Université de Strasbourg
Laboratoire ICube
Équipe ICPS

UNIVERSITÉ DE STRASBOURG

ÉCOLE DOCTORALE 269

UMR 7357

msii ÉCOLE DOCTORALE

THÈSE présentée par

Paul Godard

soutenue le **16 décembre 2019**

Pour obtenir le grade de : **Docteur de l'université de Strasbourg**

Spécialité : Informatique

Parallélisation et passage à l'échelle durable d'une chaîne de traitement graphique pour l'impression professionnelle

Thèse dirigée par :

Directeur	Cédric Bastoul	Professeur, Université de Strasbourg
Co-encadrant	Vincent Loechner	Maître de Conférences, Université de Strasbourg
Co-encadrant	Patrick Zimmermann	Directeur Solutions & Intégrations, Caldera

Jury :

Présidente	Cristel Pelsser	Professeur, Université de Strasbourg
Rapporteur	Henri-Pierre Charles	Directeur de recherches, CEA List
Rapporteur	Erven Rohou	Directeur de recherches, Inria
Examinatrice	Élisabeth Brunet	Maître de Conférences, Télécom SudParis
Examinateur	Fabien Coelho	Professeur, Mines ParisTech

Résumé

Les nombreuses avancées du vaste domaine de l'impression professionnelle ont permis la multiplication des objets imprimés dans nos quotidiens. Désormais, la flexibilité introduite par les procédés d'impression numérique promet d'associer les souhaits de personnalisation avec les avantages de la production de masse. La rapide évolution des usages et des technologies, caractérisée par des fermes d'impression toujours plus grandes et des presses numériques toujours plus rapides, pose des problèmes inédits aux systèmes informatiques actuels qui pilotent les imprimantes. Dans cette thèse, nous explorons de nouvelles techniques inspirées des systèmes de calcul haute performance afin d'accélérer l'exécution des traitements graphiques indispensables à l'impression numérique. Nous introduisons pour cela une architecture de calculs distribués flexible exploitant des techniques de traitement et de synchronisation optimisées. Nous détaillons les principes de fonctionnement et les subtilités de l'implémentation de nos travaux qui permettent de respecter les contraintes spécifiques des flux de données produits. Nous réalisons une évaluation complète de notre solution qui y démontre ses excellentes performances et sa viabilité.

Mots clés : HPC, systèmes distribués, flux de données contraints, ordonnancement, transposition et rotation de matrices out-of-core, impression numérique.

Abstract

The strong and continuous improvements in the professional printing field have led to the ubiquity of printed objects in our daily life. The flexibility introduced by the digital printing process promises to associate extensive customization with mass production. The quick growth of printing usages and technologies, illustrated by wider printer farms and faster digital presses, leads to original challenges for the computer system in charge of driving them. In this thesis, we explore new approaches inspired by the high performance computing field to speedup the graphics processing necessary to digital printing. To achieve this goal, we introduce a distributed system which provides the adequate flexibility and performance by exploiting and optimizing both processing and synchronization techniques. We present our architecture up to the subtle parts of its implementation which allows our solution to meet the specific constraints on generating streams for printing purpose. We perform a complete evaluation of our solution and provide experimental evidence of its great performance and viability.

Keywords: HPC, distributed systems, constrained data stream, scheduling, out-of-core matrices transposition and rotation, digital printing.

Remerciements

Un manuscrit de thèse ne serait pas complet sans l'expression sincère des nombreux remerciements que sa concrétisation implique.

Naturellement, mes premiers remerciements s'adressent à mes deux encadrants académiques sans qui ce projet n'aurait pas été réalisable : Cédric Bastoul pour son suivi bienveillant et ses conseils extrêmement précieux malgré son emploi du temps débordant ; et Vincent Loechner pour sa disponibilité, son expertise et ses conseils tout aussi avisés que pertinents.

À travers eux, j'en profite pour remercier l'ensemble du corps enseignant de l'Université de Strasbourg qui m'a transmis de nombreux savoirs durant ces 8 dernières années et qui m'a donné le goût et les occasions de transformer mon choix « d'études courtes » en un désir « d'études longues ».

D'autre part, je tiens à remercier chaleureusement le laboratoire ICube et tout particulièrement les membres de l'équipe ICPS pour m'avoir offert « un second bureau », mais également pour leur accueil, leur camaraderie et les nombreuses discussions intéressantes lors des départs à midi pile pour le Restaurant Universitaire.

Évidemment, j'adresse un remerciement tout particulier à l'entreprise Caldera qui m'a donné la possibilité de prolonger mes études à travers le monde professionnel par cet ambitieux projet. Je remercie spécialement Frédéric Soulier de m'avoir fait confiance pour démarrer la thèse et Patrick Zimmermann de m'avoir permis de la terminer dans les meilleures conditions.

Enfin, comment ne pas remercier toutes celles et ceux qui m'ont soutenu et encouragé depuis bien avant la thèse : ma famille, mes amis et mes collègues. Merci Maman, Papa, frangin, Laura, Arnaud, Jonathan, Ludovic, Robin, et tant d'autres.

Table des matières

Notations

Ce manuscrit porte un soin particulier à la précision des unités indiquées. Par conséquent, un usage consciencieux est fait des préfixes décimaux du Système International (SI) et des préfixes binaires de la Commission Électrotechnique Internationale (CEI), conformément aux usages et technologies sous-jacentes : les premiers, issus des valeurs multiples de 1000, sont privilégiés pour quantifier les échanges réseau ; les seconds, multiples de 1024, sont privilégiés pour quantifier les espaces mémoire. Le tableau **0.1** rappelle les noms et les symboles associés à ces deux types de préfixes, ainsi que les écarts de valeurs.

Tableau 0.1 : Liste des préfixes décimaux et binaires utilisés dans ce manuscrit

Préfixes décimaux (SI)			**Préfixes binaires (CEI)**			**Écarts**
Valeur	**Nom**	**Symbole**	**Valeur**	**Nom**	**Symbole**	$\frac{CEI-SI}{SI}$
10^3	kilo	k	2^{10}	kibi	K	2 %
10^6	méga	M	2^{20}	mébi	Mi	5 %
10^9	giga	G	2^{30}	gibi	Gi	7 %
10^{12}	téra	T	2^{40}	tébi	Ti	10 %
10^{15}	péta	P	2^{50}	pébi	Pi	13 %

Malgré l'utilisation du français dans ce manuscrit, l'usage de quelques termes anglais a volontairement été conservé. Ils n'ont, soit pas de de traduction officielle, soit leur usage serait inhabituel et pourrait être ambigu. Par exemple les termes : *switch* pour désigner les commutateurs réseau ; *cluster* pour les grappes d'ordinateurs ; et *big data* pour les « mégadonnées » ont été préférés à leur équivalent français.

Enfin, ce manuscrit indique de nombreuses références vers les pages de documentation des systèmes Linux (pages « man »). Ces références se présentent sous la forme « `page(x)` » où `x` indique le numéro de la section concernée. Toutes les pages de documentations mentionnées dans ce manuscrit sont consultables entre autres à cette adresse : `http://man7.org/linux/man-pages/`

1 Introduction

Devenus omniprésents dans nos quotidiens, rares sont les objets qui nous entourent à ne pas être composés d'une partie imprimée, que ce soit : leurs emballages, leurs étiquettes, leurs faces, ou encore les objets eux-mêmes. Initialement inventées avant l'ère commune pour répéter des motifs de décoration puis utilisées pour améliorer la copie d'œuvres religieuses, culturelles ou artistiques, les techniques d'impression ont évolué au fil des siècles jusqu'à devenir la solution privilégiée pour transcrire du texte, colorer une surface, personnaliser un contenu ou créer un objet complexe. Notre environnement journalier s'est ainsi peuplé d'une multitude d'objets ayant bénéficié de l'impression durant leur conception ou leur commercialisation, tels que : livres, affiches, textiles, revêtements ou encore les circuits électroniques communément appelés « circuits imprimés ». Pour permettre de nouvelles avancées dans les usages de l'impression numérique, cette thèse propose de nouvelles approches pour rendre les logiciels spécifiques à ce domaine plus rapides et plus flexibles.

Poussée par les nombreuses évolutions et innovations de l'automatisation et de l'industrialisation des moyens de production, le domaine de l'impression numérique s'est développée jusqu'à offrir une large palette d'utilisations qui requièrent une variété d'exigences. Premièrement, la qualité, qui s'étend de l'impression de notices d'utilisation en encre noire sur papier grossier jusqu'à l'impression d'albums photos saisissants en mélangeant plusieurs encres différentes sur papier brillant. Deuxièmement, la vitesse, qui varie d'une impression en quelques dizaines de secondes à une cadence industrielle continue capable de produire plusieurs impressions à la seconde. Troisièmement, le coût, qui en fonction des matériaux et des équipements utilisés influe sur le coût unitaire d'une impression.

Naturellement, ces différentes exigences imposent chacune des contraintes techniques particulières qui ont tendance à s'exclure mutuellement, provoquant ainsi la nécessité d'un compromis entre qualité, vitesse et coût. Par conséquent, les équipes de recherche et de développement qui composent le vaste domaine de l'impression travaillent à repousser à la fois les limites individuelles de ces contraintes, mais également les limites issues de leurs associations. L'objectif ultime est commun à de nombreux domaines de la production : faire vite et qualitatif pour un coût minimal. Parmi l'éventail des axes de recherche permettant de repousser les limites actuelles se trouvent les dernières technologies de notre époque, dont fait incontestablement partie la science informatique qui a d'ores et déjà permis de révolutionner de nombreux domaines.

1.1 Procédés d'impression

L'impression moderne est composée de différents types de procédés d'impression qu'il est commun de regrouper en deux grandes familles : d'un côté, les procédés *traditionnels* dont le nom laisse deviner l'antécédence ; et de l'autre côté, les procédés *numériques*.

Les procédés traditionnels comportent une large variété de techniques différentes dont les plus connus sont : l'*offset*, l'*héliogravure*, la *flexographie* et la *sérigraphie*. Le point commun de ces techniques repose sur l'utilisation d'une *forme imprimante* qui est chargée de définir directement ou par transfert les zones où l'encre doit être déposée pour produire l'impression voulue. Ainsi, pour l'offset il s'agit généralement de cylindres métalliques préalablement enduits d'une substance grasse formant les zones qui se chargeront d'encre avant de la restituer par transfert entre cylindres successifs sur le matériel à imprimer. L'héliogravure et la flexographie utilisent quant à elles un principe de reliefs, respectivement creux et bombés, pour composer la forme imprimante et ainsi y emmagasiner une quantité plus ou moins importante d'encre. Enfin, la sérigraphie utilise un principe de pochoirs pour définir la forme imprimante en protégeant les zones à ne pas encrer. Au-delà de leurs techniques différentes pour composer la forme imprimante, chaque procédé traditionnel dispose également de techniques de séchage particulières. Cette diversité de techniques permet d'imprimer efficacement sur des supports et matériaux aux contraintes variées : papier, textile, carton ondulé, surface rigide ou non plane, etc.

Les procédés numériques, par opposition au principe analogique des procédés traditionnels, se distinguent par l'absence de forme imprimante pour définir les zones à encrer. Ici aussi, plusieurs techniques ont été développées, mais seules deux stratégies dominent les usages. La première, communément connue sous le nom usuel d'« impression laser », est le procédé de *xérographie* qui consiste à modifier dynamiquement à l'aide d'un laser les propriétés électrostatiques d'un cylindre afin de définir le motif de l'impression qui sera ensuite encré par réaction physique et immédiatement ensuite imprimé par transfert. La seconde, tout aussi populaire, est appelée *jet d'encre* en référence à son fonctionnement d'éjection d'encre directement sur la surface à imprimer. Dans son fonctionnement le plus courant, de minuscules gouttes d'encre sont dynamiquement créées et éjectées afin de composer l'impression. Le fonctionnement pleinement dynamique de ces procédés implique une représentation numérisée des impressions à réaliser de façon à pouvoir piloter en continu la formation des zones à imprimer et ainsi produire les motifs voulus.

Si les procédés traditionnels ont su révolutionner l'impression et sont ainsi encore largement utilisés de nos jours, c'est avant tout grâce à leur capacité à produire rapidement, avec une haute qualité et en minimisant les coûts par la réalisation de grande quantité de tirages identiques. Pourtant, la flexibilité de fonctionnement des procédés numériques offre de nombreux avantages sur ces procédés traditionnels : premièrement, la possibilité de réaliser des

impressions différentes les unes des autres sans modification de l'imprimante ; secondement, la réduction des délais de préparation nécessaire à la réalisation d'une impression puisque la différenciation du contenu se fait à l'exécution ; troisièmement, conséquence des deux avantages précédents, les procédés numériques permettent de diminuer de façon drastique le nombre minimal de copies à produire pour être financièrement profitables. Ainsi, le domaine de l'impression se destine naturellement à voir ses procédés traditionnels progressivement substitués par les procédés numériques dès lors que ceux-ci sauront démontrer leur capacité à proposer les mêmes performances que celles acquises au fil des décennies par l'utilisation massive des procédés traditionnels.

1.2 Impression numérique professionnelle

Bien que les avantages des procédés d'impression numériques sur les procédés d'impression traditionnels soient indéniables, leur utilisation introduit également d'importantes contraintes inédites. En effet, en plus des contraintes mécaniques de fonctionnement spécifiques aux technologies de xérographie et de jet d'encre s'ajoutent les contraintes numériques liées aux traitements des données ainsi qu'à leur acheminement coordonné jusqu'aux têtes d'impression. La complexité nécessaire pour assurer le respect de ces contraintes réserve actuellement l'utilisation des procédés d'impression numériques à des cas d'usage restreints où la production est orientée sur une faible quantité de contenu personnalisé ; ce qui laisse ainsi une place privilégiée aux procédés traditionnels pour la production de masse.

La grande flexibilité des procédés d'impression numériques a permis le développement des imprimantes à usages personnels et bureautiques, créant de ce fait la branche de l'impression grand public. Mais leur flexibilité et leurs avantages ont également trouvé leur place dans le domaine de l'impression professionnelle en bénéficiant d'un savoir-faire hautement qualitatif ainsi que du développement de nombreux usages, services et innovations, notamment celui des impressions grand format et haute qualité.

1.2.1 Chaîne numérique de traitement graphique

Dans le domaine de l'impression, la chaîne graphique désigne l'ensemble des corps de métiers intervenant pour la réalisation d'une impression. Cette chaîne s'étend de la création du visuel jusqu'aux finitions, en passant par l'impression en elle-même. Étendue à l'impression numérique, cette chaîne graphique incorpore également l'ensemble des traitements informatiques successifs à réaliser sur les données afin de produire l'impression voulue.

L'intérêt majeur de l'impression numérique se situe dans les possibilités infinies que propose la numérisation de cette chaîne de traitement graphique. En effet, le format numérique des données permet de leur appliquer différents traitements pour les améliorer ou de les corriger

afin d'obtenir un meilleur résultat. Cette numérisation des données s'étend ainsi de la création même des images jusqu'à leur transmission à l'imprimante.

1.3 Motivations

Si les méthodes nécessaires à la mise en œuvre de l'impression numérique sont désormais connues et maîtrisées depuis plusieurs dizaines d'années, celles-ci ne cessent de s'améliorer au gré des évolutions de ce domaine constamment stimulé par la concurrence que se livrent ses nombreux acteurs. Ainsi, l'impression numérique professionnelle est en constante évolution pour intégrer les dernières innovations qui permettent d'améliorer l'ensemble de la chaîne de traitement graphique. Ces innovations concernent plusieurs domaines : la mécanique avec des imprimantes qui proposent toujours plus de paramètres et de fonctionnalités (étalonnage, impression, séchage, découpe, etc.) ; la chimie des encres qui développe de nouvelles encres et matériaux plus performants et plus écologiques ; le logiciel et l'algorithmique, que nous détaillons dans le chapitre 2 : *RIP : Raster Image Processor*, qui offrent un support efficace des innovations matérielles et produisent des impressions plus économiques et de meilleure qualité pour l'œil humain. En conséquence, ces innovations participent à complexifier la chaîne de traitement graphique en l'obligeant à être toujours plus complète, flexible et performante.

1.3.1 Accroissement de la qualité

Les nombreuses évolutions technologiques des deux dernières décennies ont déclenché une incroyable augmentation des quantités de données, aussi bien au niveau du stockage par la baisse des prix des supports de masse et des puces de mémoire vive, des traitements grâce notamment à l'augmentation de la densité des transistors sur les processeurs, que de la communication avec des bus de communication et des connexions réseau plus rapides. Touchant de nombreux domaines, cette augmentation s'est logiquement propagée dans l'impression numérique en permettant de démultiplier le niveau de détail des données traitées, conduisant ainsi à une hausse de la qualité des impressions produites. Par exemple, les images des utilisateurs sont devenues suffisamment détaillées et les capacités de traitement et de communication suffisamment performantes pour que les constructeurs d'imprimantes professionnelles jet d'encre proposent désormais des résolutions d'impression atteignant les 2 000 gouttes par pouce [1] horizontalement et verticalement, c'est-à-dire une densité de 620 000 gouttes par centimètre carré, tout en maintenant une largeur d'impression allant jusqu'à 1,60 m.

La hausse permanente de la qualité, et donc de la volumétrie des données et des traitements, pose perpétuellement de nouveaux défis afin d'exploiter efficacement ces nouvelles évolutions technologiques dans les traitements spécifiques de la chaîne de traitement graphique, tels que

1. Le domaine de l'impression utilise communément les unités de mesure anglo-saxonnes.

la rotation et la mise à l'échelle des données d'entrée, les transformations et les corrections colorimétriques, ou encore le formatage des données de sortie.

1.3.2 Évolutions des usages

En plus d'être portée par les évolutions technologiques générales, l'impression numérique comporte également son lot d'innovations spécifiques. De par la nature manufacturière du domaine, ces innovations concernent autant les technologies que les usages et l'organisation de la production. Cependant, quelles que soient leurs origines, c'est également à la chaîne de traitement graphique d'évoluer pour intégrer efficacement ces innovations afin de permettre leur application industrielle.

Fermes d'impression

Parmi les innovations liées aux usages, certaines sont directement issues de la tendance à l'industrialisation des imprimeurs afin d'améliorer leurs capacités de production. Cette industrialisation croissante repose sur de nouvelles générations d'imprimantes toujours plus stables et plus performantes ainsi qu'une meilleure maîtrise de la chaîne de production impliquant clients, fournisseurs et opérateurs. L'un des facteurs déterminants dans la capacité de production est le nombre d'imprimantes aptes à produire simultanément. Ainsi, de nombreux imprimeurs visent à augmenter progressivement la taille de leur parc d'imprimantes afin d'aborder de nouvelles opportunités de marché.

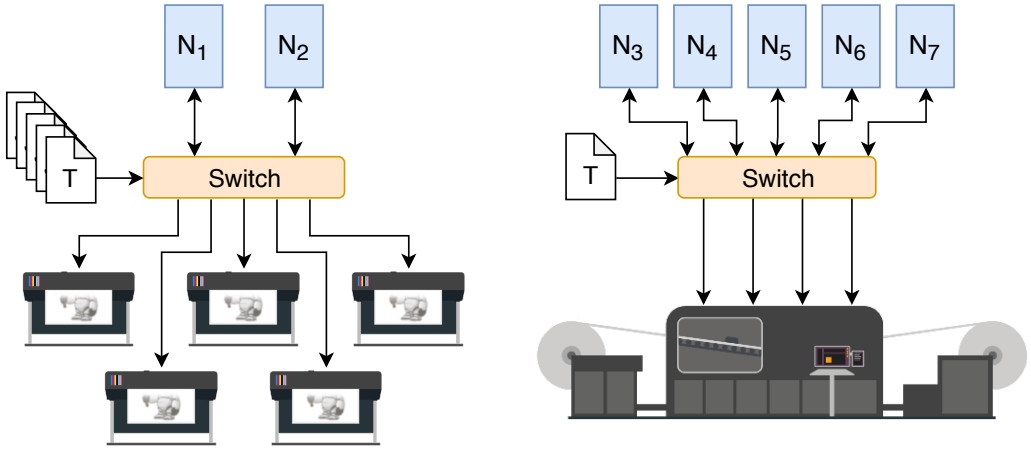
La création de ces parcs d'imprimantes se fait par l'utilisation simultanée de plusieurs dizaines à centaines d'imprimantes, ce qui a donné naissance au terme de *fermes d'impression* pour désigner ce nouvel usage. Les imprimantes couramment utilisées dans ces fermes d'impression nécessitent chacune de recevoir les données à une vitesse de seulement quelques Mio/s pour fonctionner. Cette organisation particulière est illustrée dans la figure **1.1 (a)**. En raison de leur nombre, le pilotage de ces imprimantes présente le défi inédit dans ce domaine d'une quantité importante de données indépendantes à générer continuellement, mais il offre des opportunités avantageuses de rationalisation et de résilience des capacités de calculs.

Presses numériques

Parallèlement à l'évolution des usages, les fabricants d'imprimantes travaillent à repousser les limitations techniques actuelles de celles-ci. Parmi ces limitations se trouve la vitesse maximale à laquelle l'imprimante peut déposer avec précision ses différentes encres. Ainsi, les dernières innovations en termes de mécanique et des propriétés physiques des gouttes d'encre ont permis l'arrivée d'une nouvelle gamme d'imprimantes jet d'encre appelées *presses numériques*. Celles-ci sont mécaniquement capables d'atteindre des cadences de plusieurs impressions de type poster

par seconde tout en conservant la flexibilité des procédés numériques, et sont ainsi destinées à combler le principal défaut des procédés traditionnels.

Une telle capacité de production numérique implique automatiquement en amont d'importantes contraintes sur la chaîne de traitement graphique, en particulier la capacité à générer le flux de données requis qui est susceptible d'atteindre plusieurs Gio/s de données. Ce débit, d'un ordre de grandeur jusqu'alors inédit dans le domaine de l'impression numérique, est une source importante de motivation pour réaliser les innovations nécessaires à l'exploitation du plein potentiel des procédés d'impression numérique. Ce nouvel ordre de grandeur rend nécessaire l'utilisation conjointe de nombreuses ressources de calculs. De plus, elle impose de repenser les moyens de communication des données à l'imprimante en proposant, par exemple, de multiplier le nombre de points de connexion pour y faire circuler différentes parties du flux de données, tel qu'illustré par la figure **1.1 (b)**.



(a) : Ferme d'impression **(b) :** Presse numérique

Figure 1.1 : Illustration des nouveaux usages et évolutions techniques de l'impression numérique : à gauche la multiplication d'imprimantes classiques ; à droite une presse numérique ayant une cadence de production très élevée. Avec **T** pour les travaux d'impression et **N** pour les nœuds de calculs

1.4 Limites des solutions existantes

Les principaux logiciels professionnels fournissant une chaîne de traitement graphique adaptée aux procédés de l'impression numérique sont apparus lors de l'explosion des systèmes informatisés durant les années 1990. Par conséquent, ces logiciels ont été développés en premier lieu pour résoudre les problématiques inédites de cette époque. Par la suite, les nombreuses évolutions et révolutions du domaine de l'impression numérique impulsées par les innovations mécaniques, chimiques, technologiques, ainsi que l'évolution des usages et l'accroissement des qualités ont progressivement induit une complexification importante de leur architecture pour répondre aux ajustements constants des besoins du marché.

Toutefois, les évolutions récentes portées par les fermes d'impression et les presses numériques associées aux changements majeurs de paradigmes de programmation, caractérisés par les prémices de la fin de la loi de Moore et la multiplication des cartes accélératrices, rendent les architectures monolithiques de ces chaînes de traitement graphique inadaptées. En effet, ces architectures n'ont pas été conçues pour assurer la flexibilité suffisante permettant le support de ces nouveaux usages qui requièrent à la fois des besoins de performance et de passage à l'échelle. Ces limitations logicielles sont un frein à l'exploitation du plein potentiel des innovations des procédés numériques actuels et futurs.

1.5 Déroulement de cette thèse

Cette thèse bénéficie du dispositif *Cifre* proposé par l'Association Nationale de la Recherche et de la Technologie (ANRT) [2] qui permet d'organiser la collaboration entre une entreprise française et un laboratoire de recherche académique à travers la réalisation d'un doctorat portant sur une problématique scientifique de l'entreprise. Ainsi, cette thèse est issue de la collaboration de l'entreprise Caldera [3], éditrice de logiciels pour l'impression numérique professionnelle, avec l'équipe ICPS [4], spécialisée dans les techniques de calcul haute performance, appartenant au laboratoire ICube [5] de l'université de Strasbourg et également rattachée au centre de recherche Inria [6] Grand Est. Sa réalisation est donc financée par l'entreprise Caldera et les subventions du dispositif *Cifre* accordées par l'ANRT au nom du Ministère chargé de la Recherche.

Au cours des trois années de recherche, l'organisation des travaux a évolué pour s'équilibrer autour d'une répartition équitable du temps de travail dans les deux structures. La supervision du projet au sein de l'entreprise Caldera a été successivement réalisée par deux personnes : Frédéric Soulier et Patrick Zimmermann. Cette thèse a permis l'encadrement à l'université d'étudiants sur des aspects circonscrits issus de ses travaux plus globaux, deux stagiaires de seconde année de Master Informatique : Ervin Altintas et Aurélien Rausch ; et deux plus légers Travaux d'Étude et de Recherche (TER) réalisés par des étudiants en première année de Master Informatique.

1.6 Contributions

Les travaux menés dans cette thèse ont pour objectif de repousser les limitations des solutions existantes de traitement graphique afin d'apporter à l'entreprise Caldera les moyens de suppor-

2. http://www.anrt.asso.fr
3. https://www.caldera.com
4. https://icps.icube.unistra.fr
5. https://icube.unistra.fr
6. https://www.inria.fr

ter les récentes évolutions des usages et des technologies vécues par l'impression numérique, ainsi que de préparer au mieux le support de leurs mutations futures.

Pour ce faire, nous explorons les paradigmes et les techniques de programmation qui sont issus du domaine du calcul haute performance, tels que la parallélisation des traitements sur plusieurs unités de calcul et leur distribution sur les nœuds d'un cluster. La combinaison de ces deux méthodes permet d'accélérer les vitesses de traitement grâce à une meilleure exploitation des ressources des systèmes informatiques et un passage à l'échelle efficace par leur mise en collaboration. Nous examinons la compatibilité des outils et des techniques existantes avec les contraintes, besoins et objectifs spécifiques à l'industrie de l'impression numérique dont nous définissons les limites et les domaines de validité.

De même, nous réalisons un travail d'optimisation des traitements graphiques les plus complexes qui consistent à opérer la transformation d'une matrice de données dont la taille dépasse la quantité de mémoire vive du système. Ce travail tire parti d'une utilisation efficace des mécanismes de cache et d'accès aux mémoires de masse du système d'exploitation tout en limitant habilement la quantité totale de données accédées en lecture et en écriture. Ainsi, notre solution est adaptée à l'ensemble des technologies actuelles de mémoire de masse et fournit une accélération substantielle des temps d'exécution par rapport aux solutions de l'état de l'art.

Enfin, la contribution majeure de nos travaux consiste en l'implémentation d'une solution fonctionnelle qui fournit une réponse unique prenant en compte à la fois les besoins des fermes d'impression et ceux des presses numériques, c'est-à-dire conciliant les figures **1.1 (a)** et **1.1 (b)**, au sein d'une unique infrastructure. Notre solution permet ainsi d'augmenter le potentiel de mutualisation des ressources. Cette implémentation est basée sur une architecture innovante associant la distribution et la parallélisation des traitements en accord avec les contraintes des traitements graphiques de l'impression numérique. Nous explorons et analysons en profondeur l'ensemble de ses mécanismes pour nous assurer de leur efficacité. En conséquence, l'évaluation de notre solution montre sa capacité à répondre efficacement aux besoins de performances, de passage à l'échelle et de durabilité tout en respectant des contraintes du domaine.

1.7 Organisation du manuscrit

Ce manuscrit présente les travaux réalisés sur la parallélisation et le passage à l'échelle durable d'une chaîne de traitement graphique pour répondre efficacement aux nouveaux besoins de l'impression numérique professionnelle. Il est composé de quatre parties. La partie I précise les éléments constituant la chaîne de traitement graphique explorée ainsi que l'environnement technique dans lequel elle opère. Cette clarification permet de définir les contraintes et les objectifs fixés pour les travaux exposés dans les parties suivantes. La partie II présente

les travaux réalisés pour améliorer la performance des opérations de rotation des données d'entrées dans l'étape de prétraitement de la chaîne de traitement. La partie III poursuit par la présentation des travaux menés sur la parallélisation, la distribution et le passage à l'échelle des calculs opérés par la chaîne de traitement graphique. La partie IV expose les évaluations de la solution distribuée qui a été implémentée. Les protocoles d'évaluation sont détaillés puis leurs résultats commentés et discutés. Enfin, la conclusion, partie V, revient sur les différents travaux menés et en esquisse les perspectives.

Première partie

Environnement

2 RIP : Raster Image Processor

L'utilisation des systèmes parallèles et distribués est applicable à de nombreux domaines et couvre un très large champ d'application et de technologies. Dans ce chapitre, nous précisons l'environnement dans lequel les chaînes de traitement graphique pour l'impression numérique s'exécutent ainsi que les opérations qu'elles réalisent.

