Parallélisation hybride d'une application de détection de noyaux cellulaires

Daniel Salas

UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE Mathématiques, Sciences de l'Information et de l'Ingénieur
Laboratoire ICube

THÈSE présentée par :

Daniel SALAS

soutenue le : 10 septembre 2019

pour obtenir le grade de : **Docteur de l'université de Strasbourg**
Discipline : Informatique

Parallélisation hybride d'une application de détection de noyaux cellulaires

THÈSE dirigée par :

M GUSTEDT Jens	Directeur de recherche, INRIA

THÈSE co-encadrée par :

Mme PERSEIL Isabelle	Docteur ingénieur, qualifiée Maître de conférence, Inserm

RAPPORTEURS :

Mme EMAD Nahid	Présidente du jury, Professeur, Université Versailles Saint Quentin
M BRETON Vincent	Directeur de recherche, CNRS

AUTRES MEMBRES DU JURY :

M KERN Michel	Chargé de recherche, INRIA
M NOBLET Vincent	Docteur ingénieur, Université de Strasbourg

Remerciements

Je voudrais tout d'abord remercier les membres du jury qui par leur bienveillance m'ont permis de finaliser et soutenir cette thèse dans des conditions idéales. Je remercie aussi chaleureusement Isabelle Perseil qui m'a permis d'engager cette thèse et de la réaliser dans des conditions optimales de travail. Merci aussi à Jens Gustedt pour m'avoir fait confiance et m'avoir appris tant de choses. Je remercie également Daniel Racoceanu pour m'avoir fait découvrir la méthode de détection qui allait devenir une composante essentielle de cette thèse. Je remercie également, Laurent Quaglia, mon ami, pour son écoute et sa disponibilité. Merci à mes parents, Mamica, mes sœurs et leurs conjoints, vous avez toujours été là à mes côtés. Merci enfin à mes fils, Benjamin et Jonathan, vous m'avez donné tant de courage pour toujours avancer. Et pour en finir avec cette longue liste de personnes désormais chères à mon cœur, je tiens à remercier celle qui m'a accompagné jour après jour en m'apportant tout son amour et sa confiance, mon épouse Sabrina.

Sommaire

1 Introduction

La précision toujours plus grande de nos outils de mesure génère une quantité d'informations en constante augmentation. Le traitement de ces données d'un nouvel ordre doit s'adapter en proposant de nouveaux algorithmes capables eux aussi de passer à l'échelle. Dans le contexte de l'imagerie médicale, les nouvelles générations de scanners permettent aujourd'hui de numériser les lames de biopsies à des taux d'agrandissement de 400. Les images ainsi générées peuvent atteindre des définitions de 100 000 $\times$ 100 000 pixels. Outre la possibilité d'une analyse à distance, ce changement de support a ouvert la voie aux diagnostics assistés par ordinateur [1]. Ce nouveau champ de recherche à la frontière de l'imagerie médicale et de la pathologie répond au besoin urgent d'assister les médecins dans leur tâche [2]. En sous-effectif, ils sont confrontés à un flux quotidien d'échantillons à analyser et ne peuvent y consacrer en moyenne qu'une dizaine de minutes. Les diagnostics requièrent une attention élevée et sont délivrés sur la base d'opinions subjectives. Ces deux facteurs expliquent la variabilité inter- et intra-observateurs. La fiabilité de la détection visuelle est ainsi estimée à 80% (voir illustration 1.1).

Cependant, un cancer du sein identifié suffisamment tôt peut être guéri dans neuf cas sur

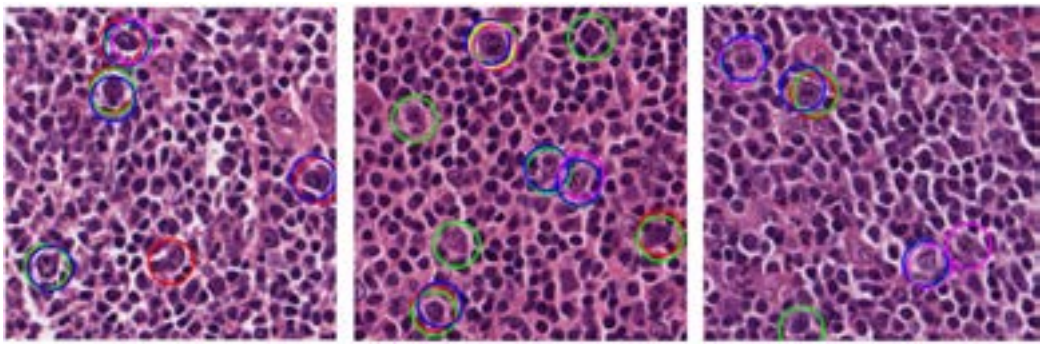
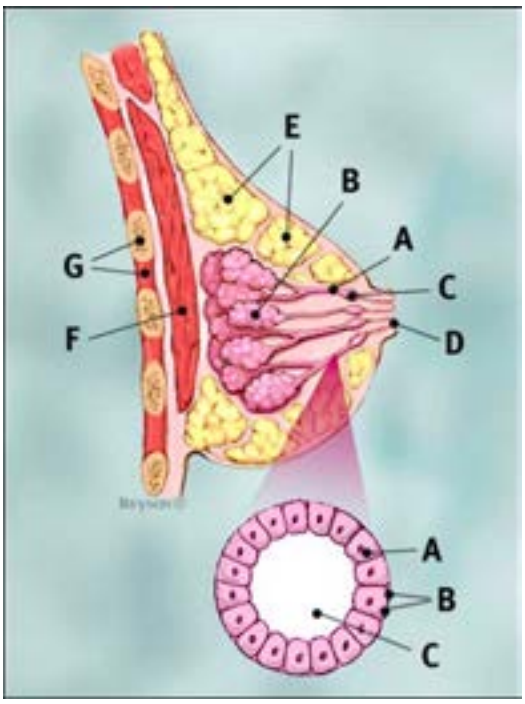


Figure 1.1: Échantillon de tissus à un agrandissement 40x. Les cellules centroblastes (grosses cellules malignes), identifiées par cinq pathologistes différents, sont indiquées par des couleurs différentes. Source : [3]



Profil de sein :

A : Ductes

B : Lobules

C : Section de ducte dilatée pour le lait

D : Téton

E : Graisse

F : Muscle pectoral majeur

G : Poitrine / cage thoracique

A : Cellules normales de ducte

B : Membrane de base

C : Lumen (centre du ducte)

Figure 1.2: Anatomie d'un sein (sources: www.breastcancer.org)

dix. Aux États-Unis, les erreurs de diagnostic constituent la troisième cause de mortalité [4].

1.1 Physionomie du cancer du sein

Notre étude est en premier lieu motivée par le dépistage du cancer du sein qui touche chaque année plus de 1.6 millions de personnes [5]. Il est la première cause de mortalité chez les femmes.
D'un point de vue anatomique, la maladie se manifeste au niveau des noyaux cellulaires contenus dans les ductes excréteurs [6]. Ces cavités acheminent le lait vers la papille mammaire au travers de pores (figure 1.2).

L'agrandissement au bas de l'image 1.2 représente un ducte de composition normale. Sa membrane périphérique contient une rangée de cellules et le lumen la cavité creuse permettant l'écoulement du lait. C'est au niveau de cette cavité que les cellules cancéreuses peuvent proliférer en l'envahissant peu à peu (voir figure 1.3).

Les cellules malignes peuvent soit rester cantonnées à l'intérieur du ducte (Carcimona *in situ*), soit se répandre dans les tissus voisins dans une forme invasive et provoquer des

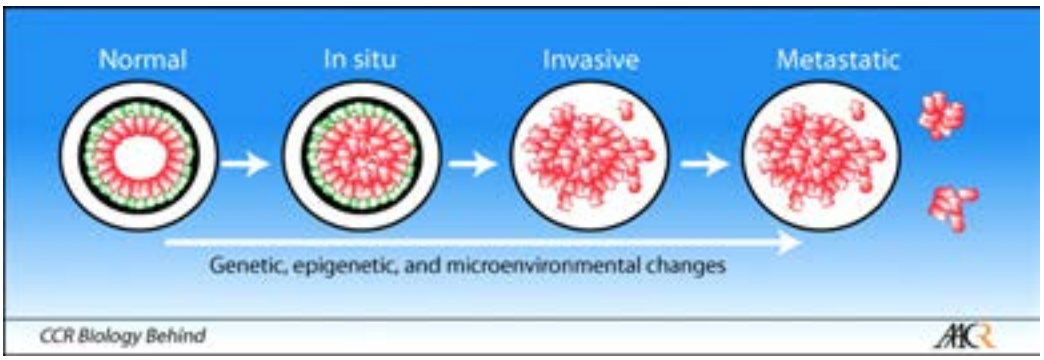


Figure 1.3: Progression schématique d'une tumeur du sein. Source: American Association for Cancer Research (AACR), Clinical Cancer Research, vol. 14 no. 2 339–341, Janvier 2008.

métastases c'est-à-dire engendrer de nouveaux foyers cancéreux dans d'autres organes. Une détection de la maladie précoce évite d'être confrontée à une forme invasive beaucoup plus difficile à combattre.

Prélèvement de tissus

Le programme national de dépistage du cancer du sein préconise aux femmes âgées entre 50 et 74 ans de réaliser une mammographie ainsi qu'un examen clinique. À l'analyse des résultats, si une anomalie indéterminée ou suspecte est détectée, un bilan diagnostic est rapidement dressé pour effectuer un prélèvement par biopsie percutanée [1]. Un tel acte peut être effectué avec une aiguille fine et fournir un échantillon de tissus de 5mm. Pour des échantillons plus importants une opération chirurgicale au bloc opératoire doit être programmée. Une lame de biopsie mesure 75 sur 25 mm.

Une fois prélevé, le tissu, pour être analysé, doit subir différentes étapes de transformation [7] : la fixation, l'inclusion, les coupes et la coloration.
Le médecin en charge de l'analyse choisit le type de coloration le plus adapté à l'élément recherché. Pour le cancer du sein, la plus utilisée est la H&E (Hématoxyline / Éosine). Elle permet d'obtenir les teintes spécifiques bleutées pour les noyaux et rose pâle pour le collagène.
L'illustration 1.4 montre le rendu final d'un prélèvement une fois scanné et agrandi 400 fois.

[1] Source : https://www.e-cancer.fr/

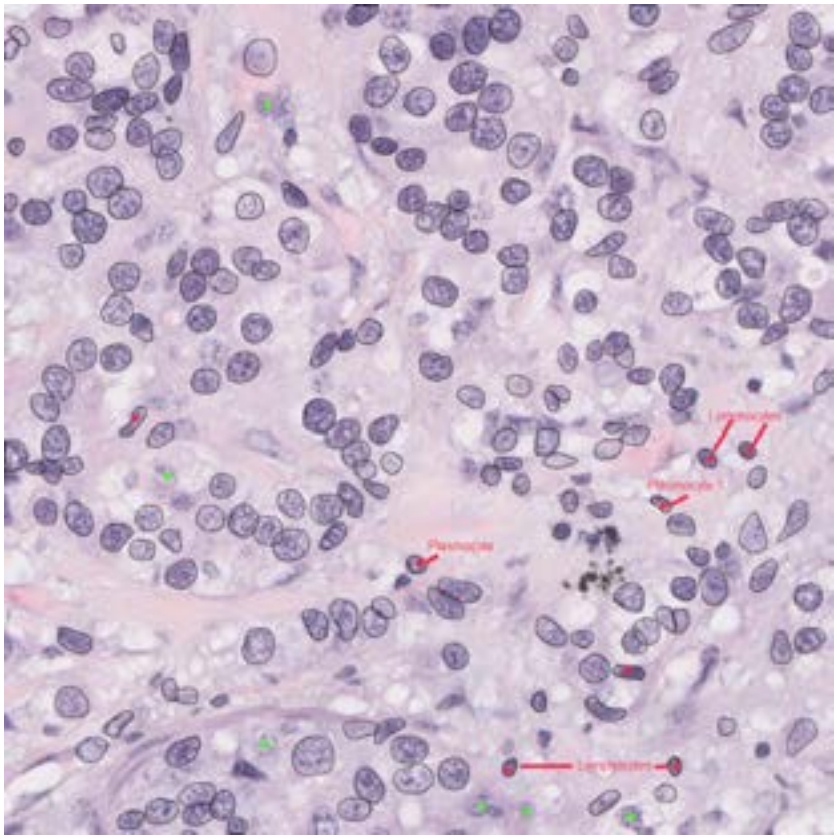


Figure 1.4: Image agrandie 400 fois d'un prélèvement de tissus coloré par H&E. Les noyaux ont été entourés manuellement en noir. Les points verts indiquent des noyaux possibles.

Scan de lames de biopsie

Les lames peuvent être analysées soit directement au microscope, soit sur un écran après numérisation. Les scanners de résolution 0.2273 μm par pixel peuvent fournir des images agrandies jusqu'à 400 fois. Selon le fabricant, les formats sont différents mais le modèle de représentation reste le même. Un seul fichier contient un ensemble d'images réparties en niveaux selon un principe pyramidal. Un tel fichier pèse 1 à 2 Go en format compressé, et peut aller de 15 à 25 Go une fois décompressé. Chaque niveau correspond à un taux de grossissement : x10, x20, x40.
L'extraction d'images à partir de ce fichier source peut être réalisée à l'aide de l'utilitaire spécifique au constructeur ou grâce à la librairie OpenSlide [2] (voir section 5.7.1).

Il est à noter que les étapes de préparation de la lame puis de son scan revêtent une

[2] https://openslide.org/

importance cruciale qui va fortement influer la future détection. Un mauvais dosage du colorant ne donnera pas la teinte souhaitée et un mauvais scan peut ajouter du flou à l'image.

Critères de détection

Une fois les images extraites, l'analyse s'effectue selon plusieurs critères. Le système de gradation Nottingham [8] est universellement utilisé et définit six critères [9] parmi lesquels la taille du noyau est l'un des plus significatifs. Le pourcentage de cellules tumorales plus grosses que la normale permet d'établir une gradation sur le stade de la maladie [9]. La section suivante présente quelques-unes des méthodes les plus efficaces utilisant ce critère comme moyen de détection.

1.2 Méthodes d'analyse d'images histologiques pour le dépistage

Les méthodes d'analyse se classent en deux grandes catégories, avec d'une part les méthodes classiques dites *computer-vision* dont le principe consiste en la segmentation de l'image et d'autre part celles issues de l'apprentissage profond qui permettent une classification des images en fonction de l'entraînement reçu.

1.2.1 Réseaux de neurones

Cette catégorie très en vogue actuellement promet des résultats spectaculaires tels que le microscope à réalité augmentée proposé par Google [10]. Le réseau de neurones servant à l'analyse est composé d'une succession de filtres s'attachant à analyser un caractère précis de l'image. Les paramètres initiaux sont affinés au fil de l'entraînement par une fonction de régression. Pour atteindre une grande fiabilité, le système nécessite un très grand nombre d'images d'entraînement. Ces images doivent par ailleurs avoir été classifiées, c'est le principe de l'entraînement supervisé. Cependant, de telles ressources annotées ne sont pas disponibles dans le domaine de cancer du sein [3] ce qui abaisse considérablement la fiabilité des résultats obtenus. Nous préférerons nous attacher ici à la première catégorie de méthode d'analyse basé sur la segmentation.

[3] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6158771/

Figure 1.5: À gauche l'image originale en niveau de gris, au milieu l'image après le seuillage, à droite l'image après l'algorithme watershed

1.2.2 Méthodes par segmentation

Le but de la segmentation est ici de séparer les noyaux cellulaires du fond de l'image. Parmi les méthodes, la plus utilisée est celle qui consiste à appliquer un seuil aux pixels de l'image à partir d'une teinte de référence. Rappelons que cette teinte est déterminée lors de l'étape de coloration de la lame. Dans sa version la plus simple, le seuillage consiste à évaluer la valeur d'un pixel par rapport à une constante. Si la valeur est inférieure, le pixel devient noir, sinon il devient blanc. Un fort contraste entre les différents éléments de l'image contribue à une plus nette différenciation des différentes structures.
En général, le seuillage constitue la première étape d'une suite de transformations [11]. Yang et al. [12] combinent ainsi le seuillage et la méthode de *watershed*. Signifiant littéralement "bassin versant", cette seconde technique considère les niveaux de gris de l'image comme un relief topographique dont on simule l'inondation [4]. Les bassins ainsi obtenus définissent des régions. Le résultat de leur algorithme est visible sur l'illustration 1.5.
Sertel et al. [3] proposent une méthode dont la première étape est la conversion de l'image multi-canal en une image *unitone* (voir illustration 1.7 image de gauche). L'espace colorimétrique RGB est ramené à la fréquence de couleur présentant le plus de contraste. Il en résulte une distribution bimodale en niveau de gris faisant ressortir les noyaux. Les cellules sont ensuite segmentées en utilisant la technique de *Gaussian mixture models* [5]. Celle-ci consiste à associer un ensemble de points en fonction de leur proximité. Dans notre cas, cela revient à identifier les pixels d'un noyau. Le terme Gaussian mixture signifie que les points sont associés à autant de gaussiennes (ellipses) qu'il y a de noyaux (voir illustration 1.6). En résolvant l'équation de la densité d'une gaussienne par rapport aux coordonnées d'un point, nous pouvons déterminer l'appartenance d'un point à un ensemble. Cependant, lorsque le nombre d'ensembles est important, le coût de tels calculs devient prohibitif. Une méthode probabiliste et itérative d'espérance-maximisation [6] (EM) permet

[4] https://fr.wikipedia.org/wiki/Ligne_de_partage_des_eaux_(segmentation)

[5] https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95

[6] https://fr.wikipedia.org/wiki/Algorithme_esp%C3%A9rance-maximisation

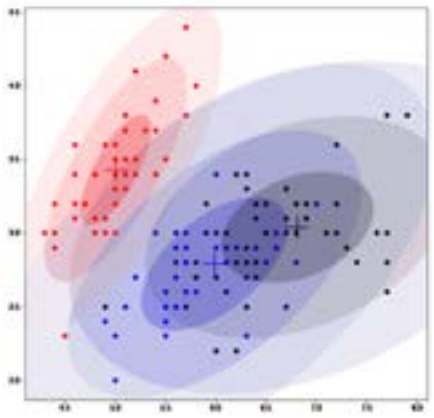


Figure 1.6: Association de point à un ensemble représenté par une gaussienne

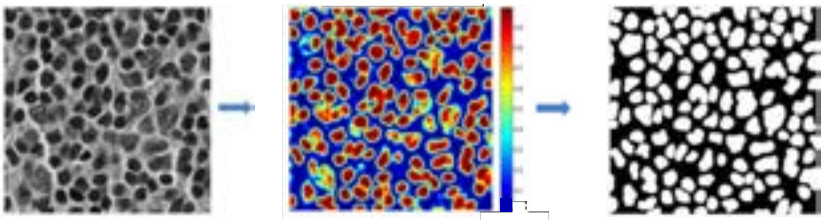


Figure 1.7: Étapes de transformation utilisant la méthode Gaussian mixture. À gauche l'image unitone, au centre l'image obtenue à partir de l'algorithme EM, à droite l'image finale des noyaux après l'application d'un seuillage adaptatif

de réduire ces coûts en estimant les paramètres avec un maximum de vraisemblance (voir illustration 1.7 image du milieu). À cette étape, les noyaux sont rassemblés en cluster. Une étape supplémentaire de seuillage adaptatif permet de définir précisément les noyaux (voir illustration 1.7 image de droite).

Les méthodes présentées jusqu'ici parviennent à correctement détecter les noyaux en les séparant du reste de l'image. Cependant, les noyaux ne sont pas identifiés entre eux par un contour ce qui permettrait notamment de prendre en compte les recouvrements. Les deux méthodes suivantes sont elles capables d'identifier clairement les noyaux.

La méthode GiPS [13] se déroule en deux phases avec une sélection des noyaux candidats puis une détection de leur contour. La sélection s'opère grâce à un processus im-

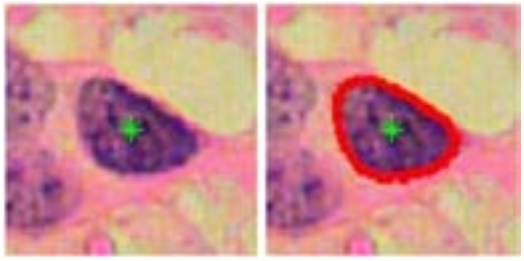


Figure 1.8: Résultat de l'extraction du contour d'un noyau avec la méthode GiPS

pliquant des étapes déjà vues précédemment :

1. extraction de couleur ;
2. application de seuil ;
3. dilatation et érosion.

La seconde phase est originale. Elle consiste à reconstituer les noyaux à partir de la segmentation des espaces polaires individuels. Le principe en est le suivant : à partir d'un angle θ_0, le radius du noyau est parcouru jusqu'à sa valeur la plus élevée. Celle-ci représente la membrane plus sombre. De là, les pixels adjacents de même couleur sont progressivement connectés en un segment. Le centre du segment sert alors de référence et l'angle est incrémenté jusqu'à la sélection complète du contour (voir image 1.8).

La méthode de détection proposée par K. Mosaligante et al. [14] (KMLS) propose une détection à partir d'images microscopiques 3D. L'ajout d'une nouvelle dimension permet entre autres de mieux détecter les recouvrements entre noyaux (voir illustration 1.9). Lors d'une première phase, un seuillage est une fois de plus appliqué à l'image pour séparer les noyaux du fond. Une seconde phase permet ensuite d'affiner le contour des noyaux en appliquant une méthode de surface de niveaux [7] (*level-set*) qui consiste à identifier la valeur d'un pixel à une élévation. Enfin, les noyaux superposés sont séparés grâce à la technique de diagrammes de Voronoi [8] qui permet de déterminer l'enveloppe convexe de chaque noyau.

La détection et l'extraction de noyaux sont donc un vrai défi. La qualité du résultat final repose sur la combinaison de diverses techniques. Le résultat final dépend alors de

[7] https://fr.wikipedia.org/wiki/M%C3%A9thode_des_surfaces_de_niveau

[8] http://villemin.gerard.free.fr/Geometri/Voronoi.htm

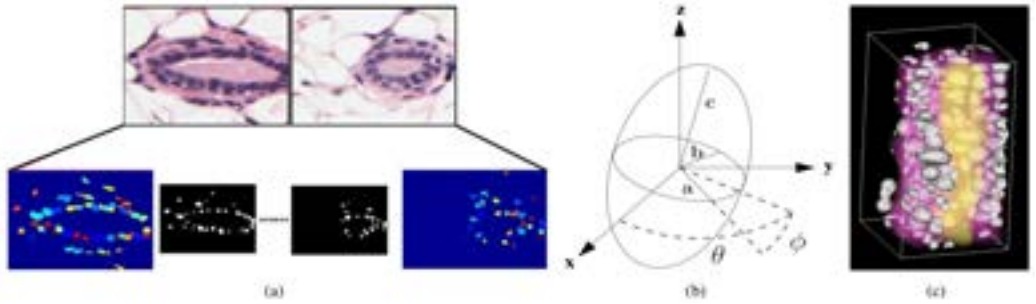


Figure 1.9: Segmentation utilisant une méthode de contour actif. (a) Noyaux en recouvrement. (b) Application d'un seuillage. (c) Application d'une méthode de surface de niveaux. (d) Séparation des noyaux par diagramme de Voronoi.

la précision avec laquelle la première phase détecte le nombre et la position des objets sur l'image.
Le modèle MPP présenté ci-après adopte une approche originale et ne présente pas cette dépendance car il ne repose sur aucune étape préliminaire d'extraction d'objets.

1.2.3 Méthode Marked Point Process (MPP)

La méthode Marked Point Process (MPP) est inspiré de la détection de forme à partir de vues aériennes. Son principe n'est pas basé sur une segmentation de l'image mais sur une mesure de valeurs d'attache aux données entre l'image et des objets générés aléatoirement. Permettant comme GiPS et KMLS une détection de la taille des noyaux, Kulikova et al. [11] ont comparé ces trois méthodes sur la base du nombre et de la qualité des détections. La méthode MPP obtient les meilleurs résultats sur les deux critères. Sur la base de ces bonnes performances, nous avons retenue cette méthode pour développer notre application. Elle sera présentée en détail dans le chapitre suivant.

1.3 Problématiques calculatoires

Pour répondre de manière concrète au besoin des médecins en terme de détection automatique, le but de cette étude est d'apporter une solution d'aide au diagnostic basé sur l'algorithme MPP. L'application devra être en mesure de traiter une lame entière en un temps compatible avec le flux d'arrivée des données.
D'après les résultats obtenu en séquentiel sur une image de 3326×2971 pixels, le traitement d'une lame complète est estimé à environ 5 jours de calcul. La parallélisation à mémoire partagée directe avec OpenMP obtient une accélération d'un facteur 12 sur 24

cœurs. Une version GPU de l'algorithme double encore ce résultat et atteint une accélération de facteur 24 [15]. Toutefois, la capacité mémoire limitée des GPU (16Go) ne permet pas le calcul d'images supérieures à 18340×15862 pixels.
Pour repousser ces limites, la parallélisation doit être portée à un niveau supérieur en combinant les modes partagé et distribué. L'utilisation conjointe de techniques telles que OpenMP, MPI et CUDA n'est toutefois pas triviale et doit être confié à des spécialistes, ce qui implique l'augmentation des coûts de développement et de maintenance.
Nous proposons d'explorer ici une autre solution basée sur le modèle Ordered Read-Write Locks (ORWL). Son approche centrée sur les données permet une parallélisation hybride au sein d'une unique implémentation.

1.4 Objectif et contributions

L'objectif de cette thèse est donc de permettre le traitement d'une lame complète de biopsie avec l'algorithme MPP. L'application d'un nouveau paradigme de parallélisation (ORWL) utilisant le parallélisme à mémoire partagée et à mémoire distribuée devra répondre à des problématiques de dépendance des données et obtenir des performances au moins équivalentes aux méthodes classiques de parallélisation. Les contributions de ce travail sont :

- La distribution parallèle de l'algorithme MPP et le traitement d'une image de plusieurs giga-octets selon cette méthode ;
- L'accélération des temps de calcul pour absorber le flux d'analyse des lames ;
- La mise en place d'un workflow de traitement d'image capable de passer à l'échelle ;
- La création d'une méthode permettant d'appliquer une distribution de Poisson sur différents nœuds ;
- L'utilisation d'ORWL pour couvrir la latence de la communication avec les calculs.

1.5 Structure du manuscrit

La structure du document est la suivante :

- Le chapitre 2 présente le modèle MPP et les enjeux de son implémentation parallèle à mémoire partagée. Une nouvelle méthode de parallélisation à mémoire distribuée est proposée avec de nouvelles problématiques qui en découlent.

- Le chapitre 3 dresse un état de l'art sur les principales méthodes de parallélisation et leurs moyens de répondre aux problèmes soulevés au chapitre 2.
- Le chapitre 4 introduit ORWL et sa capacité à répondre à la même problématique.
- Le chapitre 5 expose la solution mise en place dans cette thèse : la parallélisation de la méthode MPP avec ORWL.
- Le chapitre 6 montre les résultats de tests de passage à l'échelle de cette méthode.
- Enfin le chapitre 7 dresse un bilan des résultats et des difficultés rencontrées. Il ouvre également la voie à de nouvelles perspectives d'améliorations.

2 Marked Point Process : un algorithme de détection automatique

Les différentes méthodes de détection que nous avons pu voir au chapitre précédent reposent essentiellement sur la mise en valeur des noyaux mais ne permettent pas de concrètement en calculer la taille. Or c'est précisément le critère que l'on souhaite analyser.
La méthode MPP introduite par Descamps et Al. dans [16] a initialement été conçue pour dénombrer des colonies de flamants roses à partir de photos aériennes. Le concept clé de MPP consiste à générer de manière aléatoire une forme approchant celle d'un objet recherché. Le taux de correspondance entre l'objet et l'image est ensuite calculé grâce à la distance de Bhattacharyya . Les objets sont ensuite triés selon ce taux et un facteur aléatoire. L'opération est répétée jusqu'à ce que la meilleure combinaison soit obtenue.
Ce principe de détection ainsi que la qualité des résultats a été développé en collaboration avec le professeur Frédérique Capron [1] [17] et le Laboratoire d'Informatique Biomédicale (LIB [18]).

2.1 MPP : processus itératif de naissance et de mort

Comme nous l'avons vu lors de l'exposé de la problématique, la méthode MPP est basée sur la génération aléatoire de processus ponctuels dont la marque correspond à une ellipse. Cette forme elliptique a été choisie en concertation avec des spécialistes experts en histologie.
Pour être efficace, le procédé qui consiste à coupler une ellipse à un noyau en fonction de son degré d'attachement, doit être itéré jusqu'à ce qu'aucune ellipse ne corresponde

[1] Professeur des Universités-Praticien Hospitalier (PU-PH) au CHU de la Pitié-Salpétrière

mieux que celles déjà trouvées.
Une itération est constituée d'un processus de naissance et de mort inséré dans un processus de recuit simulé [19].

2.1.1 Recuit simulé

Cette méta-heuristique [2] est une technique issue du monde de la métallurgie visant à optimiser la solidité d'un métal. À l'état liquide, la structure homogène du matériau lui confère un état d'énergie élevé. Un refroidissement brutal par trempe fait apparaître des défauts de structure. À contrario, un refroidissement par palier accompagné de temps de réchauffe (recuit) permet d'obtenir un état solide stable [20]. La physique statistique introduit une probabilité d'atteindre un certain niveau d'énergie à partir d'une température comme suit :

$$P(E) = \exp[-E/KB * T] \tag{2.1}$$

KB la constante de Boltzmann [3] permet de convertir la température T en énergie. À température élevée (T$\rightarrow \infty$) la probabilité est de 100%. À T faible, P(E) est très faible mais non nulle. Cela permet de donner une chance de trouver un autre minimum local d'énergie.

Algorithm 1: Algorithme de recuit simulé

```
Temperature ← T0 (*température initiale*)
while condition d'arrêt non remplie do
  for i = 0 to N do
    Calculer nouvelle solution (*perturber la solution*)
    δ = coût (nouvelle solution) - coût (*solution courante*)
    if δ < 0 then
      Conserver nouvelle solution
    else if rand[0,1] < exp[−δ / Temperature] then
      Conserver nouvelle solution
    end if
  end for
  Temperature = α * Temperature (*nouveau palier de température*)
end while
```

Appliqués à la détection MPP, les paramètres de recuit simulé sont :

[2] stratégie permettant de mener à une solution optimale

[3] https://fr.wikipedia.org/wiki/Constante_de_Boltzmann

- une température initiale T0 de 50 ;
- le nombre de configurations N visitées par un palier de température correspond au nombre d'ellipses à créer par itération. Il est proportionnel à la dimension de l'image et à un facteur de densité de recouvrement delta (3.8%).
- une loi de décroissance de la température. La température est multipliée par un facteur 0.997 à chaque itération.
- Condition d'arrêt : aucun objet créé dans l'itération courante n'est retenu dans la solution.

2.1.2 Processus de naissance et de mort

Chaque itération aux paramètres définis par l'algorithme de recuit simulé, inclut un autre processus visant à déterminer la meilleure configuration d'ellipses. Ce processus de naissance et de mort [21] comprend les étapes décrites ci-après.

Naisssance Une configuration de processus ponctuels est distribuée aléatoirement sur la surface de l'image selon une distribution de Poisson.

Marquage Les processus ponctuels reçoivent pour marque les paramètres d'une ellipse calculés aléatoirement : coordonnées du centre, longueur de grand axe, longueur de petit axe, angle. Une fois le calcul d'attache aux données effectué à partir de ces éléments, il constitue l'une des marques.

Mort Les objets sont supprimés avec une probabilité dépendant de leur énergie. Pour favoriser les objets les mieux placés, les ellipses sont classées par ordre décroissant de leur valeur d'attache. En partant du début de la liste, une valeur globale d'énergie est calculée à partir des énergies d'attache et d'interaction : $U(w) = U_d(w) + U_p(w)$.
La probabilité de garder l'objet est alors de : $\frac{\delta\Delta P_i}{1+\delta\Delta P_i}$ avec

$$\Delta P_i = P(\mathbf{w}\backslash w_i)/P(\mathbf{w}) = \exp\left[-\beta(U(\mathbf{w}) - U(\mathbf{w}\backslash w_i)\right] \tag{2.2}$$

$P(w)$ est la fonction de densité de Gibbs vue sur l'équation 2.4 et β un paramètre positif permettant de favoriser une composante d'énergie par rapport à l'autre. L'expression $\mathbf{w}\backslash w_i$ signifie que le calcul est fait sur la totalité des ellipses sauf celle considérée. Après chaque suppression, la carte de voisinage est mise à jour. La probabilité de garder un objet, très

faible au début, croît au fur et à mesure que la quantité totale d'ellipses diminue. Les dernières ellipses sont donc favorisées par rapport aux premières.

Test de convergence À la fin de l'étape de mort, si le nombre d'objets est modifié, les paramètres de recuit simulé sont mis à jour de manière à diminuer le nombre de naissance et à favoriser le taux de mort.

2.2 Concepts mathématiques

MPP utilise en substance les processus ponctuels que l'on peut considérer comme des objets définis aléatoirement sur un espace fini (χ) et dont le nombre est quantifiable selon une mesure Λ [22].
L'apport d'un marque $\mathscr{M}$ à ces objets définit une nouvelle caractéristique en plus de leur position $\mathscr{P}$. Nous pouvons alors définir : $\chi = \mathscr{P} \times \mathscr{M}$.
Une distribution de Poisson permet de répartir les processus de manière homogène sur l'espace. Une telle répartition est d'intensité $\nu(.)$ si :

- le nombre de points contenus dans le sous-ensemble $\mathscr{A} \subseteq \chi$ suit une loi de Poisson d'espérance $\nu(\mathscr{A})$.
- pour k sous-ensembles distincts $\mathscr{A}_1, ..., \mathscr{A}_k$ les nombres de points $n_1, ..., n_k$ sont indépendants.

Un processus B a alors une probabilité d'apparition de :

$$\pi(B) = \int_B f(x)\pi_\nu(\mathrm{d}x) \tag{2.3}$$

où $f(x)$ représente la densité de probabilité de B, π_ν la densité du processus de Poisson et x une configuration de points de l'ensemble fini.

Les processus de Gibbs [23] sont une classe particulière des processus ponctuels où la densité s'exprime par la somme de deux composantes d'énergie :

- une valeur d'interaction entre les objets : $U_p(x)$;
- une valeur d'attache aux données : $U_d(x)$.

Sa fonction de densité s'écrit :

$$f(x) = \frac{1}{Z}\exp(-U(x)) \tag{2.4}$$

Z est une constante de normalisation sur toutes les configurations de x et $U(x)$ la somme de $U_p(x)$ et $U_d(x)$.
L'interaction entre les objets d'une configuration de points s'exprime par leur taux de recouvrement. En considérant les silhouettes de deux objets x_i et x_j ce taux représente le quotient de l'aire partagée par l'aire minimale des deux objets. L'énergie d'interaction d'un objet x_j correspond ainsi à la somme de ses interactions avec tous les objets de la configuration selon la formule :

$$U_p(x) = \gamma_p \sum_{x_i \in x} max_{x_j \sim_r x_i}\left(\frac{\Lambda(\mathscr{S}_{\mathscr{P}}(x_i) \cap \mathscr{S}_{\mathscr{P}}(x_j))}{min(\Lambda(\mathscr{S}_{\mathscr{P}}(x_i)), \Lambda(\mathscr{S}_{\mathscr{P}}(x_j)))}\right) \tag{2.5}$$

γ_p correspond à un facteur de pondération pouvant favoriser ou pénaliser l'énergie d'interaction face à celle d'attache.
Afin de simplifier les calculs d'aire, les objets sont discrétisés et la distance de Lebesgue est apparentée au nombre de pixels couverts par un objet. Avec $\mathscr{S}_{\mathscr{I}}$ l'ensemble des pixels de l'image, l'équation peut s'écrire :

$$U_p(x) \simeq \gamma_p \sum_{x_i \in x} max_{x_j \sim_r x_i}\left(\frac{Card\{p \in \mathscr{S}_{\mathscr{I}}(x_i) \cap \mathscr{S}_{\mathscr{I}}(x_j)\}_{p \in I}}{min(Card\{p \in \mathscr{S}_{\mathscr{I}}(x_i)\}_{p \in I}, Card\{p \in \mathscr{S}_{\mathscr{I}}(x_j)\}_{p \in I}}\right) \tag{2.6}$$

La mesure d'attache aux données $U_d(x)$ est dépendante de la forme recherchée. La section suivante présente la méthode retenue : la distance de Bhattacharya.

Attache aux données

Dans le cadre de notre détection, les noyaux cellulaires sont assimilés à des formes elliptiques. C'est ce qui va conditionner la marque des processus ponctuels.
Pour maintenant déterminer l'attache aux données, il est nécessaire de définir ce qui caractérise un noyau du reste de l'image. Schématiquement, un noyau cellulaire se compose d'une membrane périphérique de forme ovale et sombre par rapport à son intérieur [24], de couleur plus claire (voir illustration 2.1).

Le contraste colorimétrique entre la membrane et l'intérieur du noyau est mesuré par la distance de Bhattacharya [25]. Cette méthode statistique statue sur la similarité ou la différence entre deux distributions de points. Avec μ_1 une moyenne de la couleur des

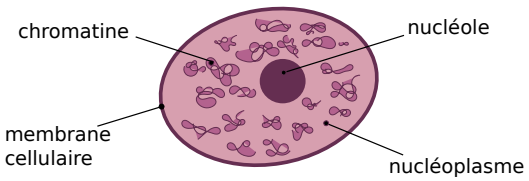


Figure 2.1: Schéma d'un noyau cellulaire

pixels situés à l'intérieur de l'ellipse, σ_1 l'écart type, μ_2 une moyenne du contour et σ_2 l'écart type, la distance de Bhattacharyya est calculée comme suit :

$$d_B(u, \mathscr{F}^p_{\mathscr{J}}(u)) = \frac{(\mu_1 - \mu_2)^2}{4\sqrt{\sigma_1^2 + \sigma_2^2}} - \frac{1}{2}\log\frac{2\sigma_1\sigma_2}{\sigma_1^2 + \sigma_2^2} \tag{2.7}$$

À partir de cette valeur, la fonction de qualité Q_d produit une valeur d'attache aux données comprise entre $[-1, 1]$:

$$Q_d(d_B) = \begin{cases} (1 - \frac{d_B}{d_0}) & \text{si } d_B < d_0 \\ exp(-\frac{d_B - d_0}{100}) - 1 & \text{si } d_B \geq d_0 \end{cases} \tag{2.8}$$

Le paramètre d_0 permet d'appliquer un seuil d'attache à partir duquel les ellipses seront considérées comme "bien placées". Il est à noter que les valeurs les plus proches de -1 seront les meilleures et correspondront aux différences de couleurs les plus franches.

Pour calculer la valeur d'attache aux données d'une ellipse, celle-ci doit être contenue à l'intérieur de l'image. Cela garantit la présence de données. Cette contrainte oblige à établir un périmètre autour de l'image à l'intérieur duquel aucune ellipse ne sera créée. Ceci aura une importance particulière que nous développerons au chapitre suivant.

2.3 Optimisations algorithmiques du modèle

Afin de minimiser le coût de certains calculs décris dans le modèle, l'algorithme implémenté adopte les postulats présentés dans cette section.

2.3.1 Condition forte de répulsion entre les objets

Le calcul de probabilité de garder un objet ΔP_i est un calcul complexe où pour chaque ellipse, il est besoin de calculer d'une part son énergie propre et d'autre part l'énergie

totale du système sans elle (voir équation 2.2). En ajoutant une condition forte de répulsion, nous pouvons simuler le fait qu'il n'existe pas de recouvrement. D'un point de vue applicatif, cela revient à ne pas considérer les noyaux qui se chevauchent. Cette nouvelle condition permet dès lors de se soustraire au calcul de l'énergie d'interaction U_p dont le numérateur représente l'intersection entre l'ellipse et ses voisins. Dès lors, seule l'énergie d'attache aux données devient significative pour calculer le taux de mort :

$$\Delta P_i = exp(-\beta(U(\mathbf{w}) - U(\mathbf{w}\backslash w_i))) = exp(-\beta U(wi)) \tag{2.9}$$

2.3.2 Assimilation de la couleur interne à une couronne

Plus le grade du cancer est élevé, moins la créatine à l'intérieur du noyau est répartie de façon homogène (voir [24]). En grade 3, des poches apparaissent vers le centre alors que la périphérie reste de couleur unie. Pour traiter l'ensemble des grades avec le même algorithme, la couleur moyenne d'une ellipse est déterminée à partir de trois ellipses concentriques (voir illustration 2.2).

