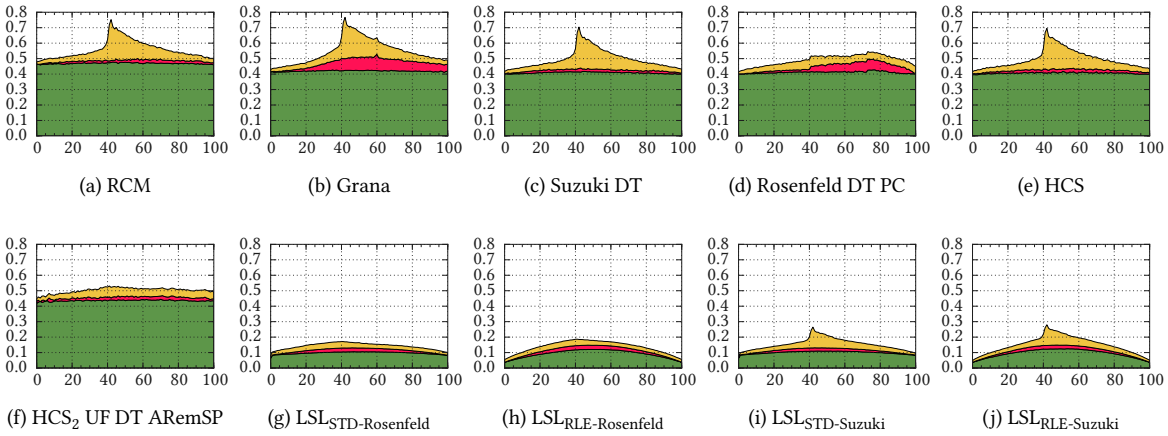


Fig. A.12 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 4$ sur 60 cœurs de la machine $IVB_{4\times15}$

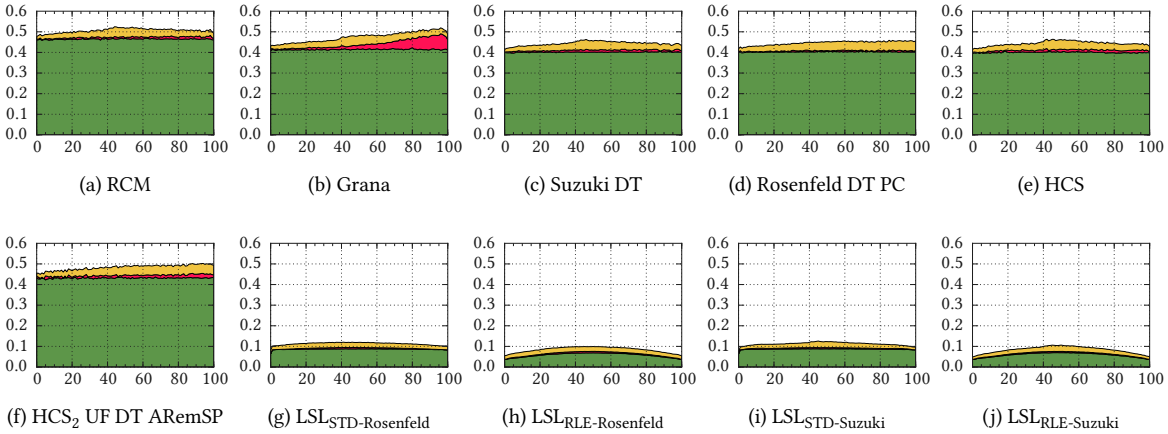


Fig. A.13 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 4096×4096 et $g = 16$ sur 60 cœurs de la machine $IVB_{4\times15}$