2.1 Définition

L'ensemble des traitements graphiques à effectuer pour transformer un fichier d'entrée en données exploitables par des imprimantes se regroupent au sein d'un logiciel nommé *Raster Image Processor* (RIP). Ce logiciel assure : l'interprétation des fichiers d'entrée ; leur traitement en accord avec la configuration voulue ; jusqu'au pilotage des imprimantes.

2.2 Travaux et flux d'impression

Les nombreuses décennies d'évolution de l'impression ont conduit cette industrie à s'organiser autour de la notion de *travaux d'impression*. Chaque travail contient l'ensemble des informations nécessaires à la réalisation d'une impression. Parmi ces informations : le document à imprimer ; les traitements à appliquer ; l'imprimante de destination au travers du pilote d'impression qui lui est associé ; ainsi que les réglages à utiliser par cette imprimante.

Cette organisation en travaux a également facilité l'interconnexion des chaînes de production avec des systèmes extérieurs permettant ainsi de développer efficacement l'automatisation de certaines opérations. En particulier, la transmission des travaux d'impression depuis les systèmes de conception graphique jusqu'aux équipements de pliage, de découpe et de conditionnement pour la livraison des travaux imprimés, en passant également par les opérations de contrôle qualité.

L'utilisation conventionnelle de cette automatisation a donné naissance à la notion de *workflow* qui se traduit littéralement par *flux de travail*. Ainsi, la production est organisée en flux de travaux passant par plusieurs étapes, dont celle du passage indispensable par la chaîne de traitement graphique au sein du RIP afin de produire les données spécifiques à ce travail et à l'imprimante cible.

2.3 Traitements graphiques

Tout au long de la production, le RIP reçoit en entrée de multiples travaux d'impression qu'il doit traiter en respectant leurs configurations et les contraintes de l'environnement détaillées dans le chapitre 3 : *Contraintes et objectifs*. Le RIP applique à chaque travail qu'il reçoit une succession de traitements graphiques qui sont issus d'illustres travaux de recherche portant d'une part sur les propriétés physiques et chimiques des matériaux utilisés, et d'autre part sur le métamérisme (qui regroupe les problématiques liées à perception des couleurs par l'œil humain en fonction des conditions d'éclairage et des individus) [Bee85 ; McL86 ; Com ; Chi+12a ; Luo16]. Ces travaux ont été adaptés aux traitements graphiques numériques afin de fournir des caractéristiques algorithmiques avantageuses. Ainsi, les traitements graphiques de l'impression numérique que nous utilisons dans cette thèse permettent une indépendance des pixels entre eux, mais n'effacent pas la dépendance entre les canaux chromatiques d'un même pixel. Dans la suite de cette section, nous détaillons le rôle et les caractéristiques de chacun de ces traitements dont l'enchaînement est illustré par la figure 2.1.

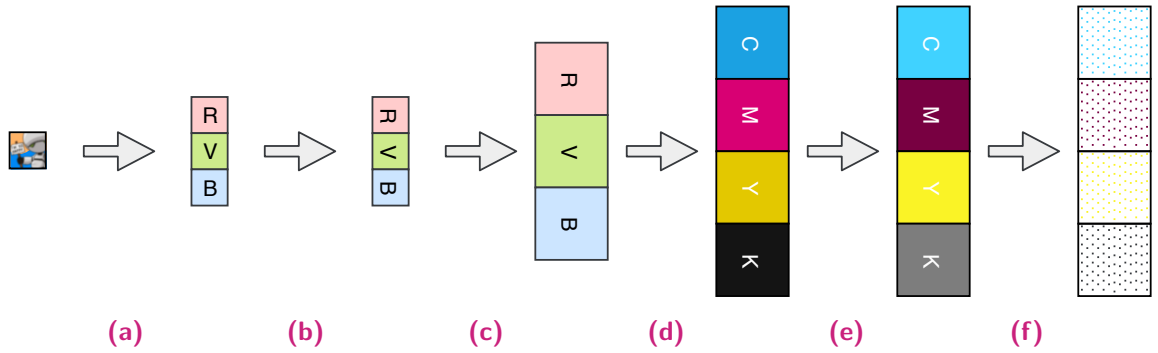


Figure 2.1 : Schématisation des différents traitements graphiques successifs réalisés par un RIP sur l'image d'entrée lors de la génération des données pour une imprimante, avec **RVB** pour *Rouge-Vert-Bleu* et **CMYK** pour *Cyan-Magenta-Yellow-Black*, **(a)** décodage, **(b)** prétraitement, **(c)** mise à l'échelle, **(d)** colorimétrie, **(e)** limitation et linéarisation, et **(f)** tramage

Prétraitements

La première étape consiste à réaliser des prétraitements sur les fichiers d'entrée composant le travail d'impression. Ces fichiers d'entrée sont constitués de l'image à convertir en données pour l'imprimante et de fichiers paramétrant les valeurs à utiliser lors des différents traitements. Ainsi, ces fichiers peuvent être modifiés afin de les améliorer ou de les enrichir.

Le premier traitement graphique indispensable consiste à décoder l'image depuis son format de stockage vers celui utilisé en interne par le RIP, figure 2.1 **(a)**. Parmi les encodages d'image les plus utilisés dans le domaine de l'impression se trouvent : les formats vectoriels, tels que PDF et PostScript ; et les formats *rasters*, tels que TIFF, JPEG, PNG. Les premiers décrivent le contenu d'une image à partir d'objets géométriques (droite, forme simple, polygone, etc.) et d'attributs (dimension, position, couleur, transparence, etc.). Ils ont l'avantage de pouvoir

être affichés à toutes les dimensions sans perte de qualité en échange d'une interprétation parfois complexe des objets. Les seconds définissent une matrice de pixels représentant l'image dans une certaine résolution, ils sont plus simples à interpréter, mais leur affichage à une résolution différente de celle de la matrice nécessite d'improviser les pixels manquants, ce qui peut créer des artefacts visuels en fonction du facteur de redimensionnement et de la méthode de mise à l'échelle employée. Le décodage de ces différents formats complexes, proposant pour certains la compression des données, peut être réalisé par des bibliothèques dédiées [Ado19 ; Sam03 ; Guy19]. Quant au format de stockage interne du RIP, nous utilisons celui consistant en une représentation séparée des canaux chromatiques, tel qu'illustré dans l'ensemble de la figure **2.1**.

Dans nos travaux, nous nous intéressons également à deux prétraitements supplémentaires impliquant des accès mémoire complexes, celui de la rotation, illustrée par la figure **2.1 (b)**, et de la transposition des images. Leur exécution par le RIP permet de répondre à un besoin d'orientation spécifique de l'impression ou à des contraintes du matériel imprimé. Ces deux transformations et leurs optimisations sont présentées dans le chapitre 4 : *Rotation et transposition de matrices rectangulaires out-of-core et out-of-place*. Réaliser ces transformations en début de chaîne, c'est-à-dire en prétraitement, procure deux avantages majeurs : optimiser l'exécution des traitements suivants en améliorant les accès aux données ; et optimiser l'exécution de travaux futurs utilisant la même image d'entrée avec la même transformation.

Mise à l'échelle

Une fois les prétraitements requis réalisés, l'étape suivante consiste à mettre à l'échelle l'image d'entrée en fonction de la résolution qui sera utilisée par l'imprimante, figure **2.1 (c)**. Ainsi, l'image d'entrée sera : agrandie ou réduite pour correspondre à la résolution d'impression ; et éventuellement déformée verticalement ou horizontalement dans le cas d'une résolution d'impression asymétrique. La mise à l'échelle peut se faire en utilisant de nombreux algorithmes fournissant chacun des avantages et inconvénients [PKT83 ; LGS99 ; TBU00]. Le plus rapide à exécuter est la mise à l'échelle par plus proche voisin qui consiste à dupliquer ou supprimer les pixels jusqu'à atteindre la résolution souhaitée. Cependant, cette approche crée de nombreux artefacts visuels dégradant sensiblement la qualité de l'image obtenue. Pour éliminer ces artefacts visuels, l'approche prédominante dans l'industrie de l'impression consiste à réaliser des interpolations pour définir les pixels manquants. De nombreuses techniques d'interpolation aux coûts d'exécution variés sont disponibles dans l'état de l'art, parmi lesquelles : les interpolations linéaires et bilinéaires exploitent respectivement une ou trois fonctions linéaires pour déterminer la valeur du pixel manquant en fonction de sa distance par rapport aux pixels existants les plus proches ; les interpolations cubiques et bicubiques en font de même avec respectivement une ou trois fonctions cubiques qui permet de mieux percevoir la tendance chromatique des pixels d'une zone ; les interpolations *edge-directed* [AW96 ; LO01] utilisent, en

plus des pixels voisins en ligne et en colonne, les pixels voisins en diagonale pour affiner le résultat.

Colorimétrie

À partir des données mises à l'échelle, le RIP applique plusieurs traitements colorimétriques issus des travaux sur la définition des espaces colorimétriques, leurs représentations et leurs algorithmiques [You02 ; SO95 ; Rei+01], ainsi que leurs applications au domaine de l'impression [Mor08]. Ces traitements ont deux objectifs : réaliser une correction des valeurs des couleurs ; et réaliser une conversion d'un espace colorimétrique à un autre. Ces opérations sont illustrées par la figure **2.1 (d)**.

— Pour le premier objectif, il s'agit d'appliquer des profils ICC [1] qui permettent d'adapter les couleurs en fonction de l'écran sur lequel l'image a été conçue et des mélanges efficaces de couleurs que l'imprimante peut réaliser. Cette étape essentielle permet notamment d'obtenir un rendu visuellement similaire sur des imprimantes travaillant sur des matériaux différents ou avec des encres de qualités différentes. Ces profils ICC font partie des fichiers transmis en entrée du travail et sont issus des opérations d'étalonnages et de calibrations des écrans et des imprimantes.

— Pour le second objectif, il s'agit d'une part de convertir, au besoin, l'image définie sous forme d'une synthèse additive des canaux chromatiques (les couleurs s'additionnent les unes aux autres jusqu'à donner du blanc) utilisée par les écrans vers une synthèse soustractive (les couleurs se soustraient les unes aux autres jusqu'à donner du noir) utilisée par les imprimantes ; et d'autre part de convertir le nombre de canaux chromatiques de l'image source vers le nombre d'encres dont dispose l'imprimante. Classiquement, les images sont enregistrées dans un format stockant la valeur des pixels à partir des trois couleurs primaires de la synthèse additive : rouge, vert et bleu (RVB) et sont à transformer vers l'espace colorimétrique de l'imprimante composé généralement des quatre couleurs d'encre primaire de la synthèse soustractive : cyan, magenta, jaune et noir (CMJN ou CMYK) ainsi que d'encres additionnelles, telles que : vert, rouge, gris, etc. qui permettent d'améliorer le rendu de l'impression.

Linéarisation et limitation

Les traitements associés à la couleur se poursuivent par l'étape commune de linéarisation et de limitation, illustrée en figure **2.1 (e)** qui consiste à ramener les valeurs des pixels dans l'intervalle de valeurs que supporte le matériel sur lequel l'impression sera réalisée. Cet intervalle définit la limite maximale à partir de laquelle le matériel va être saturé d'encre et par conséquent commencer à « baver ». Ces opérations se font par application de plusieurs *lookup table* (LUT) de conversion des valeurs pour chaque canal chromatique [Pop07 ; Qia13 ; AM97].

1. International Color Consortium – `http://www.color.org`

Ces LUT sont préalablement construites lors de l'étalonnage de l'imprimante à l'aide d'un spectrophotomètre. Par conséquent, les valeurs stockées dans les LUT dépendent du modèle de l'imprimante, du type et de la qualité du matériel qui sera imprimé, ainsi que de la qualité des encres utilisées. Les LUT font également partie des fichiers d'entrée du travail.

Tramage

Le dernier traitement graphique, figure **2.1 (f)**, avant de transmettre les données calculées au pilote d'impression consiste à aligner leur précision sur celle de l'imprimante, c'est-à-dire convertir les pixels de l'image en points de la trame, c'est-à-dire en gouttes d'encre. Dans le langage courant, cette étape s'appelle le « tramage ».

L'imprimante peut demander des précisions différentes pour chaque canal chromatique. Cette précision se décompose en deux définitions : le ton continu (*contone*) et le demi-ton (*halftone*). Le ton continu définit les encodages permettant de représenter des niveaux de couleurs sans « cassure », typiquement basé sur l'utilisation de 4 bits de valeur ou plus par canal. Par opposition, le demi-ton restreint fortement le nombre de niveaux de couleur autorisé en permettant un encodage sur moins de 4 bits. L'utilisation du ton continu permet de transmettre à l'imprimante un maximum de précision dans les valeurs des différents points. Celle-ci se chargera alors grâce à un moteur de conversion interne (et souvent propriétaire) de convertir ces données en gouttes. L'utilisation du demi-ton permet d'indiquer directement à l'imprimante l'emplacement des gouttes (sur 1 bit : valeurs 0 ou 1 pour non-goutte ou goutte), ainsi que la taille de goutte à utiliser (sur 2 bits : valeurs 0,1,2,3 pour non-goutte, petite goutte, moyenne goutte ou grosse goutte) si celle-ci supporte ce niveau de précision lors de la création des gouttes d'encre. Plus communément, le demi-ton consiste à réaliser pour chaque canal une conversion en niveaux de gris.

Cette conversion des pixels en gouttes est réalisée grâce au « tramage stochastique » [OH99] qui se décompose en deux étapes. Premièrement, l'application d'une matrice stochastique sur chaque canal afin de sélectionner les canaux du pixel qui sont pertinents à être imprimés tout en limitant l'apparition d'effets de Moiré (lorsque la disposition des gouttes de l'image rentre en résonance avec la trame de l'imprimante et provoque des artefacts visuels déplaisants pour l'œil humain) [ON63]. Secondement, une fois les canaux du pixel sélectionnés, un seuillage de leur valeur est réalisé, à l'aide de LUT établies lors de l'étalonnage de l'imprimante, pour les convertir sur le nombre de bits souhaité.

2.4 Pilotage des imprimantes

Enfin, la dernière étape composant un RIP est le pilotage des périphériques d'impression. Les pilotes d'impression font l'interface entre la chaîne de traitement graphique et les imprimantes.

Avec leur complexification croissante, les pilotes d'impression se sont également complexifiés afin de proposer aux utilisateurs de la chaîne de production toutes les informations et les fonctionnalités spécifiques offertes par chaque modèle d'imprimante.

Le pilotage des imprimantes se fait désormais majoritairement par connexion réseau, soit directement en point à point, soit avec routage des paquets au sein du réseau local d'un site géographique. Dans le cas des imprimantes classiques, il s'agit généralement d'une relation d'association « un-à-un » de telle façon qu'un pilote d'impression est conçu pour piloter un seul modèle d'imprimante et chaque imprimante ne peut être pilotée que par un seul pilote d'impression à la fois. Dans le cas des presses numériques, les pilotes d'impression peuvent être composés de plusieurs sous-pilotes d'impression complémentaires répartis sur plusieurs ordinateurs, cela afin de disposer d'une meilleure bande passante en dédiant par exemple à chacun des pilotes l'alimentation des données d'un seul canal chromatique. Dans tous les cas, la volumétrie des données des travaux est telle (cf. chapitre 3 : *Contraintes et objectifs*) que le transfert des données se fait progressivement et concomitamment à leur impression. La communication avec une imprimante ou une presse numérique peut nécessiter plusieurs connexions parallèles, par exemple une connexion pour le statut et une pour les données. L'utilisation de pilotes d'impression conçus spécifiquement pour chaque modèle d'imprimante est essentielle pour disposer d'un processus jouant le rôle d'interface standardisée entre le calcul des données, les services proposés par le RIP et l'échange des données, des ordres et des statuts avec l'imprimante.

3 Contraintes et objectifs

Les chaînes de traitement graphique exécutent un ensemble de traitements successifs dont nous avons précisé les caractéristiques dans le chapitre précédent. Cependant, ces traitements sont réalisés pour atteindre des objectifs en accord avec les besoins de l'industrie de l'impression numérique. Ainsi, nous dédions ce chapitre à la présentation des contraintes imposées par cette industrie à celles des objectifs des travaux de cette thèse.

3.1 Volumétrie et débits

Principale contrainte de fonctionnement d'un RIP, le débit de données qu'il produit doit être suffisant pour alimenter les imprimantes qu'il pilote. Ce débit est déterminé par les capacités mécaniques de l'imprimante qui définissent à quelle vitesse elle peut imprimer en garantissant un niveau de qualité stable. Pour une même imprimante, cette vitesse est variable selon la qualité d'impression indiquée dans la configuration du travail d'impression. Ce débit induit deux contraintes fortes : l'une matérielle, l'autre logicielle. En effet, d'un côté ce débit conditionne la capacité minimale du lien réseau entre l'imprimante et son pilote d'impression, de l'autre côté, la vitesse à laquelle les calculs doivent être réalisés.

Cependant, le débit n'est pas la seule métrique posant d'importantes contraintes à un RIP. C'est aussi le cas de la volumétrie des données des travaux qui découle également des capacités mécaniques de l'imprimante. En effet, celles-ci définissent les valeurs de dimension, de résolution et de précision permises par le constructeur pour réaliser une impression et, par conséquent, avec lesquelles l'utilisateur configure le travail d'impression. La volumétrie d'un travail peut se mesurer en trois points : la taille des fichiers d'entrée ; sa taille maximale lors des étapes de calcul ; et la taille des données en sortie. La volumétrie en entrée dépend exclusivement des fichiers d'entrée, alors que les deux dernières volumétries sont liées aux opérations effectuées sur ces données d'entrée et sont respectivement définies par les équations **3.1** et **3.2** ci-dessous. Ces deux équations ne divergent que par la précision des données utilisée à leur étape respective : la précision à laquelle les calculs sont réalisés, c'est-à-dire le nombre de bits sur lesquels est réalisé l'encodage de chaque canal de chaque pixel ; et la précision à laquelle les données seront transmises à l'imprimante, c'est-à-dire les notions de ton continu et demi-ton présentées dans le chapitre précédent.

Soient :

— $V_{calculs}$ volumétrie du travail (en octets) lors des calculs ;

— V_{sortie} volumétrie du travail (en octets) en sortie ;

— $D_{largeur}$ dimension en largeur de l'impression (en pouces [1]) ;

— $D_{hauteur}$ dimension en hauteur de l'impression (en pouces) ;

— $R_{horizontale}$ résolution horizontale de l'impression (en dpi [2]) ;

— $R_{verticale}$ résolution verticale de l'impression (en dpi) ;

— $P_{calculs}$ précision des données (en bits) lors des calculs ;

— P_{sortie} précision des données (en bits) en sortie ;

— C_{sortie} nombre de canaux chromatiques dans l'image de sortie ;

alors :

$$V_{calculs} = (D_{largeur} \times R_{horizontale}) \times (D_{hauteur} \times R_{verticale}) \times C_{sortie} \times \frac{P_{calculs}}{8} \qquad (3.1)$$

$$V_{sortie} = (D_{largeur} \times R_{horizontale}) \times (D_{hauteur} \times R_{verticale}) \times C_{sortie} \times \frac{P_{sortie}}{8} \qquad (3.2)$$

La volumétrie totale des fichiers d'entrée varie grandement d'un travail d'impression à l'autre. En effet, une image d'entrée, quel que soit son format d'encodage (vectoriel ou raster), peut varier de quelques kibioctets à plusieurs gibioctets selon son contenu, sa qualité et son éventuel niveau de compression. En revanche, les autres fichiers d'entrée, qui permettent de paramétrer les différents traitements, pèsent quant à eux moins de quelques mébioctets chacun. La volumétrie d'un travail d'impression lors des calculs et à sa sortie varie en fonction des nombreux critères de dimension et de qualité associés entre eux dans les équations 3.1 et 3.2. Le tableau 3.3 illustre un sous-ensemble de la combinatoire de ces différents critères. Il présente en colonne la plage de valeurs usuelles de chaque paramètre des deux équations et propose en ligne une configuration complète d'un travail d'impression avec les volumétries correspondantes. Ces configurations sont ordonnées par dimension d'impression et par qualité croissantes. La dernière colonne présente la volumétrie des travaux ramenée à une échelle commune, celle du « mètre linéaire » (que nous abrégeons « ml » pour la désigner en tant qu'unité). Un mètre linéaire désigne une impression d'un mètre de hauteur indépendamment de sa largeur. Elle permet de comparer les vitesses des imprimantes entre elles sans entrer dans le détail de leurs caractéristiques (dimension, résolution, nombre de canaux, etc.) respectives et nous sera utile pour la suite de cette section. Ainsi, la dernière colonne, exprimée en « Gio/ml » désigne la quantité de données nécessaire pour imprimer une distance d'un mètre avec cette configuration.

1. 1 pouce est égal à 2,54 cm
2. *dot per inch* – point par pouce

Tableau 3.3 : Exemples de configuration de travaux d'impression avec calcul des volumétries résultantes lors des calculs et en sortie du RIP

Dimension	Résolution	Canaux	Précision		Volumétrie		
Larg. × Haut. (cm x cm)	Horiz. × Verti. (dpi x dpi)	Sortie	Calculs (bit)	Sortie (bit)	Calculs (Gio)	Sortie (Gio)	Sortie (Gio/ml)
10 × 10	300 × 300	3	8	8	0,004	0,004	0,042
10 × 10	900 × 900	4	8	1	0,050	0,006	0,063
10 × 10	1200 × 1200	4	8	2	0,089	0,022	0,223
10 × 10	2000 × 2000	8	16	4	0,992	0,248	2,480
70 × 50	300 × 300	3	8	8	0,146	0,146	0,293
70 × 50	900 × 900	4	8	1	1,758	0,220	0,439
70 × 50	1200 × 1200	4	8	2	3,125	0,781	1,562
70 × 50	2000 × 2000	8	16	4	34,720	8,680	17,360
160 × 100	300 × 300	3	8	8	0,670	0,670	0,670
160 × 100	900 × 900	4	8	1	8,035	1,004	1,004
160 × 100	1200 × 1200	4	8	2	14,285	3,751	3,751
160 × 100	2000 × 2000	8	16	4	158,720	39,680	39,680

Les volumétries présentées dans le tableau 3.3 sont à mettre en perspective avec les cadences de production des imprimantes et des presses numériques actuelles et à venir. Naturellement, la production d'une volumétrie de données en un certain temps est moins contraignante pour les systèmes informatiques que la production de cette même volumétrie en un intervalle de temps plus court. Le tableau 3.4 présente un échantillon de modèles d'imprimantes et de presses numériques existantes. Chaque ligne expose un modèle accompagné de la volumétrie maximale en « Gio/ml » de travail qu'elle supporte et de la cadence en « mètre linéaire par heure », noté « ml/h », à laquelle elle est capable de les produire. La dernière colonne convertit ces deux valeurs dans le débit de données correspondant, à fournir en entrée de l'imprimante ou de la presse numérique. Les modèles sont ordonnés par cadence croissante.

Concrètement, les imprimantes à faible débit (quelques dizaines à quelques centaines de ml/h) utilisées dans les fermes d'impression laissent le temps suffisant aux RIP actuels pour générer les données d'un travail, quelles que soient sa taille et sa qualité. En revanche, l'apparition de presses numériques capables d'imprimer à des vitesses particulièrement importantes (plusieurs milliers de ml/h) avec des volumétries similaires à celle des imprimantes classiques élève à un nouvel ordre de grandeur le besoin en performance des RIP pour générer les débits de données correspondants. De plus, dans les prochaines années, les presses numériques se destinent à proposer de nouvelles dimensions et qualités d'impression afin d'égaler celles des meilleures imprimantes classiques actuelles. Or, comme exprimé par les équations 3.1 et 3.2, le support d'une résolution supérieure ou d'un canal chromatique supplémentaire démultiplie la volumétrie des travaux et par conséquent, si la cadence d'impression est maintenue, démultiplie également le débit des données requis en entrée.

Tableau 3.4 : Exemples d'imprimantes et de presses numériques existantes avec le débit des données requis en entrée selon la volumétrie des travaux et leur cadence d'impression maximale

Catégorie	Constructeur	Modèle	Volumétrie Entrée (Gio/ml)	Cadence Sortie (ml/h)	Débit Entrée (Gio/s)
Ferme d'impression	Roland	SG-540	0,713	7	0,001
	Hewlett-Packard	Latex 360	5,069	56	0,079
	Epson	SC-S60600	0,608	59	0,010
	Epson	Rho 1030	2,156	399	0,238
	Hewlett-Packard	PageWide XL 8000	0,874	1 284	0,311
Presse numérique	MS	LaRio Textile	1,662	4 500	2,077
	MS	LaRio Papier	1,662	7 200	3,324
	Landa	W10	1,090	12 000	3,634
	Landa	S10	0,734	13 009	2,652

Note : les imprimantes et les modes d'impression sélectionnées pour établir ce tableau sont donnés à titre indicatif et ne reflètent pas les performances globales de ces modèles, ni celles de leur constructeur.

À titre de comparaison, il est communément admis que les logiciels de RIP actuels sont capables, pour des configurations de travaux moyennement gourmands en calcul, de fournir un débit en sortie de 500 Mio/s maximum sur un système ayant une configuration matérielle grand public de type haut de gamme et combinant une exécution des traitements sur CPU et GPU. Or, les besoins des presses numériques vont actuellement jusqu'à près de 4 Gio/s, et cette valeur est appelée à augmenter progressivement dans les prochaines années.

3.2 Flux de données

Les contraintes de débit et de volumétrie sont associées à une troisième complexité spécifique au domaine de l'impression numérique : la présence en bout de chaîne de périphériques nécessitant une bande passante précise et constante pour fonctionner. En effet, une bande passante inférieure aux besoins de l'imprimante provoque au mieux le ralentissement de celle-ci, au pire la perte de l'impression en cours et donc la gâche du matériel, des encres et du temps de production utilisés. À l'opposé, fournir une bande passante supérieure aux besoins est inutile puisque la volumétrie des travaux (cf. tableau 3.3) ne leur permet pas de stocker les données dont elles n'ont pas besoin dans l'immédiat. Ainsi, les données doivent être fournies aux imprimantes au fur et à mesure de leur avancement, avec le débit requis et de façon stable pour éviter l'apparition de famine ponctuelle. Cet environnement particulier nous invite à considérer les travaux d'impression comme des flux de données à expulser progressivement pour lesquels la notion de date limite est plus importante que celle d'équité de traitement.

3.3 Topologies et passage à l'échelle

L'utilisation indispensable des pilotes d'impression pour communiquer avec les imprimantes requiert d'acheminer chacun des flux de données à des emplacements définis et non modifiables. En effet, certains d'entre eux détiennent l'unique liaison point à point avec leur imprimante. Or, les besoins spécifiques de chaque imprimante et des presses numériques requièrent des répartitions des calculs différentes : d'un côté, le regroupement de plusieurs travaux sur un même nœud afin de mutualiser les ressources ; de l'autre, la division des travaux d'impression sur plusieurs nœuds afin d'agréger les capacités de calcul et de communication des différentes ressources. Cette combinaison de points fixes avec des emplacements de calcul flexibles mène à la création de topologies flexibles évoluant avec les travaux d'impression en cours de traitement. Le support de topologies flexibles pour gérer les ressources est également la clé pour répondre aux besoins d'évolutivité et de passage à l'échelle des prochaines technologies et usages de l'impression numérique.

3.4 Cibles matérielles et pérennité

Qualifier les ressources informatiques en cause est important pour définir l'environnement dans lequel se placent et évoluent nos travaux. En effet, leurs caractéristiques sont fixées par l'impression numérique qui régit les conditions d'adhésion de notre solution à ce domaine industriel. Ainsi, nous avons choisi de cibler des systèmes ayant des configurations matérielles et logicielles répandues, tels que des composants « grand public » et un environnement GNU/Linux moderne, ainsi que des clusters de petite à moyenne dimension, c'est-à-dire étant composés de 1 à 30 nœuds. Nos travaux se consacrent également à pouvoir bénéficier des évolutions matérielles, logicielles et technologiques des prochaines années en utilisant une approche évolutive.

3.5 Conclusion

L'impression numérique repose sur une succession de traitements graphiques générant des flux de données ayant des caractéristiques, contraintes et besoins particuliers. Par conséquent, répondre à la fois aux besoins des fermes d'impression et des presses numériques en générant des flux de données aux volumétries particulièrement hétérogènes tout en garantissant des débits en sortie constants et livrés à des emplacements définis rend la problématique de nos travaux particulièrement complexe. De plus, nos travaux s'ancrent dans un environnement industriel concret qui requiert une solution flexible, évolutive et prenant en considération ses contraintes de marché et de coûts.

Deuxième partie

Prétraitements graphiques

4

Rotation et transposition de matrices rectangulaires out-of-core et out-of-place

Avant d'appliquer les traitements graphiques spécifiques au domaine de l'impression numérique, il peut être nécessaire de réaliser un prétraitement des données d'entrée. Les objectifs de ces prétraitements sont nombreux et ils ont des coûts d'exécution inégaux. Il peut s'agir de modifier l'encodage des données, tel que le format de l'image d'entrée, ou bien de modifier le contenu même des données. L'intérêt de modifier les données ou leur format est : de permettre la compatibilité des données avec la chaîne de traitement ; d'améliorer leur qualité ; ou encore d'accélérer les traitements postérieurs de la chaîne de traitement graphique.