Les valeurs comparées lors du calcul d'attache aux données sont alors celles d'une couronne interne (en bleu) et celles de l'ellipse la plus externe.

2.3.3 Utilisation d'une carte de naissances

En traitement préalable au processus de naissance et de mort, Descamps et Al. proposent d'établir une carte de naissances pour optimiser le taux de naissance. Cette carte est constituée en appliquant une version simplifiée du calcul d'attache aux données sur chaque pixel de l'image. Le calcul n'est effectué qu'une seule fois et la forme considérée n'est plus elliptique mais ronde. À partir de la valeur de fidélité d'attache, un taux de naissance est calculé à partir d'un seuil arbitraire d_0. Après normalisation de ce taux, un pourcentage de probabilité de naissance est obtenu par normalisation. Cette valeur pourra par la suite être utilisée pour définir la légitimité d'un pixel à accueillir une ellipse. Par exemple, si un pixel obtient un bon score d'attache aux données, le pourcentage de création qui figurera sur la carte sera d'autant plus élevé.

2.4 Algorithme MPP

D'un point de vue algorithmique, le processus MPP se traduit par l'enchaînement d'étapes suivant :

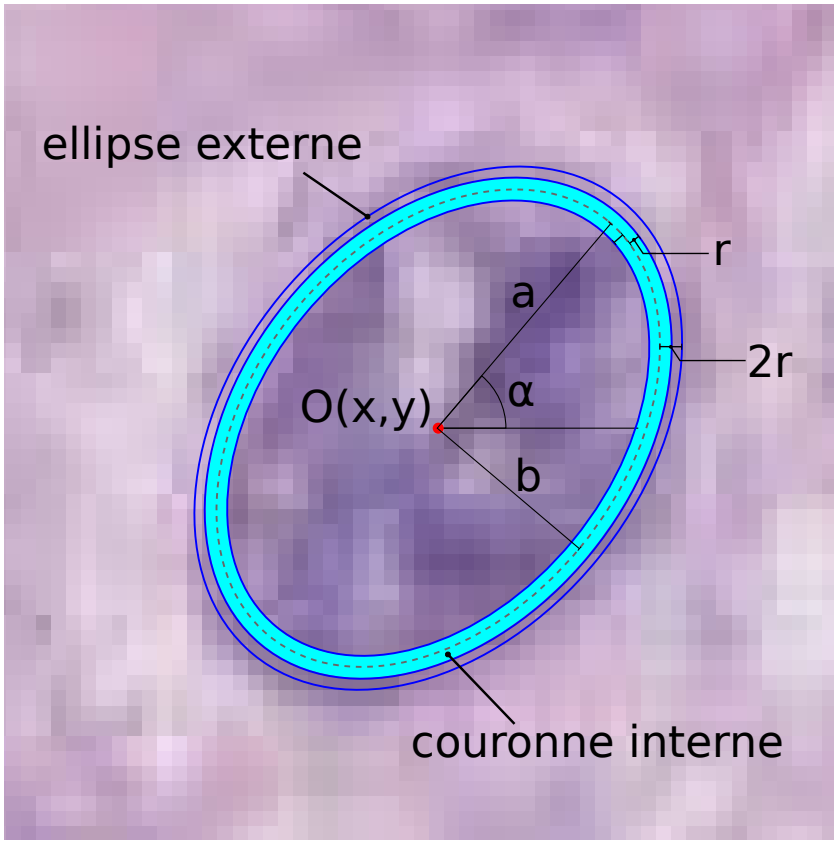


Figure 2.2: Ellipses concentriques générées autour d'un noyau

1. Chargement d'une image de largeur w et de hauteur h.

2. Boucle de convergence :
   Cette boucle encapsule l'algorithme de détection MPP dans un procédé de recuit simulé. A chaque itération, des ellipses sont créées puis leur fidélité d'attache aux données est testée.
   Un filtrage stochastique élimine alors la plupart des ellipses créées. Celles qui survivent sont remises en compétition au cours de l'itération suivante. Les composants de recuit simulé δ et `T` sont mis à jour.
   L'algorithme converge lorsque le critère suivant est atteint dix fois consécutives : tous les objets créés en début d'itération doivent être éliminés à la fin de celle-ci.

3. Création des ellipses
   Selon le modèle MPP, la répartition des ellipses doit suivre une loi de Poisson. Cette loi est ici simplifiée de la manière suivante : un nombre aléatoire entre 0 et 1 est affecté à chaque pixel de l'image. Si ce nombre est inférieur à un seuil de probabilité d'apparition, alors une ellipse est créée sur le pixel (voir justification en annexe A.6). Chacune d'entre elle reçoit les propriétés aléatoirement choisies : x, y, ax, bx, angle, Ud. Les positions x et y sont respectivement comprises entre la largeur et la hauteur de l'image. L'angle est fonction de π. Les valeurs ax de grand axe et bx de petit axe sont déterminées à partir d'une constante R représentative de la taille d'un noyau sur l'image :

   ```
   ax = R * (1 + rnd / 2)
   bx = R / (1 + rnd / 2)
   ```

   La valeur Ud est la valeur d'attache aux données de l'ellipse.
   À chaque itération, le nombre d'ellipses à générer est mis à jour en fonction du nombre de pixels de l'image et du paramètre δ compris entre 0 et 1.

4. Élimination des ellipses
   Une boucle parcourt les ellipses créées dans cette itération. Pour chaque pixel d'une ellipse, si la valeur Ud n'est pas celle de la carte de voisinage, alors l'ellipse est marquée comme mauvaise. En cas de superposition de deux ellipses, seule la mieux attachée subsiste. Un second tri garde uniquement les ellipses dont l'attache aux données passe un seuil aléatoire calculé à partir des paramètres δ et T.

5. Progression du recuit simulé
   La valeur de température T est augmentée et celle de δ diminuée.

6. Test de la convergence
   Ce test intervient à la fin de chaque itération pour statuer sur les résultats obtenus. La variable Rnew comptabilise les ellipses nouvellement détectées et la variable Rold les ellipses gardées à l'itération précédentes mais non retenues sur celle-ci. La somme de ces deux valeurs représente le nombre de changements intervenus (Rchange). En établissant le rapport du nombre total d'ellipses gardées (Rtotal) par le nombre de changements :

$$Rtotal/Rchange > \mu \tag{2.10}$$

   Si l'expression est vérifiée, nous pouvons statuer que moins d'une ellipse sur μ a été modifiée. En fixant un seuil de modification en fonction de la densité d'ellipses sur l'image, nous pouvons ainsi régler le degré de convergence à atteindre.

2.5 Parallélisation de la méthode MPP

Le LIB et le Laboratoire d'Informatique Paris 6 (LIP6 [4]) ont collaboré à l'implémentation de l'algorithme MPP. Deux versions parallèles ont également été développées [15], l'une en multi-thread CPU, l'autre sur carte GPU. Les codes de la version parallèle CPU ont servi de point de départ à l'application développée dans le cadre de cette thèse.

2.5.1 Construction d'une carte d'attache aux données

Une stratégie originale a été adoptée pour traiter l'étape de mort jusqu'alors intrinsèquement séquentielle. Le tableau utilisé pour classer les ellipses a été remplacé par une carte dont les valeurs sont mises à jour uniquement si elles sont les meilleures. L'ordre dans lequel les ellipses sont classées n'a ainsi plus d'impact. La figure 2.3 illustre comment ce procédé garantit, en cas de recouvrement, que la valeur conservée est celle de l'ellipse la mieux attachée.

Les lettres indiquent l'ordre dans lequel les ellipses sont triées. Les chiffres représentent leur valeur d'attache. Sur l'image de gauche, les pixels aux intersections prennent successivement les valeurs suivantes :

[4] https://www.lip6.fr/

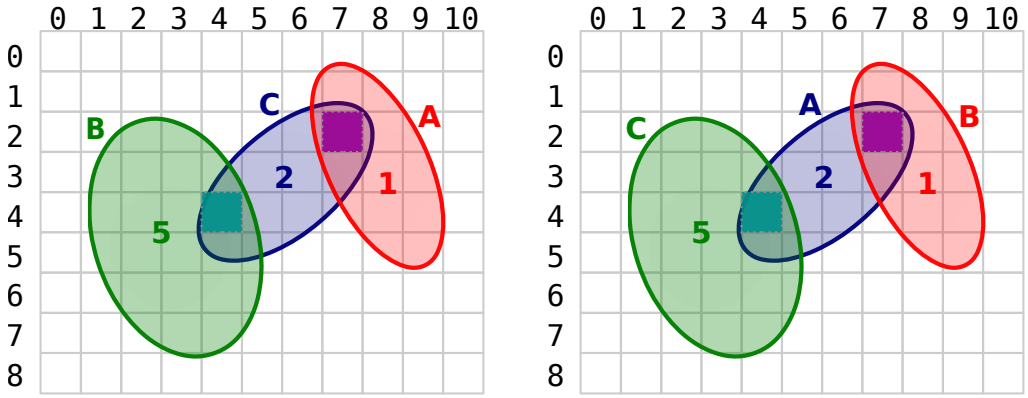


Figure 2.3: Deux possibilités de prise en compte du voisinage des ellipses

A: CARTE[4,4] = 2, CARTE[7,2] = 2
B: CARTE[4,4] = 2, CARTE[7,2] = 1
C: CARTE[4,4] = 2, CARTE[7,2] = 1
Pour le cas de droite :
A: CARTE[4,4] = 0, CARTE[7,2] = 1
B: CARTE[4,4] = 5, CARTE[7,2] = 1
C: CARTE[4,4] = 2, CARTE[7,2] = 1

Les valeurs finales des pixels considérés sont identiques. À gauche l'ellipse rouge élimine l'ellipse bleue qui élimine l'ellipse verte. À droite, l'ellipse bleue élimine l'ellipse verte qui est elle-même éliminée par la rouge. Dans les deux cas de figure seule l'ellipse rouge subsiste. Cette méthode garantit la subsistance de la meilleure ellipse et l'élimination de celles superposées. Notons qu'une remise à zéro de la carte à la fin de chaque itération est requise pour ne pas conserver les valeurs d'objets disparus.

2.5.2 Résultats

Les résultats obtenus sur une image de 4096×4096 pixels sont présentés sur les graphiques 2.4.

24 cœurs CPU [5] sont nécessaires pour rejoindre les performances obtenues sur carte GPU [6] qui atteignent des accélérations de l'ordre de x24 sur une itération.

[5] Intel X5650 hex-core @2.67GHz

[6] NVIDIA Fermi C2070 GPU

Temps moyens par itération **Accélérations moyennes**

Figure 2.4: Temps et accélération obtenus par l'algorithme MPP sur 12 cœurs CPU-hyperthread avec une image de 4096×4096

L'image testée ne correspond toutefois qu'à $1/154^{eme}$ d'une lame complète. La question est donc de savoir dans quelle mesure ces méthodes supportent le passage à l'échelle. Ce fut le point de départ des travaux présentés dans ce manuscrit.

2.6 Limites de la parallélisation à mémoire partagée de l'algorithme MPP

Les deux implémentations parallèles de l'algorithme MPP dont nous avons vu les résultats au chapitre précédent présentent toutes deux des limitations que nous allons démontrer.

2.6.1 Limitation mémoire sur GPU

Afin de déterminer la taille maximale d'image (w×h) pouvant être analysée sur GPU par l'algorithme MPP, nous allons estimer l'empreinte mémoire de l'application telle qu'elle nous a été transmise par le laboratoire du LIB.
Les objets nécessitant une attribution mémoire sont les suivants :

Nom de l'objet	Dimension	Type
`recouv_id`	w×h	float
`new_coord_X`	w×h×2	int
`new_coord_Y`	w×h×2	int
`ret_attach`	w×h×2	float
`ret_axeA`	w×h×2	float
`ret_axeB`	w×h×2	float
`ret_Angle`	w×h×2	float
`dep0`	w×h×3	float

En considérant les types `float` et `int` comme occupant quatre octets, la taille mémoire réservée par l'application représente 55 fois les dimensions de l'image.
Avec des cartes GPUs de 16 Go, le nombre de pixels calculables est de l'ordre de 0.29 Gigas, soit une image de dimension : 18340×15862 pixels.
La taille d'une image complète pouvant aller jusqu'à 100 000×100 000 pixels, un GPU ne dispose pas d'assez de mémoire.

2.6.2 Limitation des ressources CPU

Sur CPU, la limitation est d'un autre ordre, elle se situe au niveau du nombre de threads disponibles. Pour estimer la taille à partir de laquelle les CPU travaillent à 100%, nous pourrions analyser la complexité algorithmique de MPP en estimant le nombre d'instructions nécessaires à l'analyse d'une certaine image. Cependant la convergence de l'algorithme est sujette à de nombreuses variables aléatoires rendant cette méthode difficile à mettre en place. Expérimentalement, il serait possible d'observer le taux d'occupation des processeurs en augmentant progressivement la taille du problème. Toutefois une autre méthode a été utilisée sur la base des résultats expérimentaux présentés sur le graphique 2.4.
Afin de définir une limite à partir de laquelle l'augmentation du nombre de processeurs ne permet plus d'accroître l'accélération, nous allons utiliser la loi de Amdahl [7] :

$$S^a(P) = \frac{1}{f_s^a + \frac{1-f_s^a}{P}} \tag{2.11}$$

Cette expression représente le coefficient d'accélération d'un algorithme en fonction du nombre de processeurs P et du pourcentage de code séquentiel f_s^a. Le coefficient idéal est obtenu pour un nombre de processeurs tendant vers l'infini :

$$S^a_{ideal}(P) = \frac{1}{f_s^a},\ avec\ P \to \infty \tag{2.12}$$

À partir des résultats expérimentaux, le pourcentage de code parallélisable f_p^a peut être déduit comme suit :

$$f_p^a = \frac{1 - \frac{1}{S^a(P)}}{1 - \frac{1}{P}} \tag{2.13}$$

C'est ainsi que nous obtenons un score de parallélisation de 94% sur l'application. La courbe représentative des accélérations pour un tel pourcentage est visible sur le

[7] https://fr.wikipedia.org/wiki/Loi_d'Amdahl

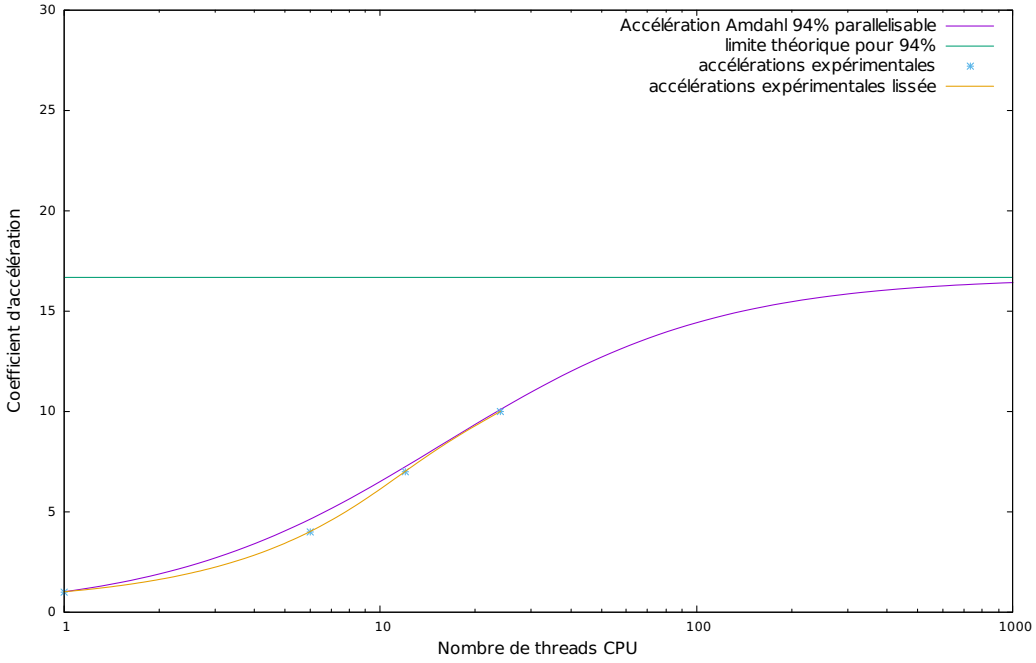


Figure 2.5: Courbe d'accélération de Amdahl pour un code parallélisable à 94%

graphique 2.5.

L'augmentation du nombre de threads est logarithmique. À 24 cœurs, l'accélération est aux deux tiers de celle idéale. Pour visualiser le taux d'augmentation nécessaire pour atteindre une accélération maximale, regardons le graphique 2.6.

Le premier point de la courbe à gauche signifie un taux d'accroissement de l'accélération de 50% en passant d'un à deux cœurs. Ce taux décroît ensuite fortement jusqu'à 40 cœurs où il atteint les 0,6%. De 39 à 40 cœurs le coefficient d'accélération n'est alors multiplié que par 1,006, soit une accélération quasi nulle. Nous pouvons en conclure qu'à partir de ce seuil, l'augmentation du nombre de cœurs n'apporte plus d'accélération significative.

2.6.3 Conclusion sur les limites de la parallélisation à mémoire partagée

La loi d'Amdahl s'attache à montrer les taux théoriques d'accélération par rapport à un problème de taille fixe. Nous avons pu ainsi constater qu'au-delà d'un certain nombre de cœurs, l'accélération n'est plus significative. Par ailleurs, la taille mémoire des GPUs est encore trop limitée pour permettre le traitement d'une lame complète.

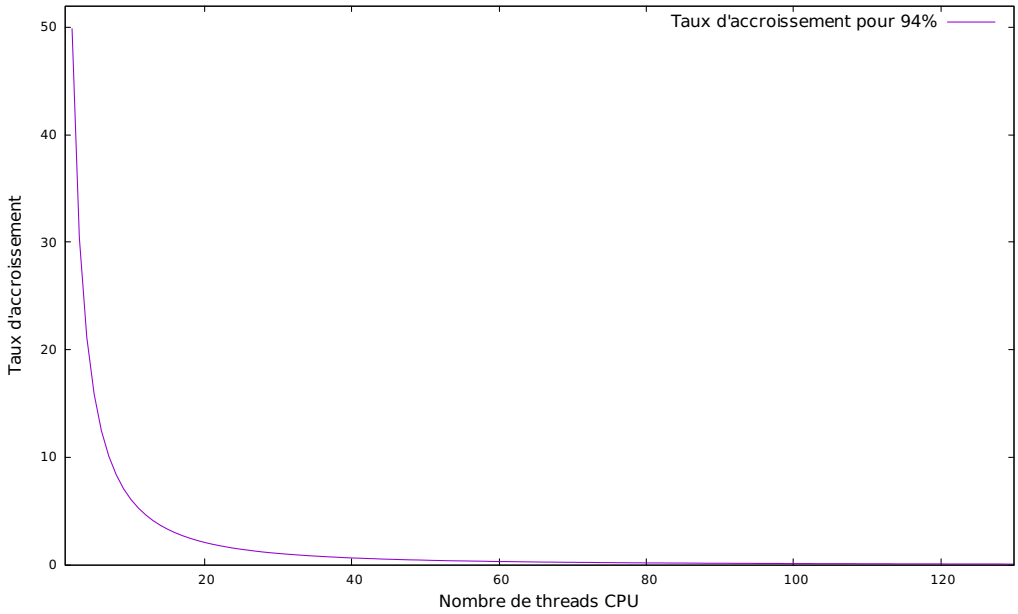


Figure 2.6: Courbe du taux d'accroissement d'accélération pour un code parallélisable à 94% en fonction du nombre de threads CPU

Pour contourner ces limitations, la parallélisation hybride conjuguant la mémoire partagée et la mémoire distribuée est la solution que nous allons explorer ici. Les avantages sont clairs, diviser la taille du problème et profiter d'une accélération maximale sur plusieurs nœuds. Malgré les bonnes performances de parallélisation obtenues sur GPU, nous ne nous sommes pas orientés vers ce type d'architecture. Nous souhaitons en effet proposer une même implémentation de l'algorithme utilisable à la fois dans un contexte à mémoire partagée et à mémoire distribuée. Les développements et la maintenance sont ainsi largement facilités et accessibles à un plus grand nombre d'utilisateurs. De plus, cette orientation fait écho aux volontés d'évolution de l'algorithme MPP énoncés dans [15]. C'est ainsi que nous avons privilégié l'utilisation d'une autre gamme de cartes accélératrices : les Xeon Phi.

2.7 Problématique de parallélisation

En distribuant l'image vers différents nœuds, les noyaux situés au niveau des lignes de coupe sont scindés en deux ou quatre parties. Le calcul d'attache aux données établi à partir de la distance de Bhattacharyya se calcule en parcourant pas à pas les points discrétisés obtenus à partir de l'équation de l'ellipse. Ces noyaux non détectables représentent une

infime partie du nombre total (voir [26]), mais un seul d'entre eux suffit à faire basculer le diagnostic d'un grade à un autre. Dès lors leur prise en compte est importante.

2.8 Conclusion

Pour atteindre notre objectif de traitement d'une lame entière de biopsie, nous proposons de dépasser les limites imposées par le parallélisme à mémoire partagée en y ajoutant une composante distribuée. La division d'une image implique cependant de prendre en considération la dépendance des données propres à l'algorithme MPP.
Afin de garantir le calcul d'attache aux données, une ellipse doit à sa création être contenue dans un périmètre à l'intérieur de l'image. Cette bande périphérique ne pouvant plus accueillir le centre d'une ellipse devient dès lors oubliée. Or, il est nécessaire de prendre en compte l'intégralité de l'image pour ne négliger aucun noyau.
Dans le chapitre suivant, nous allons voir quelle stratégie adopter face à cette nouvelle problématique et comment les principales méthodes de parallélisation peuvent y répondre.

Nous allons nous concentrer sur deux aspects : le respect de l'algorithme original et l'accélération du processus de traitement. La composante essentielle à prendre à compte devient dès lors la taille des données à analyser. La détection la plus efficace s'effectuant à des taux d'agrandissement de 400, les images atteignent de 1 à 10 GigaPixels en fonction de la lame scannée. La recherche d'une solution supportant le passage à l'échelle est donc un prérequis.

3 Méthodes et infrastructures de parallélisation adaptées au traitement d'une image de grande taille

Pour répondre à la problématique posée par la division des images de grande taille, ce chapitre propose d'explorer la méthode des cellules fantômes (ghost cells). Un état de l'art sur les principales méthodes de parallélisation est ensuite présenté de manière à introduire une implémentation de la méthode. Enfin nous conclurons sur les modalités d'utilisation de la parallélisation hybride et nous proposerons une alternative aux méthodes classiques.

3.1 Cellules fantômes

Les cellules fantômes sont des données nécessaires durant la phase de calcul mais ne font pas partie des résultats en sortie. Spécifiquement utilisée en parallélisation distribuée, cette technique permet d'économiser des coûts de communication entre les tâches. La figure 3.1 illustre un exemple avec la distribution d'une image (a) de 6×6 pixels.

Lors de la distribution de l'image vers quatre tâches, les cellules fantômes sont ajoutées de part et d'autre des traits de coupe (b). Sur l'image (c), un calcul modifie la valeur des pixels sans affecter celle des cellules fantômes. Nous pourrions en rester là, mais cet exemple illustre le cas d'un calcul itératif où une étape de mise à jour des cellules fantômes est nécessaire avant d'opérer une autre itération (d).
N'étant plus dans un contexte à mémoire partagée, la mise à jour des cellules doit faire l'objet d'un échange entre voisins. Chacun travaillant de manière autonome, pour s'assurer que l'une ne soit pas en avance par rapport à l'autre de plus d'une itération, il est nécessaire

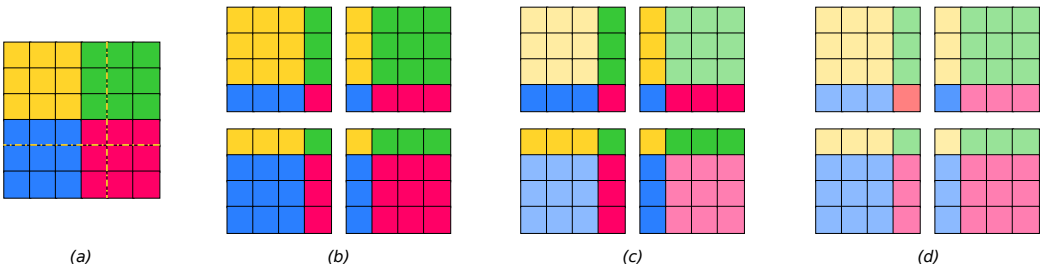


Figure 3.1: Distribution d'une image (a) sur 4 processus (b), calcul (c) et mise à jour des données fantômes (d)

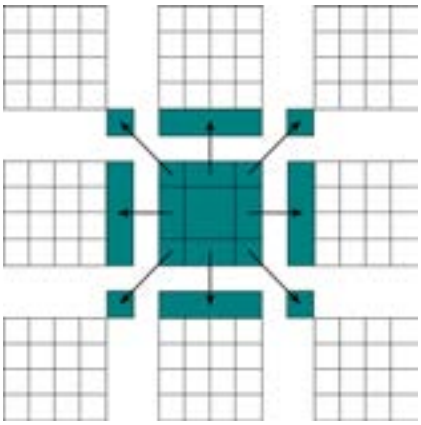


Figure 3.2: La tâche centrale envoie ses données à chacun de ses huit voisins

de synchroniser les calculs. Les communications peuvent dès lors être soit synchrones et bloquantes, soit asynchrones et non bloquantes ce qui nécessite l'ajout d'une barrière de synchronisation comme nous le verrons par la suite.
Dans [27], Palmer propose plusieurs implémentations de ces mises à jour. La première méthode nommée *simple put* (`S_P`) (voir schéma 3.2) repose sur un échange vers tous les voisins avec qui une donnée est partagée.

Cette méthode unidirectionnelle peut être déclenchée par toutes les tâches simultanément sans risque d'écriture concurrente. Ces envois doivent cependant être synchronisés afin de s'assurer que :

1. la donnée a bien été utilisée avant de la mettre à jour ;
2. la donnée a bien été mise à jour avant d'être utilisée.

Le nombre de transmissions dépend alors de la dimension du problème. En 2D, neuf transmissions sont nécessaires et pour un système à *D-dimensions* : $3^D - 1$.

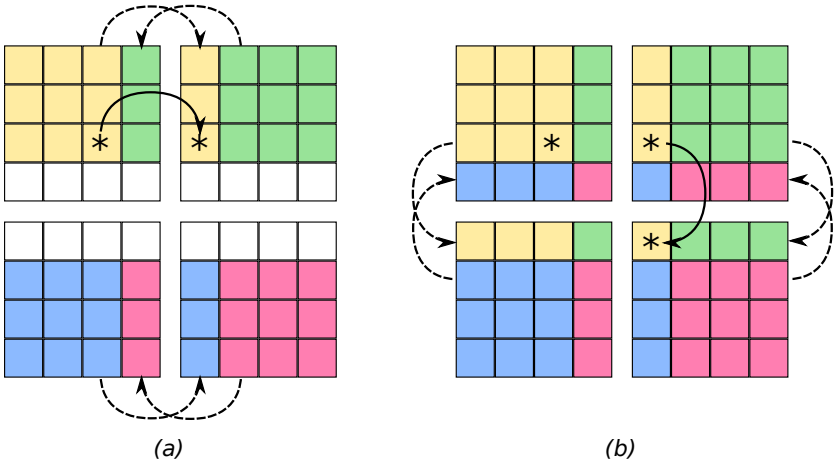


Figure 3.3: Algorithme de shift dans lequel les cellules fantômes sont échangées axe par axe. Source image [29]

La méthode *shift* [28] permet de réduire le nombre de communications en considérant le fait que certaines données sont communes à plus de deux tâches (voir illustration 3.3).

En échangeant séquentiellement les données par bandes tout d'abord verticalement puis horizontalement (ou inversement), nous pouvons voir que la donnée marquée d'une astérisque qui est à échanger avec le voisin en diagonale est mise à jour lors de la seconde vague d'échange. La méthode *shift* permet de passer de $3^D - 1$ à $2.D$ communications.

Passing Flag Dans le cas d'une parallélisation hybride, les échanges locaux à un nœud peuvent être remplacés par une écriture directe en mémoire. Cette économie nécessite toutefois l'ajout d'un élément attestant de l'état de la donnée : *à jour* / *utilisée*. Cette variante de l'algorithme *shift* nommée `P_Flag`, impose également la mise en place d'une barrière de synchronisation devant laquelle une tâche devra attendre que la donnée soit *à jour*.

Halo profond Le principe de duplication des données aux frontières peut être dans certains cas augmenté pour couvrir plus d'une itération. Il suffit pour cela d'agrandir la zone de cellules fantômes. Cette méthode dite de halo profond [29] permet de réaliser *n-1* calculs avec une bande de cellules fantômes de largeur *n* (voir illustration 3.4).

Les communications sont évitées au prix de calculs supplémentaires. Le bon ratio communication / calcul est alors à déterminer pour obtenir les meilleures performances.

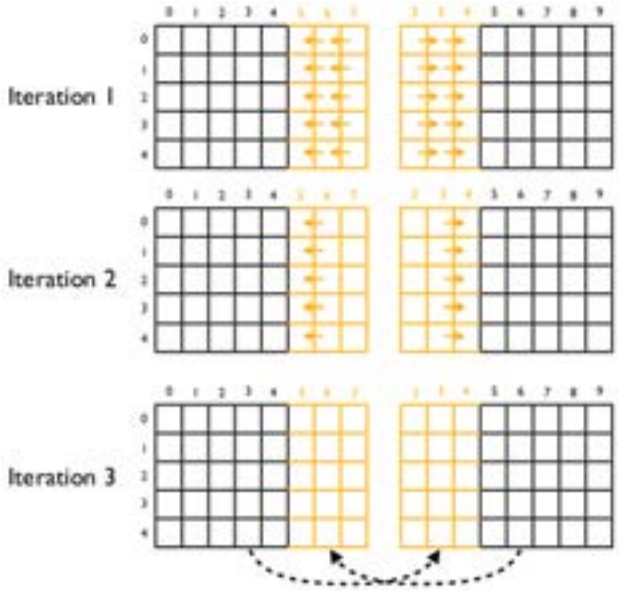


Figure 3.4: Méthode de halo profond de largeur 3. Source image [29]

Avant de présenter un comparatif des méthodes `S_P`, `M_P` et `M_P/P_Flag`, la section suivante va présenter les principales méthodes et outils de parallélisation évoqués dans cette thèse.

3.2 Outils de parallélisation

La parallélisation d'applications demande aujourd'hui de gros efforts de design, de programmation et de maintenance. Les modèles et les outils de parallélisation les plus utilisés [30] sont répartis en 5 grandes familles :

- les threads CPU : POSIX, OpenMP, TBB ;
- les threads GPU : CUDA, OpenCL, OpenACC ;
- le passage de messages : MPI ;
- le traitement vectoriel : SSE, AVX ;
- les approches fonctionnelles : BSP [31], skeletons [1], map reduce [2].

[1] https://en.wikipedia.org/wiki/Algorithmic_skeleton

[2] https://fr.wikipedia.org/wiki/MapReduce

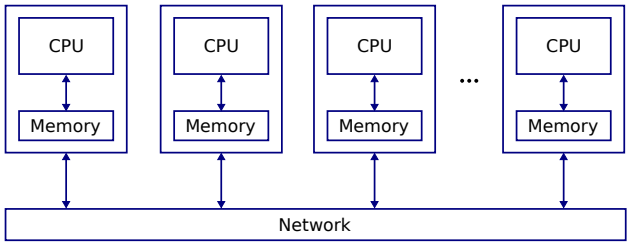


Figure 3.5: Architecture à mémoire distribuée composée d'ordinateurs reliés par un réseau

Les bases de la programmation parallèle reposent sur la bonne utilisation de trois composantes : l'unité de calcul, la mémoire, le réseau. En local, l'échange d'informations peut être géré par des variables globales. En distant, ces mêmes informations doivent transiter par le réseau. Une programmation hybride à mémoire partagée et à mémoire distribuée est possible en associant MPI et OpenMP.

3.2.1 Message Passing Interface (MPI)

Le standard MPI décrit une bibliothèque de fonctions permettant d'établir des communications point à point entre les processeurs d'un même réseau (voir illustration 3.5).

Différentes implémentations existent dont les solutions libres : MPICH [3], OpenMPI [4]. La communication entre les processus d'une même application s'effectue sous forme de messages. Le réseau d'ordinateurs participant au calcul et mettant en commun ses ressources est défini au lancement par un fichier d'hôtes. L'envoi d'un message se fait à l'intérieur d'un groupe et permet d'échanger des données de type scalaire, des tableaux ou encore des types structurés définis par l'utilisateur. Tous les nœuds appartiennent au groupe global qu'il est possible de subdiviser en sous-groupes. Les communications mettent en œuvre un émetteur et un récepteur qui doivent explicitement faire état de leur intention d'envoyer ou de recevoir un message. L'exemple 3.6 illustre les échanges nécessaires à la distribution d'une image du nœud maître vers quatre nœuds.

Cet exemple illustre la distribution de données à partir d'un nœud central avec la méthode `MPI_SScatter`. Les nœuds effectuent leurs opérations et s'échangent des bandes de données (en bleu) avec les méthodes `MPI_Send` et `MPI_Recv`. En mode standard, les échanges sont bloquants. L'émetteur doit attendre que ses données soient entièrement

[3] https://www.mpich.org/

[4] https://www.open-mpi.org/

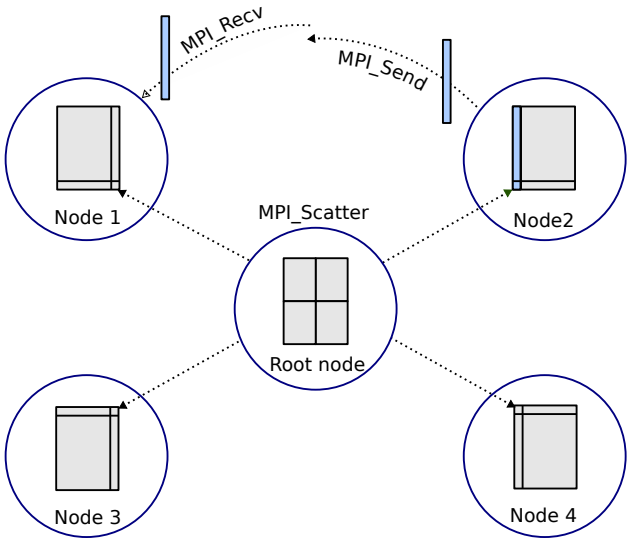


Figure 3.6: Exemple MPI d'envoi d'un message d'un vers tous (`MPI_Scatter`), puis envoi du node2 vers le node1 d'un autre message.

copiées dans un buffer côté récepteur avant de continuer son flot d'exécution. Les autres modes de communication disponibles sont :

- synchrone bloquant `MPI_Ssend` : l'émetteur n'est libéré que lorsque le buffer a été lu par le récepteur.
- non bloquant `MPI_Isend` / `MPI_Irecv` : le flot d'exécution n'est pas bloqué. Une barrière de synchronisation est alors nécessaire (`MPI_Wait`) pour garantir la validité des données.

D'autres fonctions permettent de diffuser un message en mode un vers tous : `MPI_Bcast`, ou encore de répartir un ensemble de données vers différents récepteurs : `MPI_Scatter / MPI_Gather`. Une opération de réduction `MPI_Reduce` permet d'opérer directement sur les données d'un ensemble de diffuseurs.
MPI est utilisable avec les langages C, C++ et Fortran. Sa diffusion est répandue sur toutes les plateformes ce qui en fait la solution standard en matière de parallélisme distribué. Chaque communication doit être rigoureusement définie.
La simplicité de l'implémentation est proportionnelle au nombre d'échanges et au nombre de nœuds.

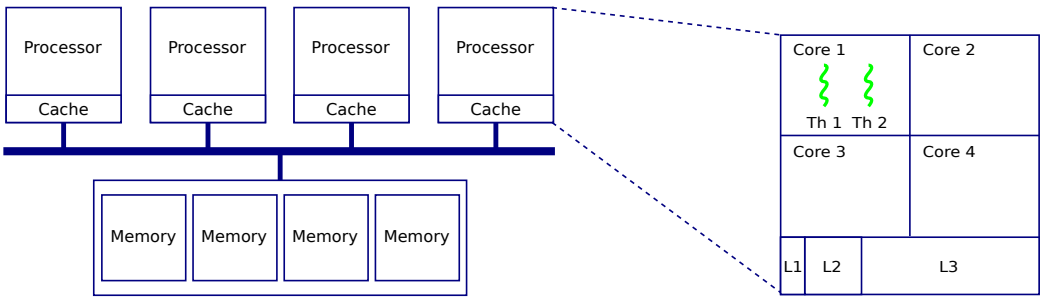


Figure 3.7: Architecture à mémoire partagée composée de quatre processeurs de quatre cœurs avec hyperthread. multicœur

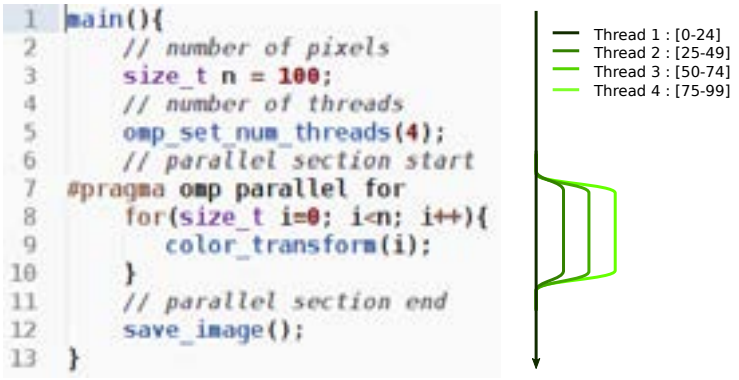


Figure 3.8: Architecture à mémoire partagée détaillant la constitution d'un processeur multicœur

3.2.2 OpenMP

OpenMP est le standard en matière de parallélisation à mémoire partagée. Il permet d'utiliser simplement l'ensemble des threads disponibles sur une machine à mémoire partagée (voir illustration 3.7).

Sur cet exemple, 32 threads sont utilisables. L'implémentation du code parallèle est partiellement implicite et passe par l'ajout de directives (`#pragma`). Les boucles `for` se prêtent particulièrement bien à ce mode de répartition, où à partir d'un processus père, des fils sont créés selon le modèle `fork / join` (voir illustration 3.8).

À l'entrée de la section délimitée par l'instruction `#pragma omp parallel for`, le thread principal (en vert foncé) est divisé en quatre threads. Le nombre de thread est défini à la ligne 5 par la méthode `omp_set_num_threads`.
À l'intérieur d'un bloc parallèle, les variables peuvent être définies comme publiques à tous les threads ou privées au thread courant. Une variable publique peut être incrémentée de manière atomique et un bloc entier de code peut n'être accessible qu'à un seul thread.

La répartition des calculs au sein d'un pool de ressources peut être :

- uniforme : tous reçoivent le même nombre d'itérations.
- dynamique : le travail est distribué au fur et à mesure.

Depuis la version 3.0 du standard, un nouveau modèle de tâches permet de paralléliser en dehors des boucles. Un thread peut ainsi créer une ou plusieurs tâches contenant des instructions. Ces tâches sont prises en charge par les threads disponibles. Les différentes parties d'un même programme peuvent ainsi se répartir aisément entre plusieurs threads à condition qu'il n'y ait pas de dépendance de données.
Les accès concurrents à une donnée constituent une des limitations de ce mode d'utilisation. Ce type d'opération rend coûteux le maintien de cohérence entre les caches (problème de *false-sharing*). OpenMP est une solution simple, non intrusive et disponible par défaut sur les compilateurs. Son exécution est toutefois limitée par le nombre de threads disponibles. La synchronisation et la création des processus légers peuvent apporter des surcoûts non négligeables qu'il faut observer pour atteindre des performances optimales.

3.3 Cartes accélératrices

L'utilisation des cartes accélératrices constitue un nouveau pan de la parallélisation. Permettant d'accroître la puissance théorique des ordinateurs de manière spectaculaire, leur utilisation relève toutefois d'un haut niveau d'expertise. Les unités de calcul, dont le nombre varie de quelques dizaines à plusieurs milliers, jouent sur un registre différent de celui des CPUs plus puissants et plus "intelligents". Les Xeon Phi misent sur des instructions vectorielles, les GPUs eux sur un mode de programmation massivement SIMD (Single Instruction Multiple Data).