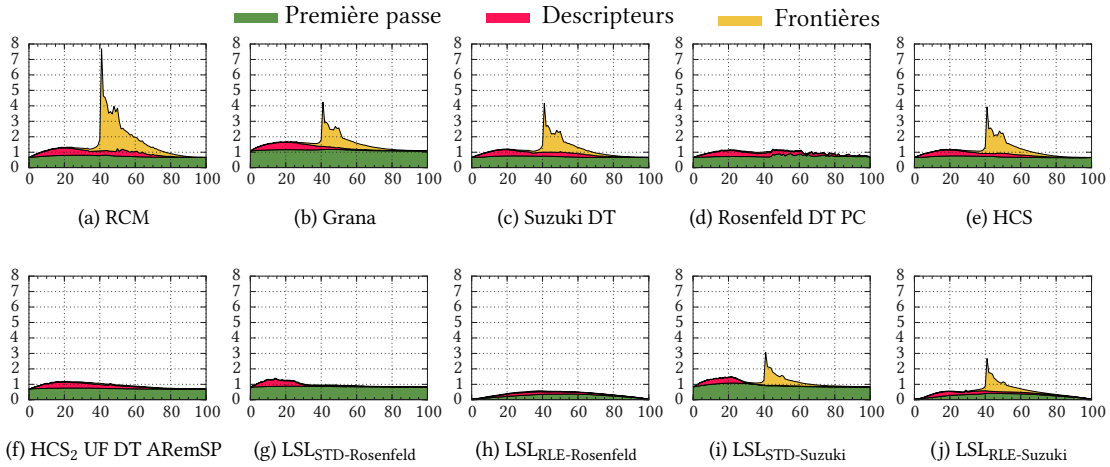


Fig. A.14 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et g = 1 sur 60 cœurs de la machine $IVB_{4\times15}$

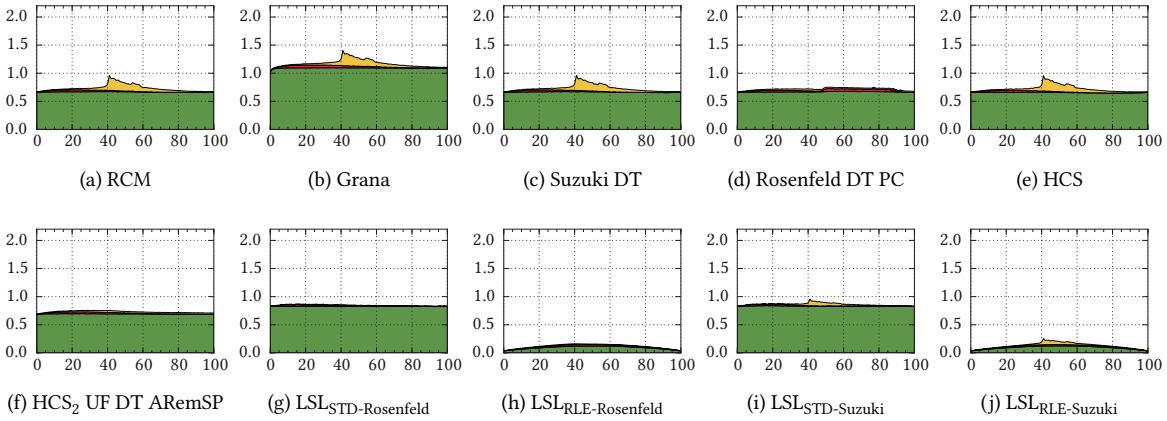


Fig. A.15 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et g = 4 sur 60 cœurs de la machine $IVB_{4\times15}$