La gestion du placement et de l'orientation des images est une composante essentielle des chaînes de traitement graphique. Elle permet notamment d'optimiser l'utilisation de la surface du support imprimé. Si l'impression doit être réalisée plusieurs fois avec la même rotation ou transposition, alors il peut être intéressant de réaliser cette rotation en prétraitement afin de générer une nouvelle image d'entrée et ainsi éviter de réaliser plusieurs fois ce même traitement. Dans notre contexte, les opérations de rotation se limitent à des rotations d'un angle de 90 degrés qui garantissent de ne pas altérer les données. Les opérations de transposition permettent quant à elles d'appliquer une rotation suivie d'une symétrie horizontale. En art graphique, cette symétrie est notamment utilisée pour les impressions au dos d'un support transparent.

La volumétrie des images traitées rend difficiles les opérations de transformations lorsqu'elle est supérieure à la quantité de mémoire vive disponible. Cette problématique est connue dans la littérature sous la classification de traitement *out-of-core*, par opposition aux problèmes *in-memory*. De plus, le besoin de conserver l'image initiale inchangée ajoute le principe de traitement *out-of-place* (la matrice transformée occupe un autre espace mémoire que la matrice d'origine), par opposition aux transformations *in-place* (la matrice transformée occupe le même espace mémoire que celle d'origine et le traitement n'utilise pas un volume de mémoire temporaire conséquent).

Ainsi, nous présentons dans ce chapitre nos travaux pour optimiser la rotation et la transposition *out-of-core* et *out-of-place* d'images stockées sous forme de matrices rectangulaires à 2 dimensions (ou *bitmap*). Nous proposons une solution efficace qui exploite les mécanismes

de cache des mémoires de masse fournis par le système d'exploitation. Les transformations sont exécutées parallèlement et optimisées par un découpage sur plusieurs niveaux en tuiles de tailles indépendantes de celle des secteurs des mémoires de masse. Nous évaluons notre solution sur quatre configurations courantes : HDD, SSD, hybride HDD-SSD et RAID 0 de plusieurs SSD. Nous montrons sa capacité à offrir une accélération significative des temps d'exécution par rapport à une implémentation optimisée manuellement par l'entreprise Caldera et nous la confrontons à la vitesse plancher d'une copie de fichier sans modification de son contenu.

4.1 Problématique et concepts

Les transformations de rotation et de transposition de matrices sont utilisées dans de nombreux domaines, par exemple au travers des *transformations de Fourier rapide* (FFT), du partitionnement en *k-moyennes* et, bien sûr, du traitement d'images. Cette section expose les problématiques et les concepts associés à ces deux types de transformations.

4.1.1 Rotation et transposition de matrices

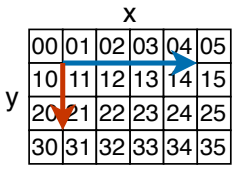
Dans ce chapitre, nous considérons une matrice source, appelée `src`, composée de 2 dimensions et de taille `H x W`, où chaque élément est défini selon ses coordonnées inscrites dans un repère d'origine `{0,0}`. Ainsi, le premier élément est situé aux coordonnées `src[0][0]` et le dernier en `src[H-1][W-1]`. Les éléments y sont stockés de façon contiguë suivant l'organisation *row-major* où les lignes sont stockées les unes à la suite des autres.

Pour une matrice `src` de taille `H=4` et `W=5`, illustrée dans la figure **4.1 (a1)**, l'application d'une rotation de 90 degrés dans le sens horaire produit la matrice illustrée dans la figure **4.1 (b1)**. De même, l'application d'une transposition produit la matrice illustrée dans la figure **4.1 (c1)**. La rotation et la transposition de matrices produisent toutes deux une matrice de destination, appelée `dst`, de taille `W x H` qui diffère par une inversion du contenu des lignes.

Les figures **4.1 (b3)** et **4.1 (c3)** présentent respectivement les algorithmes naïfs pour réaliser la rotation et la transposition. Ils implémentent ces transformations en déplaçant successivement chaque élément de la matrice `src` à sa position finale dans la matrice `dst` en suivant les formules des figures **4.1 (b2)** et **4.1 (c2)**.

4.1.2 Structure physique et adressage des mémoires

Les ordinateurs modernes continuent d'utiliser le modèle traditionnel d'un adressage linéaire pour accéder à la mémoire. Classiquement, les langages proposent un des deux grands types

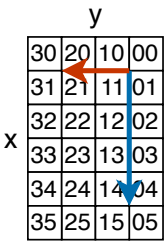


(a1) : Matrice `src`

$$i = yW + x$$

(a2) : Formule `src`

Soient :
`x` : indice de colonne de `src`
`y` : indice de ligne de `src`



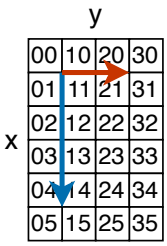
(b1) : Rotation

$$i' = (iH + H)\%(HW + 1) - 1$$

(b2) : Formule rotation

```
1 for (x = 0; x < W; x++)
2  for (y = 0; y < H; y++)
3   dst[x][y] = src[H-1-y][x];
```

(b3) : Rotation naïve



(c1) : Transposition

$$i' = \begin{cases} HW - 1, \text{si } i = HW - 1 \\ iH\%(HW - 1), \text{sinon} \end{cases}$$

(c2) : Formule transposition

```
1 for (x = 0; x < W; x++)
2  for (y = 0; y < H; y++)
3   dst[x][y] = src[y][x];
```

(c3) : Transposition naïve

Figure 4.1 : Illustration, transformation linéaire et algorithme naïf de la rotation (90 degrés, sens horaire) et de la transposition d'une matrice source de dimension `H` $\times$ `W`

de représentation pour stocker les tableaux multidimensionnels : *row-major* ou *column-major*. Ils indiquent respectivement si ce sont les lignes ou bien les colonnes qui sont stockées de façon contiguë. D'autres approches existent, telles que les *Iliffe vector* [Ili61] qui proposent de référencer les éléments à travers un premier vecteur indiquant leur adresse, mais comportent le désavantage de nécessiter un ou plusieurs accès indirects pour connaître l'adresse de chaque élément et sont généralement restreintes aux langages de haut niveau.

Dans notre environnement utilisant le langage `C` et ses dérivés, les tableaux multidimensionnels sont stockés en *row-major*. Bien que ce modèle permette d'accéder efficacement aux données d'une même ligne, il se montre inadapté pour réaliser les accès contigus en colonnes à cause des mécanismes permettant l'accès aux différentes mémoires d'un système. Or, les opérations de rotation et de transposition requièrent de transformer les lignes de données en colonnes, ou inversement les colonnes de données en lignes.

4.1.3 Hiérarchie et mécanismes d'accès des mémoires

Les ordinateurs actuels sont équipés de plusieurs technologies de stockage des données. Chacune d'elles repose sur des mécanismes physiques différents qui leur confèrent divers avantages et inconvénients. Les mémoires les plus rapides d'accès sont placées en petite

quantité au plus proche des processeurs, tandis que les mémoires plus lentes sont utilisées pour le stockage de masse des données. Cette hiérarchie des mémoires selon leur taille et leur vitesse est à la base de leur exploitation par les processeurs et les systèmes d'exploitation : basiquement, chaque niveau de mémoire est un cache pour la mémoire du dessous.

La hiérarchie classique des ordinateurs grand public actuels est composée de : trois niveaux de cache directement intégrés sur la puce du processeur (caches L1, L2 et L3), puis de la mémoire vive branchée sur la carte mère, et enfin le ou les mémoires de masse (populairement appelées « disques durs ») internes ou externes au châssis de l'ordinateur. Dans ce chapitre, nous nous intéressons aux problèmes de rotation et de transposition *out-of-core*, c'est-à-dire dont la taille des matrices à manipuler dépasse celle de la mémoire vive, et nécessite donc de réaliser des accès aux mémoires de masse. Ainsi, les accès mémoire du programme nécessitent d'être adaptés pour tous les niveaux de mémoire afin de minimiser le temps d'exécution.

Pour réduire les latences d'accès aux données de ces différentes mémoires, chaque accès à une donnée provoque automatiquement l'accès aux données contiguës et leur mise en cache par la mémoire supérieure. La quantité de données contiguës automatiquement récupérées dépend de chaque niveau de mémoire. Communément, la mémoire vive est ainsi accédée par ligne de cache de 64 octets et les mémoires de masse par *secteur* de 4096 octets. C'est de ce mécanisme que provient l'efficience des accès successifs des données d'une même ligne dans les représentations *row-major*, et d'une même colonne dans celles *column-major*.

Les systèmes d'exploitation modernes fournissent aux programmes une couche d'abstraction supplémentaire d'accès aux *secteurs* des mémoires de masse [BC05]. Cette abstraction permet un gain en portabilité et en optimisation automatique des accès à ces mémoires lentes. Sur le noyau Linux, par défaut, le contenu des périphériques de type *block*, dont font partie les mémoires de masse, est accédé via le mécanisme des *page-cache*. Une *page-cache* est une copie en mémoire vive du contenu d'un *secteur* d'une mémoire de masse, elle permet une représentation et une manipulation de plus haut niveau de leurs données tout en étant transparente pour l'utilisateur et les programmes. La gestion du nombre de *page-cache* et la synchronisation de leurs données avec le contenu réel des mémoires de masse sont entièrement assurées par le noyau. L'utilisation du système des *page-cache* permet notamment : d'optimiser les accès multiples à une même donnée en la conservant en mémoire vive ; de réordonner les opérations de lectures et d'écriture afin de les exécuter de façon optimale pour le matériel ; de temporiser l'écriture effective des données pour limiter l'écriture de *secteurs* partiels ; ou encore d'accélérer l'accès aux données suivantes grâce à des stratégies d'accès spéculatifs dénommées *read ahead*.

Cependant, bien que les *page-cache* permettent d'améliorer automatiquement les performances d'une large gamme de programmes, elles ne sont pas efficaces et sont même préjudiciables à l'exécution des algorithmes naïfs de rotation et de transposition présentées en figure 4.1 pour des matrices *out-of-core*. En effet, sur ces algorithmes parcourant les matrices à la fois

en ligne et en colonne, les mécanismes de spéculation associés aux *page-cache* augmentent sensiblement la quantité de données accédées sur les mémoires de masse, et donc par effet de bord la quantité de données mises en cache en mémoire vive. La mémoire vive étant limitée, les données inutilement récupérées provoquent alors des évictions de données utiles, ce qui génère de très nombreux défauts de cache et synchronisations d'écriture de données partielles, nuisant fortement au temps d'exécution du programme. Les performances médiocres résultant des effets de bord de la spéculation des *page-cache* sont pleinement visibles dans nos évaluations en section 4.4 : *Évaluations*.

4.1.4 Technologies des mémoires de masse

Les périphériques de mémoire de masse permettent communément de stocker individuellement entre plusieurs centaines de gigaoctets et quelques téraoctets de données, et jusqu'à plusieurs pétaoctets de données lorsqu'ils sont agrégés logiciellement. De plus, ils ont la propriété de stocker les données de façon persistante (non volatile), c'est-à-dire qu'elles sont conservées même en l'absence d'alimentation électrique. Cependant, différentes technologies, à la fois matérielles et logicielles, sont utilisées pour organiser ces mémoires à haute capacité. L'hétérogénéité de l'environnement de nos travaux rend pertinente la conception d'une solution efficace sur l'ensemble des technologies courantes de mémoires de masse.

Matériellement, deux technologies sont employées pour assurer ce stockage non volatile des données. D'un côté, celle des disques durs (HDD) qui utilise des plateaux magnétiques en rotation associés à une tête de lecture mobile. Le besoin de synchronisation entre le plateau et la tête de lecture pour accéder aux données pose une contrainte mécanique qui implique une latence importante pour accéder successivement à des données physiquement distantes. De l'autre côté, la technologie des disques électroniques (SSD) qui repose sur des puces de mémoires flash individuellement adressables. L'absence de pièce en mouvement leur permet d'accéder efficacement et parallèlement à toutes les données sous réserve d'absence de conflit entre les bus de données utilisés. Les SSD surpassent largement les vitesses maximales de lecture et d'écriture des HDD, mais sont généralement plus onéreux et limités à un nombre maximal de cycles d'écriture.

L'utilisation conjointe de plusieurs HDD ou SSD permet d'agréger les capacités, les vitesses d'accès, ou encore d'assurer la résilience des données. Cette utilisation conjointe se fait par l'application transparente pour l'utilisateur et pour les programmes d'une couche d'adressage logique des données qui vient distribuer les données et leurs accès sur les différentes mémoires de masse. Cette couche de virtualisation des mémoires de masse, appelée RAID (*Redundant Array of Independent Disks*), peut être réalisée matériellement par une carte d'extension, ou bien logiciellement au travers de pilotes informatiques et de modules noyaux spécifiques. Dans ces travaux, nous analysons exclusivement le RAID de type 0 qui permet d'agréger les capacités

et les vitesses de plusieurs mémoires de masse grâce à une répartition des données de nature *round-robin*.

4.2 État de l'art

Les implémentations naïves de rotation et de transposition de matrices *in-memory* sont génératrices d'une mauvaise localité des données ainsi que d'accès mémoire non optimisés qui produisent une explosion de la quantité d'entrées/sorties opérées sur les mémoires de masse lorsqu'elles sont étendues aux matrices *out-of-core*. Par conséquent, les problèmes d'optimisation de transformations de matrices ont été longuement étudiés dans la littérature, s'axant le plus souvent sur celle de la transposition.

Nous pouvons distinguer deux classes de techniques pour aborder la transposition de matrices *out-of-core*. La première technique consiste à utiliser une approche par bloc où la matrice est décomposée en blocs de taille acceptable pour tenir en mémoire vive. La transposition est alors réalisée bloc par bloc [Kau+93 ; AV88]. La seconde utilise le concept de transformation *multiétages* introduit en premier par Eklundh [Ekl72] pour des transpositions *in-place*. Ici, la matrice est transposée en plusieurs étapes lors desquelles les éléments sont intervertis jusqu'à l'obtention de la disposition finale. Kaushik et al. [Kau+93] ont amélioré cette approche en combinant les accès en lecture pour minimiser le nombre d'entrée/sortie engendrées. Suh et Prasanna [SP02] ajoutent la notion d'organisation et de répartition des charges d'entrée/sortie en utilisant des mémoires temporaires afin d'accélérer leur exécution. Krishnamoorthy et al. [Kri+03 ; Kri+06] ont exploré la transposition dans un contexte de matrices distribuées. Notre approche se rapporte à la technique du découpage par bloc. Cependant, nous utilisons activement les mécanismes d'abstraction des mémoires de masse fournis par les systèmes d'exploitation pour optimiser la quantité des entrées/sorties et leur exécution.

L'optimisation des opérations de transformation s'exécutant en mémoire vive permet également d'améliorer significativement les performances. Gustavson et al. [GKK12] présentent une approche parallèle efficace pour les transpositions *in-place* de matrices carrées en déterminant les principaux cycles de déplacement afin de répartir la charge des entrées/sorties et être efficace sur les problèmes *out-of-core*. Zekri [Zek15] tire avantage des instructions vectorielles du processeur pour optimiser l'exécution des permutations des données *in-memory*. Notre solution parallélise la transformation de chaque bloc en parallélisant les opérations d'écriture afin de masquer les coûts d'accès de la mémoire et repose sur l'utilisation des options du compilateur pour générer automatiquement les instructions vectorielles appropriées.

La majorité des stratégies de transposition *out-of-core* suppose que les matrices sont stockées sur des HDD et s'appliquent à pallier leurs problèmes de performance qui concernent principalement les accès non contigus. Thonangi and Yang [TY13] adressent les SSD et leurs

spécificités telle que l'asymétrie des débits de lecture et d'écriture, ainsi que leur aisance pour les accès non contigus. Pour ce faire, les opérations d'entrées/sorties sont explicitement gérées pour correspondre aux caractéristiques du SSD utilisé. Inversement, dans notre solution nous déléguons la gestion des entrées/sorties au système d'exploitation afin d'optimiser leur exécution et d'assurer la portabilité de notre solution. De plus, leur approche fournit un support pour de nombreux types de transformations en utilisant une représentation généraliste basée sur le concept *clé-valeur*. Cependant, ce mécanisme ajoute une complexité importante pour les transformations ayant un motif simple, telle que la transposition et la rotation.

4.3 Rotation et transposition rapides de matrices

Dans cette section, nous détaillons les différentes optimisations appliquées à l'algorithme naïf de rotation présenté en figure **4.1 (b3)** pour réduire son temps d'exécution. Les transformations de rotation et de transposition étant analogues, ces optimisations s'appliquent également à l'algorithme naïf de transposition. De plus, pour refléter au mieux l'environnement d'application de nos travaux, nous considérons la matrice d'entrée et celle de sortie comme des fichiers bruts (sans encodage de leur contenu) stockés sur la mémoire de masse.

4.3.1 Observations et idées initiales

La première observation de l'algorithme de rotation naïf montre l'absence de dépendance entre les données. Il est donc possible de réordonner librement leurs accès. Ainsi, nous appliquons des techniques de découpage en tuiles (*tiling*), en bandes (*strip mining*), et une parallélisation des boucles.

La première idée évidente pour accélérer la rotation, est d'éviter de lire entièrement la matrice d'entrée en mémoire pour ensuite écrire en une seule fois la matrice de sortie puisque par sa nature *out-of-core* elle est plus grande que la quantité de mémoire vive disponible et va donc générer les défauts de *page-cache* évoqués dans la section précédente. Par conséquent, la matrice d'entrée est lue par tuile via une zone mémoire temporaire de taille fixe. Pour ce faire, les boucles imbriquées d'origine sont découpées afin de former un tuilage qui couvre toute la matrice d'entrée. La rotation est ensuite réalisée sur la tuile active avant d'écrire le résultat de cette tuile.

La seconde idée est de réaliser des accès contigus pour réduire au maximum les lectures et les écritures dans le désordre d'éléments de petite taille qui sont désavantageuses pour les supports de stockage de masse. Nos expériences préliminaires ont montré que les écritures non contiguës sont plus préjudiciables que les lectures non contiguës. En effet, d'une façon générale, à cause des processus physiques nécessaires à l'écriture des données, les HDD et les

SSD présentent un débit plus faible en écriture qu'en lecture. Par conséquent, nous favorisons en premier lieu les écritures en préférant leur exécution sur des données contiguës.

Enfin, la dernière idée initiale porte sur l'accès aux données. Nos expériences préliminaires nous ont permis de nous assurer des performances équivalentes entre les écritures par appel système `write(3)` et au travers des fichiers *mappés* en mémoire par `mmap(3)`. L'utilisation de la mémoire *mappée* nous permet d'utiliser directement cette mémoire comme destination des opérations de rotation tout en déléguant le soin au système d'exploitation de réaliser les écritures physiques au moment opportun. De plus, l'utilisation de la mémoire *mappée* comme mémoire de travail nous dispense d'une mémoire tampon supplémentaire et son alignement intrinsèque avec les *page-cache* évite au système d'exploitation des copies mémoires complexes pour l'exécution des écritures. Enfin, le fait de se reposer sur le système d'exploitation pour réaliser ces écritures délicates nous assure une meilleure portabilité et intégration de notre solution au sein de logiciels existants.

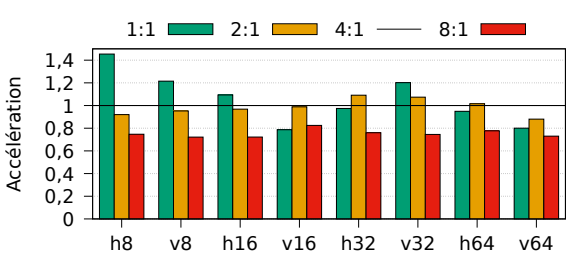
4.3.2 L'algorithme de rotation

Afin de maîtriser au mieux la quantité de mémoire utilisée par notre solution, les boucles originales de la figure **4.1 (b3)** sont découpées en tuiles rectangulaires de taille fixe. Les tuiles sont parcourues en premier sur l'ordre `y` décroissant afin d'obtenir en sortie des écritures contiguës et croissantes sur l'axe `y`, puis en second sur l'ordre `x` croissant. La taille des tuiles sur l'axe `y` doit être plus grande que sur l'axe `x` afin de favoriser la taille des écritures contiguës. Pour définir la taille des tuiles, nous utilisons la quantité de mémoire vive allouée à l'exécution de la rotation pour déterminer le volume maximal d'une tuile. Selon le type de mémoire de masse, nous créons ensuite à partir de cette volumétrie un rectangle de dimension `x_tile x y_tile` adapté à leurs propriétés. Nos expérimentations préliminaires, réalisées dans les mêmes conditions que celles du chapitre 4.4 : *Évaluations* et qui nous permettent de définir la taille idéale de la tuile, sont présentées dans la figure **4.2** où l'axe horizontal représente plusieurs matrices dont la taille est comprise entre 8 et 64 Go. À partir de leurs résultats, nous choisissons pour chaque type de mémoire de masse (HDD ou SSD) une dimension unique de tuile performante sur l'ensemble des tailles de matrices que nous évaluons :

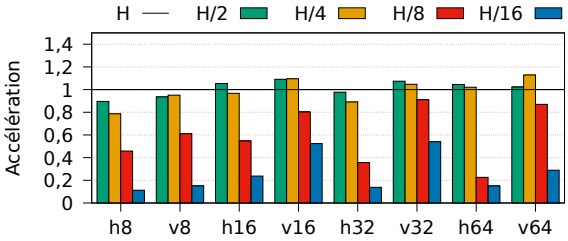
— pour les HDD, figure **4.2 (a)**, le ratio `y_tile:x_tile` que nous appliquons est de `4:1`. Ce ratio déterminé expérimentalement permet d'obtenir le meilleur compromis de performances entre les lectures et les écritures sur l'ensemble des tailles et orientations des matrices ;

— pour les SSD, figure **4.2 (b)**, nous maximisons la valeur de `y_tile` pour couvrir toute la hauteur de la matrice d'entrée, et ainsi toute la largeur de la matrice de sortie, ce qui maximise totalement les écritures contiguës.

L'exécution de chaque tuile commence par la lecture des données correspondantes de la matrice d'entrée selon les dimensions définies ci-dessus. Le rectangle de données correspondant à la tuile est lu avec des appels système `read(3)`. La lecture explicite de ces données dans la



(a) : HDD : ratio `y_tile:x_tile`



(b) : SSD : valeur de `y_tile`, avec `H` la hauteur de la matrice

Figure 4.2 : Accélération moyenne obtenue en fonction de la dimension des tuiles de lecture (`y_tile` × `x_tile`) et d'écriture (`x_tile` × `y_tile`) par rapport à la valeur choisie dans notre solution avec une mémoire temporaire `src` de 1 Go sur des matrices d'orientations et de tailles différentes (de 8 Go à 64 Go)

mémoire intermédiaire `src` permet de protéger ces données contre tout défaut de *page-cache*. Cette opération est présentée par l'algorithme 4.3.

```
1  /* (x_tile, y_tile) = taille de tuile courante          */
2  /* (bx, by)         = coordonnée de la tuile courante   */
3  for(size_t y = 0 ; y < y_tile ; y++)
4  {
5      /* ligne de la tuile à la position (by + y)         */
6      /* lecture de "haut en bas" des lignes de la tuile  */
7      off_t offset = (H -1 - (by + y_tile -1 - y)) * W + bx;
8      lseek(src_file, offset, SEEK_SET);
9      read(src_file, &src[y][0], x_tile);
10 }
```

Algorithme 4.3 : Lecture des données de la matrice d'entrée vers la mémoire intermédiaire `src`

Ensuite, le parcours des éléments de la tuile est modifié pour ajouter du parallélisme, améliorer la localité des données et favoriser l'usage des instructions vectorielles par le compilateur. Pour ce faire, l'axe `x` de la tuile est à son tour découpé en sous-tuiles de taille `INNER_X` qui sont exécutées parallèlement. Enfin, chaque sous-tuile réalise les transformations sur l'axe `x` puis sur l'axe `y`. Ces opérations sont illustrées par l'algorithme 4.4 où le parallélisme est réalisé classiquement avec l'outil OpenMP [Ope18] en suivant un ordonnancement *static* pour limiter le surcoût de synchronisation des *threads* et avec une granularité de `THD_BLOCK` que nous détaillons ci-après.

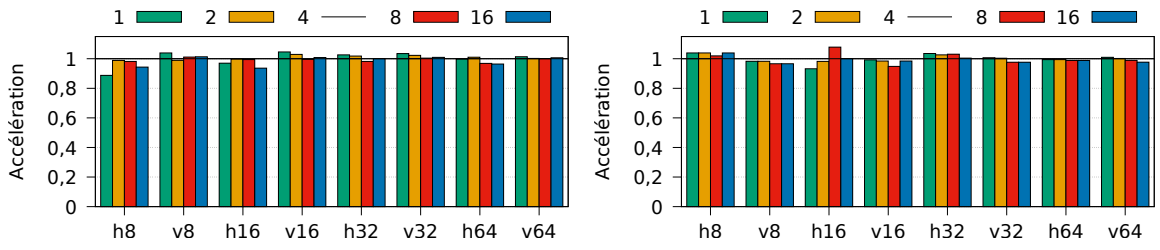
Les avantages de ce sous-tuilage sont multiples. Premièrement, les accès à la mémoire temporaire `src` bénéficient désormais d'une meilleure localité spatiale en étant contigus par bloc de `INNER_X`. Secondement, les accès à la tuile de sortie *mappée* en mémoire via la variable `dst` sont réalisés parallèlement par bloc de `INNER_X` afin de favoriser la localité spatiale tout en maintenant des écritures contiguës dans l'ordre croissant des indices par tous les *threads*.

Une valeur de `64` pour la constante `INNER_X` a été déterminée comme optimale par nos expériences. Cette valeur s'explique vraisemblablement par sa correspondance avec la taille des

```
/* (x_tile, y_tile) = taille de tuile courante */
/* boucle X */
#pragma omp parallel
#pragma omp for schedule(static, THD_BLOCK)
for(size_t out_x = 0; out_x < x_tile; out_x += INNER_X)
{
    size_t inner_max_x = MIN(out_x + INNER_X, x_tile);
    /* boucle Y */
    for(size_t y = 0 ; y < y_tile ; y++)
    {
        /* boucle intérieure X */
        for(size_t inner_x = out_x; inner_x < inner_max_x; inner_x++)
        {
            /* (bx, by) = coordonnée de la tuile courante */
            /* for the transposition use: src[y][inner_x] */
            dst[bx + inner_x][by + y] = src[y_tile -1 - y][inner_x];
        }
    }
}
```

Algorithme 4.4 : Transformation des données depuis la mémoire intermédiaire `src` vers la matrice de sortie `dst`

lignes de mémoire du cache L1 de nos plateformes d'évaluation. Conjointement, la constante `THD_BLOCK` contrôle la granularité de la boucle parallèle externe de façon à ce que chaque *thread* exécute `THD_BLOCK` sous-tuiles successives. La valeur de cette seconde constante doit être suffisamment grande pour limiter le coût de synchronisation des *threads* tout en restant suffisamment petite pour permettre une localité spatiale d'écriture raisonnable entre chaque *thread*. Bien que ce paramètre n'a qu'une faible influence dans les résultats, une valeur de 4, déterminée expérimentalement et présentée dans la figure 4.5, nous permet d'obtenir de bonnes performances sur l'ensemble des matrices. Bien sûr, les valeurs optimales des constantes `INNER_X` et `THD_BLOCK` sont susceptibles d'être différentes sur d'autres plateformes puisque leurs valeurs sont dépendantes de la configuration matérielle et du comportement du système d'exploitation. Les valeurs ci-dessus sont idéales pour nos plateformes d'évaluation équipées d'un processeur Intel Xeon D-1521 64 bits à 4 cœurs physiques *hyper-threadés*, leurs configurations sont présentées en détail dans le chapitre 4.4 : *Évaluations*.



(a) : HDD **(b) :** SSD

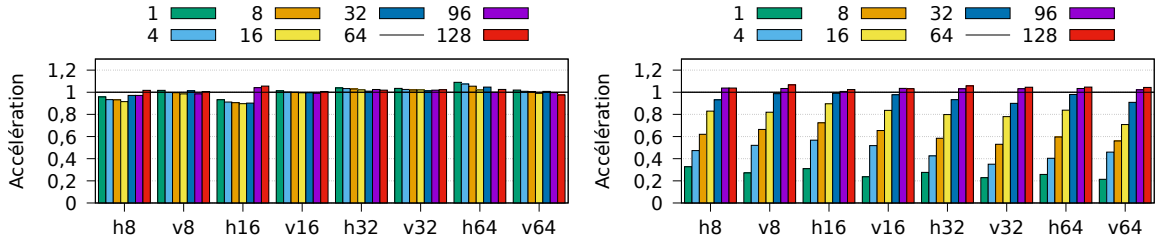
Figure 4.5 : Accélération moyenne obtenue en fonction de la valeur de `THD_BLOCK` par rapport à la valeur choisie dans notre solution avec une mémoire temporaire `src` de 1 Go sur des matrices d'orientations et de tailles différentes (de 8 Go à 64 Go)

4.3.3 Optimisations additionnelles

Nous avons également évalué l'approche d'orchestrer explicitement la synchronisation des écritures à la fin du traitement des tuiles par l'appel système `fsync(3)`. Néanmoins, nos évaluations n'ont pas montré d'amélioration tangible des performances. Nous en concluons que le noyau Linux offre nativement une gestion efficace des écritures qui ne crée pas de conflits avec les lectures effectuées lors de l'exécution de la tuile suivante.