3.3.1 Graphic Processor Unit (GPU)

Conçus pour fournir des rendus graphiques 3D temps réel, les GPUs sont devenus un nouveau mode de programmation hautement parallèle. D'un point de vue architecture, comparé à un CPU classique, le nombre d'unités de calcul est largement supérieur : 3584 sur le modèle Titan X nVIDIA. Avec une fréquence de l'ordre de 1,4GHz les performances sont alors de 11 TFLOPS sur des flottants simple précision.
Le paradigme de programmation repose sur un principe SIMT (Single Instruction Multiple Thread) où la même fonction est appliquée à tous les threads d'un *warp* (groupement

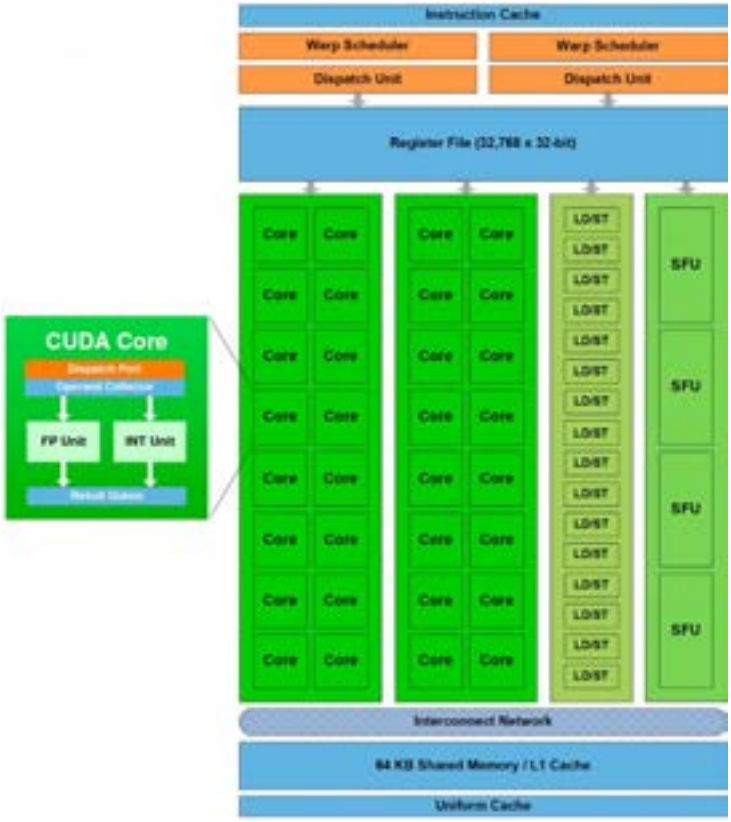


Figure 3.9: Composition d'un SM sur architecture Fermi.
Source : NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

de 32 threads).

D'un point de vue architecture, chaque thread appartient à un cœur cuda qui est réparti sur un *streaming multiprocessors* (SM). Ces derniers sont positionnés autour d'un cache commun de niveau 2 et contiennent de 8 à 64 cœurs cuda (voir illustration 3.9).

La répartition des calculs se fait à partir d'une grille définie par l'utilisateur au lancement du calcul. La dimension de cette dernière est calquée sur la taille des données à traiter. Au lancement du programme, les blocs s'exécutent en même temps dans un ordre non défini. Au sein d'un même bloc, les threads coopèrent autour d'une mémoire globale et peuvent être synchronisés.
Pour garantir un bon degré de parallélisation, les problèmes calculés doivent être de taille suffisamment grande pour occuper les cœurs. Il est ainsi recommandé d'avoir un ratio de

warps actifs au sein d'un SM de 50% [5]. En deçà, le problème est trop petit et les performances ne sont pas optimales.

3.3.2 Cartes accélératrices Xeon Phi

Initialement conçus pour intégrer les supercalculateurs et leur permettre de profiter de capacités vectorielles 512-bit [6], les cartes Xeon Phi ont eu une durée de vie relativement courte (2011-2019).
Malgré cela, les principales innovations proposées par la carte ont perduré et sont maintenant directement intégrées aux plateformes Xeon Scalable [7].

Les cartes Xeon Phi ont la particularité de disposer d'un système d'exploitation propre (le micro-système Busybox [8]) et de disposer d'un port réseau qui leur permet d'être directement accessible.
Deux modes d'utilisation sont possibles : un mode *natif*, dans lequel l'application est compilée sur le nœud hôte et exécutée depuis la carte, et un mode *offload* dans lequel une partie ou la totalité des calculs de l'application est envoyée vers la carte.
Le mode *offload* consiste en l'ajout d'instructions pré-processeurs de type `#pragma` permettant d'orienter les flux de données vers telle ou telle unité de calcul. Le mode natif ne nécessite pas de modification algorithmique par rapport à l'utilisation sur CPU. Ce mode est celui qui permet d'exploiter le plus efficacement les capacités d'une carte en évitant des communications via le bus PCIe.

La version 5110P Knight Corner de la carte est composée d'une soixantaine de cœurs cadencés à 1.1GHz disposant chacun de quatre threads hardwares (avec registre propre).
Les cœurs partagent une mémoire centrale de 8Go et sont reliés entre eux par une interface appelée Core Ring Interface.

Les quatre threads d'un cœur acheminent des instructions qui, une fois décodées, sont orientées vers deux pipelines d'exécution. L'un est spécifiquement scalaire, l'autre prend

[5] Source : CUDA C Best Practices Guide

[6] https://en.wikipedia.org/wiki/AVX-512

[7] https://www.networkworld.com/article/3296004/intel-ends-the-xeon-phi-product-line.html

[8] https://busybox.net/about.html

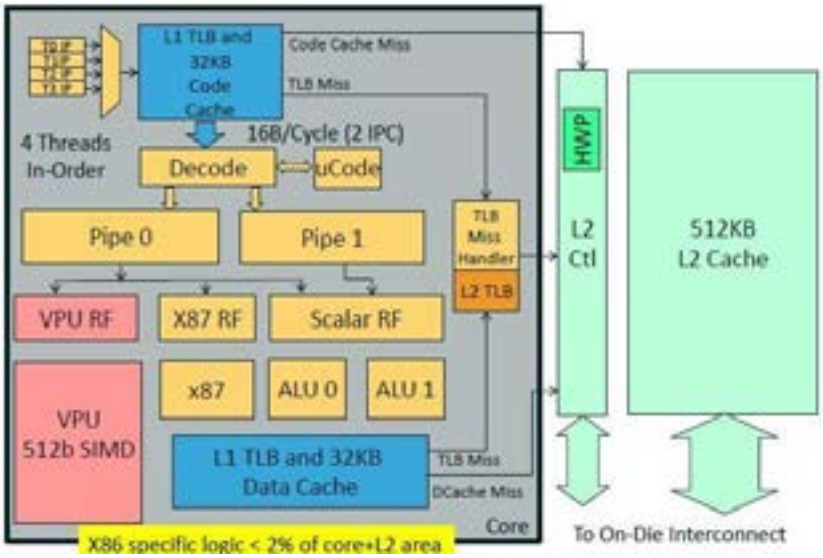


Figure 3.10: Diagramme d'architecture d'un cœur Xéon Phi KNC

en charge les opérations vectorielles, les flottants et les scalaires (voir 3.10). L'unité vectorielle (VPU) peut traiter 16 flottants simple précision ou 8 flottants double précision en un cycle. Chaque opération pouvant être une FMA (Fused Multiple Add), la puissance théorique est de l'ordre de :
16 x 2 (FMA) x 1.1 (GHz) x 60 (nombre de cœurs) = 2112 GFlop/sec en simple précision,
8 x 2 (FMA) x 1.1 (GHz) x 60 (nombre de cœurs) = 1056 GFlop/sec en double précision.

Pour être opérationnelle, la vectorisation doit traiter des données alignées en mémoire. Cet alignement peut être spécifié en C par le mot clé `alignas` [32] qui s'utilise comme suit : `alignas(sizeof(float[16]))float fvec[16]`

Un rapport d'optimisation, disponible à la compilation (option `-qopt-report`) permet de rendre compte des problèmes d'alignement ou de vectorisation. Ce dernier aspect est essentiel pour tirer réellement partie des capacités du Xeon Phi. De manière générale, la vectorisation est contrariée par : les sauts ou les branches, un nombre d'itérations non déterminé au lancement d'une boucle, les dépendances arrières [9] dans l'ordre d'exécution des itérations.
Ces optimisations, une fois effectuées, présentent l'avantage d'optimiser également l'exécution de l'algorithme sur CPU.

[9] Source : https://en.wikipedia.org/wiki/Loop_dependence_analysis

3.4 Implémentation de la méthode M_P et résultats

Après cette présentation de différents modes de parallélisme, explorons une implémentation MPI de la méthode `M_P` présentée en début de chapitre.
L'échange d'une bande de donnée (`cell[i]`) entre les voisins d'une tâche peut s'écrire :

```
for (size_t i=0; i < nb_voisins; i++) {
  MPI_Irecv(cell[i], nb_cells,
                     cell_type,
                     voisins[i], tag, comm,
                     &requetes[i]);
}

for (size_t i=0; i < nb_voisins; i++) {
  MPI_Isend(bord[i], nb_cells,
                     cell_type,
                     voisins[i], tag, comm,
                     &requetes[nb_voisins+i]);
}

MPI_Waitall(2*nb_voisins, requetes, statuts);
```

Cette implémentation décrite dans [29] utilise des fonctions asynchrones (`MPI_Irecv` et `MPI_Isend`) pour permettre un envoi simultané de toutes les requêtes. La première boucle `for` programme la réception des données de chaque voisin (`voisins[i]`) vers la zone de stockage de ses cellules fantômes (`cell[i]`). La seconde boucle déclenche l'envoi des données sur ses bords `bord[i]` vers ses voisins.
Cette portion de code représente la mise à jour des cellules fantômes devant précéder une nouvelle itération de calcul. Les fonctions étant asynchrones, il est dès lors nécessaire de s'assurer que :

1. les données des cellules fantômes correspondent à l'itération en cours ;
2. les données propres ont bien été transmises avant d'être réécrites.

La fonction `MPI_Waitall` bloque toute progression des tâches jusqu'à ce que les `2*nb_voisins` requêtes aient abouti.

Performance Les tests du graphique 3.11 comparent les temps de mise à jour des cellules fantômes avec les méthodes *simple put* (`S_P`), *shift* avec passage de message `M_P` et *shift* avec passage de message et drapeau `M_P/P_Flag`.

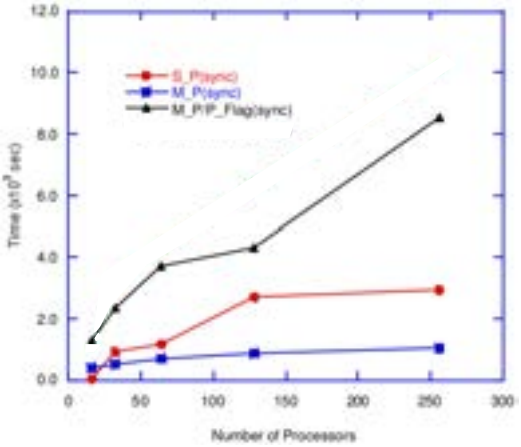


Figure 3.11: Temps de mise à jour des cellules fantômes avec synchronisation en deux dimensions. Source graphique [27]

Nous pouvons constater que les coûts de synchronisation augmentent en fonction du nombre de processeurs pour les versions `S_P` et `M_P/P_Flag` et restent presque stables pour la version `M_P`.

3.5 Limitations des méthodes exposées

Nous avons vu que pour dépasser les performances obtenues par l'algorithme MPP en mémoire partagée, un parallélisme hybride est requis. OpenMP propose un mode distribué mais le modèle perd de son efficacité notamment à cause des latences réseau. De plus, les directives ont une expressivité limitée.
MPI propose lui aussi une version implémentant le multithreading au sein de ses processus. Cependant les restrictions sont nombreuses. Deux threads ne peuvent par exemple pas utiliser en même temps des opérations collectives sur un même communicateur [33].
L'utilisation conjointe de MPI et d'un autre modèle OpenMP ou GPU sont les solutions hybrides les plus utilisées en HPC. Des compétences dans plusieurs domaines sont alors nécessaires, ce qui augmente les coûts de développement et de maintenance.
Par ailleurs, l'utilisation de MPI avec modèle `M_P/P_Flag` souffre de synchronisations locales pénalisantes par rapport à celui `M_P`.
Un nouveau paradigme de programmation parallèle hybride utilise des ressources bas niveau pour traiter de manière automatique les accès concurrents aussi bien localement qu'à distance. Avec des performances comparables à la combinaison MPI / OpenMP, ORWL est la solution que nous allons tout d'abord présenter dans le prochain chapitre puis utiliser pour implémenter notre solution MPP parallèle hybride.

4 Ordered Read-Write Locks : un modèle de programmation parallèle et distribuée

4.1 Modèle

ORWL est une libre interprétation des travaux de Leslie Valiant sur le modèle BSP [31] (Bulk-Synchronous Parallel). La mémoire, les communications et la synchronisation de bas niveau ne sont plus à la charge du développeur mais du système. Le travail est accompli par des tâches interconnectées qui peuvent être indistinctement locales ou distantes. Au lancement, les données sont réparties tout d'abord entre les processus du programme puis entre les tâches selon un modèle SPMD [34] (Single Program Multiple Data).
L'originalité d'ORWL réside dans son aptitude à gérer les accès concurrents aux données partagées. Une application ORWL se construit autour de ces points de contention qui vont permettre d'articuler les tâches entre elles.
Deux concepts forts coexistent alors : les *locations* et les handlers.
Une *location* est un objet abstrait qui symbolise une donnée partagée. Elle est composée d'un système de verrou, d'un pointeur vers la donnée et d'une file d'attente dans laquelle viennent s'insérer les requêtes provenant des handlers.

Le schéma 4.1 montre un exemple d'utilisation d'ORWL où deux tâches échangent leurs données limitrophes (en vert).

Les données partagées sont symbolisées sous forme de *locations* : `s_loc` et `n_loc` . Chacune d'entre elle est composée d'un buffer (en vert) et d'une FIFO (en jaune). Les *handlers* implémentés par les tâches (en bleu) viennent s'insérer dans la liste d'une *location* soit en écriture (w) soit en lecture (r). La position d'insertion est définie à l'instanciation et constitue la politique d'ordonnancement des tâches. Une *location* est accessible en écriture uniquement par la tâche qui la détient. Les accès en lecture ne sont

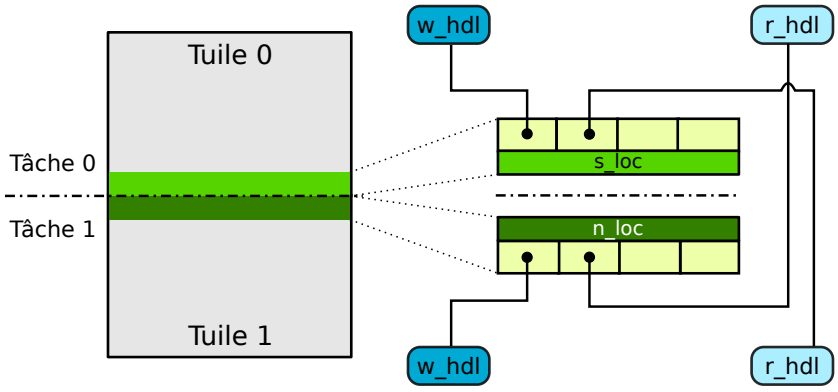


Figure 4.1: Insertion de handles (en bleu) en écriture (w) et en lecture (r) dans les FIFOs (en jaune) de *locations* (en vert).

pas exclusifs.
Pour obtenir le *lock* sur une *location*, la requête qui a été insérée dans la FIFO doit arriver en première position. L'ordre d'exécution des tâches est ainsi régulé en fonction d'un arbre de dépendances créé à partir des demandes d'accès. Il n'y a pas de *scheduler* centralisé. Les tâches suivent leur flot jusqu'à ce qu'elles aient besoin d'une ressource accessible dans une section critique. Si la tâche n'est pas prioritaire, elle entre dans une boucle d'attente active. Une fois la ressource libérée, si la tâche a la priorité la plus basse, l'accès est accordé.

ORWL garantit par ce mécanisme les notions suivantes :

1. l'absence de famine et d'interblocage ou *liveness* (voir [35]).
2. l'équité entre les processus dans l'acquisition d'une ressource. C'est l'une des différences avec le modèle voisin de *readers-writes locks* où l'OS gère la planification.
3. l'efficacité de la mise à disposition des données au moment requis.

Une fois l'application ORWL déployée et la phase d'initialisation achevée, l'intercommunication entre les tâches permet d'accéder à n'importe quelle donnée. L'illustration 4.2 schématise une telle architecture. Les tâches sont reliées entre elles et ont accès à toutes les *locations* de la zone centrale. En local, ces communications utilisent la mémoire partagée, en distant elles sont gérées par un serveur instancié sur chaque nœud.

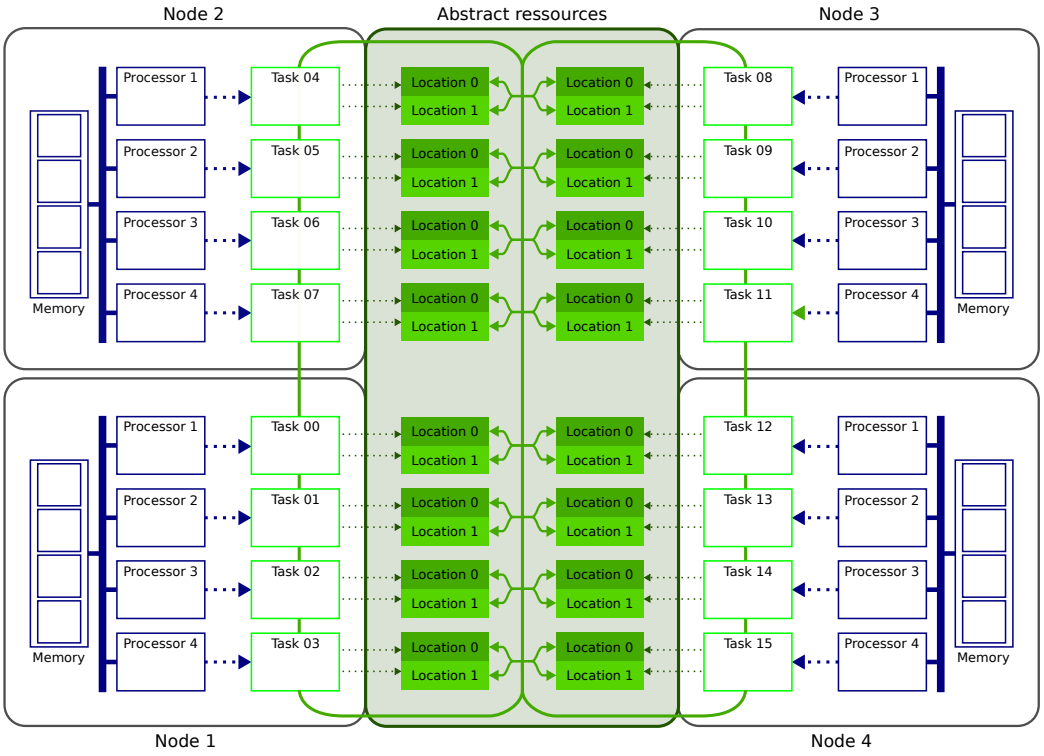


Figure 4.2: Architecture d'une application ORWL composée de 16 tâches et 32 *locations*

4.2 Domaine d'application

La structuration d'une application ORWL permet d'écrire librement à l'intérieur d'un bloc le code qui sera exécuté par une tâche. À l'intérieur de ces blocs, l'accès à toutes les ressources partagées de l'application sont permises dans les sections critiques. La modélisation des problèmes de type stencil [36] où le calcul d'un point dépend de son voisinage tire particulièrement avantage de ce type de représentation. Une phase de mise à jour globale des *locations* permet, avant le calcul, de récupérer les données voisines. Après le calcul, une mise à jour locale de ses *locations*, permet d'envoyer ses propres données aux voisins.
Les graphes de flot de données (DFG) constituent un second domaine d'application d'ORWL. Les nœuds du graphe symbolisent les tâches qui sont connectées entre elles par des sections critiques d'entrées/sorties (les arcs). La concurrence s'exprime par le fait que les fonctions peuvent délivrer leurs résultats simultanément. Une tâche est dépendante d'une autre lorsqu'elle en attend la sortie.
Un exemple d'implémentation de problème *data-driven* est présenté dans [37]. Le traitement en flux continu d'images vidéos est assigné à des tâches en pipeline qui exécutent les actions : acquisition, extraction de fond, érosion, dilatation, connexion, tracking. Les ressources partagées sont alors utilisées pour transmettre les informations d'une tâche à l'autre tout en garantissant le bon ordre d'exécution.

4.3 Performance

Les performances d'ORWL ont été comparées aux méthodes classiques de parallélisation (MPI, OpenMP, CUDA) sur une application de multiplication de matrice dense par bloc cyclique (voir [30]). Les librairies scientifiques ATLAS (CPU) et CUBLAS (GPU) ont été utilisées dans les versions ORWL - CPU, ORWL - GPU, MPI+OpenMP - GPU, MPI+OpenMP - CPU. Les benchmarks ont été menés à problème constant avec une augmentation des ressources (*speed-up*), puis avec une augmentation des ressources conjuguée à la taille du problème (*size-up*). Les résultats montrent que ORWL obtient des performances légèrement moins bonne en accélération en atteignant 360 GFlops/s contre 500 GFlops/s pour la version MPI+OpenMP optimisée. En revanche, en termes de passage à l'échelle, les performances d'ORWL sont comparables aux versions optimisées de références. Un avantage concret apparaît même dans des conditions d'utilisation hétérogènes (CPU + GPU).

4.4 Étapes de parallélisation

La légitimité d'une parallélisation ORWL s'apprécie au degré de dépendance algorithmique des données. De là, une modélisation des *locations* peut être établie et l'implémentation peut commencer. Le script de lancement joue également un rôle majeur dans le workflow de l'application de son initialisation à sa fin. Nous aborderons également le cheminement d'une requête d'insertion pour avoir un aperçu du fonctionnement interne d'ORWL.

4.4.1 Pré-étude

Les outils de profilage comme gperftools [1] permettent de mettre en évidence les points chauds d'une application, là où la parallélisation aura le plus d'impact sur les performances. L'analyse algorithmique des fonctions les plus consommatrices de calcul permet ensuite de statuer sur l'existence de dépendances au niveau des données. Si tel est le cas, la capacité d'ORWL à modéliser les accès concurrents permet de coder une application sans interblocage.

4.4.2 Compilation

L'utilisation d'ORWL passe tout d'abord par la compilation de sa librairie. Écrite en C11, son style est emprunté au *modern C* [32] avec une utilisation intensive des macros. ORWL est conçu pour s'adapter à des compilateurs n'implémentant pas les fonctions : `_Atomic`, `_thread`, `_Generic`.
L'en-tête du fichier Makefile utilisé pour la compilation décrit un certain nombre de variables ou scripts annexes à intégrer :

[1] https://github.com/gperftools/gperftools

```
include $(ORWL_DIR)/scripts/Makefile-getconf-orwl
include $(ORWL_DIR)/scripts/Makefile-target

ORWL_DIR    = \textit{chemin_vers_librairie_ORWL}
IPATH      += -I$(ORWL_DIR)/include -I$(ORWL_DIR)/p99
LPATH      += -Wl,-rpath,${ORWL_DIR}/lib -L${ORWL_DIR}/lib
              -lpthread
STD_FLAGS   = -std=c11 -D_XOPEN_SOURCE=600
```

Les premières lignes du listing incluent d'autres fichiers Makefile disponibles dans la librairie ORWL. Cette inclusion permet de personnaliser des variables d'environnement en fonction du système hôte. Des scripts permettent la détection d'informations comme : la version du standard C, le compilateur C par défaut, la version POSIX. Ils permettent également de définir une option essentielle : `-march=native`. Celle-ci renseigne le compilateur sur le jeu d'instructions disponibles sur le processeur. En l'absence de cette option, la librairie compilera normalement mais aucun thread ne pourra franchir une section critique. Cette erreur silencieuse est particulièrement difficile à détecter.
La dernière ligne indique les normes utilisées, : *C11* pour le langage C et la version de 2004 pour POSIX.

4.4.3 Script de lancement

Le lancement et le déploiement de l'application depuis un nœud maître est opéré par un script ruby : *orwlrun*. Les nœuds de calcul doivent être référencés dans un fichier texte (`ORWL_NODEFILE`). Une version locale peut également être déployée sur un unique nœud si l'option `local` est spécifié. Le lancement de l'application prend la forme suivante :

```
scripts/orwlrun output_path launch_script #nb_nodes
#nb_procs[:#task_per_proc] [executable [arguments ...]]
```

Les paramètres sont les suivants:

- `output_path` indique le dossier export des résultats (voir section 4.4.6).
- `launch_script` est le script envoyé vers les nœuds pour y exécuter l'application localement.
- `nb_nodes` définit le nombre de nœuds de déploiement. Si elle est définie à *all*, l'application s'exécutera sur tous les nœuds de la liste.

- `ORWL_NODEFILE`. Si le nombre spécifié dépasse le nombre de nœuds disponibles, la liste est reprise du début et plusieurs processus sont lancés sur un même nœud. Le déploiement local est paramétrable ici en indiquant le mot clé "local".
- `nb_procs` détermine le nombre de processus à lancer par nœud.
- `task_per_proc` détermine le nombre de tâches par processus.

Des variables d'environnement sont par ailleurs nécessaires. Elles sont copiées sur la ligne de commande en préambule au lancement. Leurs noms et leurs rôles sont les suivants :

- `ORWL_TARGET` : permet de pointer sur une version spécifique de la librairie ORWL. Par exemple sur Xéon Phi cette variable doit être définie à `mic` pour invoquer la librairie `liborwl_mic`.
- `ORWL_VERBOSE` : si supérieur à 0, permet de tracer les évènements internes à la librairie. Des indications sur la progression des transferts ou les états de connexions entre les serveurs sont imprimés dans les fichiers de logs de l'application (voir section 4.4.6).
- `ORWL_ULIMIT` : définit la taille des fichiers *coredump* générés sur les nœuds hôtes de l'application.
- `ORWL_EXECDIR` : définit le répertoire d'exécution sur les nœuds hôtes.
- `ORWL_NODEFILE` : lien vers le fichier texte contenant les adresses des nœuds.
- `ORWL_TMPDIR` : lien vers le répertoire où sont stockés des fichiers temporaires. Par défaut, `/tmp` est utilisé.
- `ORWL_SECRET` : un nombre aléatoire garantissant l'unicité d'un processus ORWL.

Après avoir défini une liste des nœuds cibles en fonction des paramètres de la commande et des ressources, le script de déploiement diffuse vers chaque nœud les fichiers nécessaires à l'exécution de l'application.
La distribution de l'exécutable et des données vers les nœuds est assurée par la librairie TakTuk [2] (voir section 5.7.3) qui assure le déploiement parallèle et l'exécution à distance de commandes. Il permet aussi la redirection des flux vers le nœud de déploiement.
`orwlrun` utilise les commandes à distance suivantes :

[2] http://taktuk.gforge.inria.fr/

1. la diffusion de l'application : `TAKTUK_BROADC_CMD` ;
2. l'exécution de l'application : `TAKTUK_EXEC_CMD` ;
3. la terminaison des processus sur les nœuds distants : `TAKTUK_KILL_CMD` ;
4. la compression, la récupération et la concaténation des fichiers de log sur les nœuds : `TAKTUK_EXIT_CMD`.

Une fois le transfert des fichiers effectué, l'application est lancée simultanément sur chaque nœud.

4.4.4 Initialisation

Le démarrage d'ORWL débute avec l'appel à la fonction `orwl_init()` depuis le `main`. Cet appel déclenche la constitution du réseau et la création d'un répertoire des *locations* : l'*addressbook*.

Réseau de communication

Le réseau de communication est constitué de serveurs instanciés sur chaque processus. Ceux-ci répondent au protocole TCP et utilisent des communications par sockets. Une fois le port assigné, le serveur est défini comme actif. L'information `up_srv` est alors envoyée au travers d'un système évènementiel propre à ORWL utilisant la fonction `p99_notifier_set` (voir section 4.5). Le serveur entre alors dans une boucle d'écoute dans laquelle il gérera les demandes de connexions entrantes.

Pour synchroniser l'ensemble des serveurs, une barrière est posée après le lancement du thread serveur dans le code principal. Elle sera ouverte à la réception de l'évènement `up_srv` et permettra de passer à l'étape suivante : l'*addressbook* (ab).

Addressbook

L'ab est un fichier texte répertoriant les adresses propres à toutes les *locations* de l'application. Le fichier est constitué à partir d'un assemblage d'ab locaux où chaque serveur recense les *locations* dont il est responsable. La figure 4.3 montre le mécanisme global d'échange, de concaténation puis de diffusion d'un l'ab global.

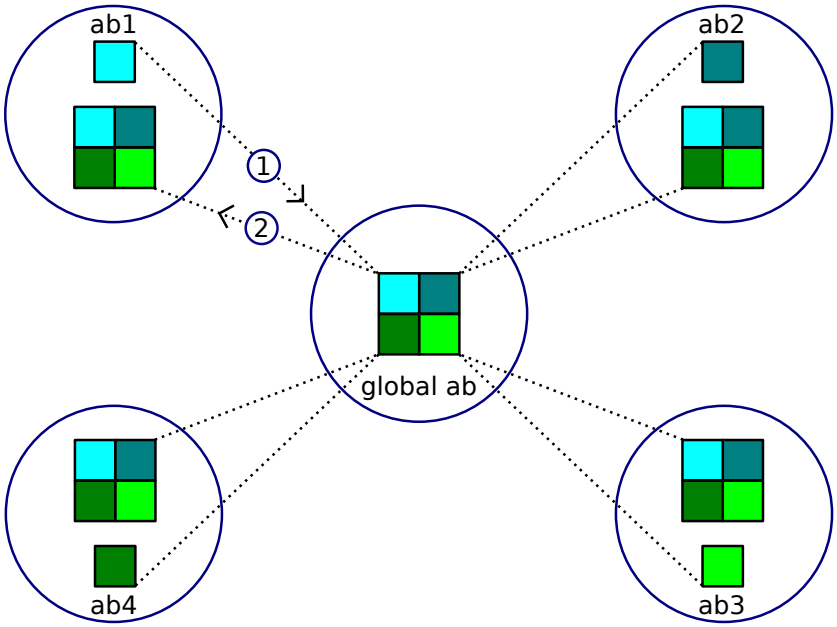


Figure 4.3: Mécanisme de création de l'*addressbook*

La récupération des ab locaux par le script de lancement utilise la fonctionnalité de redirection des flux implémentée par TakTuk. Les nœuds impriment sur leur sortie standard les urls qui sont captées par un fichier collecteur hébergé sur le nœud maître. Le rendu final d'un *addressbook* est du type :

Listing 4.1: Carnet d'adresse des ressources ORWL

```
7       orwl://192.168.1.96:33263/7
6       orwl://192.168.1.96:33263/6
5       orwl://192.168.1.96:33263/5
4       orwl://192.168.1.96:33263/4
3       orwl://192.168.1.96:33263/3
2       orwl://192.168.1.96:33263/2
1       orwl://192.168.1.96:33263/1
0       orwl://192.168.1.96:33263/
15      orwl://192.168.1.96:44103/7
14      orwl://192.168.1.96:44103/6
13      orwl://192.168.1.96:44103/5
12      orwl://192.168.1.96:44103/4
11      orwl://192.168.1.96:44103/3
10      orwl://192.168.1.96:44103/2
9       orwl://192.168.1.96:44103/1
8       orwl://192.168.1.96:44103/
```

Le chiffre en début de ligne représente l'identifiant interne de la *location*. L'url est ensuite composée de l'ip du nœud suivie du numéro de port du serveur puis d'un identifiant de la *location*. Nous remarquons ici deux numéros de port différents : 33263 et 44103. Cela signifie que l'application a été lancée localement sur deux processus. L'absence de 0 sur la dernière ligne n'est pas anodine. Le "/" final indique à la boucle de comptage le dernier élément. Si le nombre d'adresses comptabilisées équivaut au nombre attendu, la collecte s'arrête.
L'envoi du fichier vers les nœuds est déclenché par la ligne de code suivante :

```
broadcast input file [ "while␣[␣-e␣#{LOCKFILE.path}␣]␣;␣do␣
   sleep␣0.001␣;␣done␣;␣cat␣#{COLLECTOR.path}|" ]
```

Cette instruction TakTuk `broadcast input file` diffuse un fichier à un ensemble de destinataires. Le nom du fichier est ici remplacé par une commande système dont la fonction est de tester l'existence du fichier `LOCKFILE`. Tant qu'il est présent, le test est vérifié et rien ne se passe. À sa suppression, lorsque la collecte de l'address book est terminée, le chemin vers fichier COLLECTOR s'imprime dans la commande TakTuk et l'envoi de l'ab démarre.
Côté applicatif, l'ab est géré dans les trois étapes suivantes :

1. affichage sur la sortie standard des adresses locales :
   `o_rwl_write_address_book`.
2. boucle de détection de présence du fichier `addressbook.txt` sur l'hôte :
   `orwl_wait_until_file_is_here`.
3. lecture de l'`addressbook` : `o_rwl_address_book_read`.

L'*addressbook* est transformé lors de la dernière étape en un objet `orwl_address_book` dont la structure est la suivante :

```
struct orwl_address_book {
// nombre global de locations
size_t nl;
// addresses des locations sur les serveurs distants
orwl_endpoint *eps;
// addresses des locations locales au serveur
orwl_mirror * mirrors;
...
};
```

Cet objet est stocké dans l'instance du serveur. Il est accessible à tout endroit du code par un appel à la commande : `orwl_server_get()->ab`.
La phase d'initialisation du système est alors terminée et les processus peuvent lancer la création de leurs tâches.

4.4.5 Exécution

Dans le modèle ORWL, les tâches sont des ensembles d'opérations à gros grains dont le nombre est spécifié au lancement de l'application.

Définition des locations et paramétrage des tâches

La classe principale de l'application définit dans son entête un ensemble de macro nécessaire à l'initialisation d'ORWL. Parmi elles figurent la définition des *locations* utilisées et la définition des tâches de calcul. Le nombre total de tâches de l'application est le produit du nombre de processus par le nombre de tâches internes à un processus.
Elles sont créées au sein de la fonction principale de l'application. Une structure placée en en-tête de la fonction `main` définit le type d'une tâche. Elle expose les arguments qui lui seront transmis. Une fonction d'initialisation dont le nom est composé à partir du nom choisi pour la tâche et du suffixe *_init* doit également être définie. C'est cette fonction qui sera lancée au début de chaque tâche, associant les arguments passés à la tâche aux éléments de sa structure. Un exemple du processus d'implémentation d'une tâche est visible sur le listing 4.2.

Listing 4.2: Définition d'une tâche ORWL

```
ORWL_LOCATIONS_PER_TASK(n_loc, s_loc);
ORWL_LOCATIONS_PER_TASK_INSTANTIATION();
P99_DECLARE_STRUCT(task_obj);
P99_DEFINE_STRUCT(task_obj, char *arg1);

task_obj* task_obj_init(task_obj* task, char *arg1) {
if (!task) return NULL;
*task = (task_obj) { .arg1 = output_name };
return task;
}
void task_obj_destroy(task_obj *task) {
/* empty */
}
P99_DECLARE_DELETE(task_obj);
P99_DEFINE_DELETE(task_obj);
ORWL_DECLARE_TASK(task_obj);
```

La première ligne du listing définit les objets `n_loc` et `s_loc` comme étant des *locations*. La seconde exprime le fait que chaque tâche possède ses propres *locations*. La suite du listing définit la tâche est `task_obj` avec les fonctions :

- `P99_DECLARE_STRUCT` : déclare le type task_obj ;
- `P99_DEFINE_STRUCT` : définit la structure task_obj ;
- `task_obj_destroy` : crée une fonction déclenchée à la fin de la tâche ;
- `P99_DECLARE_DELETE` : crée une fonction pour nettoyer l'espace mémoire après la fin de la tâche ; `P99_DEFINE_DELETE` : rend visible la fonction `inline` définie ci-avant ;
- `ORWL_DECLARE_TASK` : déclare l'ensemble des fonctions nécessaires au thread détaché à l'origine de la tâche (voir section 4.5)

Suite à ces déclarations, les variables `orwl_lt` (nombre de tâches par processus) et `orwl_mytid` (identifiant unique de la tâche) sont créées.

Paramétrage des accès aux ressources

Les accès aux ressources dépendent souvent de la place d'une tâche dans la grille. En effet, cette position détermine le voisinage avec lequel il faudra échanger des données. À

ce titre, chaque tâche doit définir sa propre politique d'accès aux *locations*. Ces dernières ne sont pas manipulables directement, c'est le rôle des *handlers* :

Listing 4.3: Initialisation des handlers

```
orwl_handle2 w_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 r_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_scale(sizeof(float), ORWL_LOCATION(orwl_mytid, s_loc));
orwl_write_insert(&w_hdl, ORWL_LOCATION(orwl_mytid, s_loc),
    0);
orwl_read_insert(&r_hdl, ORWL_LOCATION(orwl_mytid + 1, n_loc)
    , 1);
orwl_schedule();
```

Nous pouvons voir que l'initialisation des *handlers* `w_hdl` et `r_hdl` passent par une macro : `ORWL_HANDLE2_INITIALIZER` (ligne 1-2). Dans notre exemple, le type spécial `orwl_handle2` est choisi car nos `handlers` font partie d'un calcul itératif. Cette structure est composée d'une paire de `handlers` classiques qui vont pouvoir être insérés alternativement dans la file d'attente d'une *location*. Les lignes 4 et 5 présentent la syntaxe des requêtes d'insertion : `orwl_write_insert` et `orwl_read_insert`. Dans le premier cas la demande est en écriture et dans le second cas elle est en lecture seule. Les paramètres de ces fonctions indiquent : l'adresse du *handler*, l'identifiant de la *location* et un ordre de priorité. Dans cet exemple, la tâche accédera d'abord à sa ressource `s_loc` en écriture, puis à la *location* `n_loc` en lecture seule.
Les *locations* en écriture doivent être dimensionnées selon le type de données qu'elles hébergeront. C'est le rôle de la fonction `orwl_scale` (ligne 3) à laquelle nous indiquons qu'un espace de 4 octets est nécessaire pour `s_loc`.
La fonction `orwl_schedule` (ligne 5) traite par défaut toutes les *locations* si aucun paramètre n'est spécifié. Il peut toutefois être nécessaire de procéder au *scheduling* en plusieurs étapes, par exemple lorsqu'une *location* est utilisée dans un thread séparé. Dans ce cas, il est possible d'indiquer à la fonction de prendre en compte seulement un nombre restreint de *locations* à partir d'un indice précis. Il est important de spécifier la quantité attendue car une barrière de synchronisation est calibrée pour bloquer jusqu'à ce qu'un nombre précis d'insertions soit atteint.

Processus d'insertion des requêtes

Pour prendre possession d'une ressource, une tâche doit passer par un objet tiers appelé *handler*. Ce dernier n'interagit pas directement avec la ressource mais peut en obtenir

l'accès en s'inscrivant dans une liste. Les gestionnaires de ces files d'attente sont de type `orwl_wq` (voir listing 4.4.5).

```
struct orwl_wq {
//The mutex used to control the access to the queue
  mtx_t mut;
// The rwlock used to control the access to the data
  p99_rwl rwlock;
// The head of the priority queue
  orwl_wh_ref head;
// The tail of the priority queue
  orwl_wh_ref tail;
// A counter that is increased at each event that this queue
    encounters.
  uint64_t clock;
// The data that is associated with this queue
  orwl_alloc_ref allocr;
// True if the data is not to be released from here.
  bool borrowed;
// True if the data is on hold outside a critical section.
  bool onhold;
};
```

Les *handlers* sont ordonnés selon un principe de liste chaînée. Seuls les premiers (*(head)*) et derniers éléments (*(tail)*) sont référencés. Lors de l'insertion, le nouveau intègre son prédécesseur et l'ex-dernier, s'il existe, son successeur. S'il n'existe pas, le *handler* prend la tête de la liste et acquiert automatiquement le *lock* sur la ressource. Avec ce principe, le *handler* informe lui-même son successeur lorsque sa tâche sort de la section critique.
Le système de *lock* est à deux étages. Un premier objet `mut` protège la file d'attente et un second *rwlock* protège les données.

Avant d'être transmise, une requête est transformée en un groupe (voir schéma 4.4) caractérisé par un ordre de priorité et un type d'accès (inclusif ou exclusif).