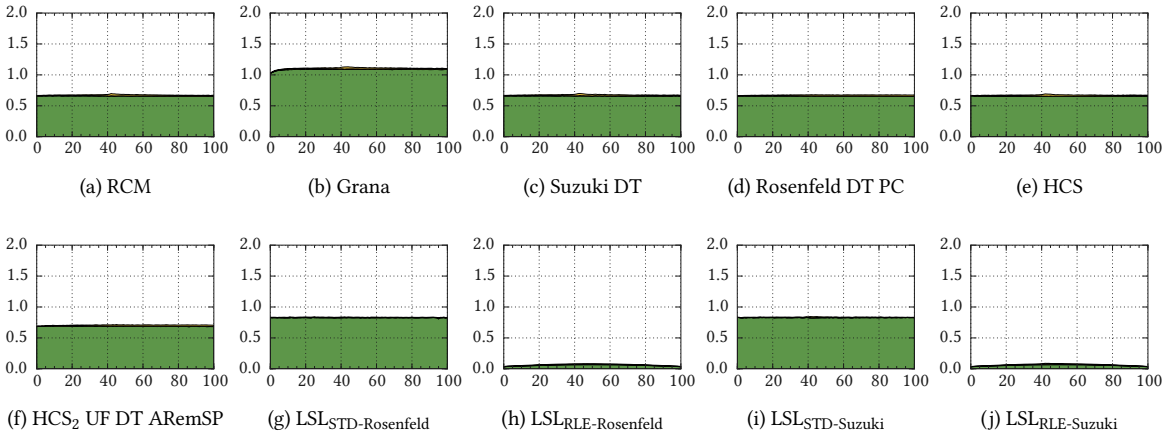
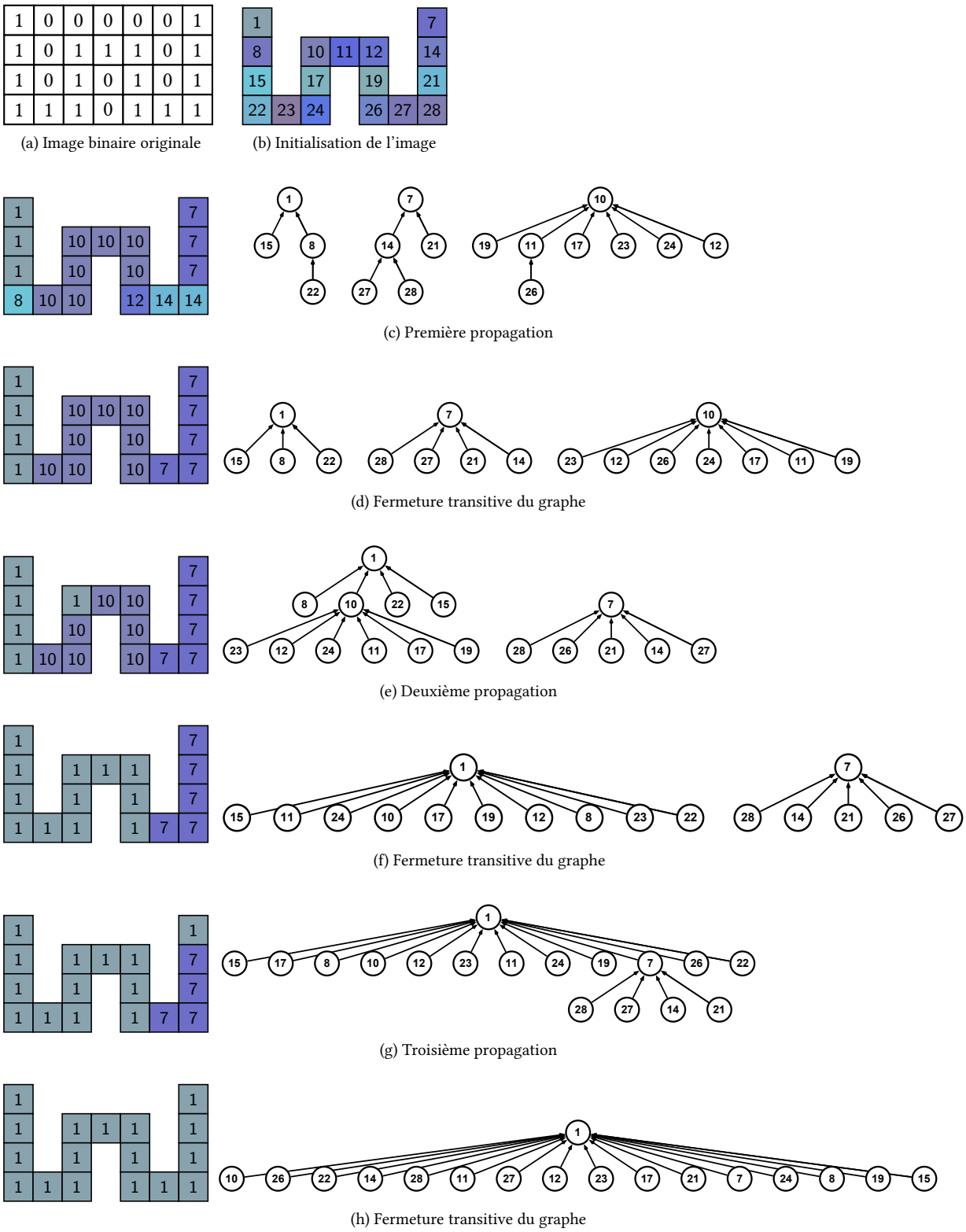