De façon complémentaire, nous avons travaillé à optimiser les opérations de lecture de la matrice d'entrée par l'utilisation des appels système `posix_fadvise(3)`. Cet appel système permet à un programme d'indiquer ses intentions d'accès à une zone d'un fichier. Fort de ce conseil, le noyau peut alors réaliser des actions proactives asynchrones, telles que la lecture anticipée des zones concernées du fichier pour éviter la survenue des défauts de *page-cache*. En plus de permettre l'exécution de lectures asynchrones, l'absence de défaut de *page-cache* inhibe le mécanisme *readahead* qui provoque des lectures spéculatives sur des zones de la matrice d'entrée qui n'intéressent pas notre algorithme dans l'immédiat. Cependant, comme ces *page-cache* résident en mémoire vive, leur quantité est soigneusement gérée par le noyau. Ainsi, il est important de trouver un équilibre entre la quantité de données qu'il est demandé de précharger et leur accès effectif qui permettra alors de les remplacer par d'autres. De cette façon, nous découpons la lecture des lignes de la tuile d'entrée vers notre mémoire temporaire `src` par bloc de `64` lignes de manière à réaliser dans un premier temps `64` appels à `posix_fadvise(3)` suivi dans un second temps par les `64` lectures effectives via `read(3)`. La valeur de `64` a été définie expérimentalement comme une valeur pivot permettant de bonnes performances sur nos plateformes d'évaluation indépendamment de l'orientation et de la taille de la matrice transformée. En effet, nous pouvons observer dans la figure 4.6 que le HDD est peu sensible au nombre d'appels système `posix_fadvise(3)` successifs réalisés, alors qu'un palier est atteint à partir de `64` appels sur SSD. Bien que l'accélération peut être légèrement perfectionnée en choisissant une valeur supérieure sur SSD, nous choisissons la valeur pivot de `64` afin de limiter les effets de bords pouvant d'écouler d'un nombre d'appels système trop important qui viendrait invalider les premières lignes sélectionnées avant leur lecture effective sur des configurations plus modestes que celle de nos plateformes d'évaluation.

L'accélération obtenue grâce à l'utilisation de l'appel système `posix_fadvise(3)` est présentée dans la figure 4.7 pour un extrait représentatif des résultats plus complets présentés en détail dans le chapitre 4.4 : *Évaluations*. Dans cette figure, l'accélération est mesurée pour chaque configuration de stockage de masse évaluée : HDD, SSD et RAID 0 de 4 SSD par rapport à la même solution sans la présence des appels systèmes `posix_fadvise(3)`. Comme prévu, les gains obtenus sur la configuration utilisant un HDD sont relativement faibles et décroissent lorsque taille de la matrice augmente. Ces faibles résultats sont liés aux contraintes de déplacement de ses pièces mécaniques et au surcoût incompressible de latence qu'elles engendrent. En revanche, pour les configurations SSD et RAID 0 qui utilisent toutes deux des



(a) : HDD **(b) :** SSD

Figure 4.6 : Accélération moyenne obtenue en fonction du nombre consécutif d'appels systèmes `posix_fadvise(3)` avant lectures effectives des données avec `read(3)` par rapport à la valeur choisie dans notre solution avec une mémoire temporaire `src` de 1 Go sur des matrices d'orientations et de tailles différentes (de 8 Go à 64 Go)

SSD, l'accélération est importante puisqu'elle réduit, respectivement, d'un facteur moyen de 4× et de 7× le temps d'exécution par rapport à la même configuration sans l'appel système `posix_fadvise(3)`. Ainsi, l'anticipation des lectures et la réduction de la quantité de données inutilement lues sont une source importante d'amélioration des performances, en particulier pour les configurations équipées de SSD.

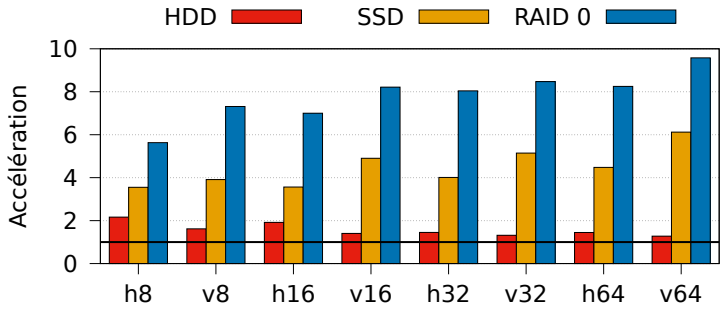


Figure 4.7 : Accélération moyenne obtenue par l'utilisation des appels systèmes `posix_fadvise(3)` avec une mémoire temporaire `src` de 1 Go sur des matrices d'orientations et de tailles différentes (de 8 Go à 64 Go)

4.3.4 Configuration hybride HDD-SSD

Les atouts de performance couplés à leur popularité croissante font des SSD une opportunité intéressante pour assister les HDD lors de l'exécution de transformations complexes. Ainsi, un SSD peut être utilisé comme support intermédiaire de travail tout en conservant les données d'entrée et de sortie sur le HDD souhaité. En dépit du surcoût lié à la copie supplémentaire des données, cette idée permet d'accélérer significativement les performances des configurations utilisant un HDD. Les gains proviennent de deux sources principales : bénéficier des très bonnes performances des SSD pour les accès non contigus ; et la possibilité d'exécuter en parallèle des lectures sur une mémoire de masse pendant des écritures sur la seconde.

Il existe deux approches pour employer cette stratégie :

1. réaliser la transformation depuis le HDD en écrivant le résultat sur le SSD, et recopier tel quel le résultat sur le HDD (HDD $\xrightarrow{T}$ SSD $\rightarrow$ HDD) ;

2. copier tel quel les données d'entrée du HDD sur le SSD, et réaliser la transformation depuis le SSD en écrivant le résultat sur le HDD (HDD $\rightarrow$ SSD $\xrightarrow{T}$ HDD).

Logiquement, la seconde approche se montre plus performante puisqu'elle tire parti des excellentes performances de lecture non contiguës des SSD tout en reposant sur la bonne performance des écritures sur HDD par le système d'exploitation.

Néanmoins, nous avons implémenté les deux approches et mesuré leurs performances respectives. La figure 4.8 présente les résultats pour les mêmes configurations que la figure 4.7 précédente : l'axe vertical présente l'accélération obtenue en fonction des différentes matrices et de l'approche utilisée par rapport à la configuration exploitant uniquement le HDD. L'accélération moyenne de la première approche est de 1,8× contre 4,3× pour la seconde. L'accélération maximale est de 8,7×. Par conséquent, la seconde approche est celle qui est retenue et qui sera évaluée en détail lors des évaluations complètes présentées dans le chapitre 4.4 : *Évaluations* sous le nom de *configuration hybride HDD-SSD*.

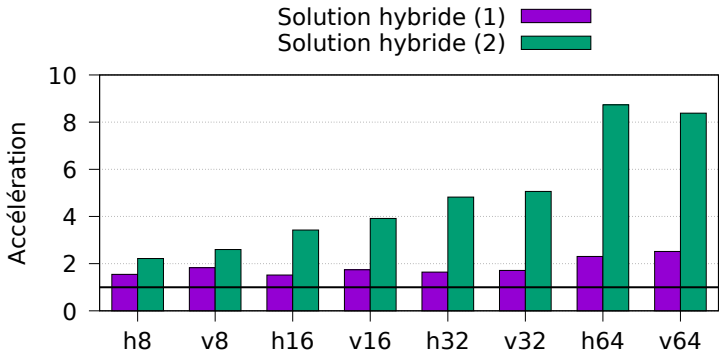


Figure 4.8 : Accélération obtenue par l'utilisation d'une configuration hybride HDD-SSD par rapport à celle d'un unique HDD avec une mémoire temporaire `src` de 1 Go sur des matrices d'orientations et de tailles différentes (de 8 Go à 64 Go), avec **Solution hybride (1)** pour la transformation du HDD vers le SSD puis copie de la sortie du SSD vers le HDD, et **Solution hybride (2)** pour la copie de l'entrée du HDD vers le SSD puis transformation du SSD vers le HDD

4.4 Évaluations

L'évaluation de notre solution de rotation et de transposition *out-of-core* d'images d'entrée, manipulées sous forme de matrice, vise à évaluer et analyser ses performances dans différentes conditions d'utilisation, aussi bien matérielles que logicielles.

4.4.1 Protocole d'évaluation

À l'exécution des programmes, le noyau Linux applique automatiquement des optimisations pour accélérer les futurs accès aux données des mémoires de masse [BC05]. Ces optimisations consistent principalement en la génération d'un système de cache utilisant la mémoire disponible. Ainsi, les mesures de performance des programmes réalisant beaucoup d'accès aux mémoires de masse sont fortement affectées par les données contenues dans ce cache. Par conséquent, avant chacune de nos expériences nous nettoyons le contenu de ces caches de lecture par l'intermédiaire de l'interface appropriée du noyau [1]. De plus, nous lui ordonnons de réaliser toutes les écritures de données qui sont en attente dans ces caches par l'utilisation de la commande `sync(1)`, au début et à la fin de chacune de nos mesures juste avant de collecter les temps d'exécution. De cette façon, nous réalisons toutes nos expériences sur un système mis dans un état stable et reproductible.

Le tableau 4.9 présente les caractéristiques des différentes matrices évaluées. Leur taille est sélectionnée pour être *out-of-core* en dépassant les 8 Gio de mémoire vive dont dispose notre plateforme d'évaluation et s'étendent jusqu'à 8 fois cette valeur pour atteindre 64 Go. La largeur et la hauteur de ces matrices sont des multiples de 1000 afin d'éviter d'aligner toutes les lignes d'une matrice sur les pages mémoire du système, puisque cela ne serait pas représentatif des caractéristiques des images d'une chaîne de traitement graphique. Enfin, les matrices étant rectangulaires nous les évaluons dans les deux orientations (portrait et paysage). Nous utilisons des matrices rectangulaires offrant un ratio tel qu'elles soient environ 2 fois plus larges que hautes (et inversement) afin de refléter le format commun des images d'entrée.

Tableau 4.9 : Noms et caractéristiques des matrices évaluées

Nom	Abréviation	Largeur (octet)	Hauteur (octet)	Ratio L/H
horizontale 8 Go	h8	125k	64k	1.95
verticale 8 Go	v8	64k	125k	0.51
horizontale 16 Go	h16	200k	80k	2.50
verticale 16 Go	v16	80k	200k	0.40
horizontale 32 Go	h32	256k	125k	2.05
verticale 32 Go	v32	125k	256k	0.49
horizontale 64 Go	h64	400k	160k	2.50
verticale 64 Go	v64	160k	400k	0.40

4.4.2 Plateforme d'évaluation et limites théoriques

Notre plateforme d'évaluation se compose de quatre systèmes identiques chacun équipé avec une configuration de mémoire de masse spécifique. Le premier, nommé *HDD*, utilise un seul HDD ; le second, *hybride*, dispose d'un SSD en plus du HDD ; le troisième, *SSD*, est équipé d'un

1. `echo 3 > /proc/sys/vm/drop_caches`

seul SSD ; le quatrième, *RAID-0* exploite un RAID logiciel composé de quatre SSD identiques pilotés par l'outil `mdadm` [2]. Le système de fichier `ext4` est utilisé sur toutes les configurations. Les références matérielles des mémoires de masse sont les suivantes :

— HDD 7200 tr/min SATA : HGST Travelstar `0J22423 - 7K1000` (version 1 To) ;

— SSD SATA : Transcend `SSD370S` (version 256 Go).

Le tableau 4.10 présente les vitesses maximales effectives de lecture et d'écriture que nous avons mesurées en réalisant des accès séquentiels à l'aide de la commande `dd(1)`. Ces vitesses correspondent aux spécifications techniques émises par les constructeurs respectifs des deux types de mémoire de masse. La configuration RAID 0 logiciel permet une accélération d'un facteur de 3,17× en lecture et de 3,87× en écriture par rapport à un SSD seul.

Tableau 4.10 : Vitesses maximales en accès séquentiels selon la configuration des mémoires de masse

	HDD	**SSD**	**RAID 0 (4 SSD)**
Vitesse en lecture	135 Mo/s	535 Mo/s	1700 Mo/s
Vitesse en écriture	125 Mo/s	310 Mo/s	1200 Mo/s

Les quatre systèmes sont équipés d'un processeur Intel Xeon D-1521 avec 2× 4 Gio de mémoire vive DDR4 fonctionnant à 2133 MT/s, sur que d'une carte mère Supermicro X10SDV-4C-TLN2F. Le système d'exploitation est Ubuntu 18.04.1 LTS doté du noyau Linux 4.15.0-43-generic. Les programmes sont compilés par le compilateur `gcc` en version 7.3.0 avec les options `-O3 -fopenmp`.

Nous réalisons la comparaison de trois implémentations différentes de rotation de matrices :

— `Simple` est l'implémentation de l'algorithme naïf présenté dans la figure 4.1 (b3) du chapitre 4 : *Rotation et transposition de matrices rectangulaires out-of-core et out-of-place* qui réalise la transformation à partir et à destination de deux fichiers *mappés* rendus accessibles par la fonction `mmap(2)` en manipulant les éléments un par un grâce à deux boucles imbriquées ;

— `Caldera` est une implémentation qui a été développée spécifiquement par la société Caldera pour la rotation rapide de matrices *out-of-core* et *out-of-place*. Elle est optimisée manuellement pour réduire la quantité de données lues et écrites en traitant la matrice d'entrée tuile par tuile tout en exécutant les écritures de façon séquentielle. Cette implémentation est utilisée dans son logiciel CalderaRIP V12.0 [Cal18] ;

— `MaRTOO` (prononcée « marteau », pour « Matrix Rotation and Transposition Out-of-core Out-of-place ») implémente notre solution utilisant l'algorithme que nous avons décrit dans le chapitre 4 : *Rotation et transposition de matrices rectangulaires out-of-core et out-of-place*.

2. Utilitaire standard des noyaux Linux pour la gestion de configurations RAID logicielles

Nous n'avons pas trouvé d'autres implémentations proposant la transformation efficace de matrices rectangulaires dans des conditions *out-of-core* et *out-of-place* pour compléter nos évaluations. Cependant, la version `Caldera` a été hautement optimisée en se basant sur les connaissances disponibles lors de son écriture, aussi la considérons-nous comme à l'état de l'art.

Contrairement à l'implémentation `Simple`, l'implémentation `Caldera` et `MaRTOO` utilisent une mémoire intermédiaire pour réaliser les transformations tuile par tuile dans la mémoire vive, avant d'écrire le résultat dans le fichier de sortie. Ainsi, nous exécutons nos évaluations avec trois tailles différentes de mémoire intermédiaire : 35 Mo, 1 Go et 5 Go. Ces trois tailles correspondent respectivement à : un usage restreint de la mémoire vive, un usage modéré, et un usage glouton. Allouer 5 Go de mémoire vive sur notre plateforme d'évaluation implique de n'en laisser que 3 Go pour le noyau et les autres programmes et services en cours d'exécution.

Nous utilisons le programme `cp(1)` comme référence pour évaluer le temps d'exécution des différentes implémentations. Celui-ci permet de réaliser une copie exacte du fichier indiqué, ce qui correspond au critère *out-of-place* sans application de transformation. Nous utilisons l'implémentation standard de `cp(1)` sans option particulière dans sa version 8.28 issue du paquet d'application *GNU coreutils*[3] fourni par défaut dans la majorité des distributions Linux. Les temps d'exécution de `cp(1)` sont collectés avec les mêmes exigences et précautions d'état stable et de reproductibilité présentées précédemment. La figure **4.11** met en évidence l'évolution linéaire des temps d'exécution de cette référence selon la taille de la matrice copiée et la configuration de mémoire de masse utilisée. De plus, les temps d'exécution confirment les vitesses de lecture et d'écriture présentées dans le tableau **4.10**. Ces deux observations démontrent la qualité et la validité de ce programme comme référence des temps d'exécution optimaux.

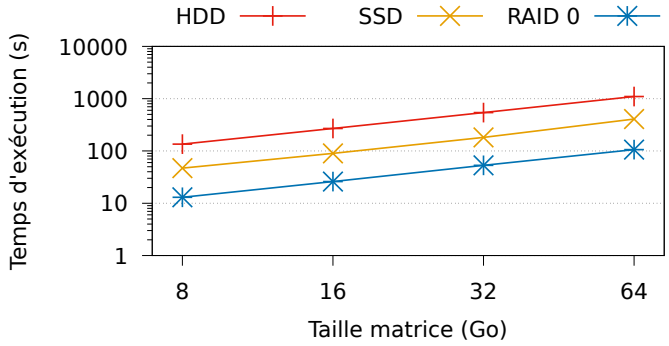


Figure 4.11 : Temps d'exécution du programme `cp(1)` en fonction de la taille de la matrice et de la configuration des mémoires de masse

Dans chacune de nos évaluations, les éléments des matrices ont une taille de 1 octet, ce qui correspond à la précision habituelle d'un canal chromatique d'un pixel d'une image

3. `https://www.gnu.org/software/coreutils/`

d'entrée. Ainsi, la taille des matrices indique également le nombre d'éléments traités. Nos expérimentations préliminaires ont montré que modifier la taille des données en conservant la même quantité totale de données n'impactait pas le temps d'exécution de la transformation. De ce fait, les résultats présentés sont valables pour d'autres types (`char`, `int`, `float`, etc.) ou tailles (`8 bits`, `16 bits`, etc.) d'éléments.

Enfin, parallèlement au temps d'exécution, nous relevons la quantité de données accédées en lecture et en écriture sur les mémoires de masse par le biais de statistiques offertes par le noyau Linux au travers de l'interface `/proc/<pid>/io/`.

4.4.3 Analyse des résultats

Les temps d'exécution de nos évaluations sur les quatre configurations sont présentés dans les figures 4.12 à 4.15. Ces figures comparent les trois implémentations pour différentes tailles de matrices. La ligne de référence représente le temps d'exécution optimal établi par l'utilisation de `cp`. Cette ligne de référence ainsi que les résultats de l'implémentation `Simple` sont dupliqués pour chaque taille de mémoire tampon évaluée puisqu'elles ne dépendent pas de ce paramètre. Pour toutes ces figures l'axe vertical est logarithmique afin d'améliorer la lecture des temps exécution des différentes matrices, cependant cet affichage minimise les écarts entre les implémentations. Pourtant, chaque unité d'écart correspond à une accélération d'un facteur $10\times$.

En conséquence des durées d'exécution importantes et des nombreuses couches logicielles et matérielles impliquées, les résultats obtenus sont particulièrement sujets à des variations de valeur entre deux mesures successives. Cependant, grâce aux précautions de reproductibilité évoquées précédemment, ces écarts de valeur restent minimes (en moyenne inférieurs à 1 % du temps d'exécution) et ne sont donc pas représentés sur ces graphiques pour améliorer leur lisibilité.

Nous avons mené les évaluations à la fois pour la transposition et pour la rotation. Cependant en raison de la similitude des résultats obtenus entre les deux transformations, nous ne présentons que les temps d'exécution de la rotation. Les valeurs manquantes pour l'implémentation `Simple` correspondent à des temps d'exécution dépassant les 10 heures. Dans l'ensemble, les résultats obtenus pour cette implémentation démontrent ses limites à traiter des matrices *out-of-core*. Cette inefficacité s'explique par l'utilisation inadaptée du cache créé par le noyau en lui imposant de mémoriser une quantité de données bien supérieure à sa capacité. Par exemple, pour la matrice *v16* de la figure 4.12, nous avons mesuré la lecture de 64 Ko de données pour l'écriture d'un seul octet !

Pour la configuration HDD présentée dans la figure 4.12, `MaRTOO` termine toutes les transformations avec un temps d'exécution quasi linéaire en fonction de la taille des matrices. Lorsque

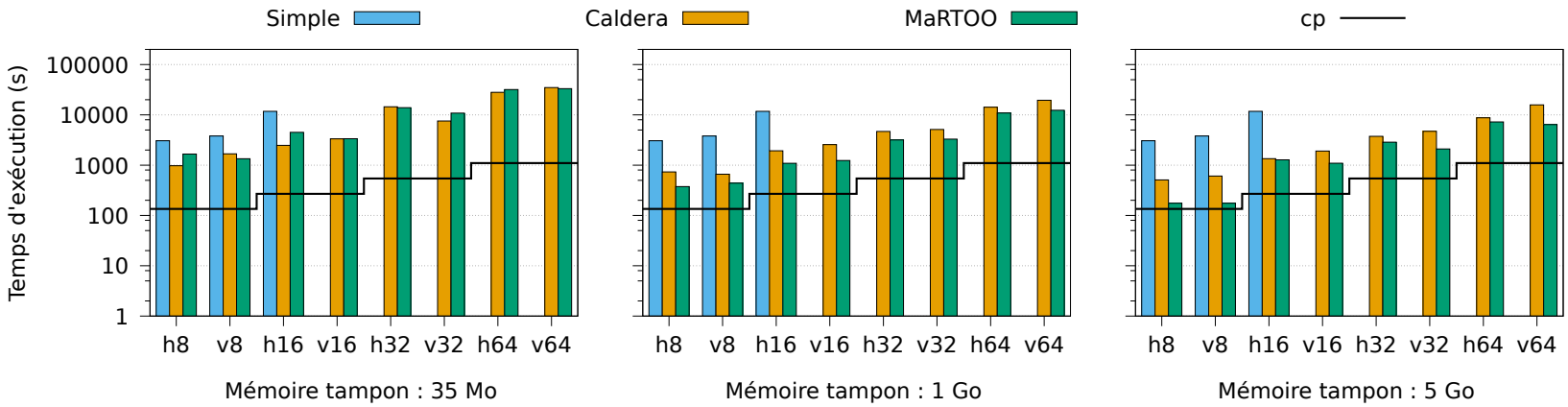


Figure 4.12 : Temps d'exécution des différentes implémentations sur la configuration **HDD**

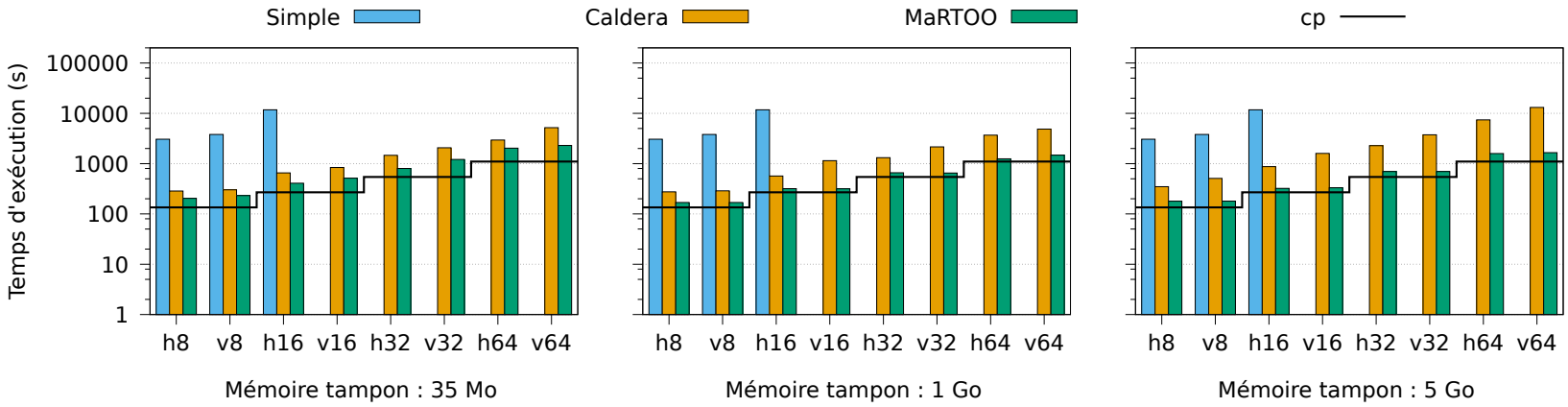


Figure 4.13 : Temps d'exécution des différentes implémentations sur la configuration **hybride HDD-SSD**

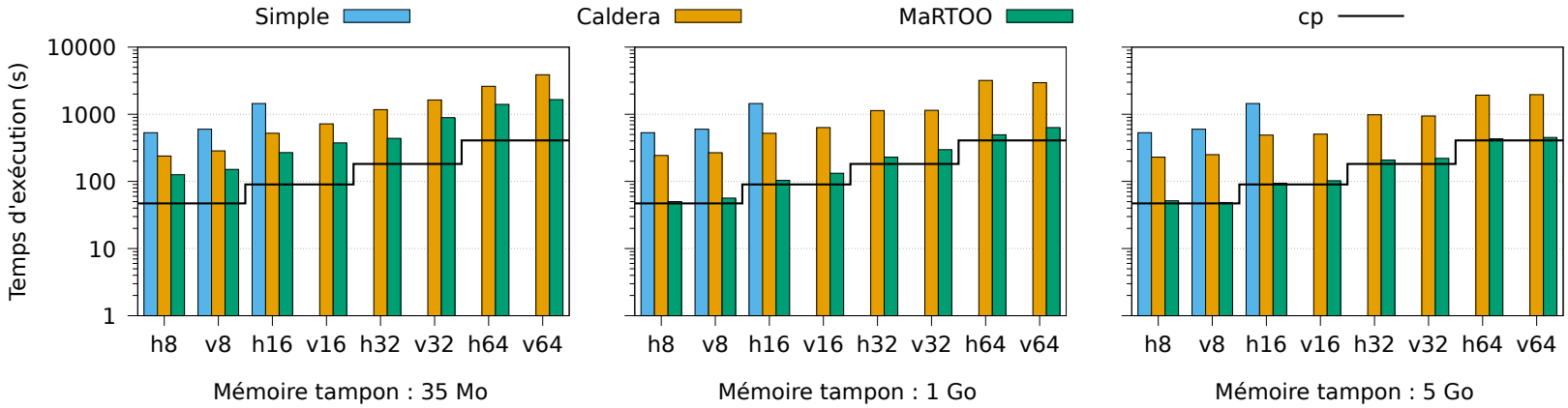


Figure 4.14 : Temps d'exécution des différentes implémentations sur la configuration **SSD**

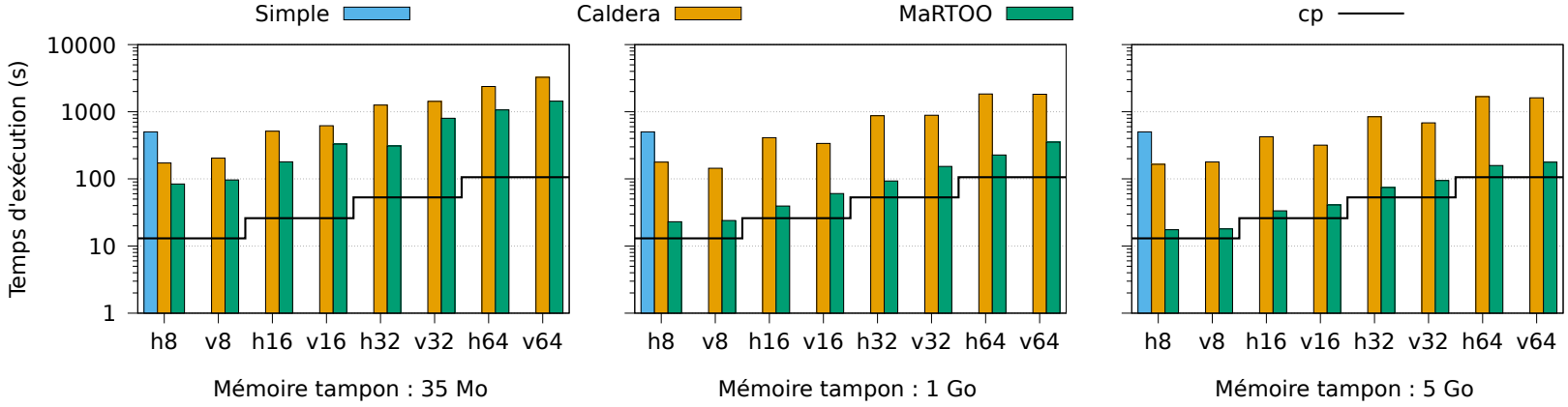


Figure 4.15 : Temps d'exécution des différentes implémentations sur la configuration **RAID 0**

la mémoire intermédiaire est très petite (35 Mo), `MaRTOO` obtient des résultats similaires à ceux de l'implémentation `Caldera`. À cause de cette faible taille de mémoire intermédiaire, les accès non contigus aux données de la mémoire de masse sont inévitables puisque la lecture et l'écriture imposent des accès divergents. Par voie de conséquence, lorsque la taille de la mémoire intermédiaire augmente (1 Go et 5 Go), les performances de `MaRTOO` dépassent celles de l'implémentation `Caldera` grâce à la meilleure gestion du ratio des accès contigus sur les accès non contigus. En outre, pour les matrices *h8* et *v8* avec une mémoire intermédiaire de 5 Gio, `MaRTOO` présente un temps d'exécution quasi optimal en flirtant avec le temps de référence. Les performances de ce cas particulier s'expliquent par le fait que seules deux tuiles suffisent pour parcourir l'ensemble de la matrice, ce qui limite fortement les déplacements inutiles du bras mécanique du HDD. Ainsi, plus la taille des matrices augmente, plus le nombre de tuiles augmente, ce qui décompose davantage les accès à réaliser par le bras mécanique, et par conséquent, dégrade les performances par rapport à la référence.