Ces groupes sont envoyés à la fois vers un tableau local au serveur de la tâche et vers un tableau du serveur hébergeur de la *location* considérée. L'ordonnancement des accès est ainsi géré au niveau des serveurs où une liste recueille les insertions de ses propres tâches et où une autre liste est construite à partir des demandes d'accès envoyées par des tâches distantes. Les requêtes sont traitées dans des threads dédiés routés en fonction de

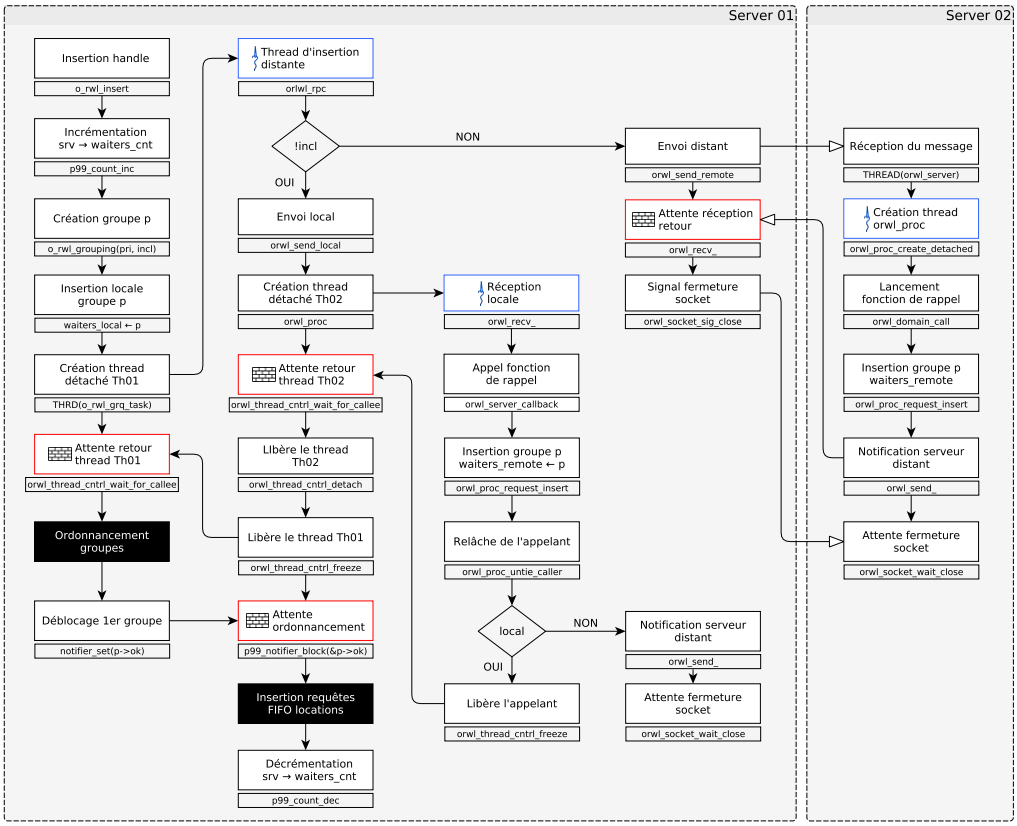


Figure 4.4: Processus d'insertion des groupes de *handlers*

leur caractère local ou distant. Le mécanisme de réception des messages est commun aux deux méthodes. Il permet l'interprétation des données transmises et le déclenchement d'une fonction de rappel. Les messages contiennent les informations suivantes :

- l'adresse du serveur de destination ;
- les identifiants de la ressource ;
- la fonction de rappel ;
- le handle à insérer ;
- le port sur lequel contacter le serveur.

Côté client, en complément du thread d'envoi, un thread de réception est également créé pour capter le retour qui garantira la bonne prise en compte de la demande et le déblocage d'une barrière. Une fois l'insertion de la requête dans le tableau distant effectuée, le thread de réception en informe son processus père qui est alors débloqué. Les tableaux peuvent alors être trié selon le type et l'ordre de priorité (voir schéma 4.5). Les groupes sont alors débloqués un à un, ce qui garantit le bon ordre d'insertion.

Ce mécanisme d'insertion des *handlers* est détaillé sur le schéma 4.6.

Nous pouvons voir, que si la liste est encore vide, une notification est directement envoyée pour faire état l'acquisition de la ressource. Un nouveau groupe peut alors être débloqué. Le transfert de la donnée est déclenché lorsque le lock est acquis. Si la ressource est disponible en local, le handler redirige vers un pointeur sur la donnée. Si la ressource est distante, une copie est envoyée par mode RPC vers le demandeur. Un buffer stocke alors la donnée jusqu'à son utilisation.

Communication entre les tâches

La communication entre les tâches repose sur un principe de Remote Procedure Call (RPC). Le protocole TCP est utilisé pour transmettre à un hôte distant des données et le nom d'une fonction de rappel. Chaque appel est encapsulé dans un thread spécifique selon le principe décrit en section 4.5. Un buffer contenant les données à transmettre est initialisé par la macro `ORWL_BUFFER_INITIALIZER64`. Il en résulte un objet de type `orwl_buffer` composé des éléments suivants :

- le nombre d'arguments de la fonction à appeler ;

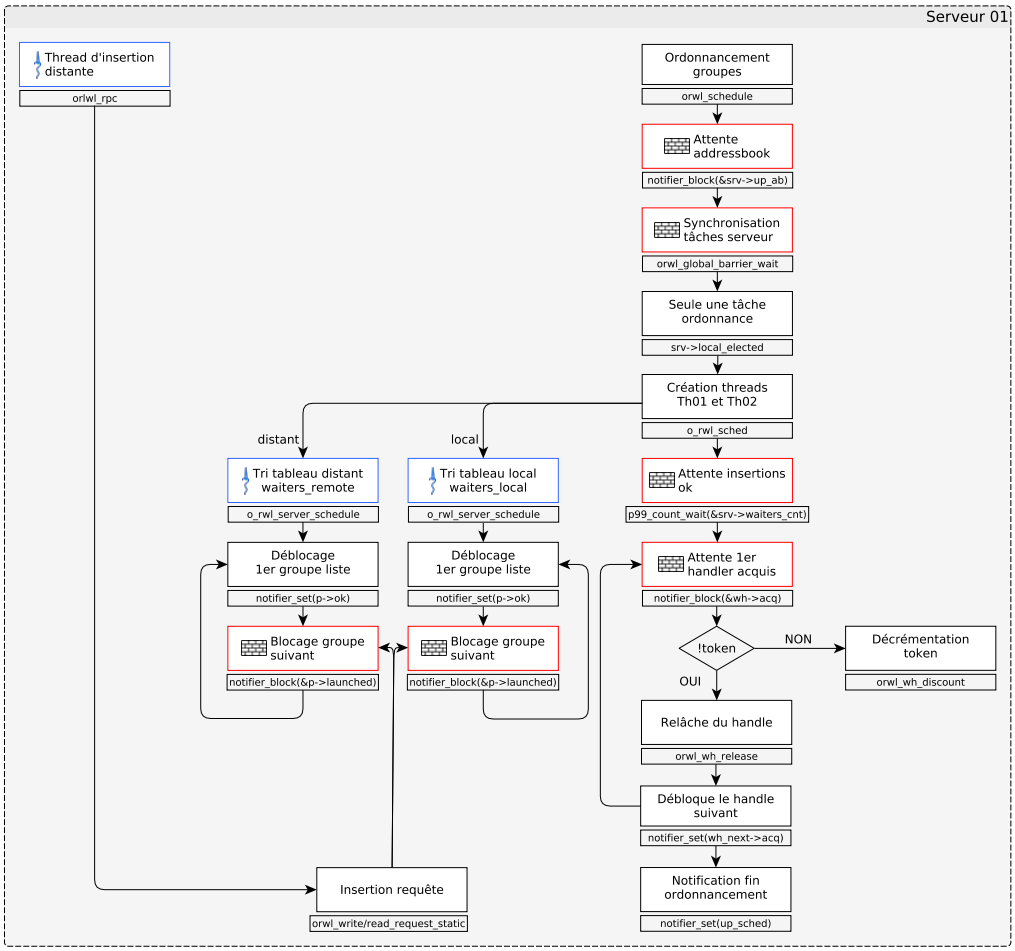


Figure 4.5: Processus d'ordonnancement des groupes de requêtes d'insertion

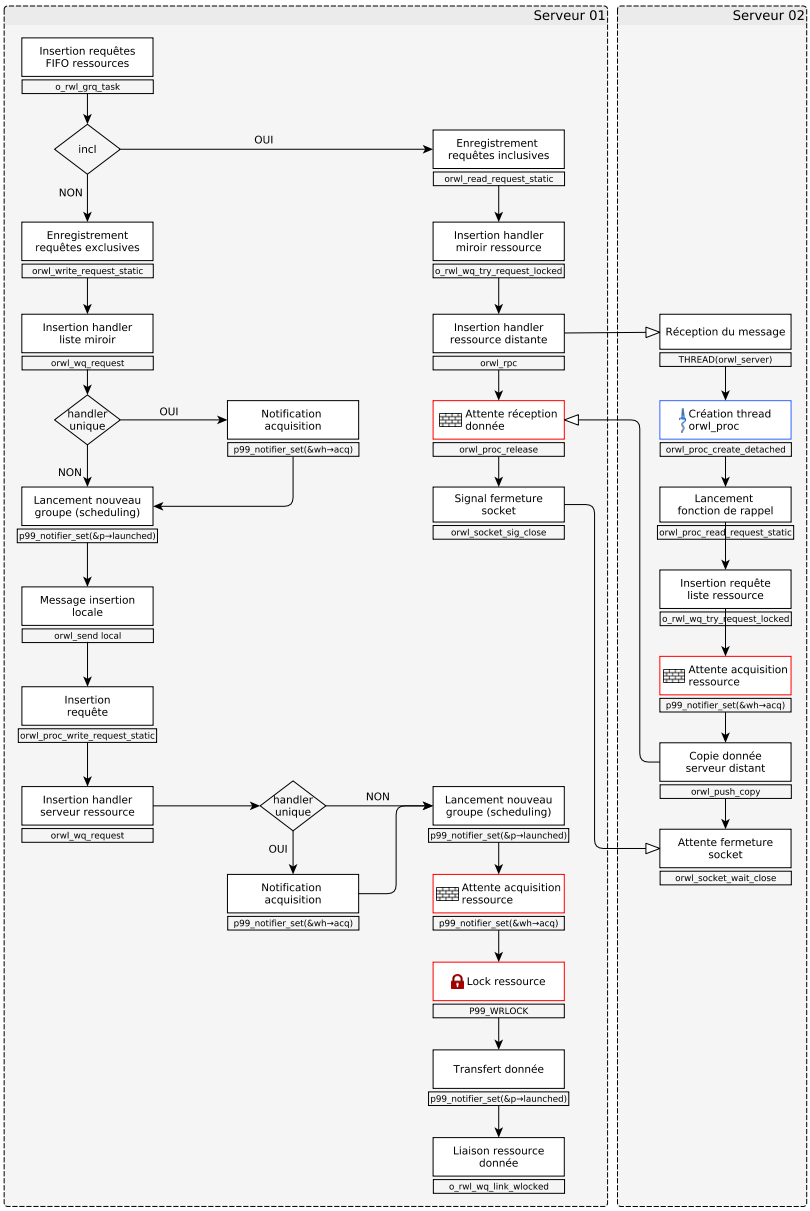


Figure 4.6: Processus d'insertion des handlers dans les listes d'attente de ressources

- un vecteur de `uint64_t` contenant l'identifiant de la fonction de rappel ainsi que les paramètres de cette fonction. Par exemple, pour l'insertion d'un handle, les paramètres sont : `[orwl_proc_request_insert, priority, inclusive]`.

L'envoi de ce buffer est géré par la fonction `o_rwl_rpc` :

```
uint64_t o_rwl_rpc(orwl_server *srv, orwl_endpoint const*
   there, p99_seed *seed, size_t n, orwl_buffer mess[n])
```

Le premier paramètre correspond au serveur d'envoi, le second définit la destination, le troisième est une graine aléatoire, le quatrième annonce le nombre d'éléments dans le buffer, le cinquième représente le contenu du message.

La transmission d'un message est dispatchée par la fonction interne `o_rwl_rpc` selon le type d'envoi. S'il est local, la fonction déclenche directement le mode réception. Pour un envoi distant, les étapes suivantes s'enchaînent :

1. création du socket ;
2. connexion à l'adresse distante ;
3. test d'envoi ;
4. test de réception ;
5. envoi de la taille du message ;
6. envoi des données ;
7. réception de la réponse du serveur distant ;
8. signal de fin de transmission ;
9. fermeture du socket.

Côté serveur, la réception est assurée par une boucle de vérification des sockets de *locations*. Une fois les données réceptionnées, la réception d'un message distant et local suit à nouveau le même processus.

Un nouveau thread détaché est créé et lance la fonction :

```
DEFINE_THREAD(orwl_proc) {
...
if (Arg->fd != -1) {
orwl_recv_(Arg->fd, &Arg->mes[0], Arg->remoteorder);
if (!Arg->srv) P99_THROW(1);
}
orwl_server_callback(Arg);
...
}
```

La condition en ligne 3 joue sur le paramètre `fd` passé à l'initialisation du thread. Dans le cas d'un envoi local sa valeur est de `-1` ce qui évite l'étape de réception du message. Ligne 7, l'appel à `orwl_server_callback` déclenche la fonction de rappel correspondant au code `funcId` (voir listing 4.4).

Listing 4.4: Macro permettant de générer une fonction d'interprétation de message

```
DEFINE_ORWL_PROC_FUNC(orwl_server_callback, uint64_t funcID)
   {
ORWL_PROC_READ(Arg, orwl_server_callback, uint64_t funcID);
orwl_domain_call(ORWL_FTAB(orwl_proc), funcID, Arg);
}
```

L'instruction `ORWL_PROC_READ` permet de récupérer les arguments du message dans un tableau. `orwl_domain_call` lance à distance la fonction de rappel avec les paramètres associés.

Section critique

Une section critique est un bloc encadré par la macro `ORWL_SECTION`. Elle est associée à un *handler*. Lorsque ce dernier arrive en tête de la FIFO de sa *location*, il en obtient l'accès. Les fonctions `orwl_read_map` (accès en lecture) et `orwl_write_map` (accès en écriture) permettent alors d'acquérir un pointeur vers la donnée. Lorsque la tâche quitte la section critique, le *lock* est relâché et le pointeur vers la donnée est invalidé. Les autres tâches en attente peuvent alors y accéder. La figure 4.7 présente comment deux sections critiques peuvent s'imbriquer pour un échange bilatéral.

Ce type d'imbrication est nécessaire dans un contexte itératif. Nous voyons sur cet

Figure 4.7

exemple deux tâches en attente du résultat de l'autre (SC1 et SC1'). Une fois l'information récupérée, ses données peuvent être mises à jour et sauvegardées dans une *location*.

4.4.6 Dernière étape

La fin de l'application intervient lorsque toutes les tâches sont terminées. Progressivement, l'information de fin est propagée jusqu'à l'arrêt effectif. Le script orwlrun prend alors en charge le rapatriement des données vers le nœud maître.

Arrêt progressif des tâches

Lorsqu'une tâche s'achève, la fonction `orwl_stop_task` déclenche une routine de nettoyage des objets créés. Cette fonction joue aussi un rôle de barrière dont le but est de maintenir la tâche active pour d'éventuels accès à ses *locations* jusqu'à une terminaison globale.
Une fois les tâches arrêtées, le script `orwlrun` reprend la main. Une commande TakTuk lance alors la récupération et la concaténation des données exportées par chaque tâche.

Fichiers générés par l'application

Chaque processus ORWL génère des fichiers de sortie dans un répertoire temporaire local paramétré dans la variable d'environnement `ORWL_TMPDIR`. À la fin de l'application, ces fichiers sont rassemblés dans un dossier unifié sur le nœud maître avec pour arborescence :

```
orwl_outputYYYYMMDD-ORWL_SECRET-4s9wus
├── logs/
│   └── orwl_outputYYYYMMDD-4999-4s9wus-localhost.tar.bz2/
│       ├── orwl_outputYYYYMMDD-4999-4s9wus-localhost-1-1
│       ├── orwl_outputYYYYMMDD-4999-4s9wus-localhost-1-2
│       ├── orwl_outputYYYYMMDD-4999-4s9wus-localhost-1-platform.txt
│       ├── orwl_outputYYYYMMDD-4999-4s9wus-localhost-2-1
│       ├── orwl_outputYYYYMMDD-4999-4s9wus-localhost-2-2
│       └── orwl_outputYYYYMMDD-4999-4s9wus-localhost-2-platform.txt
├── addressbook.txt
├── script.rb
└── timing.csv
```

À l'intérieur du dossier log, les fichiers sont compressés dans une archive `.tar.bz2`. Les numéros de fin correspondent à l'identifiant du processus et au numéro de canal dont est issu le fichier : 1 pour *stdout*, 2 pour *stderr*.
Le fichier `platform.txt` génère automatiquement la configuration du système au moment de l'exécution. Les informations reprennent les paramètres de la ligne de commande et le résultat des commandes :

- `uname -a` : numéro de version du noyau Linux et nom de la distribution ;
- `ulimit -aS` : taille des fichiers *coredump*, taille mémoire, nombre maximum de processus utilisateur, etc... ;
- `cat /proc/cpuinfo` : information sur le processeur (nombre de coeurs, cadence, taille du cache...).

Cet affichage est géré par le script `script.rb` présent dans le dossier. Ce script est la copie du fichier `orwlrun_rb` de la librairie ORWL.
Le dernier fichier *timing.csv* est généré à partir de l'analyse de données collectées par ORWL lors de l'exécution. Pour activer cette fonctionnalité, la librairie ORWL doit être compilée avec la variable d'environnement `ORWL_TIMING=1`. Un exemple des données récoltées est visible sur l'illustration 4.8.

Figure 4.8: Rapport statistique ORWL

Ce rapport donne le pourcentage d'utilisation des fonctions de l'application. Sur cet exemple, nous voyons que la fonction `orwl_write_map` (ligne 11) utilisée pour écrire une donnée dans une *location* a été appelée 1656 fois pour un temps total de 0.002365782 secondes. Les colonnes suivantes indiquent le temps moyen passé dans la fonction ou encore la déviation standard.

4.5 P99 : la boite à outils d'ORWL

La construction d'ORWL est intimement liée à l'utilisation de la librairie p99[3]. Celle-ci fournit une véritable émulation de la norme C11 au cas où le compilateur n'est pas conforme à la norme C11. D'un point de vue structurel, P99 est réparti en différentes classes de type *header* qu'il convient d'ajouter au chemin de compilation d'ORWL. Sa syntaxe repose sur l'utilisation de macros et de fonctions `inline`. Ce modèle permet une plus grande modularité de l'écriture des fonctions et la réduction de leur coût d'appel [32]. P99 automatise un certain nombre de fonctionnalités récurrentes dont nous allons détailler les plus importantes pour bien comprendre la logique orwlienne.

Création d'un nouvel objet La macro `P99_NEW` crée un objet à partir d'un type passé en paramètre et suivi d'éventuels arguments. Il convient d'avoir préalablement défini une fonction d'initialisation dont l'appel est assuré par la macro `P00_NEW_ARGS` :

```
#define P00_NEW_ARGS(T, ...) P99_PASTE2(T, _init)(P99_MALLOC(
    T), __VA_ARGS__)
```

[3] http://p99.gforge.inria.fr/p99-html/

Nous pouvons voir ici le caractère hautement dynamique de la syntaxe. Rien n'apparaît "en dur" mais fait référence aux arguments. Ici, la fonction d'initialisation est appelée avec pour premier argument un pointeur résultant de l'allocation directe d'un objet de type T.

Synchronisation La synchronisation entre les différents threads ORWL est assurée par un mécanisme de barrière dont la levée est dévolue à la mise à jour d'un objet de type futex [38] (Fast User space muTex). Pour rappel, les futex ont été introduits dans le kernel Linux en 2008 pour offrir une solution de remplacement aux sémaphores. Dans les nouveaux environnements multi-threads, ces derniers devenaient de véritables goulots d'étranglement du fait de leurs appels systématiques au noyau en cas d'accès aux ressources partagées. Les futex se limitent aux cas conflictuels. Ils sont aussi à la base d'objets plus complexes tels que les mutex ou les rwlocks.
La mise en place de la barrière est assurée par la fonction `p99_notifier_block`. L'avancée du thread ou de la tâche devient alors dépendante du statut du futex passé en paramètre. La modification de ce dernier s'opère par la fonction `p99_notifier_set(notifier_flag , value)`. et peut survenir à quelconque point du code.

Déclaration d'arguments par défaut P99 offre la possibilité d'assigner des arguments par défaut à une fonction. L'exemple suivant illustre comment la fonction `orwl_mirror_connect` est captée tout d'abord par une macro du même nom :

```
#define orwl_mirror_connect(...) P99_CALL_DEFARG(
   orwl_mirror_connect, 4, __VA_ARGS__)
#define orwl_mirror_connect_defarg_2() -INT64_C(1)
#define orwl_mirror_connect_defarg_3() orwl_server_get()
void orwl_mirror_connect(orwl_mirror *rq, orwl_endpoint endp,
    int64_t index, orwl_server* srv);
```

L'appel à fonction est alors encapsulé dans une fonction de vérification des arguments reçus. Si tous sont renseignés par l'utilisateur, rien n'est modifié, en revanche si un argument est manquant, ceux par défaut sont injectés dans la liste `__VA_ARGS__`. L'assignation d'un argument se fait par une fonction du type `orwl_mirror_connect_defarg_2` où le nombre de fin symbolise la place de l'argument.

Création de threads P9 permet la création simplifiée de thread grâce à la macro `ORWL_DECLARE_THREAD(foo)` qui instancie automatiquement la liste de fonctions nécessaires à la vie du thread :

- `foo_start(T *const Arg)` : fonction de rappel pour chaque thread de type `T`.
- `foo_join(thrd_t id)` : fonction de *join* pour un thread lancé en détaché avec `id` pour identifiant.
- `foo_create_joinable(T* arg, thrd_t *i)` : lance un thread joignable avec l'identifiant `id`. L'argument `arg` est celui passé à la fonction de rappel `foo_start`.
- `foo_create_detached(T* arg)` : lance un thread détaché de type T.
- `foo_launch(T* arg, orwl_thread_cntrl* det)` : lance un thread de type T et attend jusqu'à ce qu'il soit détaché.
- `foo_create(T* arg, thrd_t *id)` : lance un thread de type T avec un identifiant de jonction `id`.

La fonction à exécuter par le thread doit alors figurer dans le bloc de la macro `ORWL_DEFINE_THREAD` qui ajoute les lignes de codes suivantes :

```
#define O_RWL_DEFINE_THREAD_1(T)
                                                \
P99_INSTANTIATE(int, P99_PASTID2(T, join), thrd_t);
                        \
P99_INSTANTIATE(int, P99_PASTID2(T, create), T*, thrd_t*);
                \
P99_INSTANTIATE(int, P99_PASTID2(T, create_joinable), T*,
   thrd_t*);    \
P99_INSTANTIATE(int, P99_PASTID2(T, create_detached), T*);
                \
P99_INSTANTIATE(int, P99_PASTID2(T, operation), T*, size_t);
              \
P99_INSTANTIATE(int, P99_PASTID2(T, launch), T*,
   orwl_thread_cntrl*); \
P99_INSTANTIATE(int, P99_PASTID2(T, start_joinable), void*);
              \
P99_INSTANTIATE(int, P99_PASTID2(T, start_detached), void*);
              \
P99_INSTANTIATE(int, P99_PASTID2(T, user_start_detached),
   void*);      \
void P99_PASTID2(T, start)(T *const Arg)
```

Les fonctions définies lors de la déclaration ne sont pas présentes dans la table de symboles. C'est pourquoi elles sont instanciées explicitement avec la macro `P99_INSTANTIATE` . La dernière ligne du listing est la fonction de démarrage recevant le code utilisateur.

P99 fournit également les outils d'interaction avec un thread détaché. Le listing suivant montre comment créer et faire appel à un thread de type `orwl_proc`.

```
orwl_thread_cntrl* det =  P99_NEW(orwl_thread_cntrl);
orwl_proc *sock = P99_NEW(orwl_proc, ...);
P99_THROW_CALL_THRD(orwl_proc_launch, sock, det);
```

L'objet de contrôle du thread `det` (ligne 1) contient différents états du thread détaché qui permettront à l'appelant (*caller*) d'interagir avec l'appelé (*callee*) comme le montre le schéma 4.9.

La donnée partagée D joue un rôle de buffer et de temporisateur. Le thread relance l'exécution de son créateur une fois sa tâche finie.

4.6 Conclusion du chapitre

ORWL adopte une approche centrée sur les données partagées pour modéliser les problématiques induites par le parallélisme. Une seule et même implémentation de l'application est exécutable localement ou à distance. Les communications inter-tâches ou inter-processus sont gérées par le système. Une fois les modalités d'accès aux ressources définies, l'utilisateur peut se concentrer sur l'aspect algorithmique.
La complexité interne des mécanismes mis en jeu par ORWL est nécessaire car elle est soumise à des contraintes mixtes de réseau et d'accès concurrents.
L'implémentation de notre étude de cas présentée dans le chapitre suivant va permettre de tester à la fois la flexibilité du modèle et sa capacité à gérer de grands volumes de données.

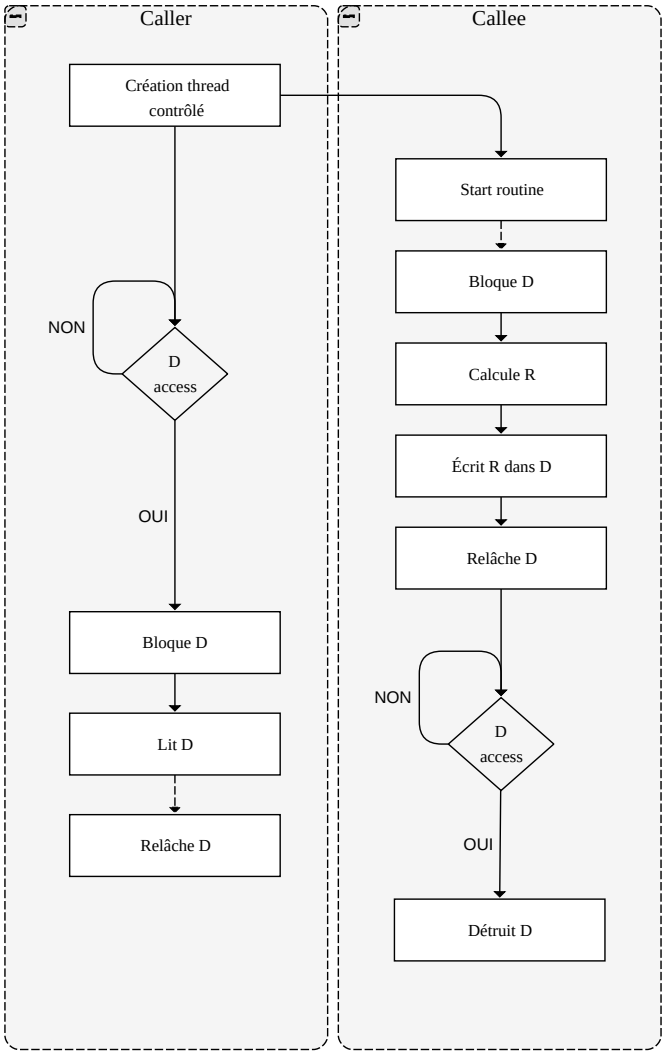


Figure 4.9: Processus de contrôle d'un thread à travers une donnée partagée D

5 Construction d'un environnement de traitement d'image basé sur ORWL

La parallélisation de l'application MPP_ORWL a tout d'abord impliqué l'appropriation complètement des deux concepts qu'elle allie. Une étape de pré-étude a été nécessaire pour démontrer l'éligibilité de l'algorithme MPP à la parallélisation avec ORWL. Une fois cette démonstration faite, les données partagée ont été modélisées et un workflow de traitement plus global a été conçu avant d'être réalisé. Durant toutes ces étapes, la préservation de l'algorithme initial a été au cœur de toute les considérations pour garantir la validité de notre travail vis à vis du modèle initial MPP.
L'implémentation de cette solution a été jalonnée de nombreux problèmes dont les résolutions seront présentées en fin de chapitre.

5.1 Pré-étude

Une étude préalable d'éligibilité d'ORWL à la parallélisation a été menée. Sur la base de cette démonstration, l'optimisation de l'algorithme séquentiel a constitué le point de départ de l'implémentation.

Profilage

L'outil Google gperftool [1] permet d'obtenir sous forme de graphe le taux d'occupation des fonctions. Dans une version non-intrusive du code, le profilage de l'application est obtenu en la compilant avec l'option `-lprofiler`. Il est également nécessaire de définir la

[1] https://github.com/gperftools/gperftools

variable `CPUPROFILE` qui précise le fichier de sortie. L'outil pprof [2] dispose d'une option permettant d'exporter le profil généré vers le format callgrind [3] :

```
google-pprof --callgrind esompp_openmp profile > profile.
    callgrind
```

L'application kcachegrind [4] permet ensuite d'en visualiser le résultat.
Le profilage de l'application séquentielle MPP est visible sous forme de graphe 5.1.

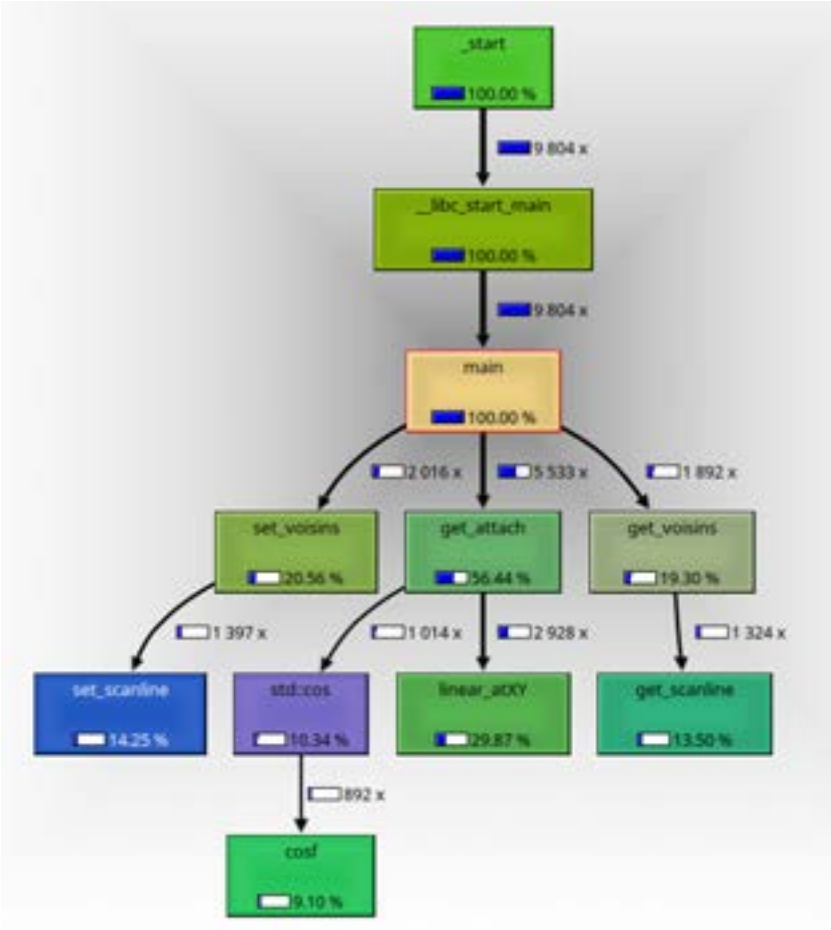


Figure 5.1: Graphe du taux d'occupation des fonctions de l'application séquentielle MPP

Le taux d'occupation de chaque fonction est représenté par une barre de progression

[2] https://github.com/google/pprof

[3] http://valgrind.org/docs/manual/cl-manual.html

[4] http://kcachegrind.sourceforge.net/html/Home.html

bleue symbolisant le pourcentage global de temps. Les arcs du graphe montrent le nombre de fois où la fonction est appelée ainsi que le pourcentage d'appels par rapport aux enfants de la fonction appelante. Il peut être ainsi établi que la fonction `get_attach` occupe 56% du temps global de calcul. Elle est appelée 5533 fois pour le calcul d'une image de dimension 1024×1024. La fonction appelle à son tour `linear_atXY` et `cos`. Une fois localisés ces points chauds, une étude algorithmique permet d'évaluer son degré de parallélisation.

Éligibilité à la parallélisation

La fonction `get_attach` est en charge de calculer le taux d'attache aux données à la création de chaque ellipse.

Algorithm 2: Calcul d'attache aux données

$nbAngle = 100$
$nbEllipses = 3$
for $i = 0$ to $nAngle$ **do**
  for $j = 0$ to $nbEllipses$ **do**
    $\theta = (i * 2 * \pi)/nAngle$
    $x \leftarrow abscisse\ en\ \theta\ sur\ ellipses[j]$
    $y \leftarrow ordonnee\ en\ \theta\ sur\ ellipses[j]$
    $val \leftarrow linear\_atXY(IMAGE, x, y, w, h)$;
    $mu[j] += val$
    $sigma[j] += val * val$
  end for
end for
$Attach \leftarrow Bhattacharyya(mu, sigma)$

Trois ellipses sont nécessaires au calcul de la distance de Bhattacharyya (voir section 2.3.2), ce qui multiplie par trois les calculs de valeurs moyennes et d'écarts types. Une ellipse est discrétisée par 100 pixels dont les coordonnées sont déduites de l'équation de l'ellipse. Ces valeurs réelles sont converties en nombres entiers avec la fonction de linéarisation : `linear_atXY`.

Le calcul d'attache aux données d'une ellipse s'obtient à partir de valeurs collectées sur son pourtour. Le calcul d'une ellipse est donc dépendant de l'ensemble des pixels qui la compose.
Une telle dépendance d'ordre spatial est modélisable dans ORWL par des *locations*.

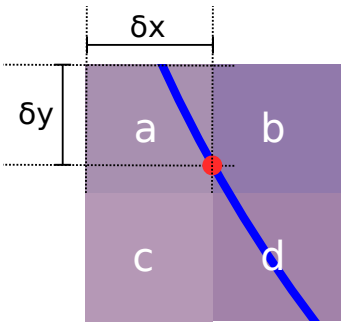


Figure 5.2: Représentation d'un point de l'ellipse discrétisé sur l'image

Analysons plus en détail la fonction `linear_atXY` :

```
real linear_atXY(...) {
  real nfx = fx < 0 ? 0 : (fx > width -1 ? width - 1 : fx),
  real nfy = fy < 0 ? 0 : (fy > height -1 ? height - 1 : fy);
  const unsigned int
  x = (unsigned int)nfx,
  y = (unsigned int)nfy;
  const real
  dx = nfx - x,
  dy = nfy - y;
  const unsigned int
  nx = dx > 0 ? x + 1 : x,
  ny = dy > 0 ? y + 1 : y;
  const unsigned char
  Icc = IMG(x,y),    Inc = IMG(nx,y),
  Icn = IMG(x,ny),   Inn = IMG(nx,ny);
  return (real)Icc + dx*(Inc-Icc +
         dy*(Icc+Inn-Icn-Inc)) +
         dy*(Icn-Icc);
}
```

Les variables `width` et `height` représentent les dimensions de l'image. `fx` et `fy` sont les coordonnées réelles obtenues à partir de l'équation de l'ellipse. Les tests ternaires en début de fonction vérifient que les points appartiennent aux limites de l'image. La suite de la fonction applique une linéarisation du point passé en paramètre :

Avec $IMAGE[n]$ la couleur du pixel n, δx la distance au pixel le plus à gauche et δy la distance au pixel le plus en haut (voir illustration 5.2). L'interpolation s'effectue en deux

Algorithm 3: Calcul de linéarisation

$TopLinearX \leftarrow IMAGE[a] + \delta x(IMAGE[b] - IMAGE[a])$
$BottomLinearX \leftarrow IMAGE[c] + \delta x(IMAGE[d] - IMAGE[c])$
$LinearXY \leftarrow TopLinearX + \delta y(BottomLinearX - TopLinearX)$

temps, horizontalement puis verticalement, en prenant la proportion de valeur d'un pixel en fonction de la position réelle du point.
Cette dépendance spatiale des données implique la création d'une zone de partage entre tâches voisines. Ces données partagées sont l'essence même du modèle ORWL qui les place au centre de son implémentation. ?

Optimisation du code séquentiel

L'optimisation du code séquentiel est une étape primordiale car elle permet de bénéficier d'accélération sans même paralléliser. La première de ces opérations a consisté à transposer le langage C++ vers le langage C pour des raisons de performance, et pour être en adéquation avec ORWL.
Les étapes de cette transformation ont été :

1. l'externalisation des méthodes C++ de manipulation de l'image ;
2. la modification du principe de stockage des ellipses, ce qui a réduit de moitié la mémoire allouée ;
3. la répartition du code partagé dans des fichiers headers ;
4. l'optimisation des types de variables.

Optimisation algorithmique La méthode de stockage des ellipses utilisait initialement deux tableaux par attribut (voir x2 dans liste page 24). Ce système permettait de garder alternativement les ellipses dans l'un ou l'autre selon la parité de l'itération. Pour éviter cette surutilisation de la mémoire, les attributs de l'ellipse ont été rassemblés en une structure :

```
struct elps_str {
        size_t x;
        size_t y;
        float Ud; // data fidelity
        float ax; // big axis
        float bx; // small axis
        float angle;
        float cs; // cos
        float sn; // sin
        // Data used for ellipse coordinates
        float a;
        float b;
        float c;
        int ymin;
        int ymax;
        // Determine if an ellipse is good
        bool good;
        // The iteration number of ellipse creation
        size_t it;
};
```

Un tableau unique (`elps_ptr`) stocke en même temps nouvelles et anciennes ellipses. En fin d'itération, une boucle est utilisée pour copier en début de tableau les ellipses survivantes (`ellipse.good=1`).

```
for (size_t i = 0 ; i < n_points_bak ; i++) {
        if (elps_ptr[i].good) {
                elps_ptr[n_points++] = elps_ptr[i];
                elps_ptr[i].good = 0;
        }
}
```

L'attribut `good` est réinitialisé car chaque ellipse entrera à nouveau en compétition au tour suivant.
Grâce à cette modification, l'empreinte mémoire de l'application se retrouve pratiquement réduite de moitié.

Externalisation des méthodes C++ Bien qu'écrite en C++, la librairie CImg (voir section 5.7.2) a été conservée car elle possède des fonctionnalités fort utiles comme : le chargement d'image, l'extraction de zones d'image ou encore le traçage de

formes géométriques.
La communication entre les deux langages est permise par la fonctionnalité C++ `extern`. Deux exécutables l'un en C et l'autre en C++ communiquent autour d'un fichier *header* commun :

```
#ifdef __cplusplus
extern "C" {
#endif
void interfaceLoadImg(img_str *str, char *name);
#ifdef __cplusplus
}
#endif
```

Du côté de l'application C++, la fonction d'interfaçage lance ici le chargement d'une image par la librairie CImg :

```
void interfaceLoadImg(img_str *str, char *name) {
        ImageUtils::loadImg(str, name);
}
```

Cette approche a été abandonnée au profit du workflow actuel qui sépare les aspects chargement, découpage et calcul.

5.2 Workflow MPP_ORWL

Un workflow spécifique au traitement d'image a été intégré de manière non intrusive à celui initialement défini par le script de lancement `orwlrun` (voir section 4.4.3).
Deux phases ont été ajoutées, l'une de pré-script intervenant avant l'exécution de l'application, l'autre de post-script intervenant une fois l'exécution terminée (voir schéma 5.3).

5.2.1 Pré-script

Le pré-script s'insère entre la définition des variables ORWL et le lancement effectif de l'application sur les nœuds. Son déclenchement dépend de la bonne définition de la variable d'environnement `ORWL_PRESCRIPT`.
Il remplit les rôles suivants :

1. l'interprétation des paramètres applicatifs. La classe Ruby OptionParser est utilisée pour gérer les erreurs ou les avertissements.