Fig. A.16 – Parallélisation multi-cœur : composition du *cpp* global par rapport à la densité (%) pour des images aléatoires de taille 8192×8192 et g = 16 sur 60 cœurs de la machine $IVB_{4\times15}$

A.4 WARP : Structure à retard



(a) Image binaire originale

(b) Initialisation de l'image

(c) Première propagation

(d) Fermeture transitive du graphe

(e) Deuxième propagation

(f) Fermeture transitive du graphe

(g) Troisième propagation

(h) Fermeture transitive du graphe

Fig. A.17 – Exemple d'une structure à concurrence de racines

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : EFFICIENT CONNECTED COMPONENT LABELING ALGORITHMS FOR HIGH PERFORMANCE ARCHITECTURES

Keywords : Image processing, Parallel processing algorithms, Multi-core architectures, Connected component labeling, Connected component analysis

Abstract : This doctoral research takes place in the field of algorithm-architecture matching for computer vision, specifically for Connected Component Labelling (CCL) for high performance parallel architectures. While modern architectures are overwhelmingly multi-core, CCL algorithms are mostly sequential, irregular and using a graph structure to represent the equivalences between labels. This aspects makes their parallelization quite challenging.

CCL processes a binary image and gathers under the same label all the connected pixels, and in doing so, CCL is a bridge between low-level operations like filtering and high-level ones like shape recognition and decision-making. It is involved in a large number of processing chains that require segmented image analysis. The acceleration of this step is therefore an issue for a variety of algorithms. At first, the PHD work focused on the comparative performance of the State-of-the-Art algorithms, for CCL as well as for the features analysis of the connected components (CCA). This was done in order to identify a hierarchy and the critical components of the algorithms. For this, a benchmarking method, reproducible and independent of the application domain was proposed and applied to a representative set of State-of-the-Art algorithms. The results show that the fastest sequential algorithm is the LSL algorithm which manipulates segments unlike other algorithms that manipulate pixels. Secondly, a parallelization framework of directs algorithms based on OpenMP was proposed with the main objective of computing the CCA on the fly and reducing the cost of communication between threads. For this, the binary image is divided into bands that are processed in parallel to each core of the architecture and a pyramidal fusion step that processes the generated disjointed sets of labels provides the fully labeled image without concurrent access to data between threads.

The benchmarking procedure applied to several machines of various parallelism levels, showing that the proposed parallelization framework applies to all the direct algorithms. The LSL algorithm is once again the fastest and the only one suitable when the number of cores increases due to its run-based conception. With an architecture of 60 cores, the LSL algorithm can process 42.4 billion pixels per second for images of 8192x8192 pixels, while the fastest pixel-based algorithm is limited by the bandwidth and saturates at 5.8 billion pixels per second.

After these works, our attention focused on iterative CCL algorithms in order to develop new algorithms for many-core and GPU architectures. The Iterative algorithms are based on a local propagation mechanism without supplementary equivalence structure which allows to achieve a massively parallel implementation (MPAR). This work then led to the creation of two new algorithms.