À titre de comparaison, la figure 4.13 montre les résultats obtenus avec la configuration hybride utilisant un SSD pour accélérer la transformation sur un HDD. Par rapport aux résultats de la solution HDD présentée dans la figure 4.12, à la fois l'implémentation `Caldera` et l'implémentation `MaRTOO` bénéficient de l'utilisation intermédiaire du SSD, malgré le surcoût engendré par la recopie des données. Cependant, grâce à nos accès plus performants aux données des mémoires de masse, `MaRTOO` surpasse l'implémentation `Caldera` pour toutes les tailles de matrices et toutes les tailles de mémoire intermédiaire. Avec une mémoire intermédiaire de 5 Go, le facteur moyen d'accélération est de 3,8× avec une valeur maximale de 7,9× obtenue sur les plus grandes matrices. Contrairement aux autres évaluations, il est important de noter que les meilleurs temps d'exécution de cette configuration sont obtenus avec la mémoire intermédiaire de 1 Go. Il apparaît ainsi que le noyau Linux réalise des accès aux mémoires de masse plus performants grâce à la plus grande quantité de mémoire cache disponible dans la mémoire vive non allouée. Il est ainsi en mesure de temporiser davantage les écritures sur le HDD et par conséquent de les réaliser parallèlement avec les lectures réalisées depuis le SSD, ce qui permet un recouvrement efficace de ces deux opérations indépendantes.

Les résultats obtenus sur la configuration contenant un seul SSD, présentés dans la figure 4.14, montrent une accélération moyenne d'un facteur 5,0× de `MaRTOO` sur l'implémentation `Caldera` pour une mémoire intermédiaire de 1 Go. Avec une mémoire intermédiaire de 5 Go, `MaRTOO` avoisine le temps de référence en présentant un surcoût moyen des opérations de rotation inférieur à 10 % du temps d'exécution de `cp`.

La dernière configuration évaluée, figure 4.15, porte sur un RAID 0 de quatre SSD. Ces résultats démontrent plus fortement encore l'utilisation inadaptée de l'implémentation `Simple` pour la rotation *out-of-core* puisque celle-ci ne complète désormais plus que la première matrice en un temps raisonnable. Nos analyses expliquent ce comportement par le fonctionnement des RAID 0 dont le mécanisme de répartition des données (*striping*) génère des lectures spéculatives (*readahead*) plus volumineuses qui polluent le système de cache du noyau encore

plus rapidement. De la même manière que pour la configuration SSD, `MaRTOO` atteint des performances d'autant plus proches de la référence que la taille de la matrice augmente. Nous surpassons les performances de l'implémentation `Caldera` d'un facteur d'accélération moyen proche de 10×.

Il est également intéressant de noter que les matrices verticales offrent des temps d'exécution légèrement plus faibles que les matrices horizontales. Cette observation est valable pour l'ensemble des configurations avec l'implémentation `Caldera`, et pour les configurations SSD, hybride et RAID 0 avec l'implémentation `MaRTOO`. Ces résultats s'expliquent par l'avantage des écritures plus longues offertes par l'orientation horizontale en sortie des matrices verticales en entrée. Nous interprétons l'exception de `MaRTOO` sur la configuration HDD comme l'utilisation d'un ratio (utilisé pour calculer la longueur des lectures et celle des écritures) légèrement sous-optimal. Selon nos expérimentations présentées en figure 4.2, il serait intéressant d'affiner dynamiquement ce ratio selon les caractéristiques exactes de taille et d'orientation de la matrice transformée.

La capacité de `MaRTOO` à avoisiner les temps d'exécution de ceux de la référence démontre le faible coût des opérations de rotation par rapport aux coûts d'entrées/sorties sur les périphériques de stockage de masse.

Pour conclure l'analyse des évaluations, il est intéressant d'observer les quantités de données lues et écrites sur les mémoires de masse par les différentes implémentations. La quantité effective de données écrites sur les mémoires de masse par `MaRTOO` pour les matrices de 64 Go est proche de la taille du fichier de sortie avec une moyenne de seulement 6 % de surplus sur la configuration HDD et moins de 1 % sur les configurations SSD et RAID 0. Ce surplus d'écriture est provoqué par le noyau lors de la synchronisation avec la mémoire de masse de secteurs de données incomplets. Cette valeur est plus élevée sur la configuration HDD à cause de la forme de nos tuiles qui génère une quantité supérieure d'accès non contigus. En comparaison, l'implémentation `Caldera` réalise pour les mêmes matrices un surplus d'écriture moyen de 16 % sur la configuration HDD et de 10 % sur les configurations SSD et RAID 0. Concernant la quantité de lecture, pour les matrices de 64 Go, `MaRTOO` génère un surplus moyen de 52 % (maximum à 77 %) sur la configuration HDD, de 20 % (maximum à 32 %) sur la configuration SSD et de 28 % (maximum à 65 %) sur la configuration RAID 0. Les maximums sont atteints lorsque la mémoire intermédiaire est la plus petite (35 Mo). Ce surplus de lecture est dû aux données inutilisées contenues dans les blocs de données (c'est-à-dire les secteurs des mémoires de masse) qui sont communs à deux tuiles, ainsi qu'aux écritures incomplètes qui nécessitent de relire la partie inchangée du bloc de données. Ici aussi nous faisons mieux que l'implémentation `Caldera` qui réalise pour les mêmes matrices un surplus de lecture moyen de 53 % (maximum à 126 %) sur la configuration HDD, de 56 % (maximum à 135 %) sur les configurations SSD et de 64 % (maximum à 194 %) RAID 0.

4.5 Conclusion

Les transformations de rotation et de transposition *out-of-place* posent d'importantes contraintes sur les périphériques de stockage de masse lorsque les matrices manipulées sont *out-of-core*. Nos travaux ont permis de mettre en avant les nombreuses optimisations à apporter aux algorithmes naïfs afin de réduire leur temps d'exécution.

Nous appliquons dans notre solution une stratégie de découpage par tuiles qui bénéficie des différents caches matériels et logiciels du système et utilise une exécution intra-tuile parallèle. Nous proposons une manière originale de réaliser les écritures des données dans le fichier de sortie en maximisant les écritures contiguës au plus proche du système de *page-cache* du système d'exploitation et en lui laissant le soin de propager les écritures sur la mémoire de masse aux moments opportuns. Cette stratégie permet d'exploiter efficacement les caractéristiques respectives des HDD, SSD et RAID 0 de SSD tout en étant indépendante des paramètres du système de fichiers et de la taille des secteurs des mémoires de masse, ce qui assure une meilleure portabilité par rapport aux autres approches de l'état de l'art. De plus, nous explorons la possibilité d'utiliser un SSD comme mémoire intermédiaire pour accélérer les transformations sur HDD.

Les résultats de nos évaluations démontrent l'efficacité de notre implémentation `MaRTOO` pour réaliser des rotations et transpositions de matrices rectangulaires *out-of-core* et *out-of-place*. En effet, notre solution surpasse l'approche optimisée de Caldera sur toutes les configurations de mémoire de masse évaluées en atteignant une accélération maximale proche d'un facteur $10\times$. De plus, grâce à sa gestion adéquate des spécificités des mémoires de masse, `MaRTOO` permet l'exécution de ces transformations de manière quasiment transparente sur les configurations à base de SSD. L'indépendance de notre solution par rapport au système de fichier et à l'exécution des entrées/sorties, grâce à une interaction saine avec le noyau, lui permet d'envisager des performances remarquables sur les prochaines générations de mémoire de masse, telle que les SSD interfacés en NVMe. La poursuite de nos travaux dans ce domaine tend à réduire la quantité de données lues sur les mémoires de masse en réalisant un cache manuel des données en bordure de tuiles.

Ce travail d'optimisation de l'étape de prétraitements de la chaîne de traitement graphique permet d'accélérer sensiblement l'exécution des travaux d'impression les plus volumineux.

Troisième partie

dRIP : Distributed Raster Image Processor

5 Génération distribuée de flux de données contraints et concurrents

Une chaîne de traitement graphique telle qu'utilisée dans l'impression professionnelle jet d'encre peut être formalisée comme un système générant des flux de données contraints à partir de plusieurs fichiers sources. La parallélisation de ce type de chaînes de traitement graphique implique de réaliser la distribution de ces productions de flux de données, ce qui fait apparaître des problématiques de gestion et de partage de ressources. En effet, cette distribution requiert d'utiliser conjointement plusieurs systèmes distants afin de générer de multiples flux de données pouvant être complémentaires ou distincts les uns des autres afin de satisfaire les contraintes des différents travaux soumis par l'utilisateur.

L'environnement précédemment détaillé dans la partie I : *Environnement* implique de pouvoir générer indifféremment sur un même ensemble de systèmes des flux aux volumétries et aux débits variés, mais aux contraintes d'exécution similaires. Ainsi, les plus petits flux permettent une mutualisation des ressources, alors que les plus volumineux exigent leur accès exclusif. Conformément à l'ordre de grandeur des flux, qui s'étend de plusieurs Mo/s à quelques Go/s, ainsi qu'aux aspirations de l'impression numérique, nous adressons des systèmes distribués composés au maximum de quelques dizaines de nœuds.

Ce chapitre présente les problématiques et les concepts généraux qui sont utilisés et discutés dans la suite de ce manuscrit. Ces problématiques et concepts sont largement présents en informatique et couvrent parfois différents domaines d'utilisation. Ainsi, ce chapitre s'applique particulièrement à les définir dans le cadre de nos travaux et de leur utilisation.

5.1 Problématique et concepts généraux

Dans cette première partie, nous présentons et définissons les différentes problématiques liées à la distribution des calculs dans notre champ d'application ainsi que les concepts associés.

5.1.1 Distribution et parallélisation

Comme l'état de l'art que nous présentons dans la section 5.2 : *État de l'art des systèmes existants* en témoigne, de nombreuses stratégies de répartition des calculs sur un système

complexe ont déjà été proposées et de nombreuses autres restent encore à explorer. L'ensemble de ces différentes stratégies s'articule autour de deux grands concepts : la distribution et la parallélisation. Bien que ces deux concepts définissent une répartition des calculs dont l'objectif est, en général, la réduction du temps d'exécution ou l'augmentation de la productivité, il est important d'établir les différences de vision qui séparent ces deux champs de recherche afin de comprendre leurs possibles interactions et imbrications.

Au sens large, le parallélisme peut être défini comme une répartition des calculs d'un même travail en les exécutant sur plusieurs unités de calcul. Celles-ci peuvent être les différentes unités fonctionnelles d'un même processeur, les différents cœurs logiques d'un processeur, les cœurs d'un accélérateur, ou plusieurs systèmes distants, hétérogènes ou non. Le point central du parallélisme consiste à réaliser ces exécutions simultanément. La distribution, quant à elle, se place généralement dans un contexte plus large en mixant la notion de parallélisation avec celle de la gestion des ressources, notamment pour obtenir une utilisation optimale de celles-ci. Selon les objectifs de l'utilisateur, la distribution vise communément à produire un ou plusieurs types de parallélisme dans un unique système composé d'une multitude d'unités de calcul, souvent localisées sur des nœuds différents. Ainsi, la distribution intègre les notions de résilience face aux pannes éventuelles des différents nœuds de calcul, mais également les notions de qualité de service, là où le parallélisme se concentre généralement davantage sur la problématique pure de performance des calculs. Ces aspects de tolérance aux pannes et de qualité de service forment à eux seuls deux domaines de recherche à part entière. Cependant, bien que nos travaux abordent leurs notions, elles ne composent pas l'essentiel de nos objectifs, et seront ainsi concentrées au sein de la problématique d'ordonnancement que nous abordons dans la suite de ce chapitre. Les nouveaux besoins de l'impression numérique requièrent une approche distribuée qui permet de concevoir une architecture plus flexible et d'exploiter efficacement plusieurs nœuds de calculs tout en assurant le respect des contraintes spécifiques à chaque travail d'impression.

Un système composé de plusieurs unités de calcul et qui doit exécuter plusieurs travaux peut exploiter deux types de parallélisme qui peuvent être qualifiés d'intra ou d'inter travaux. La parallélisation intra travaux permet de réduire le temps d'exécution de chaque travail en répartissant celui-ci sur plusieurs unités de calcul, alors que la parallélisation inter travaux permet d'avancer simultanément dans l'exécution de différents travaux. Dans les cas les plus simples, c'est-à-dire n'impliquant aucune dépendance au sein et entre les travaux, l'utilisation de l'une ou l'autre des deux stratégies donne un temps d'exécution global identique. Le fait de privilégier l'une ou l'autre stratégie permet en revanche d'exécuter les travaux à des vitesses différentes. Dans les systèmes complexes impliquant une multitude de nœuds exécutant une multitude de travaux ayant chacune des contraintes différentes, c'est le rôle de la distribution de savoir mélanger habilement la parallélisation intra et inter travaux dans un même système afin de satisfaire les contraintes.

5.1.2 Scale-up et Scale-out

Au sein des systèmes distribués comme des systèmes parallèles, il est possible de moduler la capacité de traitement du système en adaptant ses caractéristiques matérielles de façon à pouvoir traiter plus ou moins rapidement des problèmes de tailles différentes. Il existe deux approches pour répondre à cette problématique de passage à l'échelle : *scale-up* et *scale-out*, que l'on pourrait traduire en français par un passage à l'échelle vertical ou horizontal.

La première approche, *scale-up*, consiste à faire varier la capacité de traitement du système en faisant varier sa puissance de façon à traiter plus ou moins rapidement les données. En pratique, cela consiste à moduler les performances des unités de calcul ou des équipements réseau qui les interconnectent, l'objectif étant d'être en mesure de tirer efficacement parti de la vitesse de traitement du système. Cette approche est couramment utilisée dans les *datacenters* pour gagner en capacité de traitement sans modifier sa taille physique. La seconde approche, *scale-out*, repose sur l'idée de répartir les calculs sur plus ou moins d'unités de calcul de façon à moduler la capacité de stockage et de traitement afin de traiter des problèmes de tailles différentes. Cette approche est couramment utilisée dans le cadre des problèmes *big data* qui reposent en premier lieu sur un besoin de stockage correspondant à la quantité de données à traiter. La difficulté est ici de réaliser l'exécution simultanée d'un grand nombre d'unités de calcul contenant chacune une fraction des données.

Ces deux approches, qui disposent chacune d'avantages et d'inconvénients, peuvent être combinées afin de dimensionner un système distribué et parallèle aux problèmes qu'il doit traiter. Néanmoins, par construction les outils et solutions de systèmes distribués et parallèles ont tendance à favoriser davantage l'une de ces deux approches, notamment en fonction des usages ciblés.

L'impression numérique se retrouve idéalement dans ces deux approches de passage à l'échelle. En effet, à la fois les fermes d'impression et les presses numériques sont demandeuses d'une mise à l'échelle *scale-out* pour gagner en puissance de calcul par l'ajout de nouveaux nœuds au système, mais également d'une mise à l'échelle *scale-up* afin de pouvoir réduire le nombre de nœuds, et donc la taille du système, en exploitant efficacement des nœuds plus puissants.

5.1.3 Ordonnancement

Les mécaniques de distribution et de parallélisation impliquent la prise de décision pour réaliser la division et la répartition des calculs sur les différentes unités de calcul. L'organisation et la méthodologie de ces prises de décision sont connues sous le concept d'ordonnancement. Ce concept, très général, est présent à de nombreuses échelles dans les systèmes actuels, la plus connue étant l'ordonnancement des processus réalisé par le système d'exploitation sur les différents cœurs logiques des processeurs.

Pour les systèmes distribués et parallèles, l'ordonnancement consiste dans les scénarios les plus simples en une division et une attribution arbitraires des travaux à réaliser sur les unités de calcul. À l'opposé, dans les systèmes hautement complexes, l'ordonnancement assure une utilisation optimale des ressources selon leurs capacités et, selon les travaux à exécuter, d'assurer en plus une réaction adéquate face aux problèmes, éventuellement en les anticipant. La stratégie d'ordonnancement d'un système et la méthode qui en découle sont la clé de voûte d'une distribution et d'une parallélisation répondant efficacement aux besoins de l'utilisateur.

L'ordonnancement est une composante sensible dont la complexité varie en fonction de la complexité même du système et des contraintes à respecter. Ainsi, il n'est pas rare que les problèmes d'ordonnancement fassent partie des problèmes dits *NP-complets* dont il est particulièrement coûteux de trouver une solution optimale en raison de l'origine non déterministe des événements survenant dans un système distribué ouvert à la soumission de travaux durant son exécution [Ull75 ; LKB77 ; Sch00].

5.1.4 Tâches et graphes orientés acycliques

La division des travaux réalisée par l'ordonnancement pour atteindre ses objectifs de distribution et de parallélisation conduit à la création de sous-travaux de plus petite taille. Ces sous-travaux sont couramment dénommés tâches. Dans la représentation la plus largement utilisée, un travail peut contenir une infinité de tâches, alors que chaque tâche appartient à un unique travail.

L'exécution d'une tâche peut être dépendante de l'exécution précédente d'une ou de plusieurs autres tâches du même travail. Ces relations de dépendance s'expriment efficacement sous forme d'un graphe orienté qui permet l'emploi des outils mathématiques du domaine de la théorie des graphes. Dans le cadre de l'exécution de travaux bornés, tels que les traitements graphiques nous concernant, les graphes sont alors dits orientés et acycliques. C'est à l'ordonnanceur du système distribué de décomposer chaque travail en un ensemble de tâches formant un graphe dont l'exécution permettra d'exploiter efficacement les unités de calcul de façon à atteindre les objectifs du travail.

Tout comme l'existence de multiples échelles d'ordonnancement dans un système, il en est de même avec les travaux. Un ensemble de travaux peut faire partie d'un travail plus grand, chaque travail étant alors perceptible comme une tâche avec d'éventuelles dépendances. Cela offre une vision récursive qui permet de retrouver à une échelle supérieure la notion de tâches et de graphes avec les propriétés qui en découlent. Chaque niveau de tâches est généralement piloté par un ordonnancement dédié et spécifique.

5.1.5 Exécution et événements

L'exécution de calculs sur des systèmes distribués peut être réalisée statiquement ou dynamiquement. Dans le premier cas, il s'agit de prévoir l'exécution d'une liste de travaux connus à l'avance, ce qui permet de réaliser des optimisations basées sur une vision complète des travaux à ordonnancer. Cette approche est couramment qualifiée de *offline*. Dans le second cas, les travaux sont soumis sans prévision durant l'exécution du système, et leurs intégrations aux travaux précédemment soumis font partie intégrante des événements de la vie du système. On parle alors d'ordonnancement *online*.

Indépendamment de l'exécution statique ou dynamique des travaux, le nombre d'unités de calcul mises en œuvre au sein des systèmes distribués, et notamment leur aspect distant qui implique une exécution sur des nœuds indépendants, soumet le système à des risques de pannes dont la prédiction est plus ou moins prévisible. Dans le champ des pannes pouvant être anticipées, il est possible de citer les arrêts programmés pour maintenance ou bien les saturations d'espace mémoire. À l'inverse, quelques exemples de pannes imprévisibles sont : les classiques câbles réseau débranchés par mégarde, les défaillances matérielles comme celles des modules d'alimentation électrique ou encore les crashs logiciels de l'application ou du système d'exploitation. L'occurrence de ces pannes aléatoires augmente statistiquement avec le nombre d'unités de calcul et de composants matériels présent dans le système.

La gestion de ces événements aux conséquences variées incombe à la stratégie d'ordonnancement utilisée. Celle-ci doit, selon les objectifs de fonctionnement et de qualité de service, inclure des mécanismes de réaction et de prise de décision adaptés.

5.1.6 Génération de flux de données

Une chaîne de traitement graphique, telle qu'étudiée dans cette thèse, réalise des traitements qui génèrent en sortie une grande quantité de données pour chaque travail. Ces données disposent d'un ordre d'utilisation pour le pilote d'impression, et l'intérêt de ce dernier d'y accéder dans le respect de cet ordre induit un ordre dans l'ensemble des données. De plus, la volumétrie de ces données produites est en mesure de dépasser les capacités de stockage. Ainsi, la nature des données et leur transmission au pilote d'impression se représentent parfaitement sous forme de flux de données à délivrer de façon continue et progressive.

Cependant, ces flux de données présents en sortie ne sont pas présents en entrée du système. En effet, comme présenté dans la partie I : *Environnement* les données d'entrée des travaux sont constituées de fichiers dont le contenu complet peut être nécessaire afin de pouvoir commencer les traitements. Cette différence entre la nature des données en entrée et celle des données en sortie implique une génération de flux de données au sein du système, et non la réalisation d'opérations sur des flux préexistants.

5.1.7 Contraintes et concurrence

La complexité des travaux traités par un système implique le respect de contraintes variées. Ces contraintes peuvent être qualifiées de fixes ou de dynamiques. Les contraintes fixes sont composées des contraintes inhérentes à l'exécution même des travaux soumis. Dans notre cas d'étude, il s'agit par exemple de l'emplacement de sortie des flux de données qui est figé dans la définition du travail. Les contraintes dynamiques, quant à elles, sont liées à la vie du système et du traitement des différents travaux. Toujours dans notre cas d'étude, il s'agit par exemple des contraintes de débit de sortie suffisant et constant ainsi que de parer la survenue des famines.

La répartition d'un ou de plusieurs travaux sur une multitude d'unités de calcul ainsi que la génération de multiples flux de données en sortie fait apparaître l'existence d'une concurrence d'accès aux ressources. En effet, les unités de calcul et les équipements de communication les interconnectant disposent d'une capacité de traitement limitée qu'il peut être nécessaire de prendre en compte. Là aussi, il revient à l'ordonnancement de gérer ces concurrences afin d'atteindre ses objectifs.

5.2 État de l'art des systèmes existants

Dans cette seconde partie, nous présentons un état de l'art des outils et des solutions répondant aux problématiques introduites dans la partie précédente. Nous discutons des avantages et des insuffisances de ces solutions à s'appliquer efficacement à la génération distribuée de flux de données contraints et concurrents telle que rencontrée dans la problématique de l'impression numérique professionnelle.

5.2.1 Parallélisation

La parallélisation de travaux sur plusieurs unités de calcul, telle que décrite précédemment dans ce chapitre, est explorée depuis de nombreuses décennies et a été largement enrichie au fur et à mesure des évolutions technologiques des systèmes informatiques.

Actuellement, quatre outils se sont établis au rang de standards industriels bien répandues et constituent le socle de la parallélisation efficace des traitements complexes sur plusieurs unités de calcul. Le premier, OpenMP [Ope18], occupe une place privilégiée dans la parallélisation des programmes sur architectures à mémoire partagée et accélérateurs grâce à une API et des directives interprétées par le compilateur qui sont à ajouter dans le code source afin de générer automatiquement un code parallèle fonctionnel et performant. Le second, OpenACC [Ope19], étend le principe de fonctionnement d'OpenMP en fournissant une API et des directives

pour le compilateur afin d'organiser l'exécution de noyaux de calcul sur des accélérateurs. Cependant, OpenACC est concurrencé par l'extension « offload » d'OpenMP [Wie+14] qui gagne progressivement en maturité [New+13 ; Ant+16]. Le troisième, OpenCL [SGS10], permet aux développeurs d'exécuter des noyaux de calcul exprimés dans un langage proche du *C* sur une large variété de processeurs et d'accélérateurs grâce à une API unifiant leur accès. Les noyaux de calcul peuvent être explicitement écrits par le développeur ou bien générés automatiquement par des outils spécialisés [Ste+15 ; Liu+19]. Le quatrième, MPI [Gab+04 ; MPI15], est spécialement conçu pour supporter la parallélisation sur de nombreuses unités de calcul, aussi bien situées sur le même nœud que sur des nœuds distants. MPI est une bibliothèque qui offre au développeur une API de communication permettant d'échanger des données entre des unités de calcul et de synchroniser l'exécution des calculs. Le support natif des communications permet à MPI d'optimiser certaines opérations courantes, telles que les *gather*, *scatter* et *broadcast* qui permettent d'assigner des données à des unités de calcul ; lesquels après un ou plusieurs traitements vont regrouper les données produites parallèlement pour retourner un résultat synthétique. L'utilisation de MPI a été poussée jusqu'aux grilles de calculs hétérogènes [FK98 ; KTF03] grâce à l'environnement d'exécution générique qu'il fournit.

Comparativement à ces solutions, de nombreux autres outils proposent des approches spécialisées pour fournir aux développeurs des solutions efficaces à la parallélisation de programmes complexes. Par exemple, la technologie CUDA [Har+07 ; SK10] développée par l'entreprise Nvidia pour ses processeurs graphiques permet l'utilisation de ceux-ci pour le calcul parallèle de type SIMD. CUDA partage un fonctionnement similaire à celui d'OpenCL en fournissant au développeur un langage de programmation associé à une API pour accéder aux accélérateurs. Cependant, CUDA étant spécifique aux produits de l'entreprise Nvidia, il permet de bénéficier de fonctionnalités non supportées par le standard d'OpenCL, telles que l'interconnexion propriétaire NVLink permettant un débit supérieur aux interfaces PCIe actuelles.

De façon complémentaire, StarPU [Aug+11] propose une gestion commune des outils tels que OpenMP et OpenCL en fournissant une distribution automatique des calculs sur différentes unités de calcul d'un même nœud. La distribution automatique réalise un découpage dynamique des travaux en tâches et assure une répartition de charge de ces dernières sur les différentes unités de calcul de manière à optimiser (en temps, en énergie consommée, etc.) l'exécution. Pour ce faire, le développeur fournit pour chaque unité de calcul visée une implémentation des noyaux de calcul qui répondent aux contraintes de StarPU.

La spécialisation d'OpenMP, OpenCL et StarPU à proposer un parallélisme restreint à un seul nœud, et celle de MPI à organiser la communication entre des nœuds distants, a donné naissance à de nombreux travaux ayant pour objectif d'enrichir ces solutions avec leur savoir-faire respectif, tel que Hybrid MPI/OpenMP [RHJ09], Hybrid OpenCL [Aok+11] et StarPU-MPI [Aug+12]. Cependant, bien que de nombreux travaux s'attellent à combler les défauts résiduels de ces mélanges de solutions de parallélisation, elles constituent davantage des

extensions ponctuelles que des véritables solutions adaptées aux enjeux de la distribution. En effet, ces solutions adressent la parallélisation sans intégrer pleinement les notions de distribution discutées précédemment dans ce chapitre, telles que l'exécution simultanée sur des nœuds distants de travaux indépendants et la résilience aux pannes. Par conséquent, elles ne permettent pas de répondre à notre besoin de gestion des ressources d'un cluster.

Les besoins de parallélisme ont également été abordés par les langages de programmation eux-mêmes, soit au travers d'extensions, soit au travers de nouveaux langages. Par exemple, *Unified Parallel* C (UPC) [ES06] ajoute au langage C un modèle d'exécution parallèle ainsi qu'un adressage de la mémoire partagée par plusieurs processeurs et des primitives de synchronisation et de communication. De même, *High Performance Fortran* (HPF) [Koe+94] propose d'étendre le langage Fortran avec des directives et des bibliothèques afin de répartir l'exécution d'un programme sur plusieurs processeurs. Cependant, trop limitées par le langage auquel elles se rapportent, ces extensions sont peu utilisées. Dandelion [Ros+13] associe l'extension des langages C# et F# avec l'approche *.NET LINQ* (Language INtegrated Query) afin de proposer à la fois un nouveau langage, un compilateur, et un moteur d'exécution conçus pour l'exécution sur CPU, GPU, FPGA locaux ou situés dans le Cloud. Le langage X10 [Cha+05] propose un nouveau langage influencé par Java, mais enrichi des principes du *Partitioned Global Address Space programming model* (PGAS) [Sti09] qui permet une exécution parallèle sur différents processeurs et accélérateurs à l'aide d'un espace d'adressage commun ainsi que d'un placement et d'une exécution asynchrone des tâches. Par opposition, le langage Chapel (*Cascade High Productivity Language*) [CCZ04] reprend notamment les avantages des extensions de langage UPC et HPF afin de créer un nouveau langage, qui n'étend aucun langage existant, adapté à l'exécution parallèle et capable de passer à l'échelle depuis un modeste processeur jusqu'à un supercalculateur grâce également au modèle PGAS. Ces langages parallèles sont des outils remarquables pour paralléliser efficacement une application sur un cluster en simplifiant le modèle de programmation de tels systèmes. Néanmoins, ces langages sont contraignants pour la parallélisation d'applications existantes en leur imposant un nouveau langage qui peut restreindre l'utilisation de bibliothèques partenaires spécialisées, et une nouvelle représentation des données qui implique de repenser leurs structures.