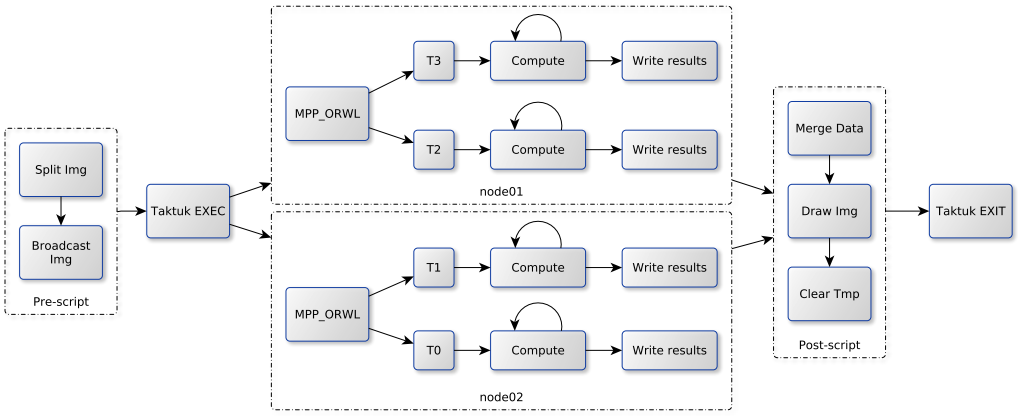


Figure 5.3: Workflow de traitement d'image intégré à ORWL

2. le découpage de l'image (voir section 5.4).
3. la transmission des images par une commande TakTuk.
4. la personnalisation des arguments en fonction du rang du processus attribué par TakTuk (voir variable `$TAKTUK_RANK`).

Découpage de l'image

La découpe de l'image est gérée depuis le nœud principal où les tuiles sont enregistrées dans un répertoire temporaire. Les dimensions de la grille de découpe sont calculées à partir du nombre de processus et de tâches définis par l'utilisateur. À partir de cette grille, les images à destination de chaque tâche sont découpées. C'est à ce moment précis que les cellules fantômes doivent être intégrées.
L'algorithme 4 s'inspire du calcul de PGCD d'Euclide pour déterminer le meilleur ratio entre le nombre de lignes et de colonnes. Une prépondérance est donnée à la largeur afin de favoriser l'alignement des données.

Algorithm 4: Dimensionnement d'une grille en fonction du nombre de tâches

Input: *nTask*
Output: grille de a lignes et b colonnes
  if checkPrimeNumber(nTask) **then**
    print Attention : nTask est un nombre premier
  end if
  $a \leftarrow 0$, $b \leftarrow 0$, $rem \leftarrow 0$
  for $i := 1$ **to** *nTask* **do**
    $rem \leftarrow nTask \mod i$
    if $rem \neq 0$ **then**
      $a \leftarrow i$
      $b \leftarrow nTask \,/\, a$
      if $a \geq b$ **then**
        return $\{a, b\}$
      end if
    end if
  end for

Listing 5.1: Algorithme de découpe de l'image

```
for (size_t j = 0; j < grid.h; j++) {
  for (size_t i = 0; i < grid.w; i++) {
    size_t ind = j * grid.w + i;
    rank = ind / nb_tasks;
    task = ind % nb_tasks;
    borderBits borders = get_borders(ind, grid.w, grid.h);
    size_t x0 = i * w;
    size_t y0 = j * h;
    size_t x1 = x0 + w - 1;
    size_t y1 = y0 + h - 1;
    if(is_border(borders, border0M)) x0 -= r_max; // West
    if(is_border(borders, border0P)) x1 += r_max; // East
    if(is_border(borders, borderP0)) y0 -= r_max; // North
    if(is_border(borders, borderM0)) y1 += r_max; // South
    CImg<unsigned char> tile = img.get_crop(x0, y0, x1, y1);
  }
}
```

Notons que la fonction `checkPrimeNumber` (ligne 3) vérifie que le nombre passé en paramètre n'est pas un nombre premier pour éviter que la grille ne soit unidimensionnelle. Une fois les dimensions établies, l'exécutable procède à l'extraction des images (voir listing 5.1). C'est à ce moment que des bandes de données supplémentaires sont ajoutées

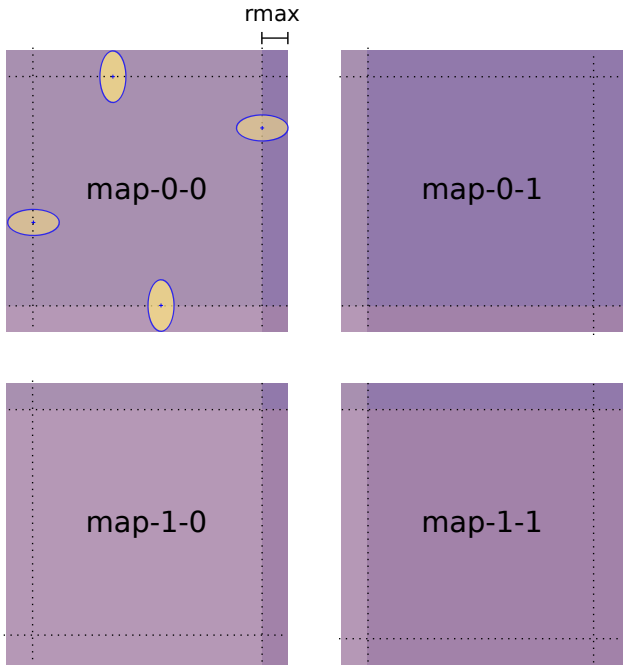


Figure 5.4: Image découpée en quatre avec zones de recouvrement

grâce aux tests de frontières `is_border`.

Les cellules fantômes

L'image 5.4 illustre la distribution d'une image vers quatre tâches. Pour ne pas déborder de l'image `map-0-0`, le centre des ellipses en jaune doit être contenu à l'intérieur d'un périmètre défini par le demi-grand axe `rmax`. Pour ne pas ignorer cette bande périphérique de données, les images empruntent à leurs voisines une bande de données : les cellules fantômes.

La largeur exacte de la bande n'est pas exactement la valeur `rmax`. Il faut en effet considérer le fait que le calcul d'attache aux données s'effectue à partir de trois ellipses (voir illustration 2.2). C'est donc le radius de celle la plus externe qui est à considérer :

$$r_{max} = \begin{array}{l} r * (1 + MAX\_RADIUS\_FACTOR) \\ + ELLIPSE\_DIST * ELLIPSE\_DIST\_MAX\_TIME \end{array} \tag{5.1}$$

La valeur maximale pouvant être attribuée à une ellipse à sa création est :
`r * (1 + MAX_RADIUS_FACTOR)`. La distance entre l'ellipse interne et la plus externe est : `ELLIPSE_DIST * ELLIPSE_DIST_MAX_TIME`.

Remarquons également sur l'image dans la tuile `map-0-0` que l'ellipse en haut à gauche, bien qu'il n'y ait pas de voisins, doit être à la même distance minimale `rmax` du coin. Cela permet d'appliquer un décalage constant lors du tirage au sort de la position des ellipses.

Détermination des voisins L'ajout des cellules fantômes dépend de la présence d'un voisin qui est fonction de la place de l'image dans la grille.
La technique utilisée pour déterminer le nombre de voisins est empruntée au benchmark *Livermore 23* de la librairie ORWL. Les bords d'une image (nord, sud, est, ouest) sont repérés de la manière suivante :

- `0` : pour la tâche au centre ;
- `P` : pour Plus si un voisin est à l'est ou au nord ;
- `M` : pour Moins si un voisin est à l'ouest ou au sud ;
- `A` : pour All si des voisins sont à la fois à droite et à gauche, ou en haut et en bas.

En une dimension, les valeurs hexadécimales suivantes sont assignées :

```
border0 = 0x0
borderP = 0x1
borderM = 0x2
borderA = border0P | border0M
```

En deux dimensions, le même principe est utilisé avec une première lettre pour l'axe vertical et une seconde pour l'axe horizontal :

```
border00 = 0x0;
border0P = 0x1;
border0M = 0x2;
border0A = border0P | border0M // (0010 | 0001 = 0011);
borderP0 = 0x4;
borderPP = borderP0 | border0P;
borderPM = borderP0 | border0M;
borderPA = border0P | border0A;
borderM0 = 0x8
borderMP = borderM0 | border0P
borderMM = borderM0 | border0M
borderMA = borderM0 | border0A
borderA0 = borderM0 | borderP0
borderAP =  borderM0 | border0P
borderAM =  borderM0 | border0M
borderAA =  borderM0 | border0A
```

Le voisinage d'une image se calcule alors en additionnant le poids de chacun de ces bords. La fonction `get_borders` renvoie ainsi une valeur selon la taille de la grille et la place de l'image dans celle-ci.

```
borderBits get_borders(size_t id, size_t global_cols,
                       size_t global_rows)
{
  size_t row = id / global_cols;
  size_t col = id % global_cols;
  return (borderBits)(
  ((row > 0) ? borderP0 : border00)
  | ((row < (global_rows - 1)) ? borderM0 : border00)
  | ((col > 0) ? border0M :  border00)
  | ((col < (global_cols - 1)) ? border0P : border00));
}
```

L'identifiant `id` est traduit en numéro de ligne et de colonne (ligne 2 et 3). Le résultat est calculé par addition successive de bits en utilisant l'opérateur |. Si la tâche est sur la première ligne, cela signifie qu'il n'y a aucun voisin au Nord, dans le cas contraire il y en a un. Et ainsi de suite...
Une fois le nombre de *borderBits* attribué à une tâche, le test de présence d'un voisin est déduit de l'opération unaire & entre le borderbit et le bit de la bordure considérée.
Une méthode similaire permettra de définir la présence d'une *location* pour recevoir les

cellules fantômes.

Enregistrement des données Une fois la taille de l'image établie, la fonction `get_crop` récupère les données images dans un vecteur de type `unsigned char`. CImg stocke les couleurs de manière désentrelacées : R1R2R3... V1V2V3... B1B2B3. Pour notre détection, la coloration H&E produisant une teinte rouge, seule cette composante présente un intérêt. Dès lors, les n (hauteur x largeur) premières valeurs uniquement sont écrites dans le fichier binaire qui représentera la portion d'image.
L'exécutable utilisé pour la découpe de l'image enregistre conjointement les informations concernant les images dans un fichier nommé Tiles. Cela permet de consigner :

- le chemin vers l'image ;
- la hauteur ;
- la largeur ;
- le nombre de canaux.

Le chemin intègre les variables de rang et de tâche pour permettre le bon routage des fichiers.

Distribution

Une fois les tuiles sauvegardées, leur distribution vers les nœuds est assurée par le code :

```
NBPROC.times do |i|
cmd4 = "taktuk␣-m␣" << $nodes3[i].to_s << "␣"
NBTASKS.times do |j|
file_name = IMG_NAME + "-#{i}-#{j}.bin"
cmd4 << "synchronize␣broadcast␣put␣[␣'#{INPUT_DIR}/#{
   file_name}'␣]␣[␣'#{INPUT_DIR}.$((#{i}␣+␣1))'␣]␣\\;"
end
cmd4 << "synchronize␣wait␣reduce␣target␣all␣\\;"
system cmd4
end
```

Les boucles imbriquées (ligne 1 et 3) permettent de parcourir l'ensemble des tâches en définissant le nom des fichiers à transmettre. La variable `cmd4` concatène les instructions afin d'envoyer une unique commande TakTuk. Le mot clé `synchronize` placé devant

l'instruction `broadcast` (ligne 5) garantit la séquentialité de l'envoi. La ligne 2 montre l'amorce de l'appel TakTuk avec l'option `-m` introduisant la liste des nœuds. La seconde boucle parcourt les tâches et forme le nom des fichiers à envoyer en fonction du rang de processus et du numéro de tâche. L'instruction d'envoi Taktuk est de la forme :

```
broadcast put [fichier source] [destination]
```

Remarquons la présence d'un `+1` pour le répertoire de destination. Cette incrémentation tient au fait que les rangs sont numérotés à partir de 1.

Paramètres d'exécution

L'application MPP_ORWL nécessite un certain nombre de paramètres composés d'éléments ORWL et d'éléments propres à l'application. La ligne de code permettant le lancement de l'application est la suivante :

```
$(ORWL_DIR)/scripts/orwlrun $(OUTPUT_DIR) default local $(
   ORWL_PROCS):$(ORWL_TASKS) $(ORWL_CALLGRIND) $(EXEC) --
   r_min 15 --r_max=25 --input_image $(BASE_DIR)/input/$(
   IMG_NAME) --exec $(EXEC) --cnv $(CNV)
```

Les variables ont pour signification :

- `$(ORWL_DIR)/scripts/orwlrun` : le script de lancement de l'application.
- `$(OUTPUT_DIR)` : le répertoire où seront collectés les éléments générés par l'application.
- default : par défaut le script `orwlrun_sh` est envoyé vers les nœuds pour lancer l'exécution distante.
- local : les calculs ont lieu sur le nœud local.
- `$(ORWL_PROCS):$(ORWL_TASKS)` : nombre de processus : nombre de tâches par processus.
- `$(ORWL_CALLGRIND)` : encapsule l'application dans Valgrind
- `$(EXEC)` : l'exécutable de l'application.
- `--r_min 15` : radius minimum d'une ellipse.
- `--r_max=25` : radius maximum d'une ellipse.

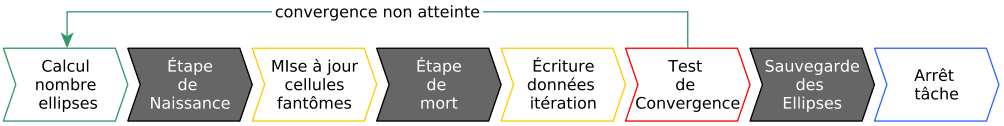


Figure 5.5: Processus applicatif MPP_ORWL

- `--input_image $(BASE_DIR)/input/$(IMG_NAME)` : chemin vers l'image à analyser.
- –cnv `$(CNV)` : nombre de fois où la convergence est atteinte

Les paramètres applicatifs commencent par un double tiret et sont interprétés au *runtime*.

5.2.2 Partie applicative

Une fois les transferts de données terminés, le nœud maître exécute à distance les processus de l'application. Chacun d'eux déploie alors un serveur et transmet les données relatives à ses *locations*. Après une consolidation de l'addressbook, la phase d'initialisation d'ORWL est terminée.
Chaque processus interprète alors le contenu du fichier *tiles* (voir A.7) et charge ses images dans des vecteurs de type `unsigned char`. Les données sont transmises individuellement aux tâches lors de leur création. La suite du calcul MPP s'effectue ensuite au sein de chacune d'entre elles selon le processus représenté sur le graphique 5.5.

Les couleurs correspondent aux éléments suivants :

- fond gris : les étapes héritées de l'algorithme MPP original ;
- contour orange : les actions menées à l'intérieur de sections critiques ;
- contour bleu : les interactions ORWL ;
- contour vert : le calcul du nombre global d'ellipses à créer et sa diffusion à toutes les tâches (voir section 5.3.1).
- contour rouge : le test de convergence appliqué de manière globale (voir section 5.5).

Finalisation de l'application

Une fois la convergence atteinte, les attributs des ellipses retenues sont transformées en fichiers `.ellipses` (voir fonction `writeData` A.12.2).
Les *handlers* utilisés par la tâches sont ensuite déconnectés, la fonction de terminaison `orwl_stop_task` déclenchée, et l'espace mémoire alloué dynamiquement dans l'application est libéré.

5.2.3 Postscript

Une fois que toutes les tâches ont terminées, le script `orwlrun` reprend la main et lance le post-script. Les résultats présents sur chaque nœud sont alors concaténés dans un fichier unique (voir fonction `merge` A.12.1) dans lequel se trouvent : le nombre d'itérations, le nombre d'ellipses trouvées et le temps de calcul, pour chaque tâche.
L'exécutable C++ `draw` finalise les résultats en traçant les ellipses sur l'image de départ à l'aide de la fonction CImg `draw_ellipse`.

5.3 Création des ellipses

5.3.1 Calcul du nombre d'ellipses à créer

L'implémentation originale de l'algorithme MPP simule la répartition de Poisson des ellipses en testant un à un chaque pixel de l'image. Une telle méthode, sur une image de 50 000 $\times$ 50 000 pixels, requiert 95 millions de nombre aléatoires. Afin d'éviter cette dépendance de calcul, une autre stratégie a été employée en estimant à priori le nombre d'ellipses à créer lors de chaque itération.
Afin d'apporter une preuve à cette nouvelle méthode, considérons le lemme suivant puis illustrons-le par deux exemples représentatifs.

Lemme 5.3.1 *Soit n un nombre d'évènements survenus pendant une durée t et répartis selon une loi de Poisson. Une redistribution uniforme aléatoire de ces évènements dans la même durée reste une répartition de Poisson.*

Preuve 5.3.1 *Considérons une urne remplie de N boules numérotées. Nous souhaitons les sélectionner aléatoirement avec une probabilité $B = \frac{1}{3}$. Pour ce faire, nous proposons les deux méthodes suivantes :*
Première méthode : *Nous tirons les boules une à une. Pour chacune d'entre elles, nous*

lançons un dé à 6 faces. Si le résultat est inférieur à 3, nous gardons la boule.
Deuxième méthode : *Nous lançons un dé à 6 faces N fois et nous comptabilisons les m fois où le résultat est inférieur à 3. Nous tirons ensuite simultanément m boules que nous garderons.*
Par la loi forte des grands nombres nous savons que ces deux méthodes sont équivalentes :

$$\lim_{N \to +\infty} \frac{m}{N} = \frac{1}{3} \tag{5.2}$$

Considérant cet état de fait, nous pouvons calculer le nombre total d'ellipses à générer sur l'image lors de chaque itération.

5.3.2 Contrainte de répartition

Le nombre d'ellipses à générer en début d'itération peut être déterminé comme expliqué précédemment. Ce nombre étant global, les ellipses doivent être réparties entre toutes les tâches de l'application. La manière la plus simple serait de répartir ces dernières équitablement. Cependant une telle répartition ne serait pas équitable par rapport aux autres possibilités de répartition. L'illustration 5.6 montre que le nombre de combinaisons possibles est déjà de 35 avec seulement quatre objets à répartir sur quatre cases.

Ce nombre de combinaisons se retrouve par la formule statistique :

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} = \frac{(4+4-1)!}{4!(4-1)!} = 35 \tag{5.3}$$

Dans cet exemple, les tirages sont effectués avec remises. Dans un tel cas, l'ordre des éléments importe peu (comme dans une combinaison classique), et les éléments peuvent être tirés plusieurs fois. Sur l'illustration, cela se traduit par le fait que tous les points violets sont identiques et qu'une case peut en contenir plusieurs.

Pour tenir compte de cette disparité de solutions, deux conditions doivent être réunies :

- le nombre d'ellipses total doit être égal à la somme des ellipses de chaque tâche.
- au sein d'une tâche, les ellipses doivent être choisies de manière indépendantes et aléatoires.

Pour satisfaire la première condition, le calcul du nombre d'ellipses ainsi que leur

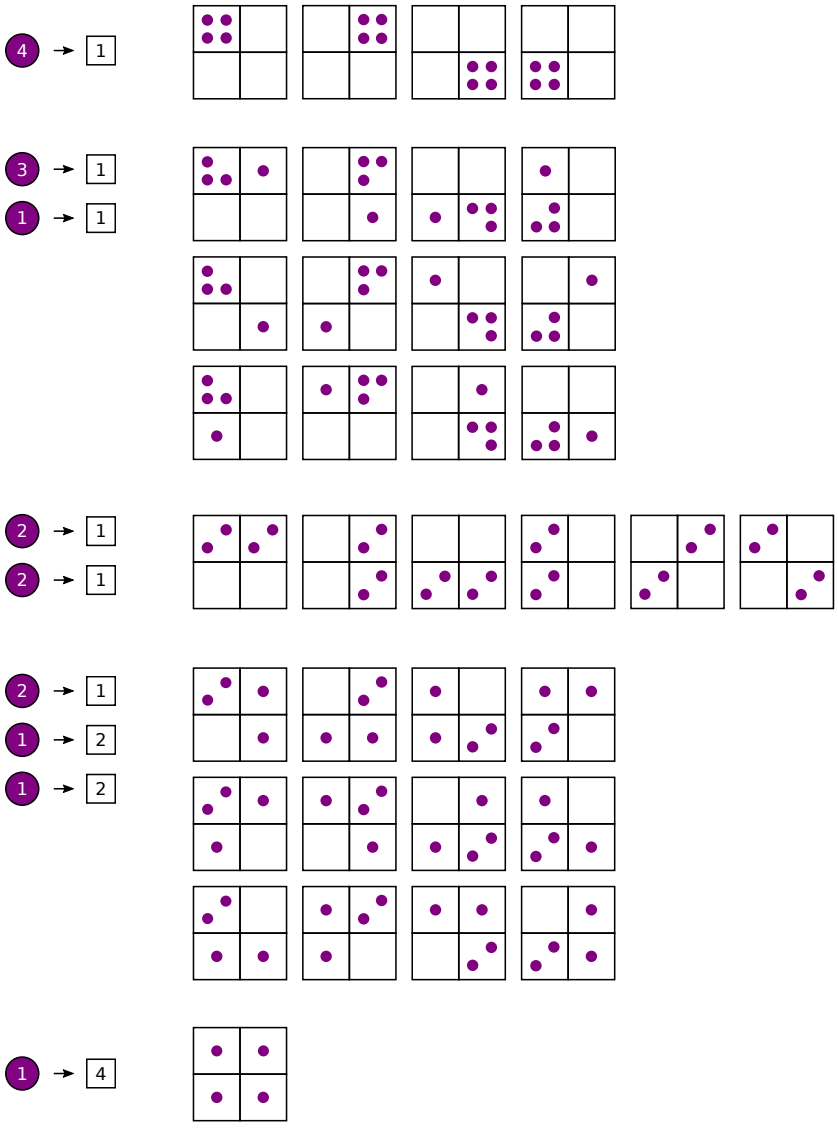


Figure 5.6: Répartitions possibles de 4 ellipses dans 4 tuiles

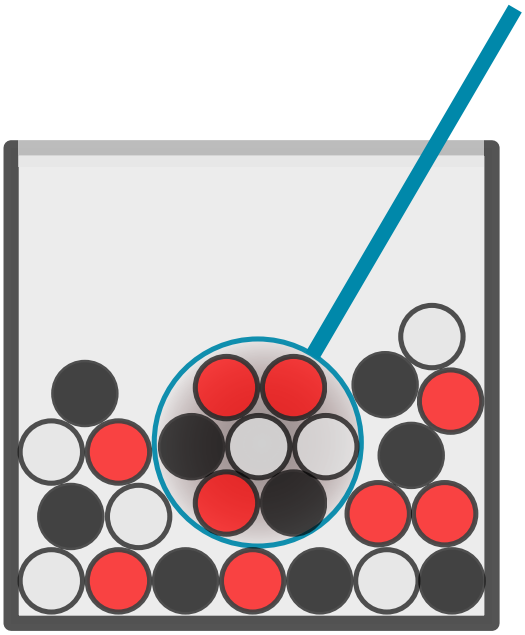


Figure 5.7: L'urne contient neuf boules blanches, neuf noires et neuf rouges. La loi hypergéométrique permet de prédire la probabilité de tirer trois boules rouges sur un échantillon de sept : $P(X = 3) = 19.8\%$

répartition devront être faits par une seule tâche. Pour la seconde condition, envisageons l'utilisation d'une loi hypergéométrique.

5.3.3 Distribution selon une loi hypergéométrique

$$P(X = k) = \frac{C_k^m \times C_{N_m}^{n-k}}{C_N^n} \tag{5.4}$$

En prélevant un échantillon n sur une population N, la loi hypergéométrique permet de prédire la probabilité de tirer k éléments caractéristiques sachant que leur effectif dans la population est m [5] (voir illustration 5.7). L'absence de remise la distingue de la loi binomiale. Lorsqu'un élément est prélevé, il n'est pas remis en jeu, ce qui augmente progressivement la probabilité d'être tiré pour les restants.

La loi hypergéométrique peut être utilisée de manière itérative jusqu'à atteindre un certain quota d'évènements, si par exemple, nous souhaitons prélever 30 boules (noires, blanches ou rouges) parmi un lot de 150.

[5] http://serge.mehl.free.fr/anx/loi_hypergeo.html

Cette loi nous permet d'estimer lors d'un premier tirage la quantité de boule noires : $m_0 = h(30, 150, 50) = 9$.
Un second tirage permet de déterminer le nombre de boules blanches parmi les restantes : $m_1 = h(21, 100, 50) = 12$.
La quantité de boules rouges se déduit des deux premiers tirages. La répartition est alors la suivante : 9 noires, 12 blanches et 9 rouges.

En apparentant les ellipses aux boules et en attribuant une couleur pour chacune des n sous-images, alors (n-1) tirages permettent de répartir les ellipses.
De plus, si le nombre d'objets à distribuer suit une loi de Poisson, alors la répartition a la garantie d'être équitable.

La condition d'unicité de résultat s'obtient en s'assurant que deux ellipses ne reçoivent pas les mêmes coordonnées.

5.3.4 Implémentation de la solution

Au début de chaque nouvelle itération, en préambule au calcul, la tâche T0 est arbitrairement choisie pour calculer le nombre d'ellipses à créer. Ce nombre dépend des facteurs suivants :

- la taille de l'image ;
- le paramètre δ de recuit simulé ;
- la distribution de Poisson.

À partir d'une valeur moyenne passée en paramètre, la fonction `orwl_Poisson` renvoie une valeur aléatoire suivant une distribution de Poisson :

```
size_t nEllipses = orwl_Poisson(*seedP, delta * w * h);
```

Le paramètre `seedP` est une graine permettant de générer un nombre aléatoire selon la méthode proposé par Marsaglia [39].

Une fois la quantité `nEllipses` déterminée, la fonction `orwl_varRandHypergeom` effectue la répartition en prenant pour paramètres : le nombre total de pixels de l'image, le nombre de pixels à sélectionner, le nombre de tâches de l'application et un tableau dans lequel stocker les valeurs :

```
orwl_varRandHypergeom(*seedH, width * height, nEllipses,
    orwl_nt, draw);
```

Le vecteur `draw` aura été préalablement rempli par les tailles individuelles de chaque image.
La tâche T0 doit maintenant diffuser aux autres tâches les données calculées. Dans ce but, ORWL dispose de la méthode `ORWL_BROADCAST` implémentée de la manière suivante :

```
ORWL_LOCS_PER_TASK(brd_loc);
ORWL_BROADCAST_LOCATION(brd_loc);

int main() { ... }

ORWL_DEFINE_TASK(task_obj) {
...
 typedef size_t aVector[orwl_nt];
 ORWL_BROADCAST_DECLARE(aVector, brd_loc, brd_loc);
...
 if (!tid) {
  size_t nEllipses = orwl_Poisson(...);
  for(size_t i = 0; i < orwl_nt; i++) {
   draw[i] = (img_size->width * img_size->height) / orwl_nt;
  }
  orwl_varRandHypergeom(...);
  for (size_t i = 0; i < orwl_nt; i++) {
   brd_loc[i] = draw[i];
  }
  ORWL_BROADCAST(brd_loc);
 } else {
  ORWL_BROADCAST(brd_loc);
 }
}
```

La *location* `brd_loc` est tout d'abord déclarée de manière classique avant de revêtir un caractère particulier avec la macro `ORWL_BROADCAST_LOCATION` qui la déclare commune à toutes les tâches. À l'intérieur de la tâche, une autre macro nommée `ORWL_BROADCAST_DECLARE` (ligne 8) assigne un type particulier à la *location* qui se transforme ainsi en une variable du même nom. La condition `if(!tid)` assigne le calcul à la seule tâche ayant pour identifiant 0. Après avoir calculé la répartition des ellipses dans le vecteur `draw`, celui-ci est copié dans la variable de diffusion `brd_loc`. Aucune section

critique n'est nécessaire du fait que la donnée sera uniquement lue. Notons la double utilisation de la macro `ORWL_BROADCAST` qui joue un rôle de diffusion pour la tâche T0, et un rôle de réception pour les autres.
Les différentes étapes de détection de noyaux peuvent alors s'enchaîner jusqu'à la prochaine itération. Une autre étape importante est également nécessaire lorsque les valeurs d'attache aux données ont été calculées : la synchronisation des cellules fantômes.

5.4 Mise à jour des cellules fantômes

Comme nous avons pu le voir à la section 5.2.1, les cellules fantômes doivent être déterminées dès la phase de découpage de l'image. À l'étape de voisinage, les valeurs des ellipses sont inscrites sur la carte d'attache aux données propres à la tâche. Ces cartes étant le reflet des images analysées, elles comportent une partie des données appartenant aux tâches voisines. Pour maintenir une cohérence entre les différentes cartes, les ellipses appartenant aux cellules fantômes doivent être prises en compte de part et d'autres. C'est ainsi qu'elles sont modélisées sous forme de *locations*.
Chaque tâche doit instancier deux handlers par voisin, l'un pour manipuler sa *location* propre et l'autre pour celle du voisin. Pour déterminer si un voisin est présent, la fonction `is_border` est une fois de plus utilisée (voir section 5.2.1). L'ensemble des handlers est initialisé ainsi :

```
orwl_handle2 ew_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 er_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 ww_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 wr_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 nw_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 nr_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 sw_hdl = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 sr_hdl = ORWL_HANDLE2_INITIALIZER;
```

Il faut alors assigner à chaque handler une *location* soit en lecture soit en écriture, puis dimensionner la *location* afin qu'elle puisse recevoir les données (voir listing 5.2).

Listing 5.2: instanciation des handlers relatifs aux cellules fantômes de la carte d'attache aux données

```
size_t xsize = width * rmax * 2;
size_t ysize = height * rmax * 2;

// North
```

```
if (bNorth) {
 orwl_scale(xsize*4, ORWL_LOC(tid, n_loc));
 orwl_write_insert(&nw_hdl, ORWL_LOC(tid, n_loc), 0);
 orwl_read_insert(&nr_hdl, ORWL_LOC(tid - W, s_loc), 1);
 scale_size += sizeof(float[xsize]);
}
// South
if (bSouth) {
 orwl_scale(xsize*4, ORWL_LOC(tid, s_loc));
 orwl_write_insert(&sw_hdl, ORWL_LOC(tid, s_loc), 0);
 orwl_read_insert(&sr_hdl, ORWL_LOC(tid + W, n_loc), 1);
 scale_size += sizeof(float[xsize]);
}
// East
if (bEast) {
 orwl_scale(ysize*4, ORWL_LOC(tid, e_loc));
 orwl_write_insert(&ew_hdl, ORWL_LOC(tid, e_loc), 0);
 orwl_read_insert(&er_hdl, ORWL_LOC(tid + 1, w_loc), 1);
 scale_size += sizeof(float[ysize]);
}
// West
if (bWest) {
 orwl_scale(ysize*4, ORWL_LOC(tid, w_loc));
 orwl_write_insert(&ww_hdl, ORWL_LOC(tid, w_loc), 0);
 orwl_read_insert(&wr_hdl, ORWL_LOC(tid - 1, e_loc), 1);
 scale_size += sizeof(float[ysize]);
}
```

Les variables `bNorth`, `bSouth`, `bEast`, `bWest` sont définies par la fonction `is_border` et réutilisées dans le reste du code. La fonction `orwl_scale` alloue la mémoire nécessaire à la *location*. `xsize` représente la surface en pixels des cellules fantômes horizontales et `ysize` les verticales. Chaque pixel recevant une donnée de type `float`, il faut multiplier la dimension par quatre. Les fonctions `orwl_write_insert` effectuent les insertions de handlers en écriture et les fonctions `orwl_read_insert` en lecture. L'utilisation de la variable `scale_size` sera décrite à la section 5.6.4.1.

Pour synchroniser les différentes cartes d'attache aux données, les handlers sont mis à jour suivant le principe de *local update / global update* décrit par Saied [26]. Ces étapes sont visibles sur l'illustration 5.8.

Une fois que les calculs locaux d'attache aux données écrits sur la carte, les bords de

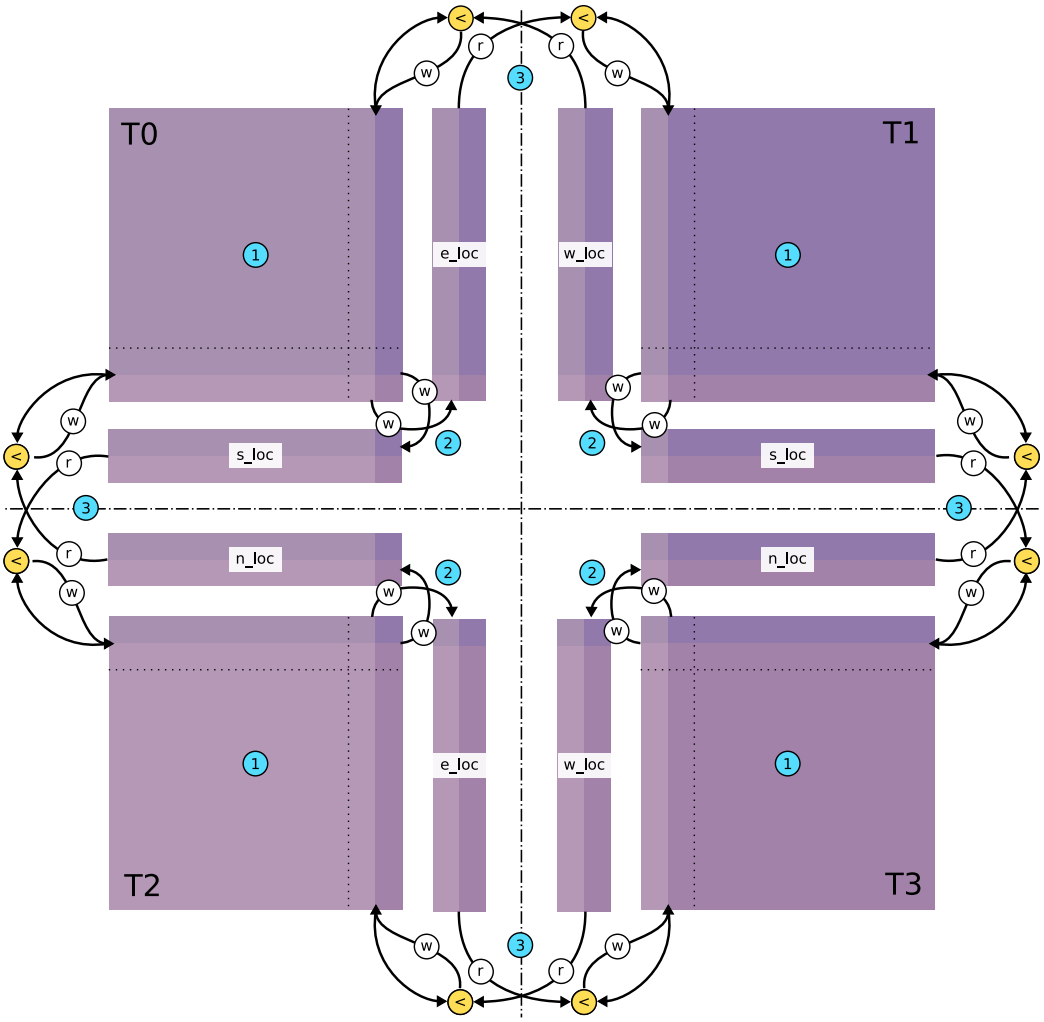


Figure 5.8: Principe ORWL de *local update / global update* appliqué à la mise à jour de la carte d'attache aux données

celles-ci sont copiés respectivement dans les *locations* s_loc, n_loc, e_loc, w_loc (étape 1). Ces écritures sont indépendantes et peuvent donc s'effectuer en même temps. Puis, les tâches récupèrent les valeurs des voisins et les comparent à leurs propres valeurs (étape 2). Enfin, seules les valeurs minimales sont écrites sur les cartes (étape 3). Les coins sont mis à jour automatiquement grâce à l'alternance des mises à jour verticales puis horizontales (voir section 3.1).

L'algorithme de copie des données diffère selon la phase de mise à jour et selon l'orientation des *locations*. Pour la mise à jour locale, s_loc et n_loc sont "bien orientées" car le sens de lecture des pixels est celui de la représentation en mémoire. La fonction `memcopy` permet de copier en bloc. Pour les *locations* e_loc et w_loc, une copie ligne à ligne sur la hauteur de la tuile est nécessaire (voir annexe A.11).
Dans la phase de mise à jour globale, le test de minima nécessite une copie valeur par valeur (voir annexe A.12).

Par ailleurs, il est intéressant de souligner un avantage de la méthode distribuée par rapport à celle partagée. La version originale d'écriture dans la carte d'attache aux données est la suivante :

```
for (y_ = ymin; y_<=ymax; ++y_) {
 omp_set_lock(&writelock[(y * lock_size)/height]);
  for (int x = nx0; x<=nx1; ++x) {
   if (Attach < IMG_ID(x,y)) {
    IMG_ID(x,y) = Attach;
   }
  }
 omp_unset_lock(&writelock[(y * lock_size)/height]);
 }
}
```

Pour chaque ellipse, l'écriture se fait en parcourant les pixels ligne à ligne (première boucle `for`). Le scan d'une ligne s'effectue entre une abscisse minimale nx0 et une abscisse maximale nx1. Afin d'éviter l'écriture simultanée de deux threads, un lock OpenMP doit être déployé par ligne.
Dans cette nouvelle implémentation, les zones communes de la carte étant dupliquées, il n'existe pas de risque d'écriture concurrente.

5.5 Thread de convergence

L'étape de convergence, telle qu'implémentée dans l'algorithme séquentiel, (voir section 2.4) n'est pas directement applicable dans un contexte distribué. Pour rappel, le critère de convergence repose sur un rapport entre le nombre d'ellipses gardées et celles détectées lors de l'itération.
En appliquant ces tests localement à chaque tâche, l'image n'est pas considérée dans son ensemble. Les résultats ainsi obtenus divergeraient de ceux obtenus avec l'algorithme à mémoire partagée. Pour être en conformité avec le modèle MPP, la convergence doit être testée à partir des résultats collectés auprès de l'ensemble des tâches.

5.5.1 Utilisation d'une location de convergence

Une possibilité consisterait à utiliser une *location* dédiée au recueil des résultats de chaque tâche pour permettre d'établir une convergence au niveau global. Localement accessibles en écriture, ces *locations* seraient lues par une tâche désignée pour en faire la somme. Le résultat du test de convergence pourrait ensuite être reporté dans une *location* de résultat. Cette *location* conditionnerait alors toute nouvelle itération.
Un inconvénient majeur à cette méthode est que la tâche la plus lente conditionne toutes les autres. En effet, pour commencer une nouvelle itération, les tâches doivent attendre les résultats du test de convergence qui sont conditionnés par la tâche la plus lente.

5.5.2 Utilisation d'un thread de convergence

Pour éviter une barrière de synchronisation forte, la solution adoptée consiste à déléguer le test de convergence à un thread détaché. Les tâches testent ainsi conjointement si elles doivent continuer les calculs ou s'arrêter. Le nouveau processus 5.9 décrit ce fonctionnement.