- An incremental improvement of MPAR using a set of mechanisms such as alternative scanning, the use of SIMD instructions and an active tile mechanism to distribute the load between the different cores while limiting the processing of the pixels to the active areas of the image and to their neighbors.
- An algorithm that implements the equivalence relation directly into the image to reduce the number of iterations required for labeling. An implementation for GPU, based on *atomic* instructions with pre-labeling in the local memory has been realized and it has proven effective from the small images.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : ALGORITHMES D'ÉTIQUETAGE EN COMPOSANTES CONNEXES EFFICACES POUR ARCHITECTURES HAUTES PERFORMANCES

Keywords : Traitement d'images, Parallélisation d'algorithmes, Architectures multi-cœur, Étiquetage en composantes connexes, Analyse en composantes connexes

Résumé : Ces travaux de thèse, dans le domaine de l'adéquation algorithme architecture pour la vision par ordinateur, ont pour cadre l'étiquetage en composantes connexes (ECC) dans le contexte parallèle des architectures hautes performances. Alors que les architectures généralistes modernes sont multi-cœur, les algorithmes d'ECC sont majoritairement séquentiels, irréguliers et utilisent une structure de graphe pour représenter les relations d'équivalence entre étiquettes, ce qui rend complexe leur parallélisation.

L'ECC permet à partir d'une image binaire de regrouper sous une même étiquette tous les pixels connexes. Il fait ainsi le pont entre les traitements de bas niveau tels que le filtrage et ceux de haut niveau tels que la reconnaissance de forme ou la prise de décision. Il est donc impliqué dans un grand nombre de chaînes de traitements qui nécessitent l'analyse d'images segmentées. L'accélération de cette étape représente donc un enjeu pour tout un ensemble d'algorithmes. Les travaux de thèse se sont tout d'abord concentrés sur les performances comparées des algorithmes de l'état de l'art tant pour l'ECC que pour l'analyse des caractéristiques des composantes connexes (ACC) afin d'en dégager une hiérarchie et d'identifier les composantes déterminantes des algorithmes. Pour cela, une méthode d'évaluation des performances, reproductible et indépendante du domaine applicatif, a été proposée et appliquée à un ensemble représentatif des algorithmes de l'état de l'art. Les résultats montrent que l'algorithme séquentiel le plus rapide est l'algorithme LSL qui manipule des segments contrairement aux autres algorithmes qui manipulent des pixels.

Dans un deuxième temps, une méthode de parallélisation des algorithmes directs utilisant OpenMP a été proposée avec pour objectif principal de réaliser l'ACC à la volée et de diminuer le coût de la communication entre les threads. Pour cela, l'image binaire est découpée en bandes traitées en parallèle sur chaque cœur de l'architecture. Ensuite une étape de fusion pyramidale d'ensembles d'étiquettes deux à deux disjoints permet d'obtenir l'image complètement étiquetée sans avoir de concurrence d'accès aux données entre les différents threads. La procédure d'évaluation des performances appliquée à des machines de degré de parallélisme variés a démontré que la méthode de parallélisation proposée était efficace et qu'elle s'appliquait à tous les algorithmes directs. L'algorithme LSL s'est encore avéré être le plus rapide et le seul adapté à l'augmentation du nombre de cœurs du fait de son approche «segments». Pour une architecture à 60 cœurs, l'algorithme LSL permet de traiter 42,4 milliards de pixels par seconde pour des images de taille 8192x8192 pixels, tandis que le plus rapide des algorithmes pixels est limité par la bande passante et sature à 5,8 milliards de pixels par seconde.

Après ces travaux, notre attention s'est portée sur les algorithmes d'ECC itératifs dans le but de développer des algorithmes pour les architectures manycore et GPU. Les algorithmes itératifs se basant sur un mécanisme de propagation des étiquettes de proche en proche, aucune autre structure que l'image n'est nécessaire, ce qui permet d'en réaliser une implémentation massivement parallèle (MPAR). Ces travaux ont mené à la création de deux nouveaux algorithmes :

- une amélioration incrémentale de MPAR utilisant un ensemble de mécanismes tels qu'un balayage alternatif, l'utilisation d'instructions SIMD ainsi qu'un mécanisme de tuiles actives permettant de répartir la charge entre les différents cœurs tout en limitant le traitement des pixels aux zones actives de l'image et à leurs voisines,
- un algorithme mettant en œuvre la relation d'équivalence directement dans l'image pour réduire le nombre d'itérations nécessaires à l'étiquetage. Une implémentation pour GPU basée sur les instructions «atomic» avec un pré-étiquetage en mémoire locale a été réalisée et s'est révélée efficace dès les images de petite taille.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France