D'autres travaux proposent des transformations automatiques de code source afin d'accélérer leur exécution. Dans ce domaine, la représentation polyédrique permet d'analyser efficacement les nids de boucles et de réaliser des transformations de celles-ci et de générer un code source de sortie dont la conformité avec le code source d'entrée est garantie [FL11]. L'idée directrice consiste à analyser le code source des programmes afin d'en extraire les dépendances d'accès aux données et d'optimiser leur exécution en améliorant la localité spatiale (proximité des données en mémoire) et temporelle (intervalle de temps entre deux accès à une même donnée) ainsi qu'en dégageant une exécution parallèle des itérations. Les outils de ce domaine se composent de bibliothèques de manipulation des polyèdres, telle que la PolyLib [Loe99] ou isl [Ver10], de générateurs de code source à partir de représentation polyédrique, tels que CLooG [Bas04] et CodeGen+ [Che12], de compilateurs source à source,

tels que Pluto [Bon+08] et CHILL [CCH08], et de compilateurs *just-in-time* (JIT), tel que APOLLO [Jim+14]. Ces outils se différencient entre eux par leur spécialité, leurs avantages de représentation et leur interopérabilité. De façon générale, les générateurs de code et les compilateurs source à source exploitent les outils OpenMP et CUDA pour exprimer le parallélisme et s'appliquent aussi bien aux processeurs généralistes, qu'aux processeurs graphiques [Ver+13 ; BRS10] et autres accélérateurs.

Enfin, partant du principe qu'il est plus efficace de demander au développeur d'exprimer son besoin avec un langage adapté à son problème plutôt que d'essayer de le déduire à partir du code source d'un langage générique, les *Domain Specific Languages* (DSL) sont des langages qui permettent de décrire une transformation à exécuter sur des données et sont couplés à un compilateur qui transformera le code source en un programme efficace en langage de plus bas niveau. Plusieurs DSL se sont spécialisés dans le domaine du traitement d'image, par exemple : Halide [Rag+13] fournit un langage conçu pour le traitement des pixels ; Diderot [Chi+12b] est dédié à l'analyse et la visualisation d'images ; et Forma [RHG15] propose une génération de code à la fois pour CPU et GPU.

Ces travaux d'optimisation des traitements sont à même de produire des codes parallèles performants qui permettent d'accélérer sensiblement les applications existantes en ne réécrivant que les portions de code concernées. Cependant, ces solutions se concentrent sur la parallélisation des traitements intranœud. Ainsi, nous articulons notre solution de façon à permettre l'utilisation de telles solutions pour les portions de code réalisant les traitements graphiques indépendamment des portions de code réalisant leur distribution.

5.2.2 Distribution et flux de données

L'explosion du *big data* et l'apparition du logiciel en tant que service (SaaS) à partir des années 2000 a mis en avant les limitations des approches parallèles pour la gestion dynamique (hétérogénéité, pannes, etc.) de nombreux nœuds de calcul exécutant de nombreux travaux concurrents simultanément et répondant à des contraintes d'exécution variées. Les nombreuses contributions scientifiques à ces problématiques ont favorisé l'existence d'une séparation claire entre la distribution des travaux et la parallélisation des calculs.

En 2004, l'entreprise Google présente son concept de MapReduce [DG10] qui propose un modèle de programmation générique permettant d'aborder le traitement distribué de grandes quantités de données. MapReduce reprend ainsi le concept de *gather* - *scatter*, notamment présent depuis les années 1990 dans MPI, sous la nomenclature *map* - *reduce*. Au-delà cette nouvelle terminologie, MapReduce propose un support natif du passage à l'échelle par *scale-out* et d'une résilience aux pannes. Ce raisonnement illustre l'angle d'approche des traitements distribués par rapport à la seule parallélisation. De nombreux projets sont nés de ce concept, ou bien s'y sont reconnus, donnant ainsi une grande variété d'implémentations du concept

de MapReduce. Aujourd'hui, une part importante de ces projets sont gérés par la fondation Apache en tant que projet de premier niveau [1].

Projet emblématique, Apache Hadoop [DG08 ; ApHado] est considéré aujourd'hui comme l'implémentation de référence du concept MapReduce. Hadoop se compose d'une collection d'outils permettant une exécution par lot (*batch processing*) d'une multitude de travaux concurrents en permettant leur distribution sur des centaines de nœuds tout en assurant la continuité de service en cas de défaillance des nœuds. Régulièrement désigné comme l'évolution de Hadoop, le projet Apache Spark [Zah+16 ; ApSpark] se propose d'améliorer les performances de Hadoop grâce aux travaux sur les *Resilient Distributed Dataset* [Zah+12]. Ainsi, Spark favorise la distribution du stockage des données directement dans la mémoire vive des différents nœuds. Cette gestion fine de la mémoire des nœuds permet à Spark de prendre un avantage considérable sur Hadoop dans les traitements qui requièrent une grande réutilisation des données produites [Seg+15 ; Men+16], tel que le *machine learning*. Thrill [Bin+16] se présente comme une variante de Spark en utilisant notamment une représentation des données sous forme de tableau, plus adaptée à la représentation de certaines données. Afin d'améliorer la localité et réduire la quantité de données échangées des programmes de type MapReduce sans sacrifier l'attribution dynamique des tâches sur les nœuds, Coded MapReduce [LMA15] propose de répliquer automatiquement certaines données sur différents nœuds en sortie des traitements.

Suivant le modèle du *dataflow programming*, ces solutions perçoivent et adressent les données des travaux à exécuter comme des tuples sur lesquels une liste d'opérations (dénombrement, modification, etc.) est à réaliser. La pertinence de ces solutions réside dans leur approche explicitement *scale-out* qui promeut la distribution des données d'entrée et de sortie sur un maximum de nœuds du système. Le stockage distribué des données permet ainsi de distribuer efficacement leurs traitements. La distribution des données est généralement fournie par d'autres projets Apache, tels que HDFS [Bor+08] qui repose sur une architecture maître-esclave pour organiser la gestion et la résilience des données entre les nœuds, HBase [Vor11 ; ApHBase] qui structure les données sous forme d'une base de données non relationnelles et peut s'installer au-dessus de HDFS, ou encore Hive [Thu+09 ; ApHIVE] qui fournit une alternative pour les données relationnelles. Toutes ces solutions offrent de nombreuses API différentes pour y accéder, par exemple sous forme de fichiers fragmentés ou bien inspirées des requêtes du langage SQL.

Le stockage intermédiaire des données avant leur traitement utilisé dans l'approche *dataflow programming* ont encouragé de nombreux projets à proposer un traitement distribué des données sous la forme de flux, aussi appelé *stream processing*, afin de minimiser ce stockage et ainsi s'affranchir des contraintes et des coûts associés. Que ce soit sous forme d'extension, telle que Apache Spark Streaming qui permet de traiter en entrée et en sortie des flux de

1. `https://projects.apache.org/`

données sous forme de *micro-batchs*, ou bien sous forme de projets totalement dédiés. C'est par exemple le cas d'Apache Apex [PRP16 ; ApApex], d'Apache Flint [Car+15 ; ApFlink] et d'Apache Storm [ApStorm] qui proposent chacun leur manière de réaliser la gestion et le traitement de flux de données. De même, la librairie TensorFlow [Aba+16] développée par Google fournit un environnement d'exécution de *machine learning* spécifique aux flux de données. Ces différents outils s'appuient sur l'exploitation forte du concept *scale-out* et proposent de réaliser diverses opérations (analyse, modification, agrégation, division, transmission, etc.) sur une multitude de flux de données entrants, provenant par exemple d'équipements de l'*IoT*. De son côté, X-Stream [RMZ13] conserve l'approche *scatter-gather* pour le traitement des données, mais organise l'exécution des traitements de façon à maximiser les accès séquentiels aux données et ainsi maximiser la bande passante des mémoires et du réseau. Enfin, Drizzle [Ven+17] propose de dissocier l'exécution des traitements des mécanismes de tolérance aux pannes afin de réduire la latence de traitement des données.

L'évolution majeure introduite par le concept de MapReduce et les nombreux défis de la distribution relevés par les implémentations qui en découlent fournissent des solutions performantes pour réaliser la distribution de travaux sur de nombreux nœuds. Cependant, leur efficacité repose sur la divisibilité des données d'entrée et de sortie, ce qui en fait des solutions clairement conçues pour l'analyse ou le traitement de grandes quantités de données existantes et non la génération de multiples flux de données contraints à partir de petites données d'entrée indivisibles. De plus, leur approche des données en tant que tuples met en avant leur conception portée sur l'exécution de petites opérations sur une grande quantité de données, et s'éloigne de la notion des traitements lourds liés la génération de données présente dans notre cas d'étude qui est davantage associée au domaine du parallélisme. Enfin, ces solutions de distribution des traitements sont conçues pour s'intégrer dans une chaîne de services successifs et comportent généralement des dépendances fortes vers d'autres outils complémentaires pour exprimer leur potentiel. Ainsi, ils sont peu adaptés à une intégration dans un environnement contraint et spécialisé.

5.2.3 Ordonnancement

L'omniprésence des besoins d'ordonnancement dans des systèmes de plus en plus complexes associé à la criticité des décisions pour permettre la bonne performance du système qu'il dirige a fait de l'ordonnancement un sujet de fond depuis plusieurs décennies [FR96 ; FR98 ; Fei+97]. Chaque solution de parallélisation et de distribution pouvant bénéficier de plusieurs stratégies d'ordonnancement, il est impossible de réaliser une liste exhaustive de ces mécanismes. En revanche, il est possible de dresser un tableau représentatif de la variété des approches existantes au sein des différents outils et solutions.

À la base des stratégies d'ordonnancement se trouvent des méthodes simples de hiérarchisation des éléments à ordonnancer : les très classiques *FIFO* et *LIFO* qui désignent respectivement

un traitement par file ou par pile ; le *round-robin* [SV96] qui assure un traitement « tour à tour » des éléments ; l'exécution en premier des plus petites tâches (Min-Min) [HSV03] ou des plus grandes (Max-Min) [ERE12 ; B+13] ou encore un mélange des deux [EN07] afin de combiner leurs avantages sans repousser indéfiniment l'exécution des tâches en fin de liste. L'avantage de ces méthodes repose sur l'absence de calcul de priorité entre les éléments tout en assurant une qualité de service stable et prévisible. L'introduction de calcul de priorité entre les éléments à ordonnancer permet d'optimiser l'usage des ressources. C'est le cas des stratégies *Rate-monotonic scheduling* (RMS) [LSD89], *Earliest deadline first scheduling* (EDF) [SB94] et *Shortest job first* (SJF) qui permettent de privilégier l'exécution d'une tâche par rapport à une autre afin de maximiser l'usage des ressources pour généralement réduire les temps d'exécution ou bien de garantir des exécutions en temps réel strict ou souple. D'autres stratégies, comme *Completely Fair Scheduler* (CFS) [Won+08 ; KS09], célèbre pour être utilisé dans le noyau Linux pour l'ordonnancement des processus, permettent d'assurer une équité de traitement à tous les éléments ordonnancés. Une approche différente, pouvant également être utilisée en complément, consiste à recourir au *work stealing* afin d'améliorer le taux d'utilisation des ressources [BL99 ; ABB00 ; Din+09]. Cette stratégie autorise les unités de calcul inutilisées à s'approprier le travail en attente chez d'autres unités de calcul. Cependant, le *work stealing* non supervisé peut créer des scénarios contreproductifs par rapport aux objectifs, en particulier sur les systèmes composés d'unités de calcul hétérogènes [Kal+14].

L'ordonnancement peut également prendre en compte de nombreux paramètres afin d'accomplir des objectifs variés. Ces ordonnancements sont alors classiquement nommés avec le suffixe *-aware*. Ainsi, il existe un large panel de stratégies d'ordonnancement favorisant des aspects différents, telles que : la localité des données *(locality-aware)*, l'utilisation des liens réseau *(network-aware)* et la consommation énergétique *(energy-aware)*. Améliorer la localité des données permet de réduire la vitesse de traitement des tâches en rapprochant les données des unités de calculs. L'approche du *Delay Scheduling* [Zah+10] propose d'améliorer la localité des données sur Hadoop en introduisant un délai d'attente afin d'augmenter les chances d'assigner la tâche sur un nœud plus proche de l'emplacement de stockage des données à traiter. D'autres approches sont possibles, telles qu'exploiter la taille et l'emplacement des données manipulées par les tâches [HS11] ou encore le placement des données et des machines virtuelles du cluster [Pal+11]. Améliorer l'utilisation des liens réseau est une source d'amélioration de la qualité de service et des performances. Cet objectif peut par exemple être accompli en exploitant les critères de qualité de service définis entre l'utilisateur et le service *Cloud* [KBK13]. Par ailleurs, Fastpass [Per+15] propose une preuve de concept d'un ordonnancement de toutes les communications d'un système distribué afin de supprimer les congestions du réseau, et donc les pertes de temps liées aux retransmissions qu'elles peuvent entraîner. S'intéresser à la consommation énergétique permet de réduire la consommation électrique d'un cluster et par conséquent son coût de fonction et son incidence environnementale. Ici aussi, plusieurs approches existent, telles que contrôler la vitesse d'exécution des tâches en fonction de la qualité de service demandée [BAB12 ; Wan+10] ou encore en limitant le nombre de nœuds allumés [Goi+10].

Pour atteindre leurs objectifs, les ordonnanceurs peuvent se baser sur plusieurs sources d'information, telles que les événements passés (*history-based*) afin d'améliorer leurs décisions futures [GRF06 ; Che+13], les caractéristiques contenues à l'intérieur des requêtes de travail (*content-based*) [Pai+98] pour déterminer les données et ressources nécessaires, ou l'exécution même des applications [LaC+13] afin de déterminer un profil type de fonctionnement (*profile-based*).

Alors que la majorité des systèmes d'ordonnancement reposent sur un modèle de prise de décisions centralisé, d'autres approches tentent d'atteindre les mêmes performances en employant une architecture distribuée [TS10]. Cet objectif peut être atteint à l'aide de langage spécifique [RLS98 ; Bos+12], ou encore d'une coopération pour réassigner les tâches des nœuds défaillants [RSZ89]. L'avantage incontestable est de supprimer le point unique de défaillance que représentent les ordonnancements centralisés, mais au prix d'un ajout de complexité au système et de contraintes dans le partage de ressources communes.

Plus largement, de nombreux outils offrent un support générique à des objectifs variés au travers de représentations généralistes de tâches à exécuter sur un système distribué. Elles se regroupent sur l'utilisation des notions de *graphe orienté acyclique* (DAG), *graphes de flot de contrôle* (CFG), *task flow* [Mae+01] ou encore *task scheduling* [Gau+13]. C'est notamment le cas de *DAGuE* [Bos+12] qui propose un moteur distribué d'ordonnancement de graphes orientés acycliques qu'il est possible de personnaliser selon les objectifs de la plateforme.

L'abondance des approches et des stratégies d'ordonnancement illustre la variété des besoins et des objectifs des applications s'exécutant sur les systèmes parallèles et distribués. Par extension, au-delà des besoins spécifiques des chaînes de traitement graphique que nous traitons dans ces travaux, à notre connaissance il n'existe pas dans la littérature un unique ordonnanceur capable de répondre aux priorités de chaque utilisateur. Ainsi, nous proposons dans nos travaux un système d'ordonnancement personnalisable qui repose sur une approche par gestion des ressources, utilisée par exemple dans YARN [Vav+13] qui est le le gestionnaire de ressources de Hadoop, et de leurs réservations [Hov+03] à laquelle nous intégrons nos contraintes et objectifs spécifiques : volumétrie des traitements, constance des débits, gestion des points de sortie logiques des données, tolérances aux pannes, etc. De plus, nous optimisons cette approche par la combinaison de stratégies *FIFO*, *locality-aware* et *network-aware* qui sont détaillées dans le chapitre 7 : *Génération distribuée des données de travaux concurrents*.

5.2.4 Partage de données inter-processus

Les systèmes parallèles ou distribués requièrent l'échange de messages entre les différents processus qui le composent. Ces messages permettent aux processus de coordonner leur action, ou bien de s'échanger des données de travail. La complexité à établir et maintenir des communications efficaces varie en fonction de nombreux paramètres tels que la distance

physique ou logique entre les processus, le nombre et le type de relation qu'ils entretiennent. L'organisation de ces communications est essentielle pour maximiser l'utilisation des différents processus et atteindre les performances maximales du système.

La littérature décrit de nombreux patrons de conception qui permettent de classifier les communications entre les processus selon leur relation. Parmi les plus usuels : l'exclusion mutuelle [Lam87 ; YA95] qui définit l'accès exclusif à une ressource ; la cohorte qui est son opposé naturel ; le rendez-vous [Wal72 ; Pat+90 ; JPA99] où les processus se retrouvent après avoir chacun effectué un traitement ; le lecteurs-rédacteurs [MS91 ; HW92 ; Cal+13] où une ressource ne peut pas être accédée lorsqu'elle est en cours d'écriture ; et le producteurs-consommateurs [BF99 ; CCD07] où les processus producteurs génèrent des données qui sont consommées par les processus consommateurs. Dans cette thèse, nous avons choisi d'utiliser la relation de producteurs-consommateurs pour organiser la génération distribuée des données tout en assurant le respect des contraintes spécifiques à une chaîne de traitement graphique. De nombreux travaux s'intéressent à l'échange efficace de données dans le patron producteurs-consommateurs avec des topologies particulières : producteur et consommateur uniques [Tor10 ; Lê+13] ; producteurs et consommateurs multiples [GST10]. Cependant, nos travaux reposent sur un cas particulier qui associe plusieurs producteurs avec un unique consommateur, ils sont présentés en détail dans le chapitre 6 : *Approche générale pour la génération distribuée de flux de données*.

La performance des échanges de données dans le patron de conception producteurs-consommateurs repose, entre autres, sur l'utilisation efficace de la mémoire tampon intermédiaire. Selon le contexte d'utilisation, cette mémoire tampon peut être organisée de différentes manières. La première basiquement, sous forme de file *premier-entré premier-sorti* (FIFO) [Tor10 ; Lê+13] force l'écriture ordonnée des données par les producteurs. D'autres représentations étendent le modèle FIFO en permettant d'utiliser une unique zone mémoire pour stocker successivement les différentes données : *circular queue*, *circular buffer*, *ring buffer* [LBC09] ou *double buffering* [Whi84]. Plus récemment, la stratégie *Bip Buffer* [Coo14] vise à garantir l'accès en écriture et en lecture de zones mémoires contiguës par l'utilisation d'un pointeur d'écriture qui est replacé en début de buffer lorsque celui-ci n'a plus la place nécessaire pour stocker la prochaine donnée de façon contiguë. L'utilisation d'une zone mémoire fixe permet de l'optimiser pour les interactions d'entrées/sorties grâce aux mécanismes de *mémoire mappée* et de moteurs *DMA* [ZK06]. Nos travaux, présentés dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*, se rattachent aux mécanismes basés sur l'allocation d'une zone mémoire fixe et tire parti des particularités de son environnement pour proposer l'écriture concurrente et désordonnée des données par les producteurs.

Complémentaires au stockage des éléments dans la mémoire tampon, les mécanismes de synchronisation des accès pour l'écriture et la lecture génèrent une attention particulière de par leur criticité dans les performances résultantes. Dans nos travaux, nous exploitons les principes de *lock free* [Bar94 ; FH07 ; LBC09] et *compare-and-swap* de données [Val95] qui permettent de

réduire les latences de synchronisation en évitant les coûteuses opérations de verrou proposées par les classiques mécanismes *mutex* et *futex* [WT95 ; Dre05].

Enfin, de nombreux travaux sur l'échange de données entre producteurs et consommateurs tirent parti des mécanismes de *Remote Direct Memory Access* (RDMA) qui permettent d'écrire ou de lire directement dans une mémoire distante [ST03 ; LWP04] et embarquent la synchronisation de ces accès à travers les échanges réseau afin de minimiser les surcoûts de latence [BH15]. Les transferts de données par RDMA permettent d'optimiser les débits des transferts et de décharger les processeurs des copies de mémoire. Néanmoins, malgré les évolutions pour s'intégrer dans les réseaux traditionnels [BK11 ; RA07] il est nécessaire de disposer de cartes réseau compatibles. Par conséquent, nous avons choisi de ne pas faire usage de RDMA pour réaliser les transferts, ce qui toutefois ne nous empêche aucunement d'atteindre les performances maximales de notre plateforme d'évaluation, tel que présenté dans la section 11.2 : *Débits maximaux de l'architecture*.

5.3 Conclusion

Ce chapitre a permis de mettre en avant l'étendue des travaux menés dans les domaines de la parallélisation, de la distribution, de l'ordonnancement des travaux et de l'échange de données entre processus qui sont indispensables dans ces systèmes mettant en œuvre plusieurs unités de calcul. Néanmoins, à notre connaissance, aucune des solutions actuellement présentes dans l'état de l'art ne permet de résoudre efficacement la problématique de génération distribuée de flux de données contraints et concurrents. Cependant, la variété des travaux existants fournit des stratégies et des méthodes qui sont indispensables à la résolution efficace de ce défi.

6 Approche générale pour la génération distribuée de flux de données

L'état de l'art ne fournissant pas de solution efficace pour réaliser la génération distribuée de flux de données contraints et concurrents, nous présentons dans ce chapitre la solution conçue et développée dans cette thèse [God+19]. Pour ce faire, ce chapitre expose une vue globale du système permettant de présenter ensuite les différentes parties de ce système distribué, leurs rôles et leurs interactions. Les chapitres suivants de ce manuscrit, chapitres 7, 8 et 9, abordent ces différentes parties et le détail de leurs principes de fonctionnement.

La solution présentée ici offre une approche généraliste pour représenter et traiter tout problème similaire à celui de RIP distribué abordé dans cette thèse, tel que la génération de flux de données pour d'autres types de périphériques que les imprimantes, ou encore la génération de flux multimédia. Afin d'exprimer de façon plus complète ses principes de fonctionnent, nous présentons à la fois la structure générique de notre solution ainsi que son application au domaine spécifique de l'impression numérique jet d'encre professionnelle. Le support matériel de cette organisation est un cluster aux nœuds et interconnexions standards.

6.1 SPC : Scheduler Producers Consumers

Nous adressons les défis de la génération distribuée de flux de données contraints et concurrents par la conception d'une architecture distribuée que nous désignons sous l'acronyme *SPC*. Cet acronyme reflète l'utilisation de trois éléments principaux respectivement nommés : *Scheduler*, *Producers* et *Consumers*. La figure **6.1** propose une vue d'ensemble de ces trois éléments ainsi que leur organisation et leur intégration afin de former un système distribué performant. Les détails de cette figure sont expliqués tout au long de ce chapitre. L'existence explicite de ces trois éléments offre à notre solution une séparation claire des rôles qui permet de combiner une représentation flexible avec une exploitation efficace des ressources.

Similairement aux diverses solutions étudiées à travers l'état de l'art, nous abordons la génération des flux de données avec une approche haute performance, sans considération d'une exécution en temps réel strict ou souple. Bien que l'application de notre solution au pilotage d'équipements électroniques soit fortement liée au respect d'une exécution temps réel fort, deux principales raisons s'opposent à proposer une solution en ce sens. Premièrement, la génération du volume de données visées ainsi que les débits associés sont en déséquilibre avec les capacités

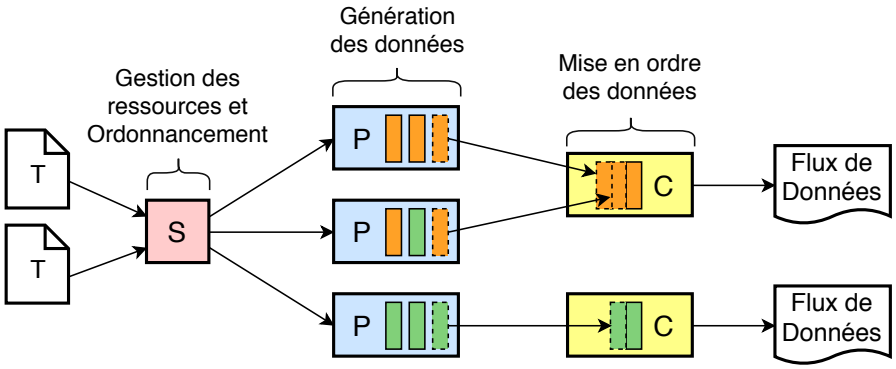


Figure 6.1 : Vue d'ensemble et répartition des rôles des trois éléments de notre solution, avec **T** pour Travail, **S** pour *Scheduler*, **P** pour *Producer* et **C** pour *Consumer*

de communication réseau des solutions temps réel existantes qui sont de l'ordre de 100 Mb/s [1] à l'heure où les réseaux locaux non-temps réel permettent communément un débit minimum de 1 Gb/s. Secondement, les traitements graphiques comportent de nombreuses étapes de calcul et requièrent l'utilisation de bibliothèques propriétaires. Ces spécificités rendent difficile l'évaluation des temps limites d'exécution nécessaires au support du temps réel. Ainsi, bien que les exécutions temps réels fournissent des garanties intéressantes, une approche temps réel entraîne un fort ajout de complexité et induit une sous-utilisation des ressources réseau. En revanche, nous proposons de fournir une qualité de service compatible avec des besoins de génération de flux de données constants, tout en permettant une utilisation maximale des unités de calcul et des équipements réseau associés, grâce à une utilisation de tampons de données paramétrables permettant d'absorber une grande variété d'aléas. Nos résultats concernant cet objectif sont spécifiquement évalués dans la section 11.4 : *Qualité de service des traitements graphiques*.

6.1.1 Interaction avec l'utilisateur

Notre solution s'intègre pleinement dans un environnement concret qui interagit avec elle. Nous nommons cet environnement générique sous le terme d' *utilisateur*, celui-ci pouvant aussi bien être un utilisateur physique interagissant directement avec le système distribué, ou bien être constitué d'un ou de plusieurs autres logiciels faisant ainsi de notre solution une brique d'un flux plus complexe.

Le premier rôle de l'*utilisateur* est la soumission de travaux à notre système. Cette soumission se fait de façon centralisée auprès de l'élément nommé *Scheduler*. C'est également par cette unique interface de communication que l'utilisateur peut également contrôler les travaux, par exemple demander leur arrêt. Le second rôle de l'utilisateur est bien évidemment de récupérer les résultats issus des travaux qu'il a soumis. Conformément aux différentes contraintes que

1. `https://www.ethercat.org/en/technology.html`

notre solution permet de supporter, l'utilisateur accède aux données produites sur le ou les nœuds de son choix, les données étant alors formatées et organisées en respect des contraintes indiquées dans le travail duquel elles sont issues.

L'organisation des interactions avec l'utilisateur, c'est-à-dire de part et d'autre du système, permet de représenter notre solution en tant que service, ayant pour rôle de réaliser les travaux soumis en respect avec les contraintes de ceux-ci. Cette représentation permet de faciliter l'intégration dans une grande variété de systèmes plus complexes ayant des contraintes spécifiques.

6.1.2 Distribution et parallélisation

Conformément à la séparation entre distribution et parallélisation évoquée dans le chapitre précédent, notre solution permet une exploitation simultanée de ces deux concepts. La distribution est assurée par une double répartition : premièrement, une répartition des différents travaux sur les différents nœuds afin de tirer parti de leur exécution concurrente pour maximiser l'exploitation des ressources du cluster ; secondement, la répartition d'un même travail sur différents nœuds pour lui faire bénéficier de la puissance de plusieurs nœuds. Le concept de parallélisation est quant à lui pleinement exploité dans l'objectif de maximiser l'usage des ressources internes aux nœuds. Afin d'éviter d'éventuelles interférences entre distribution et parallélisation, tout en permettant une flexibilité dans l'utilisation des unités de calcul, nous intégrons les éléments *Producers* chargés de produire les données et que nous définissions plus en détail dans la suite de ce chapitre.

6.1.3 Ordonnancement centralisé

L'exécution coordonnée des travaux sur le cluster est assurée par un ordonnanceur dédié : le *Scheduler*. Ce processus organise l'exécution des travaux en leur assignant les ressources du système. Son objectif principal est de réaliser une organisation efficace des travaux permettant d'en exécuter un maximum en concurrence tout en assurant le respect de leurs contraintes.

Nous avons choisi un ordonnanceur centralisé pour faciliter la prise de décision. Cette centralisation fournit de nombreux avantages dans cet environnement. Premièrement, toutes les décisions sont prises en un point unique, ce qui réduit les risques de latence et supprime les conflits entre les décisions qui peuvent être difficiles à résoudre. En effet, des recherches précédentes ont tenté de conserver ces avantages sur des architectures d'ordonnanceur distribués, mais ont inévitablement mené à compliquer la prise de décision [TS10 ; RSZ89 ; RLS98], réduire ses possibilités ou bien introduire le besoin de messages supplémentaires de synchronisation. Deuxièmement, un ordonnanceur centralisé offre la possibilité de fournir à l'utilisateur un point unique de communication pour interagir avec notre solution, ce qui simplifie gran-

dement son utilisation, cf. section 6.1.7 : *Organisation et protocoles de communication*. Ainsi, nous faisons de l'ordonnanceur le processus chargé de recevoir les travaux soumis au système par l'utilisateur et de contrôler les différentes requêtes les concernant. Troisièmement, cela simplifie également les communications avec les éléments de notre solution puisque ceux-ci se réfèrent à une autorité unique.