Le thread de convergence est créé dès le début de la tâche. Après l'initialisation de deux handlers lui permettant de communiquer avec sa tâche mère, il entre dans une boucle de vérification de la convergence. À l'intérieur de cette boucle, une première section critique lui permet d'accéder aux données de convergences sauvegardées par la tâche. Ces données sont disponibles en fin d'itération. Le thread va alors utiliser une méthode de réduction pour obtenir les données de tous ses semblables. Possédant toutes les informations, l'écriture des résultats peut être fait dans une nouvelle section critique.
Du point de vue de la tâche, les résultats de convergence ne sont réellement nécessaires

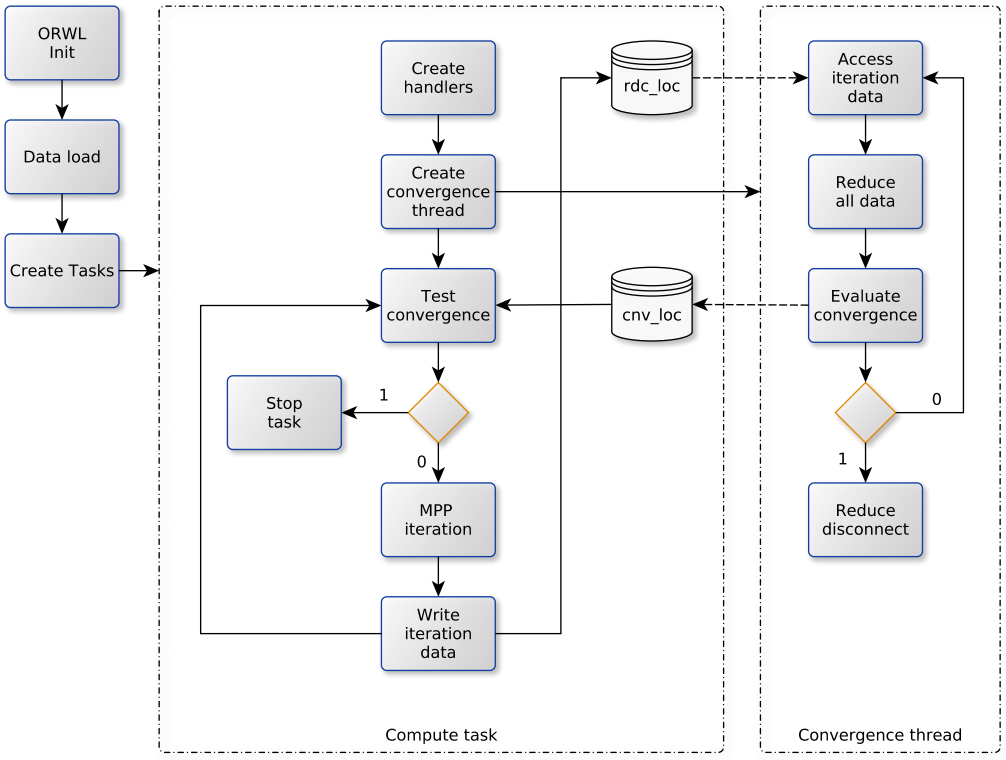


Figure 5.9: Processus de test de convergence dans un thread détaché

qu'au moment de mettre à jour les ellipses gardées, soit une itération plus loin. C'est là qu'une section critique est placée et permet d'accéder à la *location* partagée par le thread. S'il y a convergence, le tableau d'ellipses n'est pas mis à jour et la tâche s'achève. Sinon, les ellipses gardées sur cette itération remplacent celle de l'itération précédente.
Avec ce décalage, le temps nécessaire à la synchronisation des threads est masqué par le calcul des tâches. Cette méthode oblige a effectuer une itération supplémentaire. Le thread de convergence est implémenté en début de tâche de la manière suivante :

```
task_thread* thr = P99_NEW(task_thread);
rdc_str *thrr_hdl_ptr = P99_NEW(rdc_str, 0, 0, 0);
task_thread_operation(thr, orwl_myloc + rdc_loc);
```

Le pointeur `thrr_hdl_ptr` (ligne 2) est une structure contenant le nombre d'ellipses gardées, le nombre de changements et le booléen indiquant la convergence. Ce dernier paramètre est utilisé comme test d'arrêt de la boucle générale de calcul :

```
for (IT = 0 ; thrr_hdl_ptr->convergence == 1; IT++)
```

Le code exécuté par le thread de convergence est le suivant :

```
DEFINE_THREAD(task_thread) {
 ORWL_THREAD_USE(task_thread, cnv);
 orwl_server *srv = orwl_server_get();
 ORWL_ALLREDUCE_DECLARE(rdc_str, rdc_loc) =
 { .points = 0, .changes = 0, .convergence = 0 };
 orwl_handle2 thrw_hdl = ORWL_HANDLE2_INITIALIZER;
 orwl_handle2 tskr_hdl = ORWL_HANDLE2_INITIALIZER;
 orwl_write_insert(&thrw_hdl, ORWL_LOC(tid, thr_loc), 0);
 orwl_read_insert(&tskr_hdl, ORWL_LOC(tid, tsk_loc), 1);
 rdc_str *thrw_hdl_ptr = P99_NEW(rdc_str, 0, 0, 0);
 orwl_schedule(orwl_myloc, 1, srv);
 for (size_t t = 0; thrw_hdl_ptr->convergence < cnv; t++) {
  ORWL_SECTION(&tskr_hdl){
   rdc_str *tskr_hdl_ptr = orwl_read_map(&tskr_hdl);
   rdc_loc.points = tskr_hdl_ptr->points;
   rdc_loc.changes = tskr_hdl_ptr->changes;
  }
  ORWL_ALLREDUCE(rdc_loc, other) {
   rdc_loc.points += other.points;
   rdc_loc.changes += other.changes;
  }
  ORWL_SECTION(&thrw_hdl){
```

```
    if (!t) {
     orwl_truncate(&thrw_hdl, sizeof(rdc_str));
    }
    thrw_hdl_ptr = orwl_write_map(&thrw_hdl);
    if (rdc_loc.points / (rdc_loc.changes + 0.0001) > 500) {
     thrw_hdl_ptr->convergence = 1;
    } else {
     thrw_hdl_ptr->convergence = 0;
    }
   }
  }
  ORWL_REDUCE_DISCONNECT(rdc_loc);
  orwl_disconnect(&thrw_hdl);
  orwl_disconnect(&tskr_hdl);
}
```

Les *handlers* créés ont pour fonctions de récupérer les données émises par la tâche et de les rendre publiques. Le test de convergence est inclus dans une boucle `for` à l'intérieur de laquelle une première section critique écrit le résultat de la tâche une fois celui-ci disponible. La macro `ORWL_ALLREDUCE` réduit ensuite les résultats de toutes les tâches et les injecte dans la *location* `rdc_loc`. Là, une seconde section critique donne accès à la *location* `thrw_hdl` qui comptabilise le nombre de convergences atteintes. C'est ce facteur qui détermine l'arrêt de l'application lorsqu'il atteint la limite fixée.
Les trois dernières lignes permettent de déconnecter la mécanique de réduction et libérer les *handlers*.
Notons que l'appel à la fonction `orwl_schedule` est nécessaire pour transmettre les requêtes d'insertion des handlers (voir section 4.4.5).

5.6 Problèmes rencontrés et démarches de résolution

Cette section présente les problèmes les plus significatifs rencontrés au cours du développement de l'application MPP_ORWL.

5.6.1 Problèmes de détections

Au fil des modifications apportées à l'algorithme d'origine, la qualité des résultats s'est dégradée. Un tel glissement a eu lieu sans provoquer d'erreur de code. La représentation visuelle des résultats, a permis de rectifier les deux erreurs suivantes :

Mauvaise orientation des ellipses

Les deux images 5.10 et 5.11 illustrent le problème.

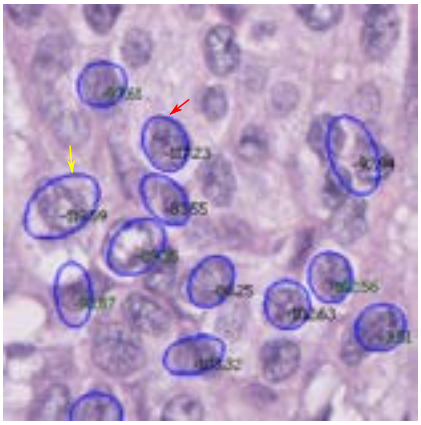


Figure 5.10: Affichage de l'attache aux données sur une détection incorrecte

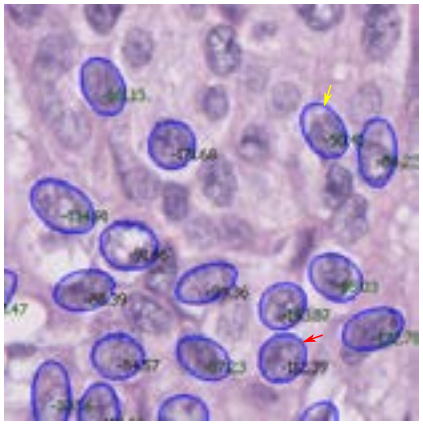


Figure 5.11: Affichage de l'attache aux données sur une détection correcte

Sur l'image de gauche, la flèche jaune pointe sur une ellipse de toute évidence mal positionnée. Son score est pourtant élevé (-119.59). En comparaison, la flèche rouge qui elle est bien placée obtient un score légèrement inférieur (-117.23).
Sur l'image de droite où la détection est bonne, nous constatons qu'une ellipse avec un score de -50.41 (flèche jaune) est clairement moins bien attachée qu'une ellipse avec un score de -112.39 (flèche rouge).
Une analyse plus poussée de l'image de gauche révèle un bug d'affichage au niveau de l'orientation des ellipses. Après vérification du code, il s'est avéré que l'angle assigné aux ellipses lors de leur création était différent de celui transmis sous forme de sinus et cosinus, d'où le décalage.

Recouvrement des ellipses

Une autre erreur d'affichage a permis de mettre en lumière un problème d'écriture des *locations*. L'image 5.12 montre, au centre, deux ellipses superposées. Or, le principe de constitution de la carte d'attache aux données, empêche un tel comportement en ne gardant que la meilleure des ellipses en cas de chevauchement (voir section 2.5.1). En observant l'image dans son ensemble, il s'est avéré que ce type de problème se produisait uniquement au niveau de l'axe central de l'image. C'est ce qui a orienté les recherches

vers les cellules fantômes. Pour matérialiser les données échangées entre les tâches, les bandes fantômes ont été capturées sur l'image 5.13.

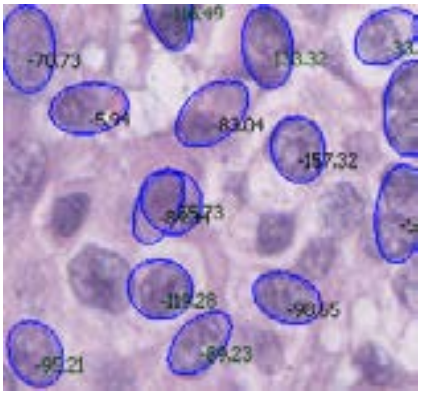


Figure 5.12: Problème de recouvrement de certaines ellipses

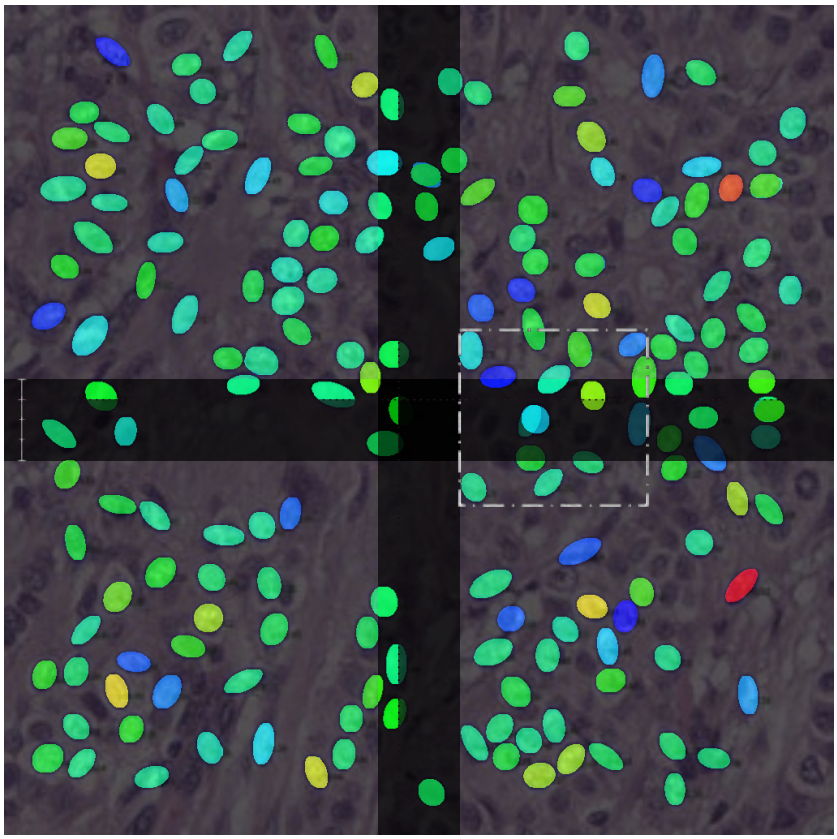


Figure 5.13: Affichage des cartes d'attache aux données pour chacune des quatre tâches

Les bandes sombres représentent les zones de recouvrement. Les lignes en pointillés violets font apparaître une ligne de démarcation au-delà de laquelle les ellipses sont tronquées. Les données au-delà de cette zone ne sont donc pas échangées. À l'intérieur du cadre blanc, nous retrouvons les ellipses superposées. Elles appartiennent à deux tuiles différentes qui n'ont pas connaissance de l'existence l'une de l'autre. Les deux sont donc consignées dans les résultats locaux puis tracées sur l'image finale.
La cause du problème a pu être ainsi rapidement identifiée : une erreur de typage dans la fonction copie des cellules fantômes. Alors que les valeurs d'attache sont de type `float`, des `char` étaient copiés, soit quatre fois moins de données. Nous retrouvons bien ce ratio si l'on observe la règle graduée placée sur la gauche de l'image.
Après correction, les cartes coïncident parfaitement comme nous pouvons le voir sur l'image 5.14.

5.6.2 Changement de version

Suite à des mises à jour système, l'application était dans l'impossibilité de se lancer. Aucune indication n'indiquait la provenance de la panne. Un débogage pas à pas a été nécessaire pour circonscrire l'origine du problème : le changement de version de l'interpréteur Ruby. Dans la nouvelle version 2.2, l'instruction :

```
\code{trap("KILL") do |sig|}
```

du script de lancement, provoque une erreur. En effet, comme le montre la documentation, le langage ne prend désormais plus en compte le signal KILL :

Quoting from the man page: KILL (non-catchable, non-ignorable kill)

L'erreur a été résolue en mettant en commentaire cette interception de signal.

5.6.3 Addressbook

La constitution de l'addressbook est une étape préliminaire au lancement des tâches. Son mode de création (voir section 4.4.4) repose sur un script analysant les sorties standards de chaque processus ORWL. Lors des tests sur Xeon Phi, deux types de problèmes sont apparus, l'un ayant pour cause une incompatibilité système, et l'autre une limitation du script de transmission des flux.

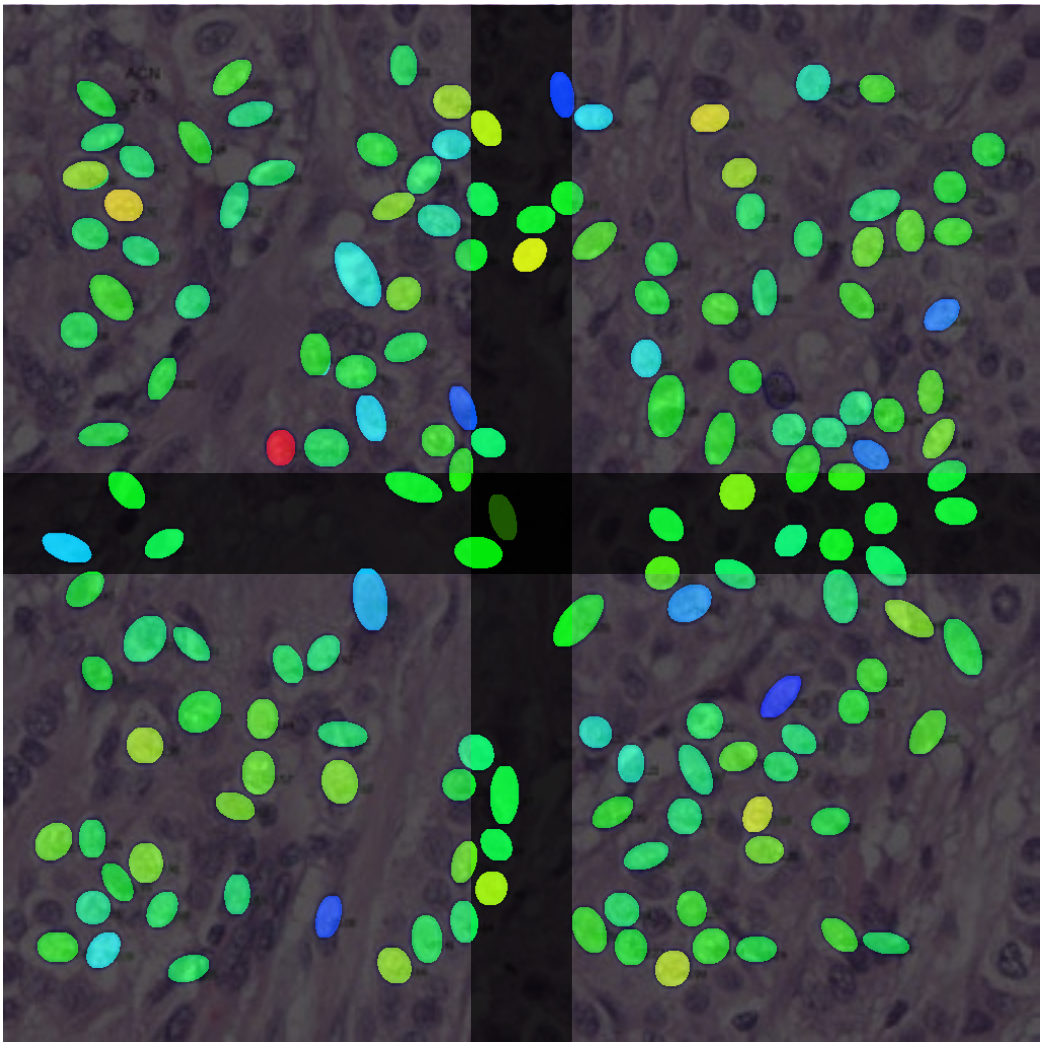


Figure 5.14: Zones de recouvrement des cartes d'attache aux données après correction du bug de copie

Incompatibilité système

Les tests d'exécution de l'application sur carte Xeon Phi ont révélé un problème au niveau de la génération de l'addressbook. La première ligne du fichier était polluée par deux 0 entraînant ainsi une erreur de lecture silencieuse. La recherche de l'émetteur de ces caractères s'est tout d'abord soldée par un échec. Une solution de contournement a en premier lieu consisté à supprimer la ligne indésirable. Mais le problème est réapparu de façon sporadique. En élargissant le périmètre de recherche aux fichiers de configuration ORWL, la source du problème a été identifiée : le script orwlrun_sh. La commande `ulimit` utilisée pour paramétrer la limite des ressources de l'application a un comportement différent sur les compilateurs gcc et icc. Sur le second, la fonction renvoie un 0 sur la sortie standard. Or c'est précisément ce flux qui est intercepté par TakTuk pour construire l'*addressbook*. La correction du problème a consisté à rediriger des sorties de la commande vers un périphérique nul :

```
ulimit -S ${ORWL_ULIMIT} > /dev/null 2>&1
```

Surcharge mémoire

Un autre type de problème sur Xeon Phi a été rencontré lors de tests impliquant la création de 240 tâches. Avec sept *locations* par tâche, le nombre total de *locations* à répertorier s'élevait ainsi à 1680. À l'exécution, le fichier présentait l'erreur d'affichage suivante :

Listing 5.3: Addressbook problème de retour à la ligne

```
1619 61  orwl://10.141.128.1:39037/61
1620 60  orwl://10.141.128
1621 .1:39037/60
1622 59  orwl://10.141.128.1:39037/59
```

Pour remonter à l'origine de ce problème, il a fallu explorer les méthodes utilisées par TakTuk pour rediriger les flux.
La fonction responsable se nomme : `post_write`. Les données reçues des nœuds sont stockées dans un buffer avant d'être retransmises par protocole TCP au nœud root. La taille d'envoi des messages est alors limitée à 32768 octets. Au delà, le message est scindé et envoyé en plusieurs parties.
En doublant la taille du buffer d'écriture, le problème a été réglé. Cependant, cette solution n'étant pas évolutive, une expression régulière a été ajoutée à la réception des lignes pour contrôler qu'elle soit bien de la forme : `orwl://adresse_ip:port/location_id`.

Listing 5.4: Vérification de la validité des lignes de l'addressbook

```
if (!/^[0-9]+\sorwl:\/\/\d{,3}\.\d{,3}\.\d{,3}\.\d{,3}\:\d
   +\/\d*$/.match(line)) then puts "problem␣on␣line␣#{line}"
```

5.6.4 Passage à l'échelle

Lors des tests de passage à l'échelle, l'augmentation de la taille de l'image a révélé un certain nombre de problèmes sur toute la chaîne de traitement. Les réglages recommandés dans cette section sont applicables à tout projet manipulant des images gigapixels et au-delà.

Paramétrage systèmes

La manipulation d'une image implique l'utilisation d'un fichier temporaire. Lorsque l'image est de l'ordre du gigapixel, ce fichier peut quant à lui, être cinq à dix fois plus gros quand il est en cours de traitement. Il faut ainsi veiller à adapter la taille du dossier temporaire. Il se peut également que la mémoire vive soit insuffisante. L'augmentation de la partition de swap peut permettre de contourner cette limite. Le cas de figure s'est produit lors de l'extraction des images à partir du fichier source issu du scan de lame.

Paramètrage ImageMagick

Une fois le dossier temporaire à la taille adéquate, il faut paramétrer l'exécutable en charge de la conversion. Pour l'application, ImageMagick a été utilisé. Par défaut, la taille des images est limitée [6] pour des raisons de sécurité. Pour aller au-delà des 8000×8000 pixels, le fichier de configuration `policy.xml` doit être mis à jour. Les paramètres suivants ont été appliqués dans le cadre de l'application MPP_ORWL :

```
<policy domain="resource" name="temporary-path" value="/home/
   tmp"/>
<policy domain="resource" name="memory" value="6GB"/>
<policy domain="resource" name="map" value="512MiB"/>
<policy domain="resource" name="width" value="40KP"/>
<policy domain="resource" name="height" value="40KP"/>
<policy domain="resource" name="area" value="128MB"/>
<policy domain="resource" name="disk" value="6GB"/>
```

[6] https://imagemagick.org/script/security-policy.php

Le premier paramètre est particulièrement important car il représente le répertoire temporaire utiliser par ImageMagick. Par défaut, le dossier `/tmp` est utilisé. Jusqu'au traitement des images de dimension 38913 x 33310 20 Go . Au-delà, le dossier atteint saturation et l'exécution s'interrompt sur une erreur.

5.6.4.1 Saturation d'allocation mémoire des locations

Lors des tests sur carte Xeon Phi, une limitation de la librairie ORWL a été mise à jour. La gestion de l'attribution mémoire des *locations*. Avec quatre *locations* par tâches, si 200 tâches sont lancées simultanément, cette phase de dimensionnement engendre 800 threads concurrents. En régime permanent, la probabilité que tous ces threads se connectent en même temps aux *locations* est relativement faible. En revanche, c'est le cas en phase d'initialisation.
Une fois connue, cette limitation a pu être contournée en déportant les requêtes à l'intérieur des sections critiques juste avant l'utilisation des *locations*. Le dimensionnement a alors lieu avant la première utilisation d'une *location* ce qui étage les insertions de requêtes et évite la surcharge.
Il est à noter, que l'allocation étant faite après l'étape de synchronisation du `orwl_schedule`, les *locations* en écriture doivent être redimensionnées et non pas juste dimensionnées. En effet, une taille nulle par défaut est appliquée si aucun `orwl_scale` n'est détecté. Cette fonction est `orwl_truncate` et s'utilise comme suit :

$$orwl\_handle\_truncate(orwl\_handle * rh, size\_t data\_len, p99\_seed * seed)$$

5.6.4.2 Carte des naissances

La carte des naissances est une image en niveau de gris dont la valeur des pixels est calculée par une approximation de l'algorithme MPP (voir section 2.3.3). Cette image en niveau de gris est de dimension égale à l'image à analyser. Son processus de création comporte les étapes suivantes :

1. pré-script
   (a) découpage de l'image ;
   (b) envoi des images vers les nœuds ;
2. application

(a) attribution d'une valeur d'attache aux données à chaque pixel ;

(b) écriture des résultats ;

3. post-script

(a) récupération des résultats ;

(b) conversion des résultats en image ;

(c) collage des images.

Les images sont constituées individuellement et assemblées ensuite car à partir d'une certaine taille, l'écriture en une seule passe était impossible avec l'outil convert. Une boucle de collage (voir annexes A.12.3) a donc été mise en place pour assembler les images en bandes horizontales. Ces dernières sont ensuite assemblées verticalement pour produire l'image finale.
L'enregistrement en jpeg d'une image dépassant le gigapixel n'est cependant pas possible avec le taux de compression standard de 80%. Le message d'erreur suivant apparaît :

```
convert: Bogus Huffman table definition
```

Une option de non-optimisation lors de la conversion au format jpeg doit être ajoutée comme suit pour contourner ce bug :

```
-define jpeg:optimize-coding=false
```

5.6.5 Compilation d'ORWL sur compilateur Intel

La compilation d'ORWL avec le compilateur Icc 16.0 a remonté des problèmes de conformité à la norme C11. Les mots clés : `generic`, `atomic`, `threads`, `inline` ne sont ainsi pas pris en compte malgré les données publiées par Intel (voir tableau A.3).
Pour les fonctions atomiques, tous les types ne sont pas pris en charge [7].
P99 permet de combler ce manque en émulant les fonctions atomiques. Il suffit pour cela de définir la variable `__STDC_NO_ATOMICS__` à 1.

D'autres erreurs de compilations comme celles du listing ci-dessous sont aussi apparues à l'exécution :

[7] Source : https://software.intel.com/en-us/forums/intel-c-compiler/topic/681815

```
__typeof__(
__builtin_choose_expr (
__builtin_types_compatible_p(__typeof__((char(*)[8]){0}),
   char(*)[4]),
(uint16_t){ 0 },
(uintptr_t){ 0 } ) ) p00_m;

BAD => internal error: assertion failed at:
"shared/cfe/edgcpfe/expr.c", line 259
(uintptr_t){ 0 } )) p00_m;
```

Cette erreur était provoquée par la syntaxe utilisée pour définir les *compound literals* : `0 (uint16_t)`.
Pour contourner ce problème, une autre syntaxe a été utilisée :

```
uint16_t p00_val = {0}; p00_val;
```

Cette nouvelle forme est alors correctement interprétée par le compilateur Intel mais son utilisation se limite au code à l'intérieur de fonctions.
Le code suivant levait ainsi de nouvelles exceptions :

```
union {
__typeof__(
__builtin_choose_expr (0, ({uint16_t p00_val = {0}; p00_val
   ;}), ({uint32_t p00_val = {0}; p00_val;}))) p01_m;
};
```

Finalement une troisième forme a remporté l'adhésion générale : `*(T*)0`.

L'ensemble des modifications apportées aux bibliothèques ORWL et p99 pour une conformation aux cartes Xeon Phi sont consultables sous forme de fichiers diff en annexes A.13.

5.6.6 Exécution d'application ORWL sur Xeon Phi

Le système d'exploitation présent sur les cartes Xeon Phi ne dispose pas d'un interpréteur perl. Or ce langage est celui de la librairie TakTuk utilisé dans ORWL.
La société Paratools propose sur son site [8] une version de l'interpréteur compatible. Le

[8] http://www.paratools.com/XeonPhi/Perl

script suivant permet d'automatiser son installation :

```
#!/bin/sh

cp -r /cm/shared/apps/perl-5.18.2-mic/ /cm/local/perl

echo "PermitUserEnvironment␣yes">>/etc/ssh/sshd_config
echo "AcceptEnv␣PERL5LIB␣PATH␣MIC_PERL5LIB␣MIC_PATH">>/etc/
   ssh/sshd_config

ln -s /cm/local/perl/bin/perl /usr/bin/perl

/etc/init.d/sshd restart
```

Après la copie du dossier de l'hôte vers la carte, des variables d'environnement doivent être initialisées dans les fichiers de configuration `sshd_config`. Cette étape est primordiale car ces variables doivent être disponibles avant l'appel des commandes `ssh` de TakTuk. Dans notre contexte, le serveur est la carte MIC et le client est le nœud ou le master qui s'y connecte. Le script de création des variables d'environnement `perlvars_config.sh` est placé sur le nœud maître dans : `/etc/profile.d` et contient les variables suivantes :

```
prefix=/cm/local/perl
export PERL5LIB=$prefix/lib/perl5
export PATH=$PATH:$prefix/bin
export MIC_PERL5LIB=$prefix/lib/perl5:$MIC_PERL5LIB
export MIC_PATH=$prefix/bin:$MIC_PATH
```

À chaque connexion le script est sourcé et les variables d'environnement perl sont ainsi définies.
Pour maintenant transmettre ces variables vers les cartes Xeon Phi, il faut paramétrer les fichiers de configuration `ssh`.
Côté client : `/etc/ssh/ssh_config`

```
SendEnv PERL5LIB PATH MIC_PERL5LIB MIC_PATH
```

Côté serveur : `/etc/ssh/sshd_config`

```
PermitUserEnvironment yes
AcceptEnv PERL5LIB PATH MIC_PERL5LIB MIC_PATH
```

Pour redémarrer les services côté client :

```
service sshd restart
```

Côté serveur MIC :

```
/etc/init.d/sshd restart
```

Les variables d'environnement sont disponibles même dans le cas d'une connexion ssh non-interactive comme c'est le cas dans la transmission de commande TakTuk.

5.7 Outils

Cette section présente les outils et librairies annexes à l'application et néanmoins indispensables à la manipulation d'image.

5.7.1 Open Slide

Pour des raisons de confort et de possibilité de partage, les lames issues de biopsies sont de plus en plus souvent numérisées. Les taux d'agrandissement peuvent aller jusqu'à 400 fois, ce qui produit des images dont la taille se compte en dizaines de gigas [40]. Le format image généré par les scanners sont multi-résolution, c'est-à-dire qu'un fichier contient différentes résolutions de l'image assemblées selon un principe pyramidal.
Il n'existe pas de format universel pour ce type de fichiers et chaque vendeur implémente ses propres solutions et outils de visionnage. Il existe cependant une solution libre de droit développée à l'université de Carnegie Mellon : OpenSlide[9]. Cette API propose une interface permettant d'extraire et de visionner les formats des plus grands fabricants de scanner, parmi lesquels : Aperio (.svs, .tif), Hamamatsu (.vms, .vmu, .ndpi), Leica (.scn), MIRAX (.mrxs), Philips (.tiff), Sakura (.svslide), Trestle (.tif), Ventana (.bif, .tif), Generic tiled TIFF (.tif).

La librairie, implémentée en C99, expose des interfaces servant de relais aux fonctions propriétaires. Le listing 5.7.1 montre comment extraire une image à sa plus forte

[9] https://openslide.org/

résolution.

```
openslide_t *osr = openslide_open(image_path);
int64_t w, h, x, y;
openslide_get_level0_dimensions(osr, level, &w, &h);
uint32_t *dest = malloc(sizeof(uint32_t[w*h]));
openslide_read_region(osr, dest, x, y, level, w, h);
openslide_close(osr);
unsigned char *dest2 =
malloc(sizeof(unsigned char [w*h*3]));
for(size_t i=0; i<w*h; i++) {
        dest2[i] = (dest[i] & 0x00ff0000) >> 16; // R
        dest2[i+w*h] = (dest[i] & 0x0000ff00) >> 8; // G
        dest2[i+2*w*h] = dest[i] & 0x000000ff; // B
}
interfaceSaveImg(str, "6569.png");
```

Après avoir récupéré le scan dans un objet `openslide` (ligne 1), la fonction `openslide_get_level0_dimensions` permet de récupérer les dimensions de l'image. Ces données sont ensuite injectées dans la fonction `openslide_read_region` pour lire l'image. Les coordonnées `x, y` et les longueurs `w, h` permettent d'extraire une zone précise de l'image. Les données sont collectées dans un vecteur de type `uint32_t` dans lequel un pixel est représenté avec 8 bits de rouge, 8 bits de vert, 8 bits de bleu et 8 bits de transparence. Une boucle (ligne 8) est ensuite utilisée pour extraire ces valeurs dans un vecteur dont les composantes couleur sont contiguës. L'image finale est enfin enregistrée.

5.7.2 CImg

CImg [10] est la librairie open-source de traitement d'image. Le projet est soutenu par le laboratoire GREYC [11] et compte des contributeurs du monde entier. La librairie consiste en un unique fichier `CImg.h`. Écrite en C++, elle est portable sur les systèmes Linux, Windows et MacOS.
La classe principale est de type `CImg`. C'est elle qui permet la manipulation des images en une, deux, trois ou quatre dimensions. En mémoire, une image est représentée par un vecteur de dimension : longueur $\times*$ largeur $\times*$ profondeur $\times*$ dimension $\times*$ sizeof(T), dans lequel T est le type de valeur d'un pixel. La dimension représente le nombre de couleurs utilisées et la profondeur le nombre de plans dans le cas d'une image en trois

[10] http://cimg.eu

[11] www.greyc.ensicaen.fr

dimensions.
Les pixels d'une image couleur sont stockés en mémoire de manière désentrelacée : R1R2R3...V1V2V3...B1B2B3... Le premier pixel est celui en haut à gauche de l'image. Sa composante rouge est obtenue par l'appel : `img(0,0,0,0)`.
Une large palette d'effets est disponible ainsi que les fonctionnalités classiques de recadrage ou découpage. Des fonctions de dessin permettent de tracer toute sorte de formes simples. L'interface exporte dans la plupart des formats images standards et de ceux utilisés en imagerie médicale.
Les deux autres classes de CImg permettent de gérer des listes d'éléments ou encore d'afficher les images dans une fenêtre graphique.

5.7.3 TakTuk

TakTuk [12] est un outil open-source de déploiement et d'exécution d'applications parallèles. Capable de s'auto-propager selon un algorithme adaptatif [41], TakTuk permet l'exécution de commandes à distance.
Son utilisation est spécialement conçue pour l'administration de machines parallèles avec le protocole `ssh`.
La syntaxe d'envoi de commandes s'écrit :

```
taktuk -s -m node001 -m node002 broadcast exec { foo }
taktuk -s -f hosts broadcast exec { bar }
```

La fonction `foo` est ici envoyée vers les nœuds `node001, node002` (ligne 1) puis la fonction `bar` vers la liste de nœuds contenue dans le fichier `hosts` (ligne 2).
TakTuk gère un ensemble de flux auquel il est possible de s'inscrire. Un canal info permet notamment de suivre les évènements de connexion et d'envoi/réception de commandes. Un élément fort appréciable est la redirection de ces flux. Il est ainsi possible de récupérer l'affichage envoyé sur la sortie standard des machines distantes en utilisant l'option -R :

```
taktuk -s -m node001 -R output=>/tmp/taktuk.log broadcast
       exec { uptime }
```

Chaque nœud possède un rang qu'il acquiert lors du déploiement de l'application. Lors de l'envoi de fichiers, cette variable peut être utilisée pour faire référence à une destination spécifique. Un exemple de commande d'envoi est :

[12] http://taktuk.gforge.inria.fr

```
taktuk -f nodes_list synchronize broadcast
       put [ src/file ] [ dest/file.$TAKTUK_RANK ]
```

La variable `$TAKTUK_RANK` est interprétée par le shell client comme le rang du nœud. Notons l'utilisation du mot clé `synchronize` qui force à attendre la fin du déploiement, du numérotage des nœuds et de la commande précédente avant de s'exécuter.

De la même manière, la fonction `get` permet la récupération de fichiers issus des nœuds spécifiés dans le fichier `nodes_list` :

```
taktuk -f nodes_list synchronize broadcast get [ src/file ] [
    dest/file.$TAKTUK_RANK ]
```

Dans un script Perl, TakTuk peut également être utilisée dans une forme orientée objet. L'instruction suivante va créer un nouveau répertoire sur le nœud client :

Listing 5.5: utilisation de la syntaxe objet avec TakTuk

```
my $cmd = "broadcast␣exec␣[␣mkdir␣new_dir␣]␣";
my $tak = TakTuk::Pilot->new();
my %args = (tak=>$tak, step=>"2");
$tak->add_callback(callback=>\&callback_stream, stream=>'
   state', argument=>\%args, fields=>['host', 'rank', 'line',
    'command', 'type']);
$tak->send_command($cmd);
$tak->send_termination();
$tak->run(command=>"taktuk␣-s␣-m␣node001␣-m␣node002");
```

Dans cette forme, il est possible aussi bien d'ajouter une fonction de rappel à la commande que d'envoyer et récupérer des arguments dans cette fonction.

5.8 Conclusion du chapitre

Les modifications apportées à l'algorithme ont permis d'en proposer une version plus rapide et distribuable à la fois sur CPU mais aussi sur cartes Xeon Phi.
Les tests de performances du chapitre suivant vont permettre de valider le bon passage à l'échelle de l'application.

6 Tests et résultats

Le but de ces tests est d'observer les performances de l'algorithme parallèle MPP_ORWL sur une image de gigapixels. Les tests sont réalisés sur le cluster (voir section 6.1). Ce dernier est composé de quatre nœuds possédant chacun 16 cœurs avec hyperthreading. Au total, 64 cœurs réels et 128 threads sont disponibles.

6.1 Plateforme de test

Les tests ont été réalisés sur un cluster composé d'un serveur frontal et de quatre nœuds équipés comme suit :

1. serveur frontal :
   (a) bi-processeurs 8 cœurs E5-2623 à 3GHz ;
   (b) 32Go de RAM DDR4 à 2133MHz ;
   (c) 2 disques SSD de 256Go ;
   (d) 5 disques de 3To en RAID 6 ;
   (e) OS : linux CentOS 7.1.1503., compilateur Intel ICC.
2. 4 nœuds de calcul :
   (a) bi-processeurs 8 cœurs E5-2640 (32 threads disponibles) ;
   (b) 128Go de RAM DDR4 à 2133MHz ;
   (c) carte GPU GTX TITAN X 12Go GDDR5 (3072 cœurs cuda) ;
   (d) carte Intel Xeon Phi 5110P Knight Corner, 1.053GHz, 8Go de RAM ;
   (e) OS : linux CentOS 7.1.1503.
3. connectique : liaison infiniband mellanox 40Go/s.

Bien que les nœuds soient équipés de cartes GPU, nous avons privilégié l'utilisation des cartes Xeon Phi pour offrir une implémentation de l'algorithme unique rendant ainsi plus facile son usage et sa compréhension.
Les architectures physique et logicielle sont respectivement présentées sur les schémas 6.1 et 6.2.

Figure 6.1: Architecture physique du cluster de test. Source : Neoteckno

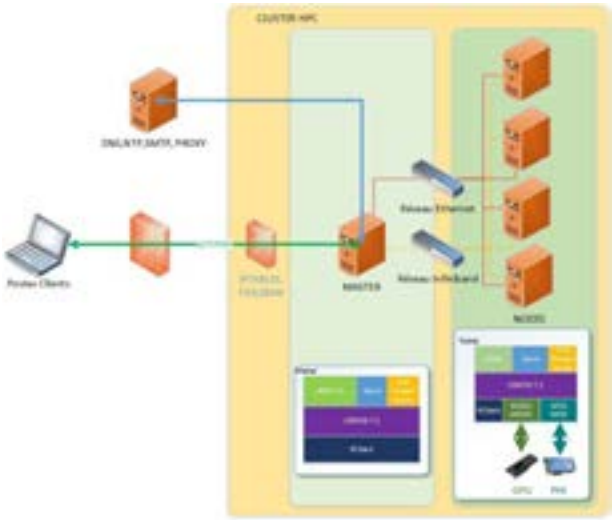


Figure 6.2: Architecture logicielle du cluster de test. Source : Neoteckno

L'ensemble occupe 19U soit un demi-rack (≈ 91cm), pour un poids total de 125kg et une consommation électrique de 9000W. Ce faible encombrement permettrait son instal-

lation à proximité d'un scanner de lames.
D'un point de vue logiciel, le nœud frontal est accessible par connexion ssh. Les nœuds sont alors interconnectés sur un réseau infiniband avec un débit de 40Go/s. Les cartes Xeon Phi disposent de leur propre adresse et sont accessibles directement pour des exécutions en mode natif.
La puissance globale du cluster est de :

Type	cadence	cœurs	quantité / nœud	flops	précision
CPUs	2133MHz	8	2	0.14 Tflops [1]	double
GPUs	1089MHz	3072	1	0.209 Tflops [2]	double
Xeon Phi	1053MHz	60	1	1.01 Tflops [3]	double
Total (4 nœuds)				5.99 Tflops	

6.2 Protocole de test

Les données utilisées pour les tests proviennent d'un test de dépistage du cancer du sein opéré à l'hôpital de la Pitié Salpétrière. La lame de biopsie a été numérisée sur un scanner Aperio [4]. Le fichier, au format .svs, contient trois niveaux de zoom différents :

Niveau	agrandissement	taille image
0	400	58630×44216
1	100	14657×11054
2	25	3664×2763

Les images à analyser ont été extraites du niveau 0 en divisant par deux le format sept fois consécutives (voir 6.3).

L'image 6.4 montre une vue d'ensemble de la lame complète ainsi découpée. Un aperçu des différents niveaux de zoom disponibles dans le fichier source sont visibles sur l'illustration 6.5.