Bien que l'ordonnanceur centralisé procure de nombreux avantages, son choix dans un système distribué implique la formation d'un point individuel de défaillance dont la panne mène à l'arrêt complet du système. Néanmoins, nous privilégions cette architecture qui permet une grande efficacité de décision pour toutes les politiques d'ordonnancement et se dimensionne sainement jusque dans les clusters de taille moyenne, puisque ceux-ci sont soumis à des probabilités et des taux de pannes relativement faibles [SG09 ; CAK16] qui sont acceptables avec les scénarios d'utilisation de notre solution dans un environnement industriel d'impression numérique. De plus, en section 6.2 : *Limites et extensions* nous proposons plusieurs méthodes permettant de modérer l'impact d'une défaillance tout en conservant l'approche centralisée. La centralisation présente également des risques de congestion des communications que nous adressons en section 6.1.7 : *Organisation et protocoles de communication* par la définition claire de protocoles de communication qui limitent le nombre et la quantité d'information de contrôle circulant depuis et vers l'ordonnanceur.

En tant que seul élément décisionnaire de notre système, l'ordonnanceur centralise également les informations à propos des différentes ressources du cluster. Il dispose ainsi d'une vision complète du système et de l'attribution des ressources aux différents travaux en cours d'exécution. Cette vision permet d'optimiser les décisions en fonction des caractéristiques des travaux soumis, mais également en fonction des caractéristiques des unités de calcul présentes et du réseau.

L'ordonnanceur réalise le découpage des travaux en *tâches* qu'il assigne aux différentes unités de calcul. L'assignation de ces tâches mène à l'exécution parallèle des différents travaux et au respect de leurs contraintes. C'est à travers ces tâches que l'ordonnanceur pilote la génération des données ainsi que leur formatage et acheminement aux points de sortie indiqués par l'utilisateur. L'environnement d'exécution du système impliquant des événements imprévisibles (soumissions des travaux, pannes, etc.), notre ordonnanceur opère en prenant l'intégralité de ses décisions durant l'exécution du système. Cette exécution dynamique lui offre la possibilité d'assurer un équilibrage de charge entre les unités de calcul en fonction des caractéristiques et des besoins des nouveaux travaux soumis.

6.1.4 Producteurs et consommateurs

Conformément à l'acronyme *SPC*, en plus du *Scheduler*, notre solution comporte deux autres éléments : les *Producers* et les *Consumers*. Ces deux éléments sont des entités logiques qui, contrairement à l'ordonnanceur, peuvent chacune avoir plusieurs instances dans le système.

Le ou les *Producers* sont chargés de produire les données des différents travaux exécutés sur le système distribué. Leur rôle est d'exécuter avec les meilleures performances possible les différentes tâches assignées par l'ordonnanceur et de l'informer quand celles-ci ont été réalisées. Comme évoqué précédemment, ces producteurs de données permettent d'abstraire les ressources physiques du système et apportent ainsi une flexibilité dans la composition physique du cluster. Ceux-ci sont des éléments génériques pouvant représenter diverses formes d'unités de calcul, telles que par exemple : un ou plusieurs cœurs logiques d'un processeur, une carte accélératrice, ou encore l'ensemble des ressources d'un nœud. L'exécution des tâches est laissée à la discrétion des producteurs, en fonction de leur implémentation et de leur configuration. Par exemple, un producteur regroupant plusieurs unités de calcul peut choisir d'exécuter une tâche sur un cœur spécifique du processeur ou bien sur une carte accélératrice particulière, ainsi qu'exécuter les tâches assignées séquentiellement ou parallèlement. Cette abstraction des unités de calcul permet deux avantages majeurs. Premièrement, cela offre une forte capacité d'adaptation à l'exécution de problèmes variés. Secondement, cela permet à l'ordonnanceur de piloter de façon homogène toutes les unités de calcul du système, quelles que soient leurs éventuelles hétérogénéités. Ces mécanismes de prise en compte de l'hétérogénéité sont détaillés le chapitre 7 : *Génération distribuée des données de travaux concurrents*.

Le ou les *Consumers* du système ont un rôle complémentaire à celui des *Producers*. Conçus en tant que consommateurs des données produites par les producteurs, ils sont en charge d'agréger les données produites par ceux-ci afin de former les flux de données à transmettre à l'utilisateur. Les consommateurs jouent ainsi le rôle d'interface entre l'utilisateur et le système distribué. Ils permettent d'abstraire la complexité du système distribué en formant les points de sortie logiques des flux de données produits, indépendamment du ou des producteurs ayant généré les données du flux. L'intégration des points de sortie logiques au sein de la solution permet à l'ordonnanceur d'optimiser le calcul des données et leur transmission jusqu'à eux.

En tant qu'éléments logiques, nous n'introduisons aucune contrainte de placement des producteurs et des consommateurs dans le cluster. Ceux-ci sont parfaitement indépendants les uns des autres et peuvent aussi bien être placés sur le même nœud que sur des nœuds distants. Le seul besoin est d'être en mesure de communiquer entre eux et avec l'ordonnanceur.

Notre solution permet l'ajout et le retrait dynamique de producteurs ou de consommateurs. Cela apporte une grande flexibilité dans l'usage matériel des ressources du cluster. En fonction de ces événements d'ajout ou de retrait, l'ordonnanceur modifie sa représentation interne du

cluster et prend les décisions appropriées pour tirer parti des nouvelles ressources, ou bien assurer la continuité de l'exécution des travaux en cours. Cette gestion native des retraits permet également à l'ordonnanceur de supporter d'éventuelles pannes ou déconnexions de producteurs ou de consommateurs. Le fait pour une ressource de quitter le cluster (de façon volontaire ou inattendue) introduit des conséquences différentes qu'il s'agisse de producteurs ou de consommateurs. Le retrait d'un producteur contraint l'ordonnanceur à réassigner dynamiquement ses tâches à un autre producteur disponible. Si aucun producteur n'est disponible, alors l'ordonnanceur peut être dans l'obligation d'arrêter le traitement du travail ou des travaux concernés ou bien d'interrompre d'autres travaux, en fonction de sa politique. En revanche, le retrait d'un consommateur implique l'impossibilité pour l'ordonnanceur d'honorer la contrainte de point de sortie logique des données des travaux lié à ce consommateur, ce qui, sauf politique spécifique de l'ordonnanceur, entraîne nécessairement l'arrêt de ces travaux.

L'utilisation dans notre solution de producteurs et de consommateurs peut facilement être associée au patron de conception *publish-subscribe* (ou *pub/sub*) [Eug+03]. Cependant, plusieurs différences avec ce concept sont à signaler. Premièrement, la connexion des deux éléments n'est pas initiée par le consommateur, mais par le producteur sur ordre de l'ordonnanceur. Secondement, nous ne proposons pas de duplication des données produites par le producteur vers de multiples consommateurs, chaque donnée produite est à destination d'un unique consommateur. En effet, il n'existe pas dans l'impression numérique de cas pratique où deux drivers différents requièrent les mêmes flux de données avec la même synchronisation.

6.1.5 Topologies flexibles

L'exécution distribuée et parallèle de travaux sur diverses unités de calcul associés à la transmission des données à l'utilisateur depuis des points de sortie logiques qui sont contraints dans la définition même des travaux, impose à notre solution de supporter plusieurs topologies logiques d'interconnexion des producteurs et des consommateurs, d'autant plus que notre solution permet un placement flexible des producteurs et des consommateurs sur le cluster.

Chaque travail soumis à l'ordonnanceur entraîne ainsi la formation d'une topologie liant un ou plusieurs producteurs à un ou plusieurs consommateurs. Ces différentes topologies logiques résultant de l'emplacement des calculs décidé par l'ordonnanceur et des points de sortie imposés par le travail peuvent être répertoriées en quatre classes illustrées par les figures **6.2 (a)**, **(b)**, **(c)** et **(d)**. Ainsi, chaque travail traité par notre solution forme l'une des topologies suivantes :

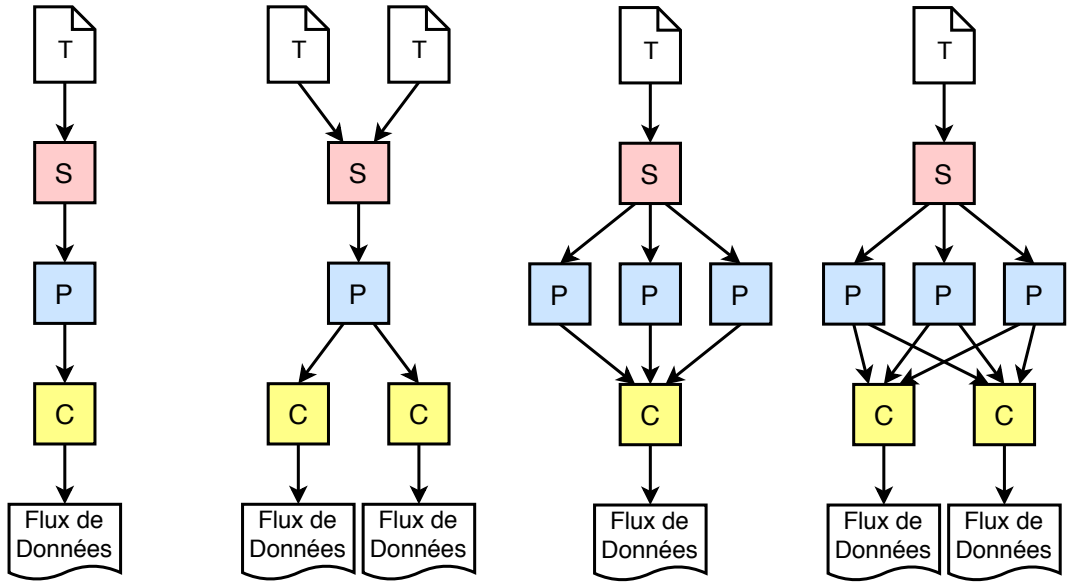
— **one-to-one** : association simple d'un seul producteur délivrant toutes les données produites à un seul consommateur (dans notre contexte, cas d'un unique nœud produisant les données pour une unique machine pilotant une seule imprimante standard) ;

— **one-to-many** : association d'un seul producteur à plusieurs consommateurs dans le cas d'une livraison des données produites sur plusieurs points de sortie distincts (dans

notre contexte, cas d'un unique nœud produisant en parallèle les données pour plusieurs machines pilotant chacune une seule imprimante standard) ;

— **many-to-one** : association de plusieurs producteurs à un seul consommateur afin de profiter de la puissance de calcul de plusieurs producteurs tout en conservant un unique point de sortie des données produites (dans notre contexte, cas de plusieurs nœuds produisant les données pour une unique machine pilotant une seule presse numérique) ;

— **many-to-many** : association de plusieurs producteurs délivrant chacun les données produites à plusieurs consommateurs, utilisée lorsqu'il y a une dépendance entre les données produites différente de la division des données demandée (dans notre contexte, cas de plusieurs nœuds produisant les données pour plusieurs machines pilotant une même presse numérique avec séparation des canaux chromatiques).



(a) : one-to-one **(b) :** one-to-many **(c) :** many-to-one **(d) :** many-to-many

Figure 6.2 : Représentation des différentes topologies possibles entre les producteurs et les consommateurs, avec **T** pour Travail, **S** pour *Scheduler*, **P** pour *Producer* et **C** pour *Consumer*

L'exécution simultanée de plusieurs travaux sur le cluster implique la présence simultanée de plusieurs topologies logiques, identiques ou non, pouvant partager ou non, des producteurs et consommateurs communs. Notre solution assure une parfaite concurrence de ces topologies en laissant à l'ordonnanceur le soin de les mapper sur les différentes ressources du cluster afin d'optimiser l'exploitation de ses ressources. De plus, l'équilibrage de charge sur les différents producteurs assuré dynamiquement par l'ordonnanceur lors de son exécution implique une éventuelle évolution des topologies logiques au cours de l'exécution des travaux.

Le support natif et unifié de ces quatre topologies ainsi que l'absence de prérequis matériels assure à notre solution de s'adapter à une large collection de cas d'usage tout en permettant à l'ordonnanceur d'appliquer une politique capable exploiter au mieux l'ensemble des ressources du cluster.

6.1.6 Passage à l'échelle et pérennité

Nous suivons une approche à la fois *scale-out* qui permet l'utilisation de nœuds supplémentaires pour exécuter des travaux plus nombreux et plus complexes, ainsi qu'une approche *scale-up* par l'exploitation du parallélisme au sein des nœuds fournit par l'abstraction offerte par les producteurs.

Le support de l'ajout et du retrait dynamique de producteurs et de consommateurs durant l'exécution permet à notre solution de proposer un passage à l'échelle sur une large variété d'usage et de clusters. Cet ajout ou retrait dynamique de ressources permet à la fois de faire évoluer les ressources disponibles en fonction de la charge du système et des besoins, mais aussi de supporter aisément des politiques liées au *green computing*, où les nœuds sont allumés et éteints à la demande.

De plus, l'absence de prérequis matériels permet à notre solution d'être pérenne vis-à-vis de l'évolution des technologies de calcul et de réseau, ce qui facilitera leur exploitation.

6.1.7 Organisation et protocoles de communication

Une bonne organisation des communications est indispensable pour atteindre de bonnes performances lors de l'exploitation d'un cluster. Pour ce faire, nous établissons une organisation claire des communications entre les différents éléments de notre solution. Elle permet de renforcer les rôles de chacun et leur permet d'exploiter pleinement les différentes ressources du cluster en limitant les délais d'attente, les opérations et les messages de contrôle inutiles.

La figure 6.3 présente la connexion des producteurs et des consommateurs auprès de l'ordonnanceur afin de s'y enregistrer comme ressources disponibles. Le détail des informations fournies lors de cet enregistrement est présenté dans le chapitre 7 : *Génération distribuée des données de travaux concurrents*. Cette identification se limite à un unique message fournissant toutes les informations nécessaires suivies d'une unique réponse de l'ordonnanceur validant ou refusant l'enregistrement de la ressource. En cas d'acceptation, le producteur ou le consommateur restent considérés comme disponibles en l'absence d'événements contraires, tels qu'une déconnexion explicite ou bien une perte de connexion détectée par notre système de communication présenté dans le chapitre 9 : *Aspects techniques de la gestion des communications*.

La figure 6.4 présente les différents messages échangés pour l'exécution de travaux au travers de leurs découpages en tâches que nous présentons dans la section 7.1.4 : *Tâches*. Elle se déroule en trois phases :

1. Premièrement, l'utilisateur soumet à l'ordonnanceur le travail, celui-ci vérifie la capacité du système à l'exécuter en vérifiant la présence des consommateurs requis pour les

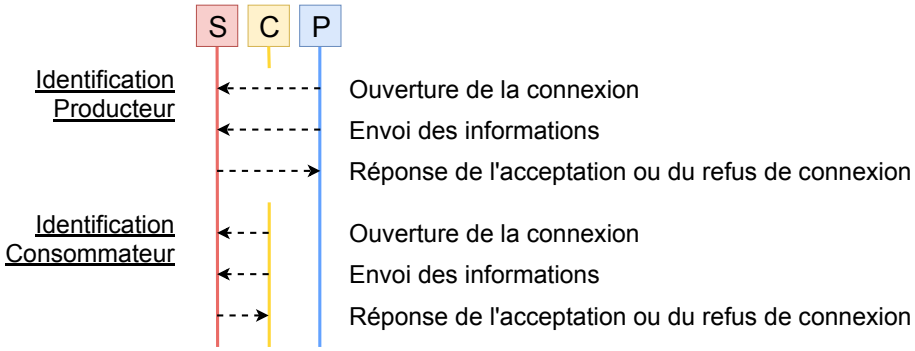


Figure 6.3 : Chronologie des messages échangés pour l'identification des producteurs et des consommateurs auprès de l'ordonnanceur, avec **S** pour *Scheduler*, **C** pour *Consumer* et **P** pour *Producer*

emplacements de sortie des données et demande de façon asynchrone à chacun s'ils sont en mesure d'allouer les ressources (mémoire, etc.) nécessaires à gestion des données de ce travail. Si tous les consommateurs concernés donnent leur feu vert, l'ordonnanceur répond à l'utilisateur que le travail est accepté et lui fournit les informations nécessaires pour récupérer les données sur les consommateurs. Dans le cas où le travail ne pourrait pas être exécuté, l'ordonnanceur indique la raison du refus à l'utilisateur et l'échange de messages reprendra avec la soumission d'un nouveau travail ;

2. Deuxièmement, lorsque l'utilisateur confirme être prêt à lire les données générées depuis les différents points de sortie logiques, il envoie à l'ordonnanceur le signal de démarrage de l'exécution du travail. L'exécution du travail se déroule par l'assignation dynamique des tâches (détaillée dans la section 7.1.6 : *Génération et assignation dynamique des tâches*) aux producteurs sélectionnés par l'ordonnanceur et la lecture par l'utilisateur des données produites et acheminées sur les différents points de sortie. Chaque information de tâche terminée est remontée auprès de l'ordonnanceur ;

3. Troisièmement, lorsque toutes les données résultantes d'un travail ont été lues par l'utilisateur, celui-ci prévient l'ordonnanceur qui peut alors informer les différents producteurs et consommateurs impliqués dans l'exécution de ce travail de libérer les ressources qui lui ont été allouées.

Le système se doit d'être résistant à l'apparition d'une erreur lors de l'exécution d'un travail, mais doit également laisser la possibilité à l'utilisateur d'annuler un travail en cours de façon à libérer au plus vite les ressources associées et permettre à l'ordonnanceur de les réattribuer à d'autres travaux. Dans le premier scénario, un producteur remonte l'erreur concernant l'exécution de la tâche à l'ordonnanceur. Si celui-ci n'est pas en mesure de la résoudre, alors il informe l'utilisateur que le travail est avorté et libère les ressources associées. Si c'est l'utilisateur qui souhaite annuler le travail, alors il en informe l'ordonnanceur qui applique une logique similaire en interrompant la production de données et en libérant les ressources des producteurs et des consommateurs.

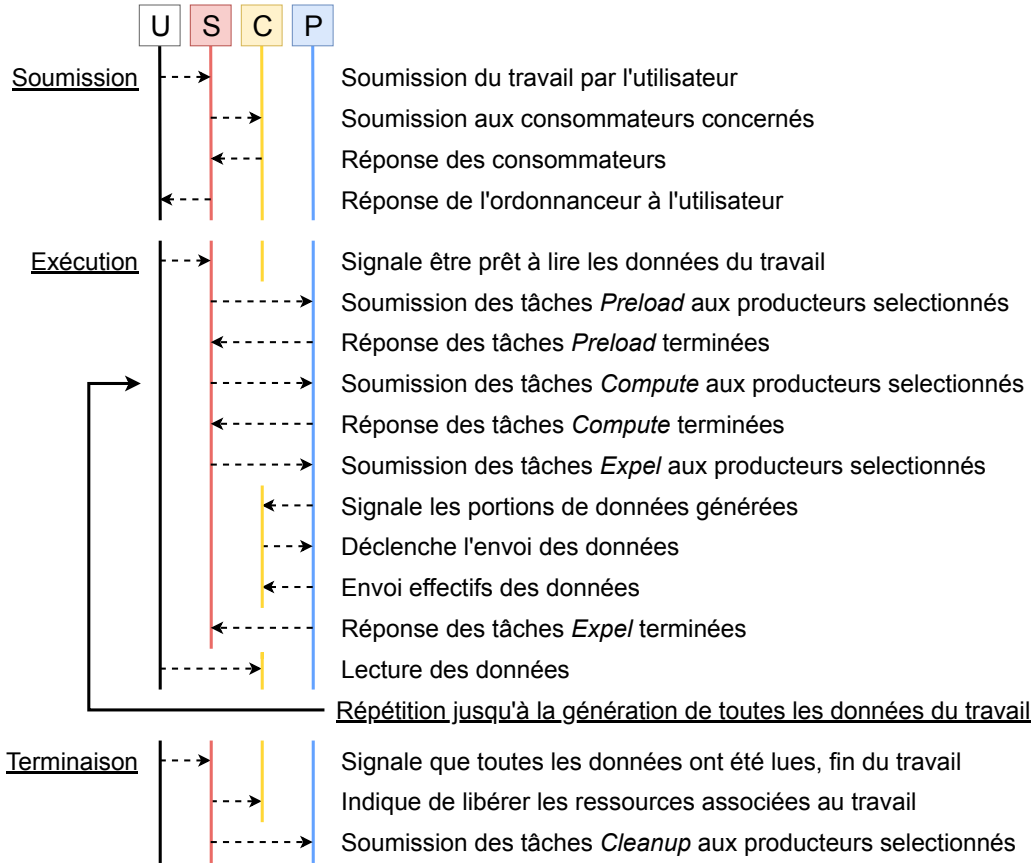


Figure 6.4 : Chronologie des messages échangés depuis la réception d'un travail jusqu'à sa terminaison en passant par son découpage en tâches successives (*Preload*, *Compute*, *Expel*, et *Cleanup*), avec **U** pour Utilisateur, **S** pour *Scheduler*, **C** pour *Consumer*, **P** pour *Producer*

6.1.8 Haut débit et faible latence

Bien que l'ajout de mémoires tampon pour stocker temporairement les données générées permet de faciliter l'augmentation du débit moyen d'une solution complexe, cette pratique peut être contre-productive. En effet, l'introduction de grandes mémoires tampons a pour effet d'augmenter la latence du système qui mène généralement à une perte de performance en condition réelle due au manque de réactivité du système induit.

Pour permettre à notre solution de concilier hauts débits en sortie et faible latence de fonctionnement, nous avons décomposé l'architecture de notre solution en deux parties complémentaires qui sont visibles sur la figure 6.5. Nous distinguons un chemin de communication à faible latence (partie gauche) qui permet de piloter efficacement un chemin haut débit (partie droite) dédié au traitement des données. Ce chemin haut débit est conçu sur le modèle des *pipelines* de données en permettant un cheminement simplifié des données et limitant l'utilisation et la taille des mémoires tampon. Ainsi notre *pipeline* commence dès l'espace de stockage des données sources et va jusqu'à la transmission à l'utilisateur des données produites.

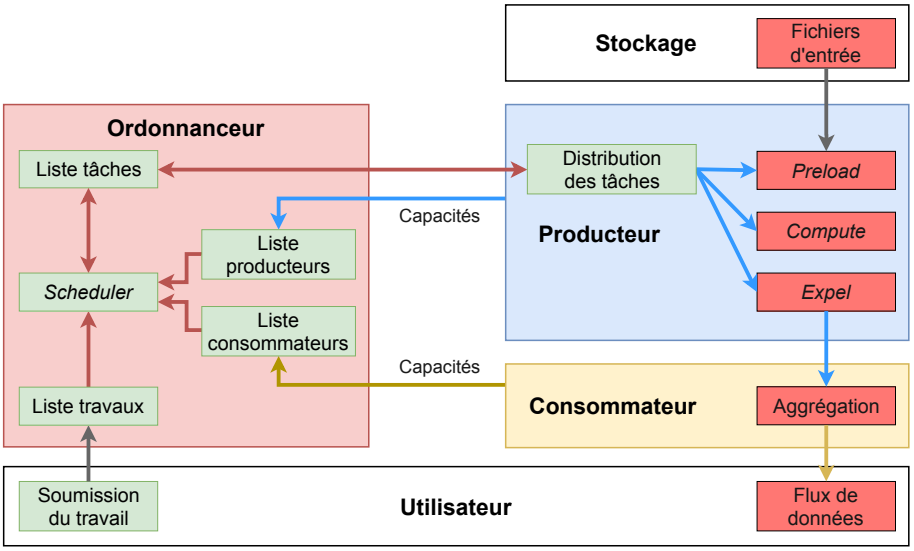


Figure 6.5 : Organisation interne de notre solution avec mise en avant du chemin à faible latence (partie gauche) pilotant celui à haut débit (partie droite)

6.2 Limites et extensions

Les risques de pannes de l'ordonnanceur centralisé peuvent être atténués en appliquant plusieurs stratégies. Nous proposons par exemple de définir automatiquement un ou plusieurs nœuds de secours qui seraient capables d'assurer le rôle d'ordonnanceur en cas défaillance de celui actuellement en exécution. Ce mécanisme de remplacement à chaud serait opérationnel en un délai minimal par une stratégie de réplication périodique ou continue des données de l'ordonnanceur sur ces nœuds de secours. À la détection d'une panne de l'ordonnanceur principal, un nœud de secours prendrait le relais en indiquant aux différents producteurs, consommateurs et utilisateurs le changement de nœud ordonnanceur.

Pour étendre les cas d'application de notre solution, nous proposons de supporter l'exécution de graphes orientés acycliques (DAG) complexes en créant une structure logique consistant à empiler et chaîner entre elles plusieurs couches de producteurs, tout en conservant une couche unique de consommateur. Ainsi, des scénarios complexes avec dépendances et échanges de données entre plusieurs producteurs seraient gérés nativement tout en conservant les différents avantages de notre solution.

Dans la mesure où notre solution souhaiterait disposer d'une capacité à dupliquer les flux de données vers plusieurs consommateurs, que ce soit lors d'une évolution des technologies de l'impression numérique ou bien pour une application à d'autres domaines, nous recommanderions de déléguer cette mission à des outils spécialisés capables d'utiliser habilement des protocoles et des stratégies réseau de type *multicast* [Ban+99].

6.3 Conclusion

Cette vue d'ensemble de notre solution permet de définir les contours d'une solution que nous jugeons généraliste pour la génération de flux de données concurrents et contraints, mais également particulièrement flexible pour s'adapter efficacement à des environnements variés. Ces objectifs sont atteints grâce à la définition claire de ses trois éléments principaux et de leur protocole de communication, ainsi que leur haut niveau d'abstraction permettant là aussi une grande flexibilité des usages. Ainsi, notre solution est conçue pour permettre une exploitation maximale des ressources d'un cluster de taille moyenne, aussi bien sur l'aspect distribué que parallèle. De plus, nous sommes également confiants dans sa capacité à pouvoir être étendu en cas d'évolutions des besoins sans pour autant remettre en cause ses principes de fonctionnement.

7 Génération distribuée des données de travaux concurrents

Nous avons présenté dans le chapitre précédent l'architecture globale de notre solution de génération distribuée de flux de données. Celle-ci se décompose en trois types de processus principaux : l'ordonnanceur, les producteurs et les consommateurs. Dans ce chapitre, nous abordons le fonctionnement interne du processus ordonnanceur et des processus producteurs, ainsi que leurs interactions.

À eux seuls, ces deux types de processus organisent et exécutent la génération des données décrites dans les travaux qui sont soumis à l'ordonnanceur par l'utilisateur. Notre solution, développée pour le domaine de l'impression numérique, fournit un cadre de fonctionnement qui est d'une part générique en étant applicable à d'autres domaines partageant des contraintes de flux similaires, et d'autre part personnalisable afin de s'adapter aux besoins et caractéristiques spécifiques de l'environnement de chaque utilisateur.

7.1 Distribution et ordonnancement des calculs

L'ordonnanceur constitue le centre névralgique de notre solution : il est en charge de coordonner le travail des différentes unités de calcul et d'orchestrer l'utilisation des ressources réseau afin d'assurer le respect des contraintes des travaux soumis pour exécution. La qualité des décisions de l'ordonnanceur est intimement liée aux performances résultantes du système. Si des décisions sous-optimales ont une incidence réduite sur un système de petite taille ayant une faible charge de travail, elles peuvent en revanche fortement dégrader les performances de tout le système dans un environnement hautement concurrentiel. D'autant plus que les décisions d'un ordonnanceur opérant dynamiquement à l'exécution du système (*online*), tel que celui présent dans notre solution, sont soumises à la résolution de problèmes complexes de gestion de ressources en évolution permanente.

7.1.1 Politiques d'ordonnancement et personnalisation

La très grande variété des stratégies d'ordonnancement présentées dans l'état de l'art en section 5.2.3 : *Ordonnancement* exprime un constat simple : il n'existe aucune stratégie d'ordonnancement idéale et universelle. En effet, en fonction des systèmes, de leur composition,

de leurs objectifs et de l'environnement dans lequel ils évoluent, chaque stratégie d'ordonnancement possède des avantages et des inconvénients. Forts de ce constat, nous avons choisi de doter notre ordonnanceur d'une solution de personnalisation permettant aux développeurs de concevoir l'ordonnancement qui correspond à leurs besoins. Ainsi, la politique de notre ordonnanceur est entièrement personnalisable par le biais de *hooks* permettant d'interagir avec chaque événement remonté à l'ordonnanceur. Ce support natif de personnalisation offre une grande flexibilité permettant à notre solution d'être adaptable à une grande variété de contextes et d'environnements.