[1] https://cads.iiap.res.in/tools/flopsCalc

[2] https://www.techpowerup.com/gpu-specs/geforce-gtx-titan-x.c2632

[3] https://en.wikipedia.org/wiki/Xeon_Phi

[4] https://www.leicabiosystems.com/digital-pathology/scan/

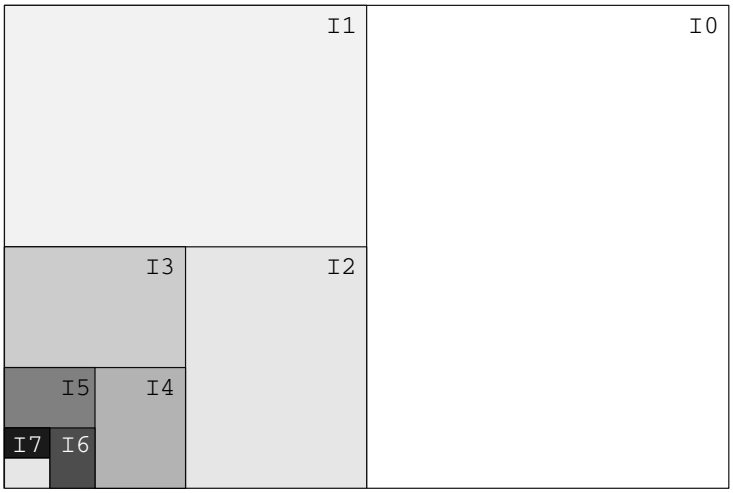


Figure 6.3: Formats d'image utilisés pour les tests

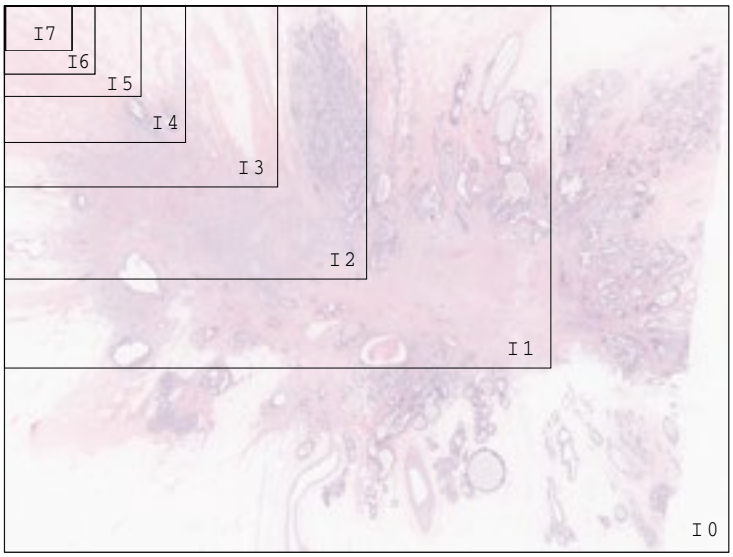


Figure 6.4: Vue d'ensemble de la lame et des différents formats d'images

À un grossissement de 400, les noyaux mesurent environ 20 pixels de long.

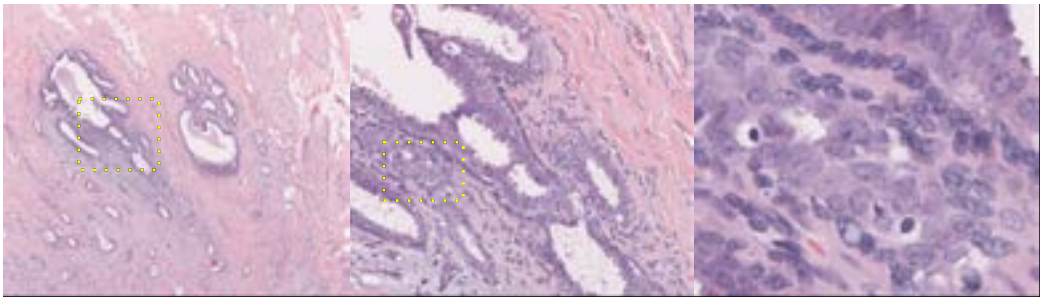


Figure 6.5: Zone de l'image représentée à différents niveaux de zoom : 25, 100, 400

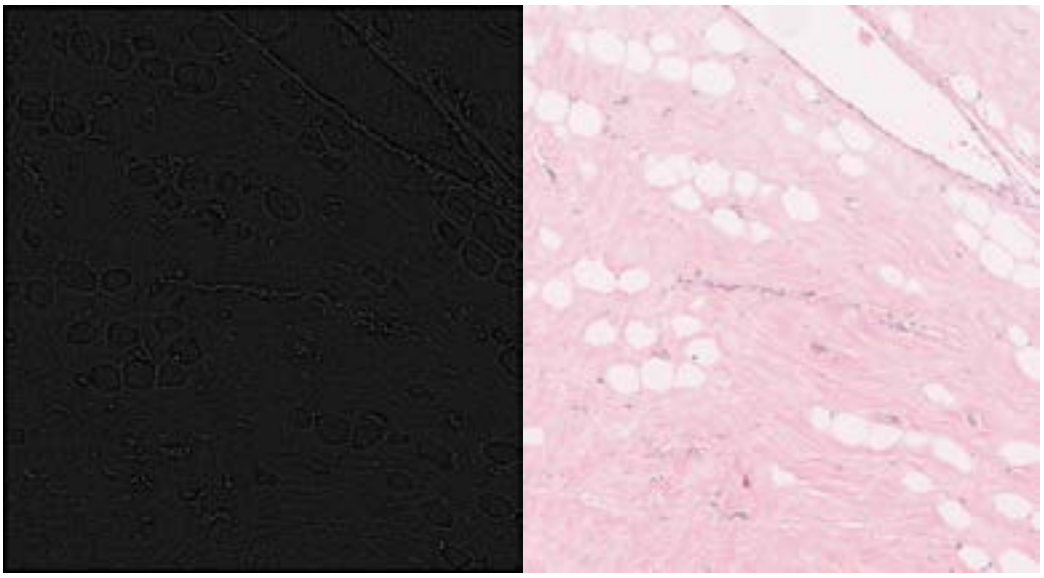


Figure 6.6: À gauche, rendu en niveau de gris de la carte des naissances. À droite, l'image originale I7.

6.3 Utilisation de la carte des naissances

L'utilisation de la carte des naissances telle que présentée à la section 2.3.3 consiste à créer une image sur laquelle la valeur d'un pixel va refléter son aptitude à recueillir une ellipse. Le parallèle entre la carte et l'image de départ est visible sur l'illustration 6.6.

Les valeurs d'attache aux données, comprises entre -1 et 1 sont converties en cinq niveaux de gris différents, où le noir représente une zone avec la plus faible probabilité de noyaux.
Lors de l'étape de naissance, la carte préalablement chargée au même titre que l'image à analyser, permet d'affiner la sélection des pixels en agissant comme un masque. En effet,

le coefficient de probabilité correspondant à la taille de l'image et à un facteur de recuit simulé est multiplié par le pourcentage issu de la carte de naissance.
Dans les faits, les résultats obtenus avec et sans la carte sont identiques. Aucune amélioration au niveau de la qualité de la détection ou de la rapidité de convergence n'a été détectée. De plus, la carte a constitué un frein supplémentaire aux tests sur Xeon Phi. Les ressources mémoires réquisitionnées pour la carte obligeaient à réduire d'autant la taille de l'image analysée. Pour ces raisons, la carte n'a finalement pas été utilisée lors des tests présentés ci-après.

6.4 Threads CPU

Pour mesurer le degré de passage à l'échelle de l'application sur CPUs, nous allons comparer les résultats obtenus sur 1, 2, 4, 8, 16, 32, 64 threads. Ces 7 mesures coïncident avec les formats images de I7 à I1, en doublant à chaque montée. Le format cible I0, représentant une lame complète, devait être analysé par 128 threads. Cependant, les tests ont montré l'inefficacité du *simultaneous hyperthreading* (SMT) dans le contexte de notre utilisation avec ORWL. Une discussion sur la comparaison des résultats avec et sans SMT est proposée à la section 6.4.3.
La nomenclature définie plus haut permet de paramétrer simplement différents types de configurations. Les benchmarks suivants font usage de 32 tâches :

- B05 = 0.1.2.16.2 : 2 processus lançant chacun 16 tâches ;
- B05b = 0.1.1.32.2 : 1 processus lançant 32 tâches ;
- B26 = 0.2.2.16.2 : 2 processus répartis sur deux nœuds, lançant chacun 16 tâches.

Les configurations B05 et B05b obtiennent des résultats similaires. Ce qui indique que le système répartit automatiquement les tâches sur les deux processeurs disponibles, qui sont rappelons-le, dotés chacun de 8 cœurs physiques.
Malgré cette déconvenue, les tests menés permettent de conclure sur l'aptitude de l'algorithme à passer d'un mode partagé à un mode distribué. Et même si l'objectif de traiter une lame complète n'est pas totalement atteint, une preuve de concept est apporté sur la faisabilité de traiter une image gigapixel (I1).

6.4.1 Tests de passage à l'échelle

Afin d'assigner les paramètres applicatifs et ainsi d'automatiser le lancement des tests, une nomenclature a été mise en place.
Par exemple, le code 0.1.1.1.7 dans lequel :

- 0 : CPU, (1 : MIC) ;
- 1 : 1 nœud (1 à 4) ;
- 1 : 1 processus (1 à 4) ;
- 1 : 1 tâche (1 à 64) ;
- 7 : image I7 (I0 à I7).

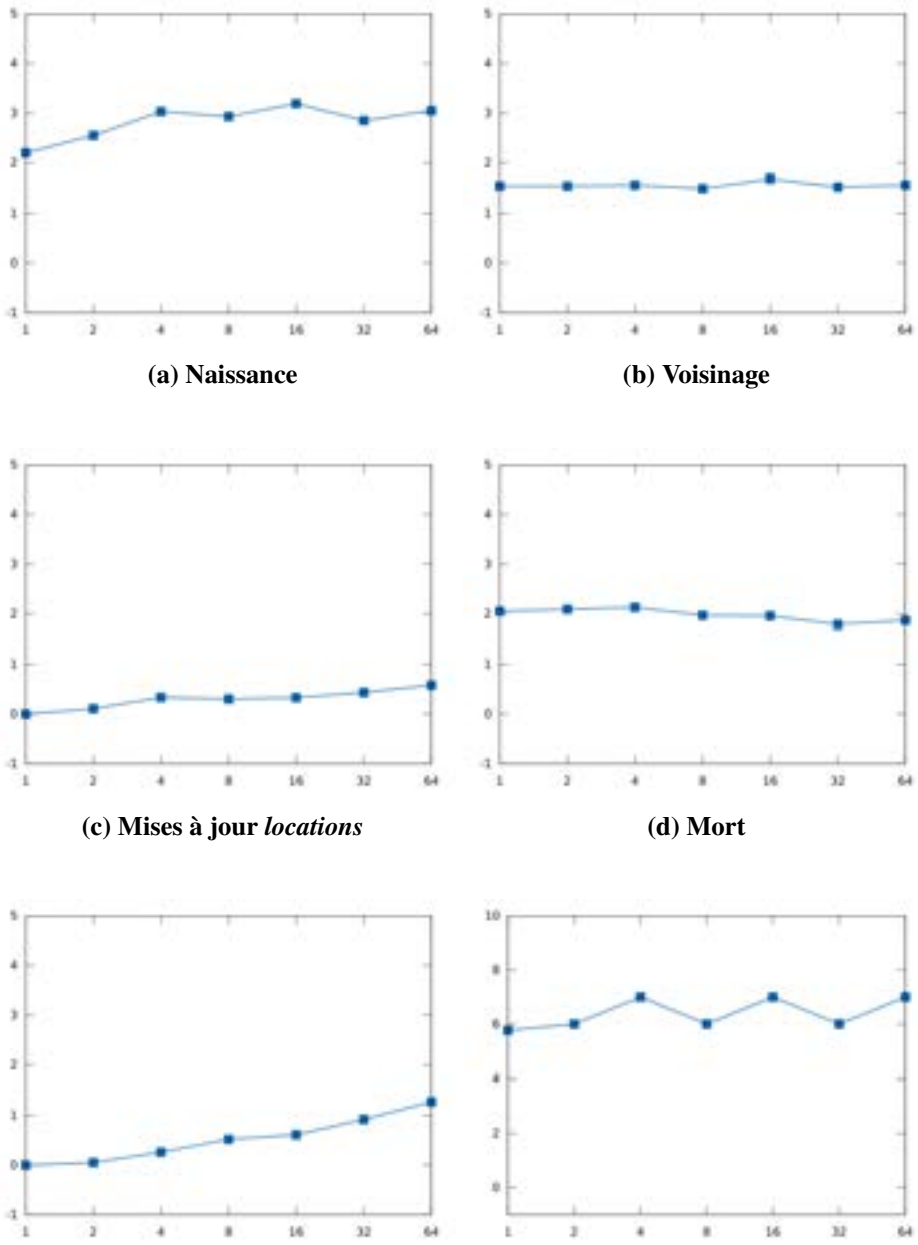
Le nombre de processus est global à l'application, alors que le nombre de tâches est relatif à un processus.
Les tests menés sont alors répertoriés comme suit :

Nom	Configuration	Matériel	Nœud	#Procs	#Tâches	Image
B00	0.1.1.1.7	CPU	1	1	1	I7 : 4301 x 4710
B01	0.1.1.2.6	CPU	1	1	2	I6 : 6281 x 6450
B02	0.1.1.4.5	CPU	1	1	4	I5 : 8600 x 9420
B03	0.1.1.8.4	CPU	1	1	8	I4 : 12900 x 12560
B04	0.1.1.16.3	CPU	1	1	16	I3 : 18840 x 17200
B26	0.2.2.16.2	CPU	2	2	32	I2 : 27636 x 23452
B27	0.4.4.16.1	CPU	4	4	64	I1 : 38913 x 33310

Les résultats de ces tests sont visibles sur les graphiques 6.7.

Les étapes les plus consommatrices de calculs (naissance, voisinage et mort) ont un comportement quasi-linéaire. L'étape de naissance augmente jusqu'à quatre threads puis se stabilise autour des trois secondes. L'étape de mort semble même être en légère décroissance. Ces aspects confirment le bon passage à l'échelle de l'application, notamment lors de la transition entre le mode partagé et le mode distribué, entre 16 et 32 threads, qui est transparente pour ces trois étapes.
Les temps de mise à jour des *locations* augmentent graduellement en fonction du nombre de threads et de la taille de l'image. Ce phénomène s'explique par l'augmentation de la quantité de données à échanger entre les tâches.

Figure 6.7: Étapes du calcul MPP_ORWL en temps par itération (s)



(a) Naissance **(b) Voisinage**

(c) Mises à jour *locations* **(d) Mort**

(e) Convergence **(f) Total**

Le temps de calcul pour la convergence augmente lui aussi de manière régulière. Cela est du à l'augmentation du nombre de tâches à synchroniser pour obtenir un résultat. Notons toutefois que ce temps est masqué par le temps de calcul des étapes de naissance, de

voisinage et de mort.
Remarquons enfin que le temps total par itération suit les fluctuations de l'étape de naissance avec une progression en dents de scie à partir de deux threads. Là encore le passage en mode partagé est à peine perceptible du fait de la compensation de l'augmentation de la mise à jour des *locations* et de la baisse de l'étape de mort.

6.4.2 Tests d'accélération

La capacité d'accélération de l'application a été calculée à partir de l'image de référence I3 (18840 x 17200 pixels). À problème constant, le nombre de threads a été doublé en sept étapes pour passer de 1 à 64 threads sans SMT. La configuration des tests est :

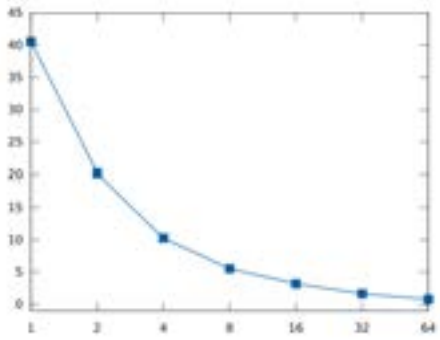
Table 6.1: Tests d'accélération sur CPU

Nom	Configuration	Matériel	Nœud	#Procs	#Tâches	Image
B13	0.1.1.1.3	CPU	1	1	1	I3
B14	0.1.1.2.3	CPU	1	1	2	I3
B15	0.1.1.4.3	CPU	1	1	4	I3
B16	0.1.1.8.3	CPU	1	1	8	I3
B17	0.1.1.16.3	CPU	1	1	16	I3
B18	0.2.2.16.3	CPU	1	2	32	I3
B19	0.4.4.16.3	CPU	2	4	64	I3

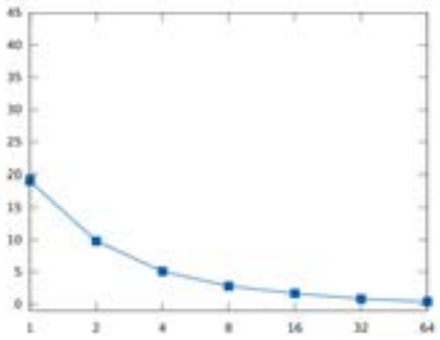
Les résultats sont visibles sur la série de graphiques 6.8.

Nous pouvons voir sur les étapes a, b et d que les temps de calcul sont diminués de moitié lorsque l'on double le nombre de threads. Là encore, le passage du mode partagé au mode distribué (de 16 à 32 threads) n'impacte pas négativement sur les résultats. Pour preuve, les temps de mise à jour des *locations* (graphique c) restent constants malgré l'augmentation du nombre de threads. Nous pouvons en déduire que les mises à jour des *locations* ont lieu simultanément. Ce phénomène s'explique par la nature du calcul qui reste homogène sur l'ensemble des tâches.
Les temps de convergence sont eux aussi stables ce qui révèle l'efficacité des threads mis en place à cet effet.
Le temps total par itération (f) présente logiquement les mêmes caractéristiques d'accélération que les étapes de naissance, de voisinage et de mort.

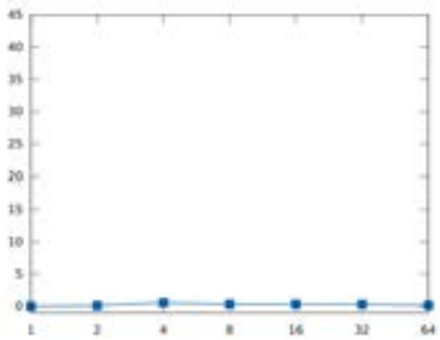
Figure 6.8: Temps par itération en secondes des tests d'accélération threads CPU sur l'image I3



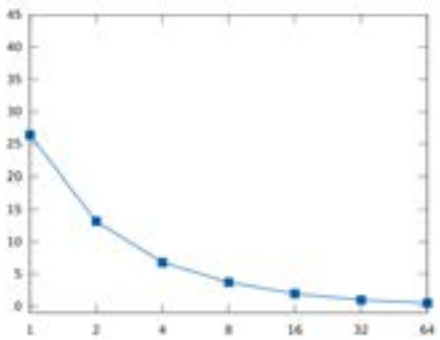
(a) Naissance



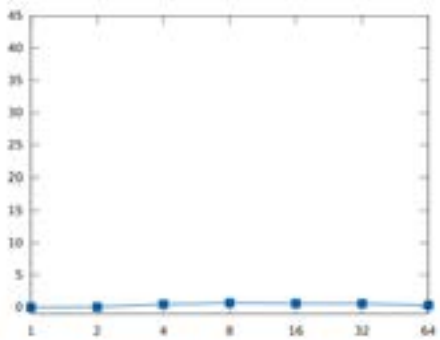
(b) Voisinage



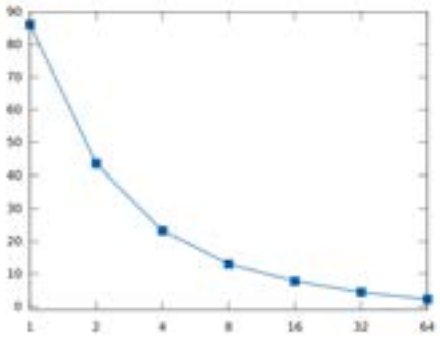
(c) Mises à jour *locations*



(d) Mort



(e) Convergence



(f) Total

6.4.3 Utilisation du SMT

Les nœuds du cluster comprennent à eux quatre 64 cœurs physiques. Avec l'utilisation du SMT cette capacité est doublée. Le découpage de l'image et les tests ont été conçus en prenant en compte un nombre maximum de 128 threads. Cependant, les tests de passage à l'échelle ont révélés l'inefficacité de l'hyperthreading dans notre cadre d'utilisation (voir graphique 6.9).

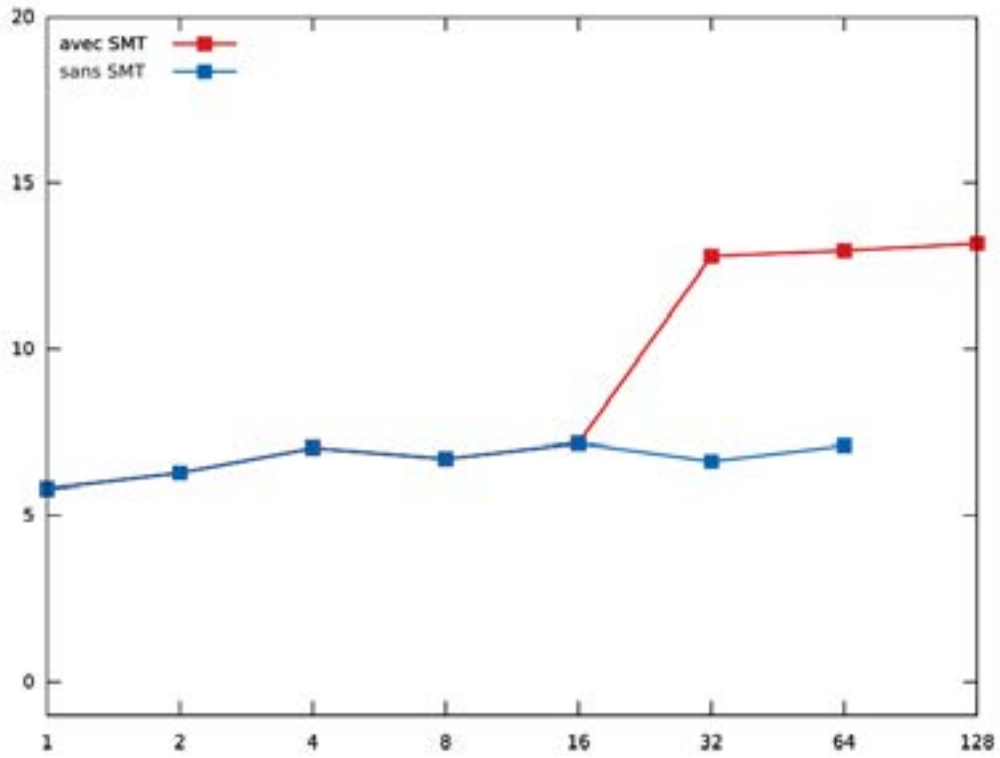


Figure 6.9: Test de passage à l'échelle avec SMT

Contrairement aux points bleus, les points rouges utilisent le SMT. Dans le premier cas, les 32 threads sont répartis sur deux nœuds, alors que dans le second cas, un seul nœud est utilisé ce qui déclenche l'utilisation du SMT.
De même, sur 64 threads, le point bleu représente une répartition sur quatre nœuds, contre seulement deux pour le point rouge.
Les temps de calcul de la version SMT sont doubles, ce qui révèle l'inefficacité de cette fonctionnalité dans notre cadre d'utilisation.
Cet état de fait peut s'expliquer par l'usage intensif de threads pour gérer le fonctionnement interne à ORWL. Le plus bel exemple en est l'insertion des requêtes décrite à

la section 4.4.5. Chaque insertion monopolise plusieurs threads qui restent actifs durant toute la durée d'utilisation d'un handler.

6.5 Xeon Phi

L'utilisation des cartes Xeon Phi a demandé un fort investissement pour permettre une correcte exécution de l'application (voir section 5.6.6). Les résultats obtenus ne reflètent toutefois pas ces efforts car de nouveaux problèmes survenus durant la phase de test n'ont pas permis de mener à bien toutes les mesures. L'utilisation hybride des processeurs CPU et des cartes Xeon Phi n'a pas été possible à cause de problème de communications au sein du réseau ORWL. Par ailleurs, la capacité mémoire des cartes étant limitée à un peu plus de 7Go, nous avons pu analyser d'image supérieure au format I3 (18840 x 17200). Malgré ces aspects, les résultats obtenus sur une carte sont prometteurs, ce qui justifie la poursuite des développements visant à corriger les erreurs de communication entre les cartes.

6.5.1 Estimation de la mémoire

Les cartes Xeon Phi utilisées pour ces tests disposent d'une capacité mémoire de 8 Go. Ce facteur est à prendre en compte pour dimensionner correctement la taille des images à analyser. C'est pourquoi une estimation de l'empreinte mémoire de l'application est nécessaire. L'analyse de l'algorithme révèle les éléments consommateurs de mémoire suivants :

- l'image à analyser (/tmp) ;
- vecteur image : hauteur * largeur (unsigned char) ;
- vecteur ellipses : hauteur * largeur * delta * 2 * 66 (structure ellipse) ;
- carte d'attache aux données : hauteur * largeur * 4 (float) ;
- données des *locations* :
  Nord : largeur * 40 (rmax) * 2 * 4 (float)
  Sud : largeur * 40 (rmax) * 2 * 4 (float)
  Est : hauteur * 40 (rmax) * 2 * 4 (float)
  Ouest : hauteur * 40 (rmax) * 2 * 4 (float)

En additionnant les composants listés ci-dessus, nous obtenons l'estimation suivante en fonction de la largeur (w) et de la hauteur (h) de l'image :

$$10 * w * h + 640 * w + 640 * h + poidsimage \tag{6.1}$$

À partir de cette équation et de la taille des images à charger en mémoire, nous pouvons comparer l'éligibilité de trois images.

Image	Dimension	Poids	Estimation	+10% marge
I3	17200×18340	180 Mo	3.44 Go	3.78 Go
I2	23452×27636	364 Mo	6.88 Go	7.57 Go
I1	33310×38913	741 Mo	13.75 Go	15.12 Go

En appliquant une marge de 10% à l'estimation, seules les images I2 et I3 sont éligibles. Cependant, en procédant aux premiers tests avec l'image I2, il s'est avéré que l'application était en surcharge mémoire. En effet, sur les 7.8 Go d'espace mémoire, 800 Mo sont environ nécessaires au fonctionnement du micro système. La taille maximale analysable avec l'application MPP_ORWL est donc celle de l'image I3.

6.5.2 Tests de passage à l'échelle

Pour être probants, les tests de passage à l'échelle auraient dû suivre les configurations suivantes :

Nom	Configuration	Matériel	Nœud	#Procs	#Tâches	Image
B09	1.1.1.60.5	MIC	1	1	60	I6
B10	1.1.1.120.4	MIC	1	1	120	I5
B12	1.1.1.240.3	MIC	1	1	240	I3
B13	1.2.2.240.2	MIC	2	2	480	I2
B14	1.4.4.240.1	MIC	4	4	960	I1

Le principe est ici le même que sur CPU avec une progression d'un facteur deux du nombre de threads et de la taille d'image à analyser. À partir de B13, la parallélisation est distribuée sur plusieurs nœuds. C'est à partir de là que des problèmes de communication entre les cartes Xeon Phi ont été révélés. Celles-ci ne peuvent pas accéder aux *locations* distantes, comme en témoigne le message de log suivant :

```
orwl_recv_:270: recv error,
```

```
abandonning: Connection reset by peer

orwl_send_remote:1357:
send request orwl://10.141.128.2:51666/ to
             orwl://10.141.128.1:52966/1110
did not succeed: Connection reset by peer
```

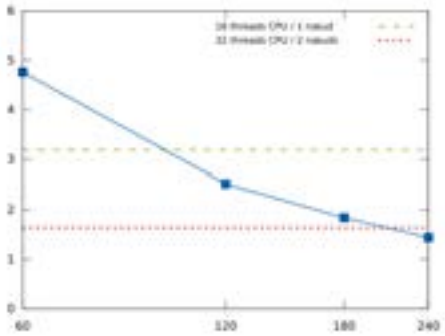
Nous voyons ici que la carte appartenant au nœud 2 reçoit une fin de non-recevoir de la part de la carte du nœud 1 qui ferme la connexion. La raison de ce problème est encore actuellement à l'étude.
Ne disposant que des mesures B09, B10 et B12, aucune conclusion ne peut être tirée. Toutefois, l'observation de l'activité mémoire de la carte durant le test B12 a permis de valider l'estimation réalisée au chapitre précédent avec une consommation de 3.72 Go.

6.5.3 Tests d'accélération

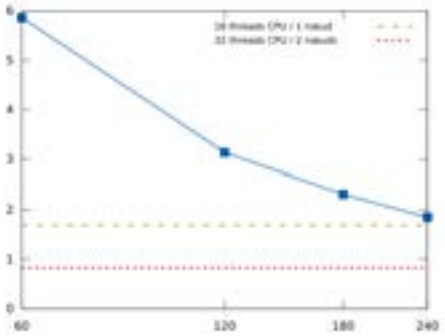
Les tests d'accélération ont été réalisés à problème constant sur l'image I3 et avec un nombre de cœurs maximal (60). La variation s'établit autour du nombre de threads par cœur en progressant de un à quatre. Les résultats obtenus sont visibles sur les graphiques 6.10.

À titre de comparaison avec les résultats obtenus sur CPU, les lignes en pointillés représentent les temps d'analyse sur 16 et 32 threads. De manière globale, nous constatons sur les trois étapes de calcul (a, b, d) une accélération de 1.9 entre 60 et 120 threads, ce qui se rapproche des performances constatées sur CPU. Le passage de 120 à 240 threads bénéficie d'une accélération moins grande avec une accélération de 1.75.
La mise à jour des *locations* est plus rapide sur Xeon Phi que sur CPU jusqu'à trois threads par cœur. L'utilisation du quatrième thread amène les temps à ceux observés pour 16 et 32 threads CPU.
Les temps de convergence suivent également cette tendance, avec cette fois une augmentation de 87% lors de l'utilisation du quatrième thread. Malgré ces deux augmentations, la vitesse de résolution sur 240 threads demeure la plus élevée (voir graphe f).
Il est intéressant de constater les bonnes performances du Xeon Phi sur l'étape de naissance, où les 240 sont plus rapides que les 32 threads CPU. Par ailleurs, cette étape est sur Xeon Phi plus rapide que celles de voisinage et de mort, alors que sur CPU, la naissance est deux fois plus coûteuse que les deux autres. Cela révèle la bonne vectorisation de cette étape et laisse entrevoir de possibles optimisations.

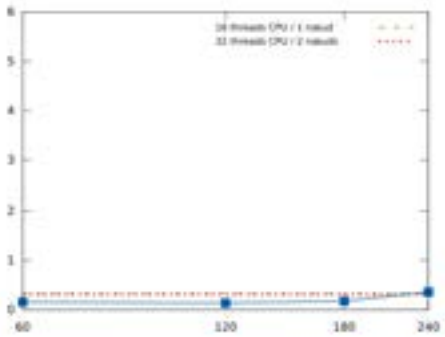
Figure 6.10: Temps par itération en secondes des tests d'accélération Xeon Phi sur l'image I3



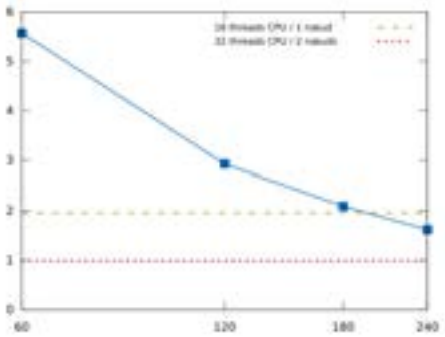
(a) Naissance



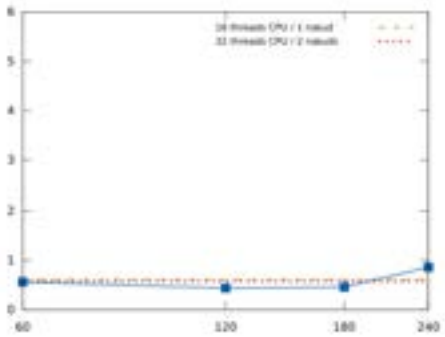
(b) Voisinage



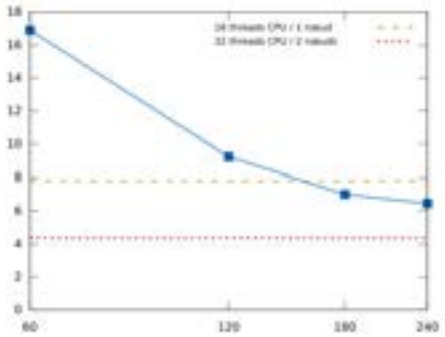
(c) Mises à jour *locations*



(d) Mort



(e) Convergence



(f) Total

6.6 Comparaison détection CPU / Xeon Phi

À partir des tests d'accélération effectués sur l'image I3, nous pouvons comparer les résultats obtenus sur Xeon Phi et CPU.

Th. CPU	1	2	4	8	16	32	64
Nb ellipses	**11901**	11958	**11971**	11922	11934	11956	11914
Nb itérations	**252**	261	265	255	258	**266**	258

Th. Xeon Phi	60	120	180	240
Nb ellipses	11922	**11891**	11922	**11934**
Nb itérations	254	**252**	**259**	257

Les résultats en gras représentent les valeurs minimales et maximales. Nous constatons que sur CPU, le nombre d'ellipses détectées varie entre 11901 et 11971. Ce qui représente un écart à la moyenne de 0,29%. Sur Xeon Phi, cet écart est de 0,18%. En comparant maintenant la moyenne des valeurs optimales sur CPU et Xeon Phi, nous obtenons 11936 ellipses sur CPU contre 11912,5 sur Xeon Phi. Ces valeurs sont extrêmement proches ce qui nous permet de considérer que le nombre de threads en jeu lors du calcul n'influence pour ainsi dire pas les résultats. Nous pouvons alors statuer que l'algorithme MPP_ORWL est reproductible à la fois sur un nombre variable de threads à la fois en mode partagé et en mode distribué, mais qu'également il y a reproductibilité entre les deux supports matériels CPU et Xeon Phi. Nous entendons ici par le terme reproductibilité, l'observation de phénomène semblables, nous ne statuons pas ici sur la répétabilité des expériences.

6.7 Qualité des résultats

L'analyse de l'image I1 (voir image 6.11) a permis d'identifier une problématique relative à la densité des noyaux sur l'image.

Prise dans sa globalité, une lame présente une grande disparité du point de vue de la répartition des ellipses (voir images 6.12).

L'image (a) montre une zone extraite du centre de la lame et particulièrement peuplée en noyaux. En haut, de nombreux faux positifs sont détectés. Au centre, en revanche, les

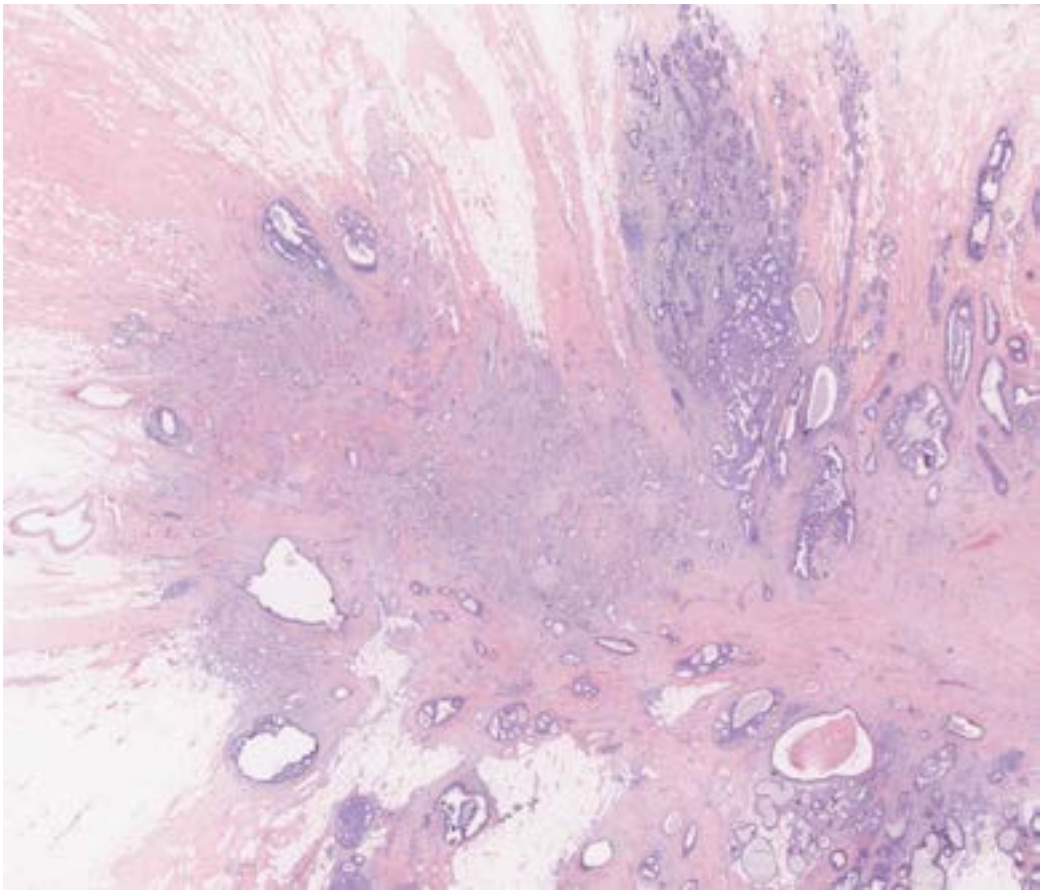
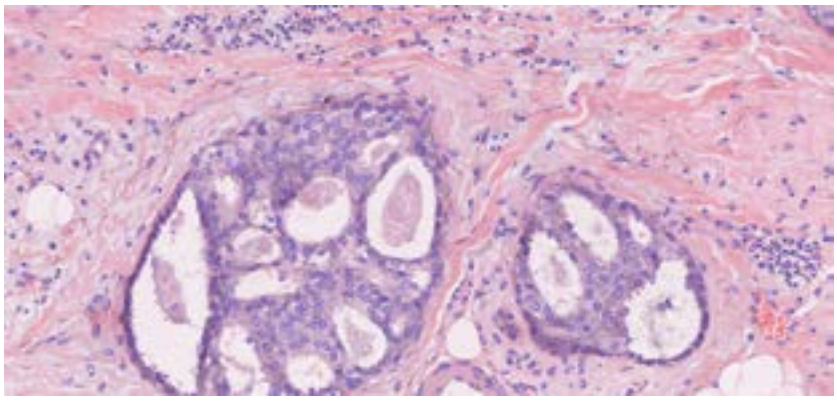
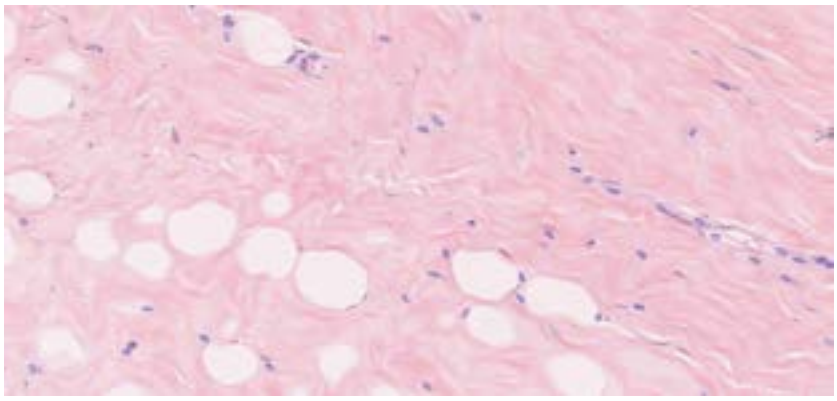


Figure 6.11: Vue d'ensemble de l'image I1

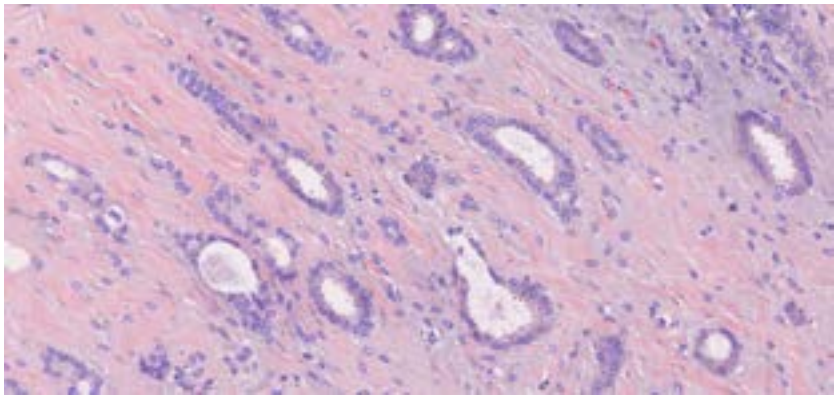
Figure 6.12: Agrandissement de l'image I1 analysée avec diverses densités de noyaux



(a) Forte concentration de noyaux



(b) Faible concentration de noyaux



(c) Concentration moyenne de noyaux

noyaux sont plutôt bien détectés avec cependant beaucoup de faux négatifs oubliés.
L'image (b) a été prélevée sur les bords et nous pouvons y constater une absence totale de noyaux. Quelques faux négatifs sont tout de même détectés par l'application. L'image (c) montre enfin une zone où la répartition est plus homogène. C'est le cas où la détection est la meilleure avec une bonne proportion de vrais positifs.
Cette image a été analysée par 64 threads CPU en 43 minutes. Un total de 78816 noyaux a été détecté en 306 itérations.

La combinaison des threads CPU et Xeon Phi permettrait d'abaisser cette durée. La qualité de la détection quant à elle bénéficierait de l'utilisation d'une carte des naissances plus précise.
En effet, outre les difficultés rencontrées pour produire ces dernières dans les résolutions les plus grandes, les tests utilisant le masque de détection ont produit des résultats similaires à ceux obtenus sans masque. C'est pourquoi cette fonctionnalité n'a pas été utilisée lors des tests.
Toutefois, l'utilité d'une telle carte est bien réelle. Avec une bonne détection des zones d'intérêt, les créations d'ellipses pourraient se concentrer sur les zones de fortes concentrations augmentant ainsi la qualité globale de l'algorithme.

7 Conclusion et perspectives

La parallélisation de l'algorithme MPP avec ORWL a permis de faire évoluer la méthode vers une solution extensible dont la multiplication du nombre de nœuds de calcul n'affecte pas les performances. En résolvant les difficultés inhérentes à la distribution et celles inhérentes à l'utilisation de données de grandes tailles, preuve a été faite de l'efficacité du modèle ORWL.

Les résultats obtenus sur CPU au chapitre 6 ont validé l'utilisation d'ORWL pour le traitement d'une image histologique GigaPixels avec la méthode MPP. Les temps de communications inhérents au modèle augmentent en fonction du volume des données mais de manière très peu significative par rapport aux temps de résolution. La courbe d'accélération suit une progression linéaire en réduisant de moitié les temps de résolution lorsque l'on double le nombre de cœurs. Ces résultats sont obtenus en mémoire partagée mais également lors du passage de la mémoire partagée vers la mémoire distribuée.

L'objectif initial de parallélisation d'une lame entière (I0) n'a pu être concrètement testé sur le cluster car le protocole de test prévoyait d'utiliser l'ensemble des 128 threads disponibles pour ce format cible. Or, nous avons montré l'inefficacité de l'hyperthreading sur l'application. Toutefois, l'analyse de ce format d'image sur 64 threads physiques n'est pas remise en cause d'une part, et d'autre part, les résultats obtenus sur 1, 2 et 4 nœuds nous permettent de prédire le comportement de l'application sur 8 nœuds.

Les tests d'accélération sur CPU et sur Xeon Phi, nous ont permis de statuer que l'intervalle de confiance de l'algorithme MPP_ORWL sur l'une ou l'autre architecture est le même, ce qui est encourageant dans l'optique d'assurer la portabilité de l'application. Sur ce même sujet, nous nous sommes efforcés de mettre à disposition presque intégralement le code de l'application MPP_ORWL. La lame de biopsie analysée a quant à elle été fournie par le laboratoire du LIB dans le cadre de ce travail et n'est donc pas divulguable.

Contrairement à ce que l'on peut intuitivement penser d'une expérience à caractère stochastique, telle que MPP, une reproduction à l'identique serait possible grâce à l'aspect déterministe des générateurs de nombres pseudo-aléatoires.

L'obtention de résultats rigoureusement identiques est toutefois soumise à de nombreuses contraintes (voir [42]) parmi lesquelles l'ordre des opérations peut conduire à des résultats faux [43]. Nous avons ainsi veillé à conserver la correction de l'algorithme séquentiel tout au long de la conception de la solution MPP_ORWL, notamment lors de la distribution des ellipses.

L'utilisation de la carte des naissances n'a pas apporté d'amélioration ni du point de vue de la qualité des détections ni des temps de calcul. Le mode de création basé sur une mesure statique d'attache aux données n'a pas fourni un support suffisamment précis pour orienter de manière efficace la création des ellipses. Toutefois, l'utilité d'une telle carte n'est pas remise en question car elle permettrait de focaliser la création des ellipses sur les zones d'intérêt.

Du point de vue de l'utilisation d'ORWL, nous avons démontré ses performances et sa flexibilité à travers la création d'un workflow de traitement spécifique à un domaine. Nous avons aussi mis en évidence une limitation système au niveau de l'insertion des *handlers* lors des tests sur Xeon Phi. Une évolution de l'implémentation avec un enregistrement par lots est à l'étude pour réduire le nombre de threads nécessaires aux requêtes.

L'une des difficultés techniques rencontrées a été l'utilisation du compilateur ICC qui s'est révélée être non conforme à la norme C11 exigée par ORWL. Des contournements ont cependant été trouvés et la compilation sur ICC est désormais fonctionnelle.

L'utilisation des cartes accélératrices Xeon Phi a demandé un fort investissement pour parer aux limitations du système d'exploitation embarqué. De plus, les tests de passage à l'échelle ont montré que la capacité mémoire effective de la carte (7Go) contraint à l'analyse d'image de format I3 (18840×17200).

L'ajout récent d'un module d'affinité à la librairie ORWL permet désormais d'optimiser la gestion mémoire d'un thread (voir [37]) associé à une tâche en le clouant sur un cœur physique. Les threads dévolus aux besognes de communication sont quant à eux mobiles. Une telle amélioration pourrait résoudre les problèmes constatés lors de l'utilisation de l'hyperthreading.

La prise en compte de ces fonctionnalités accélérerait le traitement de la lame entière et rendrait l'algorithme plus efficace. Le caractère extensible de cette solution permettrait par ailleurs de dimensionner les ressources en fonction du flux d'arrivée des données pour répondre au mieux aux besoins des médecins.

Références

[1] A. Madabhushi, “Digital pathology image analysis: opportunities and challenges,” 2009.

[2] M. N. Gurcan, L. E. Boucheron, A. Can, A. Madabhushi, N. Rajpoot, and B. Yener, “Histopathological image analysis: A review,” *Biomedical Engineering, IEEE Reviews in*, vol. 2, pp. 147 – 171, 2009.

[3] O. Sertel, G. Lozanski, A. Shana’ah, and M. N Gurcan, “Computer-aided detection of centroblasts for follicular lymphoma grading using adaptive likelihood-based cell segmentation,” *IEEE transactions on bio-medical engineering*, vol. 57, pp. 2613–6, 10 2010.

[4] M. A. Makary and M. Daniel, “Medical error—the third leading cause of death in the us,” *BMJ*, vol. 353, 2016. [Online]. Available: http://www.bmj.com/content/353/bmj.i2139

[5] World Health Organization, “GLOBALCAN 2012: Estimated Cancer Incidence, Mortality and Prevalence Worldwide in 2012,” Tech. Rep., 2012. [Online]. Available: http://globocan.iarc.fr/

[6] C. Genestie, “Mammary pathology,” Private Communication, Tech. Rep., May 2011, hôpital de la Pitié-Salpêtrière, Paris.

[7] J. Hinsinger, “Les colorations histologiques : colorations de routine et colorations spéciales,” Tech. Rep., 2015, université de Montréal. [Online]. Available: http://www.u999.u-psud.fr/images/stories/PDF/protocoles/Lescolorationshistologiques.pdf

[8] E. Rakha, J. Reis-Filho, F. Baehner, D. J Dabbs, T. Decker, V. Eusebi, S. Fox, S. Ichihara, J. Jacquemier, S. Lakhani, J. Palacios, A. L Richardson, S. J Schnitt, F. C Schmitt, P.-H. Tan, G. Tse, S. Badve, and I. Ellis, "Breast cancer prognostic classification in the molecular era: the role of histological grade." *Breast cancer research : BCR*, 2010.

[9] International Conference on Pattern Recognition, "Mitos-atypia-14," Tech. Rep., 2014. [Online]. Available: http://mitos-atypia-14.grand-challenge.org/

[10] Y. Liu, K. Gadepalli, M. Norouzi, G. E. Dahl, T. Kohlberger, A. Boyko, S. Venugopalan, A. Timofeev, P. Q. Nelson, G. Corrado, J. D. Hipp, L. Peng, and M. Stumpe, "Detecting cancer metastases on gigapixel pathology images," *CoRR*, vol. abs/1703.02442, 2017. [Online]. Available: http://arxiv.org/abs/1703.02442

[11] M. Kulikova, A. Veillard, L. Roux, and D. Racoceanu, "Nuclei extraction from histopathological images using a marked point process approach," *Proc SPIE*, vol. 8314, pp. 75–, 02 2012.

[12] X. Yang, H. Li, and X. Zhou, "Nuclei segmentation using marker-controlled watershed, tracking using mean-shift, and kalman filter in time-lapse microscopy," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 53, pp. 2405 – 2414, 12 2006.

[13] J.-R. Dalle, H. Li, C.-H. Huang, W. K. Leow, D. Racoceanu, and T. C. Putti, "Nuclear pleomorphism scoring by selective cell nuclei detection," 01 2009.

[14] K. Mosaliganti, L. Cooper, R. Sharp, R. Machiraju, G. Leone, K. Huang, and J. Saltz, "Reconstruction of cellular biological structures from optical microscopy data," *IEEE transactions on visualization and computer graphics*, vol. 14, pp. 863–76, 07 2008.

[15] C. Avenel, P. Fortin, and D. Béréziat, "Parallel birth and death process for cell nuclei extraction in histopathology images," *2013 42nd International Conference on Parallel Processing*, pp. 429–438, 2013.

[16] S. Descamps, X. Descombes, A. Béchet, and J. Zerubia, "Détection de flamants roses par processus ponctuels marqués pour l'estimation de la taille des populations," INRIA, Research Report RR-6328, 2007. [Online]. Available: https://hal.inria.fr/inria-00180811

[17] D. Racoceanu and F. Capron, "Towards semantic-driven high-content image analysis: An operational instantiation for mitosis detection in digital histopathology," *Computerized Medical Imaging and Graphics*, vol. 42, pp. 2–15, 2015.

[18] A. Veillard, M. S. Kulikova, and D. Racoceanu, "Cell nuclei extraction from breast cancer histopathology images using colour, texture, scale and shape information," *Diagnostic Pathology*, vol. 8, no. 1, p. S5, Sep 2013. [Online]. Available: https://doi.org/10.1186/1746-1596-8-S1-S5

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: http://science.sciencemag.org/content/220/4598/671

[20] K. Ghedira, *Optimisation combinatoire par métaheuristiques*, Technip, Ed. Technip, 2007.

[21] F. W. Crawford and M. A. Suchard, "Transition probabilities for general birth–death processes with applications in ecology, genetics, and evolution," *Journal of mathematical biology*, vol. 65, pp. 553–80, 2011.

[22] D. Li, "Construction de la mesure de Lebesgue," Jan. 2008. [Online]. Available: http://lefevre.perso.math.cnrs.fr/PagesPerso/enseignement/Archives/Lebesgue.pdf

[23] J.-F. Coeurjolly, "Introduction aux processus ponctuels de Gibbs : modélisation, identification et validation," Journées MAS et Journée en l'honneur de Jacques Neveu, INRIA Bordeaux - Sud-Ouest, Aug. 2010. [Online]. Available: https://hal.inria.fr/inria-00510351

[24] C. Avenel and M. Kulikova, "Marked point processes with simple and complex shape objects for cell nuclei extraction from breast cancer H&E images," in *SPIE Medical Image*, 2012.

[25] F. J. Aherne, N. A. Thacker, and P. I. Rockett, "The Bhattacharyya metric as an absolute similarity measure for frequency coded data," *Kybernetika*, vol. 34, no. 4, pp. 363–368, 1998.

[26] D. Salas, J. Gustedt, D. Racoceanu, and I. Perseil, "Resource-centered distributed processing of large histopathology images," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE)*, Aug 2016, pp. 367–370.

[27] B. J. Palmer and J. Nieplocha, "Efficient algorithms for ghost cell updates on two classes of mpp architectures," in *IASTED PDCS*, 2002.

[28] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[29] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLoP '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:9. [Online]. Available: http://doi.acm.org/10.1145/1953611.1953615

[30] J. Gustedt, S. Vialle, and P. Mercier, "Resource Centered Computing delivering high parallel performance," in *Heterogeneity in Computing Workshop (HCW 2014)*, ser. Heterogeneity in Computing Workshop (HCW 2014), workshop of 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014). Phenix, AZ, United States: IEEE, May 2014. [Online]. Available: https://hal.inria.fr/hal-00921128

[31] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[32] J. Gustedt, *Modern C*, n/a, Ed. Manning Publications, 2019.

[33] P.-F. Lavallee and P. Wautelet, "Programmation hybride mpi-openmp," Tech. Rep., Nov. 2015. [Online]. Available: http://www.idris.fr/media/formations/hybride/form_hybride.pdf

[34] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972. [Online]. Available: http://dx.doi.org/10.1109/TC.1972.5009071

[35] P.-N. Clauss and J. Gustedt, "Iterative Computations with Ordered Read-Write Locks," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010. [Online]. Available: https://hal.inria.fr/inria-00330024

[36] M. Saied, J. Gustedt, and G. Muller, "Automatic code generation for iterative multi-dimensional stencil computations," in *2016 IEEE 23rd International Conference on*

High Performance Computing (HiPC), vol. 00, Dec. 2017, pp. 280–289. [Online]. Available: doi.ieeecomputersociety.org/10.1109/HiPC.2016.040

[37] J. Gustedt, E. Jeannot, and F. Mansouri, "Automatic, Abstracted and Portable Topology-Aware Thread Placement," in *IEEE Cluster*, ser. Cluster Computing (CLUSTER), 2017 IEEE International Conference on, Hawaï, United States, Sep. 2017, pp. 389 – 399. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01621936

[38] H. Franke, R. Russell, and M. Kirkwood, "Fuss , futexes and furwocks : Fast user-level locking in linux," 2002.

[39] W. Wan Tsang and G. Marsaglia, "The Ziggurat method for generating random variables," *Journal of Statistical Software, Articles*, vol. 5, no. 8, pp. 1–7, 2000. [Online]. Available: https://www.jstatsoft.org/v005/i08

[40] A. Goode and M. Satyanarayanan, "A vendor-neutral library and viewer for whole-slide images, computer science department, carnegie mellon university," *Computer Science Department, Carnegie Mellon University, Technical Report CMU-CS-08-136*, Aug. June 2008.

[41] B. Claudel, G. Huard, and O. Richard, "TakTuk, Adaptive Deployment of Remote Executions," in *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC)*. Munich, Germany: ACM, 2009, pp. 91–100. [Online]. Available: https://hal.inria.fr/hal-00788923

[42] V. T. Dao, "High performance computing and stochastic simulation : Study of numerical reproducibility on multicore and manycore architectures," Theses, Université Clermont Auvergne, Mar. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01610378

[43] W. Gropp, "Issues in accurate and reliable use of parallel computing in numerical programs," *Accuracy and Reliability in Scientific Computing*, pp. 253–263, 2005.

A Annexes

A.1 Utilitaires disponibles sur le système linux BusyBox pour Xéon Phi

ar, ash, awk
basename, bunzip2, bzcat
cat, chattr, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, cut
date, dc, dd, deallocvt, df, diff, dirname, dmesg, dnsdomainname, du, dumpkmap, dumpleases
echo, egrep, env, expr
false, fbset, fdisk, fgrep, find, flock, free, fsck, fsck.minix, fuser
grep, groups, gunzip, gzip
halt, head, hexdump, hostname, hwclock
id, ifconfig, ifdown, ifup, insmod, ip
kill, killall, klogd
less, ln, loadfont, loadkmap, logger, logname, logread, losetup, ls, lsmod
md5sum, microcom, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, modprobe, more, mount, mv
nc, netstat, nohup, nslookup
od, openvt
patch, pidof, ping, pivot_root, poweroff, printf, ps, pwd
rdate, readlink, realpath, reboot, renice, reset, rm, rmdir, rmmod, route, run-parts
sed, seq, setconsole, sh, sleep, sort, start-stop-daemon, strings, stty, swapoff, swapon, switch_root, sync, sysctl, syslogd
tail, tar, tee, telnet, test, tftp, time, top, touch, tr, traceroute, true, tty
udhcpc, udhcpd, umount, uname, uniq, unzip, uptime, users, usleep
vi
watch, wc, wget, which, who, whoami
xargs
yes
zcat

A.2 Script de création d'un module sur Bright Cluster Manager

```
#%Module1.0
## App_name modulefile
##
proc ModulesHelp { } {
 puts stderr "\tAdds␣APP_NAME␣to␣your␣environment"
}
module-whatis   "Adds␣APP_NAME␣to␣your␣environment"
set             app_name_root        /cm/shared/apps/app_name
prepend-path     PATH              $app_name_root/bin
prepend-path     MANPATH           $app_name_root/share/man
prepend-path     LD_LIBRARY_PATH   $app_name_root/lib
prepend-path     LIBRARY_PATH      $app_name_root/lib64
prepend-path     CPATH             $app_name_root/include
```

A.3 Support C11 du compilateur Icc

Table A.1: Fonctionnalités C11 prises en charge par le compilateur Intel ICC. Source : https://software.intel.com/en-us/articles/c11-support-in-intel-c-compiler

Fonctionnalité C11	V15.0	V16.0	V17.0	V18.0
Static Assert	Yes	Yes	Yes	Yes
Typedef redefinition	-	Yes	Yes	Yes
Anonymous struct and union	-	Yes	Yes	Yes
New macros in float.h	-	Yes	Yes	Yes
`_Alignas, _Alignof`	-	Yes	Yes	Yes
`_Atomic`	-	-	-	Yes
Atomic operations	-	Yes	Yes	Yes
Macros for Complex values	-	Yes	Yes	Yes
Unicode strings	-	Yes	Yes	Yes
uchar.h	-	-	-	-
`_Noreturn`,stdnoreturn.h	-	Yes	Yes	Yes
`_Generic`	-	Yes	Yes	Yes
`__STDC_VERSION__` == 201112L	-	Yes	Yes	Yes
`_thread_local`	-	Yes	Yes	Yes

A.4 Liste des types atomiques non pris en charge dans ICC 16.0

- `atomic_char16_t`
- `atomic_char32_t`
- `atomic_int_least8_t`
- `atomic_uint_least8_t`
- `atomic_int_least16_t`
- `atomic_uint_least16_t`
- `atomic_int_least32_t`
- `atomic_uint_least32_t`
- `atomic_int_least64_t`
- `atomic_uint_least64_t`
- `atomic_int_fast8_t`
- `atomic_uint_fast8_t`
- `atomic_int_fast16_t`
- `atomic_uint_fast16_t`
- `atomic_int_fast32_t`
- `atomic_uint_fast32_t`
- `atomic_int_fast64_t`
- `atomic_uint_fast64_t`
- `atomic_intptr_t`
- `atomic_uintptr_t`
- `atomic_size_t`
- `atomic_ptrdiff_t`
- `atomic_intmax_t`
- `atomic_uintmax_t`

A.5 Fonction de vérification des nombres premiers

Algorithm 5: Vérification d'un nombre premier

Input: $nTask$
Output: true **or** false
$ret \leftarrow false$
$num \leftarrow nTask$
$numSqrt \leftarrow \lfloor \sqrt{nTask} \rfloor$
$nMaxFactor \leftarrow \lfloor log_2(nTask) \rfloor$
$FACTOR[nMaxFactor]$
$nFactor \leftarrow 0$
while $num \mod 2 = 0$ **do**
  $num \leftarrow num/2$
  $FACTOR[nFact++] \leftarrow num$
end while
for $i := 3$ **to** $numSqrt$ **step** 2 **do**
  while $num \mod i = 0$ **do**
    $num \leftarrow num/i$
    $FACTOR[nFact++] \leftarrow num$
  end while
end for
if $num = nTask$ **then**
  $ret \leftarrow true$
end if
return ret

A.6 Justification de l'implémentation MPP d'une répartition selon la loi de Poisson

Notons k le k^{ieme} pixel de l'image et Y_k la variable aléatoire valant 1 si le pixel reçoit une image, 0 sinon.
Considérons X_k, un nombre aléatoire suivant une loi uniforme sur [0;1] et p la probabilité d'apparition d'une ellipse sur un pixel.
Nous savons que la probabilité que X_k soit inférieur à p est de p :

$$P(0 < X_k < p) = p \tag{A.1}$$

En posant :

$$\begin{cases} 0 < X_n < p & \Rightarrow Y_k = 1 \\ p \leq X_n < 1 & \Rightarrow Y_k = 0 \end{cases} \tag{A.2}$$

Nous pouvons dire que Y_k est assimilable à une variable suivant une loi de Bernouilli de paramètre p.
En répétant sur les n pixels de l'images, n tirages aléatoires et indépendants, la loi de Bernouilli devient une loi binomiale de paramètres (n,p).
Or, avec $n > 30$ et $p < 0,1$, nous pouvons conclure que cette loi binomiale est assimilable à une loi de Poisson de paramètre $\lambda = np$.

A.7 Interprétation du fichier "tiles" contenant les informations sur les tuiles d'image

```
struct size_str {
 size_t width;
 size_t height;
 size_t dim;
};

struct tile_str {
 size_t tile_width;
 size_t tile_height;
 size_t tile_dim;
 char tile_path[512];
};

static inline
void loadTileFile(char* fileName, size_t nTasks,
                  tile_str * tiles, size_str * size) {
 FILE* size_file = fopen(fileName, "rb");
 if (size_file) {
  size_t nTile = 0;
  char line[1024];
  const char dem[2] = "␣";
  while (fgets(line, 1024, size_file)) {
   if (line[0] == '#') {
    size_t ind = 0;
    char* token;
    token = strtok(line, dem);
    while (token) {
     switch (ind) {
      case 1: size->width = strtoull(token, 0, 10);
      break;
      case 2: size->height = strtoull(token, 0, 10);
      break;
     }
     token = strtok(NULL, dem);
     ind++;
    }
   } else {
    size_t ind = 0;
    char* token;
    token = strtok(line, dem);
    while (token) {
     if (token[strlen(token) - 1] == '\n')
      token[strlen(token) - 1] = 0;
     switch (ind) {
      case tile_path :
      strcpy(tiles[nTile].tile_path, token);
      break;
```

```
      case tile_width :
      tiles[nTile].tile_width = strtoull(token, 0, 10);
      break;
      case tile_height :
      tiles[nTile].tile_height = strtoull(token, 0, 10);
      break;
      case tile_dim :
      tiles[nTile].tile_dim = strtoull(token, 0, 10);
      break;
     }
     token = strtok(NULL, dem);
     ind++;
    }
    if (nTile == nTasks) {
     fprintf(stderr, "Too much Tiles\n");
     break;
    }
    nTile++;
   }
  }
 }
 fclose(size_file);
}
```

A.8 Pre-script exécuté avant le lancement de l'application MPP_ORWL

```
#!/usr/bin/ruby
require 'optparse'
require 'ostruct'
#Option parser
$options = OpenStruct.new
opt_parser = OptionParser.new do |opts|
opts.banner = "Usage: orwlrun [$options]"
 opts.on("--r_min N", Integer,) do |n| $options.r_min = n end
 opts.on("--r_max N", Integer,) do |n| $options.r_max = n end
 opts.on("--proc N", Integer,) do |n| $options.proc = n end
 opts.on("--task N", Integer,) do |n| $options.task = n end
 opts.on("--inimg S", String,) do |s| $options.inimg = s end
 opts.on("--exec S", String,) do |s| $options.exec = s end
 opts.on("--loop N", Integer,) do |n| $options.loop = n end
 opts.on("--verb B", Integer,) do |b| $options.verb = b end
 opts.on("--cnv C", Integer,) do |c| $options.cnv = c end
end

opt_parser.parse!(ARGV)

INPUT_DIR = Dir.mktmpdir("mpp_output")
BIN = Dir.chdir("../bin")
BIN_DIR = Dir.pwd
IMG_NAME = /^.*\/(.*)\..*$/.match($options.input_image)[1]
COMPILER = /^.*\/(.*)$/.match(ENV['CC'])[1]
TILES_FILE = "tiles"
node_list = ""
$nodes3.each { |m|
 node_list += " -m #{m} "
}

# SPLIT
cmd1 = "
#{BIN_DIR}/split_#{COMPILER} #{NBPROC} #{NBTASKS}
#{$options.r_max} #{$options.input_image}
#{TILES_FILE} #{INPUT_DIR};"
system cmd1;

# CREATE INPUT DIR
cmd2 = "taktuk #{node_list} broadcast
exec [ 'mkdir #{INPUT_DIR}.$TAKTUK_RANK' ]"
system cmd2

# DISTRIBUTE TILES FILE
cmd3 = "taktuk #{node_list} synchronize broadcast
put [ '#{INPUT_DIR}/#{TILES_FILE}' ]
[ '#{INPUT_DIR}.$TAKTUK_RANK' ]"
```

```
system cmd3

# TILES
NBPROC.times do |i|
 cmd4 = "taktuk␣-m␣" << $nodes3[i].to_s << "␣"
 NBTASKS.times do |j|
  file_name = IMG_NAME + "-#{i}-#{j}.bin"
  cmd4 << "synchronize␣broadcast
␣␣put␣[␣'#{INPUT_DIR}/#{file_name}'␣]
␣␣[␣'#{INPUT_DIR}.$((#{i}␣+␣1))'␣]␣\\;"
 end
 cmd4 << "synchronize␣wait␣reduce␣target␣all␣\\;"
 system cmd4
end

# ARGS FOR APP
$argv.clear
$argv.push($options.exec)
$argv.push("--r_max␣#{$options.r_max}")
$argv.push("--r_min␣#{$options.r_min}")
$argv.push("--loop␣#{$options.loop}")
$argv.push("
--tiles_file␣#{INPUT_DIR}.$TAKTUK_RANK/#{TILES_FILE}")
$argv.push("--tmp_dir␣#{INPUT_DIR}.$TAKTUK_RANK")
$argv.push("--cnv␣#{$options.cnv}")
```

A.9 Liste d'arguments de l'application MPP_ORWL

```
struct arguments
{
  // MPP
  char * input_mask;
  size_t cnv;
  float d0;
  float alpha;
  float beta;
  float logamma;
  float delta;
  float decrease;
  float Temp;
  float decreaseT;
  float r_min;
  float r_max;
  bool verbose;
  // ORWL
  char* tmp_dir;
  char* tiles_file;
  char* filter_flag;
  size_t loop;
};
```

A.10 Calcul de la position d'une tâche

```
inline
size_str get_task_offset(tile_str *tiles, size_str grid,
                         size_t tid, size_t rmax) {
 size_t y = tid / grid.width;
 size_t x = tid - (y * grid.width);
 size_str size = (size_str){ 0, };
 for (size_t i = 0; i < y; i++) {
  size.height += tiles[i * grid.width].tile_height - 2 * rmax;
 }
 for (size_t j = 0; j < x; j++) {
  size.width += tiles[j].tile_width - 2 * rmax;
 }
 return size;
}
```

A.11 Mise à jour locale des données

```
if (bNorth) {
 ORWL_SECTION(&nw_hdl) {
  if (!IT) {
   orwl_truncate(&nw_hdl, sizeof(float[xsize]));
  }
  float *nw_hdl_ptr = orwl_write_map(&nw_hdl);
  memcpy(nw_hdl_ptr, &cover_map[0], xsize * sizeof (float));
 }
}
if (bSouth) {
 ORWL_SECTION(&sw_hdl) {
  if (!IT) {
   orwl_truncate(&sw_hdl, sizeof(float[xsize]));
  }
  float *sw_hdl_ptr = orwl_write_map(&sw_hdl);
  memcpy(sw_hdl_ptr, &cover_map[(width * height) - (xsize)],
         xsize * sizeof (float));
 }
}
if (bEast) {
 ORWL_SECTION(&ew_hdl) {
  if (!IT) {
   orwl_truncate(&ew_hdl, sizeof(float[ysize]));
  }
  float *ew_hdl_ptr = orwl_write_map(&ew_hdl);
  for (size_t i = 0; i < height; i++) {
   memcpy(&ew_hdl_ptr[rmax * 2 * i],
          &cover_map[(width * (1 + i)) - (rmax * 2)],
          rmax * 2 * sizeof (float));
  }
 }
}
if (bWest) {
 ORWL_SECTION(&ww_hdl) {
  if (!IT) {
   orwl_truncate(&ww_hdl, sizeof(float[ysize]));
  }
  float *ww_hdl_ptr = orwl_write_map(&ww_hdl);
  for (size_t i = 0; i < height; i++) {
   memcpy(&ww_hdl_ptr[rmax * 2 * i],
          &cover_map[width * i], rmax * 2 * sizeof (float));
  }
 }
}
```

A.12 Mise à jour globale des données

```
size_t offset_loc = 0, offset_map = 0;
//
if (bNorth) {
 ORWL_SECTION(&nr_hdl) {
  float *nr_hdl_ptr = orwl_read_map(&nr_hdl);
  for (size_t i = 0; i < rmax * 2; i++) {
   offset_loc = i * width;
   for (size_t j = 0; j < width; j++) {
    if (nr_hdl_ptr[offset_loc + j] < cover_map[offset_loc + j]) {
     cover_map[offset_loc + j] =
     nr_hdl_ptr[offset_loc + j];
    }
   }
  }
 }
}
if (bSouth) {
 ORWL_SECTION(&sr_hdl) {
  float *sr_hdl_ptr = orwl_read_map(&sr_hdl);
  for (size_t i = 0; i < rmax * 2; i++) {
   offset_loc = i * width;
   offset_map = width * (i + height) - xsize;
   for (size_t j = 0; j < width; j++) {
    if (sr_hdl_ptr[offset_loc + j] < cover_map[offset_map + j]) {
     cover_map[offset_map + j] = sr_hdl_ptr[offset_loc + j];
    }
   }
  }
 }
}
if (bEast) {
 ORWL_SECTION(&er_hdl) {
  float *er_hdl_ptr = orwl_read_map(&er_hdl);
  for (size_t i = 0; i < height; i++) {
   offset_loc = i * rmax * 2;
   offset_map = (width * (i + 1)) - (rmax * 2);
   for (size_t j = 0; j < rmax * 2; j++) {
    if (er_hdl_ptr[offset_loc + j] <
     cover_map[offset_map + j]) {
     cover_map[offset_map + j] =
     er_hdl_ptr[offset_loc + j];
    }
   }
  }
 }
}
if (bWest) {
```

```
 ORWL_SECTION(&wr_hdl) {
  float *wr_hdl_ptr = orwl_read_map(&wr_hdl);
  for (size_t i = 0; i < height; i++) {
   offset_loc = i * rmax * 2;
   offset_map = i * width;
   for (size_t j = 0; j < rmax * 2; j++) {
    if (wr_hdl_ptr[offset_loc + j] <
     cover_map[offset_map + j]) {
     cover_map[offset_map + j] =
     wr_hdl_ptr[offset_loc + j];
    }
   }
  }
 }
}
```

A.12.1 Concaténation des résultas depuis le post-script

```
def merge(suffix)
 merge_file_name = "#{OUTPUT_PATH}/#{NEW_ID}/#{IMG_NAME}.#{suffix}"
 t_it = t_el = t_ti = 0
 pathtofiles = File.join("#{OUTPUT_PATH}/#{NEW_ID}", "*")
 files = Dir[pathtofiles]
 typefile = files.select { |file| /(.+\.#{suffix})$/ =~ file }
 puts typefile
 File.open(merge_file_name, 'w') do |w|
  typefile.each do |file|
   File.foreach(file) do |line|
    if line.include? '#'
     res = line.split(/[^0-9\.]/)
     res.delete("")
     t_it = res[0].to_i > t_it ? res[0].to_i : t_it
     t_el += res[1].to_i
     t_ti = res[2].to_f > t_ti ? res[2].to_f : t_ti
    else
     w.puts line
    end
   end
  end
 end
end
```

A.12.2 *Écriture des ellipses détectées dans des fichiers textes*

```
static inline
 void writeData(elps_str* elps_ptr, char *output_name, size_t n,
                size_t IT, float T, size_str offset, size_t width)
                   {
 char *ell_name = (char*)malloc(sizeof(char[256]));
 char *csv_name = (char*)malloc(sizeof(char[256]));
 sprintf(ell_name, "%s.ellipses", output_name);
 sprintf(csv_name, "%s.csv", output_name);
 FILE *output_file = fopen(ell_name, "w");
 FILE *csv_file = fopen(csv_name, "w");
 fprintf(output_file, "# IT:%zu, ELL:%zu, TIME:%.4f\n", IT, n, T);
 size_t x, y;
 for (size_t i = 0; i < n; i++) {
  x = elps_ptr[i].x + offset.width;
  y = elps_ptr[i].y + offset.height;
  fprintf(output_file, "%zu,%zu,%f,%f,%f,%.2f\n", x, y,
          elps_ptr[i].ax, elps_ptr[i].bx, elps_ptr[i].angle,
             elps_ptr[i].Ud);
  print_ellipse(csv_file, elps_ptr[i], offset);
 }
 fclose(output_file);
 fclose(csv_file);
 free(ell_name);
 free(csv_name);
}
```

A.12.3 *Collage de la carte des naissances*

```
#!/bin/bash

WIDTH=$1
HEIGHT=$2
INPUT_DIR=$3
IMG_NAME=$4
OUTPUT_DIR=$5
OUTPUT_NAME="$OUTPUT_DIR$IMG_NAME.birth.jpg"
V=""

for ((i=0; i<$WIDTH; i++)); do
 H=""
 for ((j=0; j<$HEIGHT; j++)); do
  NAME="$INPUT_DIR/$IMG_NAME-$i-$j.birth.jpg"
  H="$H $NAME"
 done
 HNAME="$IMG_NAME-$i.birth.jpg"
 convert $H -define jpeg:optimize-coding=false +append $HNAME
```

```
 V="$V␣$HNAME"
done

convert $V -define jpeg:optimize-coding=false -append $OUTPUT_NAME
```

A.13 Modifications apportées à la bibliothèque ORWL

A.13.1 p99

```
diff --git a/p99/p99_atomic.h b/p99/p99_atomic.h
--- a/p99/p99_atomic.h
+++ b/p99/p99_atomic.h
@@ -399,10 +399,16 @@ typedef enum memory_order memory_order;
-__typeof__(P99_GENERIC_SIZE_LIT(sizeof(T), (uintptr_t){ 0 },
    P00_UINT_TYPE_LIST))
+__typeof__(P99_GENERIC_SIZE_LIT(sizeof(T), *(uintptr_t*)0,
    P00_UINT_TYPE_LIST))
@@ -734,14 +740,16 @@ P99_DECLARE_ATOMIC(long double _Complex,
    atomic_cldouble);
- ((T){ 0 },
+ (*(T*)0,
P99_GENERIC_SIZE_LIT
(sizeof(T)+1,
-    (struct P99_PASTE3(p99_atomic_, T, _struct)*){ 0 },
+    *(struct P99_PASTE3(p99_atomic_, T, _struct)**)0,

diff --git a/p99/p99_compiler.h b/p99/p99_compiler.h
index 348e17f..d9bbd1d 100644
--- a/p99/p99_compiler.h
+++ b/p99/p99_compiler.h
@@ -186,6 +186,12 @@
#  define __GNUC_MINOR__ 2
#  undef __GNUC_PATCHLEVEL__
#  define __GNUC_PATCHLEVEL__ 0
+# elif (__ICC < 1800)
+#  define p00_has_feature_c_generic_selections 0
# endif
#endif

@@ -316,7 +322,7 @@ signed p00_trailing_comma_in_initializer__(void
    ) {
# if P99_GCC_VERSION >= 40600UL
#  define p00_has_feature_c_static_assert 1
-#  define p00_has_attribute_externally_visible 1
+#  define p00_has_attribute_externally_visible 0
# endif
```

```
@@ -331,8 +337,12 @@ signed p00_trailing_comma_in_initializer__(
   void) {
#  define p00_has_feature_stdalign_h 1
#  if __STDC_VERSION__ > 201100L
#   define p00_has_feature_c_max_align_t 1
-#   define __STDC_NO_ATOMICS__ 1
-#   define p00_has_feature_c_generic_selections 0
+#   ifndef __STDC_NO_ATOMICS__
+#    define __STDC_NO_ATOMICS__ 1
+#   endif
+#   ifndef p00_has_feature_c_generic_selections
+#    define p00_has_feature_c_generic_selections 0
+#   endif

@@ -344,7 +354,6 @@ signed p00_trailing_comma_in_initializer__(void
   ) {
#if P99_COMPILER & P99_COMPILER_INTEL
-# define p99_inline __attribute__((__weak__,__always_inline__))
+# define p99_inline __attribute__((__weak__,__always_inline__))
   inline

@@ -402,7 +414,7 @@ signed p00_trailing_comma_in_initializer__(void
   ) {
# else
/* Intel now has a file stdatomic.h, but this is still quite
incomplete and not yet worth it. */
-//#  define p00_has_feature_stdatomic_h 1
+#  define p00_has_feature_stdatomic_h 0
# endif

diff --git a/p99/p99_generic.h b/p99/p99_generic.h
index 43035ac..042e455 100644
--- a/p99/p99_generic.h
+++ b/p99/p99_generic.h
@@ -28,7 +28,8 @@
-#define P00_GENERIC_LIT_(T, EXP) (EXP){ 0 }
+#define P00_GENERIC_LIT_(T, EXP) *(EXP*)0

diff --git a/p99/p99_libc.h b/p99/p99_libc.h
index d3586b8..e34b118 100644
--- a/p99/p99_libc.h
+++ b/p99/p99_libc.h
@@ -187,7 +187,7 @@ typedef size_t rsize_t;
#if __STDC_VERSION__ > 201100L
# define p00_has_feature_stdalign_h 1
# define p00_has_feature_stdnoreturn_h 1
-# ifndef  __STDC_NO_ATOMICS__
+# if !defined(__STDC_NO_ATOMICS__)
```

```
diff --git a/p99/p99_qsort.h b/p99/p99_qsort.h
index e26e931..803acda 100644
--- a/p99/p99_qsort.h
+++ b/p99/p99_qsort.h
@@ -420,7 +420,7 @@ errno_t p00_qsort_s(void *p00_base,
-              )((B), (N), alignof(*(B)), (S), (CMP), (CTX)),
+              )((B), (N), alignof(typeof(*(B))), (S), (CMP), (CTX)
   ),

@@ -453,7 +453,7 @@ errno_t p00_qsort_s(void *p00_base,
-              )((B), (N), alignof(*(B)), (S), (CMP), (CTX)),
+              )((B), (N), alignof(typeof(*(B))), (S), (CMP), (CTX)
   ),
```

A.13.2 ORWL

```
diff --git a/lib/p99_tp.c b/lib/p99_tp.c
new file mode 100644
index 0000000..473014d
--- /dev/null
+++ b/lib/p99_tp.c
@@ -0,0 +1,6 @@
+#include "p99_compiler.h"
+#include "p99_tss.h"
+#include "p99_tp.h"
+
+p00_tp_glue p00_tp_get(register p99_tp volatile*const p00_tp);
+p00_tp_glue p99_tp_xchg(p99_tp volatile* p00_tp, void* p00_val);
```

Daniel SALAS

Parallélisation hybride d'une application de détection de noyaux cellulaires

Résumé

L'algorithme Marked Point Process parallélisé en mémoire partagée permet d'accélérer la détection de noyaux cellulaires. Cependant, les limites imposées par le nombre de cœurs CPU ou la capacité mémoire des cartes GPU ne permet pas d'analyser une image histologique entière (50 000 $\times$ 50 000 pixels).
Pour y parvenir, nous proposons d'ajouter une dimension distribuée à cette parallélisation en utilisant le modèle hybride Ordered Read-Write Locks.
Afin de garantir la validité de l'algorithme original, nous avons mis en place différentes stratégies pour d'une part garantir le traitement de l'ensemble des noyaux et d'autre part, pour que les traitements locaux soient considérés à l'échelle globale.
Les tests menés ont tout d'abord permis de valider le passage à l'échelle de l'application puis d'afficher un facteur d'accélération de 40 sur 64 cœurs CPU.

Mots-clés : Calcul parallèle hybride, imagerie médicale, détection automatique de cancer du sein

Abstract

The shared memory parallelized version of the Marked Point Process algorithm allows to speed-up cells nuclei detection. However, the limitations imposed by the number of CPU cores or the memory capacity of GPU cards do not allow to analyse an entire histological image (50,000 $\times$ 50,000 pixels).
To achieve this, we propose to add a distributed dimension to this parallelization using the hybrid Ordered Read-Write Locks model.
In order to guarantee the validity of the original algorithm, we have implemented different strategies to ensure that all nuclei are processed and that local processing is considered on a global scale.
The tests carried out first validated the scalability of the application and then showed an acceleration factor of 40 on 64 CPU cores.

Keywords : Hybrid parallel computing, medical imaging, automatic breast cancer detection