Pour aider cette personnalisation et offrir un ordonnancement de qualité, notre solution se charge de proposer un moteur d'ordonnancement assurant les fonctionnalités de base. Ainsi, tous les événements de la vie du cluster : ajout et retrait de producteurs et de consommateurs, soumission de travaux, remontée d'erreur, etc. sont recueillis de façon asynchrone et centralisés par le moteur d'ordonnancement. Le caractère asynchrone d'occurrence et de réception de ces événements est une source de difficulté pour écrire un ordonnanceur robuste. Ainsi, afin de passer du caractère asynchrone de ces événements à leur traitement déterministe, nous empilons ces événements dans une liste *FIFO* qui permet d'ordonner les événements par date d'arrivée. Notre moteur d'ordonnanceur dépile ensuite séquentiellement cette liste, permettant ainsi d'éviter le traitement simultané d'événements concurrents. Dans le pire des scénarios, des événements contradictoires provenant de sources différentes peuvent apparaître dans la liste, mais le moteur d'ordonnancement est assuré de pouvoir les traiter de façon déterministe, ce qui facilite la cohérence du système [1]. L'exécution de chaque événement déclenche les *hooks* personnalisables correspondants. Le moteur d'ordonnancement et la politique personnalisée communiquent avec les différents éléments du système par une même bibliothèque dédiée que nous présentons dans le chapitre 9 : *Aspects techniques de la gestion des communications*. De l'extérieur, cette unification des communications offre une fusion de l'exécution du moteur et de la politique personnalisée.

La personnalisation de l'ordonnanceur passe également par la personnalisation des travaux exécutés. En ce sens, notre solution fournit le moyen de personnaliser la définition des travaux afin de leur ajouter toute information nécessaire à leur exécution. De même, les métriques utilisées par l'ordonnanceur sont pleinement personnalisables afin de supporter diverses contraintes et d'atteindre divers objectifs.

1. Il pourrait être judicieux de traiter en priorité les événements qui concernent les défaillances afin d'obtenir au plus vite la mise à jour des ressources disponibles et des tâches en défaut, cependant, en accord avec l'environnement cible de notre solution, nous considérons ces événements comme rares et donc cette optimisation se montre coûteuse au vu de la complexité ajoutée. De plus, nous préférons favoriser un traitement efficace des événements afin de minimiser la taille de la liste et ainsi accéder rapidement à tous les types d'événements, dont les défaillances.

7.1.2 Recensement et exploitation des ressources

L'ordonnanceur centralisé a pour rôle de recenser les différentes ressources du cluster afin d'organiser la répartition des travaux. De manière à prendre des décisions pertinentes, l'ordonnanceur doit connaître la liste des ressources ainsi que leurs propriétés. Pour ce faire, chaque producteur et chaque consommateur s'enregistre auprès de l'ordonnanceur pour faire partie des ressources utilisables. Cet enregistrement se déroule lors de leur connexion à l'ordonnanceur. Dès la connexion établie, ils envoient des informations. Certaines informations sont communes aux producteurs et aux consommateurs :

— un identifiant numérique dont l'ordonnanceur s'assure de l'unicité dans le système. Si l'identifiant est déjà utilisé, l'enregistrement est rejeté et la connexion est fermée. Cet identifiant est utilisé au sein des configurations des travaux pour désigner les consommateurs devant former les points logiques de sortie des données et permet de différencier les producteurs appartenant éventuellement au même nœud d'un même rack ;

— un identifiant numérique permettant de désigner le rack sur lequel il s'exécute. Un rack désigne un ensemble de nœuds disposant d'une capacité d'interconnexion totale ;

— un identifiant numérique permettant de désigner le nœud au sein du rack. Les différents éléments d'un même nœud peuvent communiquer localement entre eux sans consommer la bande passante du réseau ;

— la liste des interfaces de communication dont ils disposent avec leur bande passante théorique (indiquée par le standard de communication négocié par celles-ci et les caractéristiques matérielles), ce qui regroupe les interfaces réseau pour les communications distantes, ainsi qu'une interface locale interne au nœud.

Ces identifiants de rack et de nœud permettent à l'ordonnanceur de modéliser la topologie du cluster et de mieux visualiser les capacités d'interconnexion entre les éléments. Cette cartographie réseau lui permet de privilégier la localité des traitements, mais également de prendre en compte la bande passante réduite des liens interconnectant les nœuds appartenant à des racks différents. Actuellement, les trois identifiants numériques sont définis manuellement dans la configuration des producteurs et des consommateurs afin de permettre à l'utilisateur qui configure les ressources de personnaliser la topologie logique perçue par l'ordonnanceur, et ainsi d'influencer ses choix.

En plus de ces informations communes listées précédemment, des informations spécifiques aux producteurs et aux consommateurs sont transmises :

— les producteurs envoient une estimation de la capacité de traitement dont ils disposent. Cette information apporte à l'ordonnanceur une idée des capacités de chaque producteur et lui permet de prendre en compte l'hétérogénéité des unités de calcul. Cette capacité de traitement est exprimée en octets par seconde de données produites en sortie ;

— les consommateurs transmettent les informations nécessaires pour que les producteurs puissent leur transmettre les données produites. Ces informations contiennent les va-

leurs nécessaires pour établir des connexions locales et distantes compatibles avec la bibliothèque de communication détaillée dans le chapitre 9 : *Aspects techniques de la gestion des communications*. Elles seront transmises en temps voulu par l'ordonnanceur aux producteurs concernés.

Concernant la capacité de traitement et de communication des producteurs, celle-ci est actuellement indiquée manuellement par l'utilisateur dans la configuration de chaque producteur, mais nous proposons dans la section 7.3 : *Limites et extensions* des méthodes issues de l'état de l'art pour définir automatiquement et affiner dynamiquement sa valeur.

7.1.3 Travaux

Les travaux soumis par l'utilisateur consistent en un ensemble de propriétés contenant toutes les informations nécessaires à leur exécution. Ces informations listent : les données sources requises ainsi que leur emplacement de stockage ; les traitements à effectuer sur ces données sources ; et les contraintes sur les flux de données à générer. Dans notre contexte de chaîne de traitement graphique pour l'impression jet d'encre, les informations (introduites dans la partie I : *Environnement*) sont les suivantes :

— données sources : l'image source ainsi que les valeurs de configuration des traitements graphiques (profils ICC, LUT, etc.) ;

— traitements : dimensions, propriétés et règles de l'image de sortie à générer ;

— flux de sortie : débit nécessaire en octets par seconde, besoin de constance et d'ordre, éventuel découpage en plusieurs flux (par exemple, séparation des canaux chromatique en sortie) et emplacements de sortie des flux.

L'expression du besoin de constance implique à l'ordonnanceur de faire correspondre le débit moyen généré avec le débit demandé dans le travail. L'amplitude de variation autorisée est définie par la taille des mémoires tampon que nous détaillons dans la suite de ce chapitre.

Bien que chaque travail soit indépendant des autres travaux, ils peuvent néanmoins partager des ressources communes. Ces ressources concernent des données sources ou des points logiques de sortie des données. Ainsi, ces liens entre travaux sont explicites et il revient à l'ordonnanceur de superviser la gestion de ces ressources communes. Celui-ci a également la possibilité de dégager des optimisations de ces ressources communes en profitant par exemple de données sources déjà chargées par un nœud pour un travail précédent.

Modèle pour l'exécution des travaux

Contrairement aux systèmes distribués qui souhaitent généralement exécuter les travaux le plus rapidement possible, l'impression numérique requiert avant tout leur exécution progressive au même rythme que leur impression afin de limiter la quantité de données à stocker temporairement. Cependant, l'impression numérique présente les mêmes besoins que les systèmes

distribués usuels pour d'une part paralléliser les traitements sur différents nœuds et d'autre part mutualiser les ressources. Ainsi, nous abordons l'exécution des travaux du domaine de l'impression numérique avec l'objectif de minimiser l'empreinte des travaux dans le cluster, c'est-à-dire de réduire le nombre de nœuds et de liens réseau impliqués dans leur exécution, auxquels nous intégrons les contraintes de débit requis, de constance et d'emplacement de sortie des données.

En théorie, la distribution maximale d'un travail dépend de sa capacité à être décomposé en tâches plus petites et exécutables simultanément. Ces tâches sont alors assignables à tout autant d'unités de calcul indépendantes. Cette idée d'accélération des traitements par leur exécution parallèle est décrite dans la littérature sous différentes formes complémentaires. La loi d'Amdahl [Amd67] intègre la notion qu'un travail possède des parties séquentielles qui limitent le facteur d'accélération en fonction de leur proportion dans le travail. La loi de Gustafson [Gus88] reprend ce postulat en y intégrant que la durée d'exécution des parties séquentielles et parallèles dépend également de la quantité de données traitée, et qu'ainsi, une augmentation du nombre d'unités de calcul permet de traiter plus de données en autant de temps. La métrique de Karp-Flatt [KF90] reprend la loi d'Amdhal en ajoutant aux durées d'exécution la prise en compte des surcoûts liés aux instructions permettant la mise en œuvre de l'exécution parallèle. Selon leurs équations associées, un travail parallélisé devrait voir sa durée d'exécution évoluer linéairement avec le nombre d'unités de calcul qui lui sont allouées, de telle manière qu'en doublant le nombre d'unités de calcul la durée d'exécution de la portion parallèle serait réduite de moitié, et donc le débit doublé. Cependant, les architectures matérielles actuelles comportent des mécanismes de fonctionnement qui sont susceptibles de générer des accélérations super- ou sous-linéaires non prises en compte par ces trois modèles théoriques. En effet, les effets de cache sont connus pour fournir des accélérations super-linéaires ; alors que les tailles minimales de message sur les bus de communication brident le potentiel d'accélération.

Similairement, la loi générale des rendements décroissants soutient l'idée que l'exécution de tâches trop petites est génératrice de surcoûts venant impacter négativement leur durée d'exécution. Ces surcoûts peuvent par exemple être dus à : une sous-utilisation des capacités vectorielles des processeurs ; un débit de transfert sous-optimal ; ou encore un surcoût imputable à la synchronisation simultanée d'un trop grand nombre de tâches. Par conséquent, la taille des tâches est critique pour permettre une exécution efficace qui maximise l'utilisation des ressources du cluster. La difficulté réside dans le fait de trouver la taille optimale de ces tâches qui dépend de la configuration matérielle du cluster. Dans notre cas, cette taille optimale des tâches peut être obtenue à partir des informations renseignées par l'utilisateur sur les producteurs. Les performances issues de la taille des tâches dépendent de la précision des informations fournies par l'utilisateur. Cette approche suit les principes du *profile-based*.

Au-delà du concept de parallélisation, la distribution d'un travail sur plusieurs nœuds implique des surcoûts inhérents à l'utilisation même de ces nœuds. Dans notre cas, ces surcoûts sont : la

durée nécessaire pour charger sur les producteurs les données sources nécessaires à l'exécution des traitements ; et la durée nécessaire pour rapatrier les données produites entre les nœuds producteurs et le ou les consommateurs formant les points de sortie logiques voulus par l'utilisateur. Puisque ces durées sont propres à chaque producteur et pour chaque travail (puisqu'elles varient en fonction des emplacements des données d'entrée et de sortie), il est intéressant de chercher à les minimiser indépendamment de l'exécution des traitements. Ainsi, l'ordonnanceur doit réaliser judicieusement le placement des travaux sur le cluster afin de minimiser l'utilisation des ressources de communication du cluster.

Par extension, l'utilisation de nœuds distincts ajoute des tâches de communication dont l'exécution peut recouvrir l'exécution de tâches de calcul, ou inversement. Maximiser ce recouvrement est une stratégie élémentaire employée dans les systèmes distribués pour améliorer leur performance. L'ordonnanceur doit sélectionner les producteurs idéaux permettant d'atteindre le débit demandé par le travail sur la base de leur capacité renseignée par l'utilisateur en tenant compte du recouvrement des tâches. Dans notre architecture, l'ordonnanceur connaît les caractéristiques du cluster et comprend les travaux exécutés. Par conséquent, il est en mesure d'évaluer simplement (cf. équations **3.1** et **3.2**) pour chaque producteur si l'exécution des tâches sera recouverte par celles de calcul, ou bien par celles de communication et ainsi en déduire quelle est la capacité maximale (calcul ou communication) du producteur pour ce travail.

Enfin, nous n'incluons pas la durée nécessaire pour réaliser l'ordonnancement dans notre modèle d'exécution pour deux raisons. Premièrement, la durée de ces traitements d'ordonnancement est négligeable par rapport aux multiples traitements graphiques exécutés. Secondement, les opérations d'ordonnancement sont en quasi-totalité réalisables simultanément à l'exécution des tâches sur les producteurs : l'ordonnancement des nouvelles tâches est recouvert par l'exécution simultanée des tâches précédentes.

Modèle pour la gestion des mémoires tampons

Les flux de données que notre solution permet de produire peuvent avoir à respecter des contraintes diverses, telles que la contrainte de délivrer en sortie un débit constant qui requiert une attention particulière. En effet, un débit constant ne peut pas être résumé à la livraison d'un débit moyen, comme notre cas d'application l'illustre : un débit constant consiste à délivrer les données produites à intervalles réguliers afin d'alimenter en continu les imprimantes en données sans provoquer de famine, ni dépasser la capacité maximale de leur mémoire tampon.

Comme évoqué précédemment dans le chapitre 6 : *Approche générale pour la génération distribuée de flux de données*, nous n'employons pas de principes d'exécution temps réel pour la production des données, mais plutôt une approche orientée hautes performances avec

l'utilisation contrôlée de mémoires tampons. C'est sur ce mécanisme de mémoires tampons que nous appuyons notre capacité à livrer des flux constants à l'utilisateur. Ainsi, dans les consommateurs, nous plaçons au niveau de chaque point de sortie logique une mémoire tampon. L'objectif de l'ordonnanceur est d'assurer qu'elles contiennent toutes toujours les prochaines données à transmettre à l'utilisateur. Les détails de fonctionnement de cette mémoire tampon ainsi que ses interactions avec l'utilisateur sont présentées dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*.

Par conséquent, il est évident qu'un travail dont les données produites sont accédées avec un faible débit par l'utilisateur n'impose pas la même pression sur cette mémoire tampon qu'un travail voyant ses données produites accédées à très haut débit. Bien qu'une contrainte de flux constant respectée implique que la vitesse moyenne d'écriture dans la mémoire tampon soit supérieure ou égale au débit de lecture par l'utilisateur, la réciproque qui suggère qu'il suffit de fournir un débit moyen d'écriture supérieur au débit de lecture pour assurer une constance des données, est fausse. En effet, à lui seul, le débit moyen d'écriture ne donne aucune garantie sur l'absence de famine de données dans la mémoire tampon qui viendrait rompre la livraison constante. Ainsi, pour respecter une contrainte de constance, il est nécessaire que le débit moyen d'écriture soit supérieur ou égal au débit de lecture, mais également que les variations du débit d'écriture au cours de l'exécution du travail ne mènent pas à dépasser pas la capacité de la mémoire tampon.

De cette façon, pour assurer une contrainte de constance, il existe une relation directe entre le débit de lecture de la mémoire tampon et sa capacité. Cette relation exprimée sous la forme de l'équation 7.1 montre bien qu'agir sur la capacité de la mémoire tampon fait varier le temps disponible pour la maintenir alimentée en données. Cette équation permet également de confirmer l'idée intuitive qu'à capacité égale le débit de lecture détermine la pression temporelle pesant sur la mémoire tampon.

$$Temps\ de\ réaction\ maximal = \frac{capacité\ de\ la\ mémoire\ tampon}{débit\ lecture} \tag{7.1}$$

Maintenir la présence de données dans cette mémoire tampon est donc l'objectif de l'ordonnanceur, qu'il réalise à travers l'organisation de la production de données sur les producteurs suivie de leur transfert dans cette mémoire tampon située dans les consommateurs. L'équation 7.1 définit donc indirectement le temps maximal disponible pour générer et transférer les données depuis les différents producteurs. Son extension en équation 7.2 transcrit l'aspect dynamique de l'exécution et par conséquent le temps maximal disponible à chaque instant pour respecter la contrainte de constance.

$$Temps\ de\ réaction\ courant = \frac{quantité\ de\ données\ dans\ la\ mémoire\ tampon}{débit\ lecture} \tag{7.2}$$

Il est très attrayant de vouloir augmenter la capacité de la mémoire tampon de façon à augmenter le temps disponible pour produire les prochaines données, d'autant plus si nous considérons la quantité moyenne de données contenues dans la mémoire tampon comme étant égale à la moitié de sa capacité. Néanmoins, cette stratégie se heurte rapidement aux contraintes physiques des nœuds. En effet, en prenant pour exemple les débits de lecture visés par notre cas d'application, vouloir un temps de réaction d'une seconde en moyenne pour un débit de lecture de 1 Go/s impose de dédier deux gigaoctets à cette mémoire tampon. En plus d'être particulièrement gourmande en mémoire, cette approche borne le temps maximal de réaction à la capacité de mémoire physique dont dispose le nœud accueillant le consommateur en fonction du débit de lecture des données par le pilote d'impression.

Pour pallier aux défauts de l'augmentation de la capacité de la mémoire tampon, nous abordons la problématique du temps de réaction en fonction du débit de lecture et de la capacité de la mémoire tampon dans l'environnement dans lequel elle prend place. Conformément à notre stratégie de découpage des travaux en tâches, les producteurs génèrent des données par blocs. Une fois générés, ces blocs de données ont pour vocation d'être transférés dans la mémoire tampon du consommateur. Or, avant d'être transférés, ces blocs de données forment des mémoires tampons intermédiaires. La différence majeure est leur distribution sur les producteurs qui permet de répartir le coût mémoire sur plusieurs nœuds, mais également de pouvoir passer à l'échelle en permettant des débits plus importants. En poussant l'application de cette stratégie plus loin, nous arrivons sur la notion de *N-buffering* qui consiste à employer N mémoires tampon intermédiaires composées de données générées. Ainsi, nous étendons l'équation 7.2 en l'équation 7.3 qui inclut l'environnement d'exécution et en tire avantage. À partir de cette dernière équation, il est possible de distribuer la quantité de mémoire tampon voulue dans les différents producteurs en tenant compte de leur éventuelle hétérogénéité.

Soient :

— N le nombre de producteurs alloués à l'exécution du travail ;

— M_c quantité de données dans la mémoire tampon du consommateur ;

— $M_p(n)$ quantité de données dans le *N-buffering* du producteur n ;

alors :

$$Temps\ de\ réaction\ courant = \frac{M_c + \sum_{n=1}^{N} M_p(n)}{débit\ lecture} \tag{7.3}$$

En finalité, la quantité minimale de mémoire tampon à conserver dans le consommateur en est réduite au strict nécessaire permettant la continuité des transferts entre les mémoires intermédiaires des producteurs et celle-ci en fonction de la qualité des interconnexions réseau et non plus de la capacité de réaction de l'ordonnanceur et des producteurs. Cette quantité est alors dépendante de la taille des blocs de données générées et du nombre que l'on veut pouvoir recevoir en parallèle.

7.1.4 Tâches

Durant son exécution, la principale activité de l'ordonnanceur est la décomposition progressive des travaux en tâches. Ces tâches constituent l'ensemble des étapes nécessaires pour compléter le travail de façon distribuée et leur contenu est l'ensemble de toutes les informations requises pour permettre aux producteurs qui les recevront de les exécuter sans délai.

De façon analogue à la personnalisation de la politique d'ordonnancement et de la personnalisation des travaux, notre solution permet la personnalisation des tâches. De cette manière, notre système est en mesure de supporter la décomposition et l'exécution de travaux variés issus de cas d'applications diversifiés. Toutefois, nous définissons une base composée de quatre tâches qui permettent de distribuer sainement l'exécution de nombreux travaux de génération de flux. Ces quatre tâches sont :

— *Preload* : fournit la liste des fichiers d'entrée nécessaires pour participer à la production des données d'un travail. À sa réception le producteur démarre leur collecte. En conséquence de la grande variété possible des formats des fichiers d'entrée et de leurs options d'encodage, telle que la compression, nous considérons que les producteurs requièrent de récupérer le contenu intégral des fichiers avant de pouvoir s'en servir. La tâche liste les fichiers ainsi que la source auprès de laquelle il doit les récupérer ;

— *Compute* : demande à un producteur de produire les données d'une portion d'un travail. Le travail et la portion associée sont pleinement décrits dans la tâche ;

— *Expel* : demande à un producteur de transmettre une portion de données produites à leurs consommateurs. La tâche précise la portion concernée ainsi que les informations sur les ressources réseau (interface, adresse, port, etc.) à utiliser pour chaque consommateur ;

— *Cleanup* : indique à un producteur qu'il peut libérer les ressources associées à un travail. Cette tâche est envoyée à tous les producteurs ayant reçu des tâches de *Preload* une fois que le travail a été entièrement complété.

Il est intéressant de noter que les tâches *Preload* et *Expel* font toutes les deux un usage des liens réseau. Cependant, les liens réseau étant communément *full duplex*, leur exécution concurrente sur un même nœud ne provoque pas de conflit d'accès à l'interface réseau puisque l'une effectue des lectures pendant que l'autre fait des écritures. Néanmoins, les conflits de partage de la bande passante du réseau restent possibles entre différents nœuds, peu importe le type de tâche *Preload* ou *Expel*.

La figure 7.4 présente sous la forme d'un graphe orienté l'ordre dans lequel l'ordonnanceur peut générer ces quatre types de tâches afin de compléter l'exécution d'un travail. Ce graphe s'applique pour chaque producteur impliqué dans le travail. Il débute par une tâche *Preload* qui découle sur :

— `(a)` une ou plusieurs tâches *Compute* pour démarrer la production de données en constituant éventuellement un *pipeline* (dont l'intérêt est expliqué dans la section 7.1.6),

— ou `(g)` une tâche de *Cleanup* si ce producteur n'a eu aucune donnée à produire.

Chaque tâche *Compute* découle sur :

— `(b)` aucune ou plusieurs tâches *Compute* pour continuer la production de données sur ce producteur,

— et `(c)` aucune ou plusieurs tâches *Expel* pour transmettre les données générées à chaque consommateur.

Chaque tâche *Expel* découle sur :

— `(d)` aucune ou plusieurs tâches *Compute* pour reprendre la production de données sur ce producteur,

— et `(e)` aucune ou plusieurs tâches *Expel* si des données ont été générées précédemment sans créer de tâches *Expel*,

— ou `(f)` une tâche de *Cleanup* si ce producteur n'a plus de données à produire ni de données à transmettre.

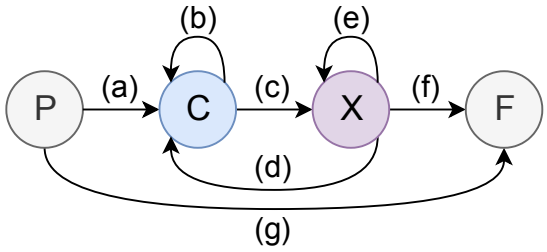


Figure 7.4 : Graphe orienté présentant l'enchaînement possible des tâches sur un producteur pour un travail donné, avec **P** pour *Preload*, **C** pour *Compute*, **X** pour *Expel* et **F** pour *Cleanup*

L'ordonnanceur suit ce graphe orienté pour chaque travail afin de générer progressivement et dynamiquement (*online*) un graphe orienté acyclique qui couvre l'ensemble des données à générer. Cette génération progressive des tâches est présentée en détail dans la section 7.1.6 : *Génération et assignation dynamique des tâches*.

7.1.5 Assignation dynamique des travaux

Tel qu'ils ont été présentés dans le chapitre I : *Environnement*, les travaux d'impression peuvent être de volumétrie et de débit très variables. Ainsi, la contrainte de génération progressive des flux (puisque ceux-ci ne peuvent pas être envoyés en entier aux imprimantes numériques) impose de fournir une vitesse de génération spécifique à chaque travail. Ces vitesses de générations différentes imposent dans certains cas d'utiliser plusieurs producteurs pour un travail, et, au contraire, permettent de placer plusieurs travaux sur un même producteur dans d'autres cas. Cependant, chaque producteur qui participe à la génération des données d'un travail doit préalablement disposer des fichiers d'entrée de ce travail. Or, la récupération de ces fichiers d'entrée est coûteuse en bande passante réseau. Ainsi, il est évident de restreindre la récupération des fichiers d'entrée aux seuls producteurs qui seront concernés par le traitement de ce travail.

Pour ce faire, l'ordonnanceur de notre solution applique une stratégie de réservation des ressources [LMD94 ; CT90 ; LNR09 ; Pol+11] en assignant chaque travail à un nombre restreint de producteurs en fonction de ses caractéristiques. Cette assignation est dynamique pour suivre les évolutions du cluster et de l'environnement (nouveaux travaux, ajout de nœuds, pannes, etc.). Le nombre de producteurs nécessaires est déterminé en fonction des capacités de calcul de chaque producteur qui est transmise à la connexion de ceux-ci à l'ordonnanceur. Pour chaque nouveau travail, l'ordonnanceur va assigner autant de producteurs que nécessaire pour fournir le débit demandé par le travail.

Afin d'optimiser les performances du système, l'ordonnanceur adopte une approche *locality-aware* en cherchant à réduire la distance entre les producteurs et les consommateurs d'un travail. La réduction de cette distance se traduit par une diminution de l'utilisation des liens réseau. En effet, l'ordonnanceur va favoriser l'assignation des producteurs sur le même rack que les consommateurs, et idéalement sur la même machine, ce qui va permettre de limiter la quantité de données circulant entre et dans les racks. Pour ce faire, l'ordonnanceur se base sur la cartographie réseau qu'il connaît grâce aux identifiants de rack et de nœud communiqués par chaque producteur et consommateur à leur connexion.

À l'arrivée d'un nouveau travail, l'ordonnanceur va successivement essayer de l'assigner aux producteurs du même nœud que les consommateurs, puis à ceux du même rack, puis aux autres. Dès que l'ordonnanceur rencontre un producteur ayant une capacité de calcul non saturée, il lui assigne proportionnellement une partie du travail, jusqu'à satisfaire la capacité de calcul requise par le travail. Si la capacité de calcul demandée par le travail ne peut être satisfaite par les producteurs disponibles, alors notre ordonnanceur refusera le travail. Les possibilités de personnalisation de notre ordonnanceur lui permettent de supporter d'autres réactions, telles que par exemple un score de priorité des travaux afin de stopper ou d'annuler les travaux d'importance moindre.

Cependant, chaque travail dispose de consommateurs potentiellement différents, ce qui induit une classification différente de la distance des producteurs pour chaque travail. En effet, un producteur qui présente une localité rack (c'est-à-dire, situés dans le même rack) avec un consommateur pour un travail, peut également présenter une localité hôte (c'est-à-dire, situés sur le même nœud) avec le consommateur d'un autre travail. Dans ces scénarios, notre ordonnanceur va alors favoriser la localité hôte sur la localité rack afin d'améliorer la localité globale de l'ensemble du système.

Pour construire une localité globale qui tend vers l'idéal, notre ordonnanceur assigne séquentiellement les différents travaux (au fur et à mesure de leur arrivée dans le système). Si lors de son parcours d'assignation, il rencontre un producteur assigné à un précèdent travail avec une localité moins bonne que la localité que permettrait d'avoir ce producteur sur ce nouveau travail, alors il réassignera le précédent travail sur un autre producteur qui lui présente une localité équivalente. Cette approche est illustrée par l'algorithme 7.5. Symétriquement, lors-

qu'un travail est terminé, celui-ci libère de la capacité de calcul sur les producteurs. Lors de cet événement, l'ordonnanceur va parcourir les assignations ayant les plus mauvaises localités et les réassigner sur les producteurs nouvellement libres. Si un déplacement a lieu, alors un traitement récursif est appliqué pour redescendre successivement les travaux sur les producteurs disponibles proposant la meilleure localité.

Enfin, afin de fournir une continuité de service (c'est-à-dire une continuité de production des données et ainsi ne pas créer de famine) lors des déplacements des travaux d'un producteur sur un ou plusieurs autres producteurs (malgré le temps d'initialisation obligatoire avant de pouvoir produire les données), notre ordonnanceur utilise un double compteur de capacité : *réservée* et *allouée* qui débitent tous deux la capacité totale des producteurs. Les travaux sont placés en tant que réservés durant la récupération des fichiers d'entrée, et à la fin de celle-ci, ils sont déplacés en tant qu'alloués. Ainsi, lorsqu'un travail quitte un producteur, il n'est pas sorti de son allocation tant que la capacité correspondante n'est pas récupérée par le ou les nouveaux producteurs. De cette manière notre ordonnanceur continue de faire produire des données sur le premier producteur tout en évitant une double réservation de la même capacité sur le ou les nouveaux producteurs par un autre travail.

```
var job      // Le travail à ordonnancer
var cluster  // Les informations du cluster

/* Vérifie la capacité actuelle du cluster à exécuter le travail */
if capacity_available(cluster) < capacity_required(job)
    /* Applique la politique choisie :
     *   rejet, réessayé plus tard, etc. */
    job_retry_later(job)
else
    /* Tri des producteurs par localité par rapport au(x) point(s) de
     * sortie de ce travail (même nœud, même rack, cluster). Au sein de
     * chaque localité, tri par capacité libre décroissante afin de
     * diminuer la fragmentation du travail */
    producer_list = producer_sort(cluster)

    var producer = first(producer_list)

    /* Réserve une capacité sur autant de producteurs que nécessaire
     * en déplaçant, si besoin, d'autres travaux à une localité qui leur
     * est équivalente */
    while ! job_fully_reserved(job)
        capacity_reserve(producer, job)

        if ! job_try_to_move_to_same_locality(producer)
            producer = next(producer_list)
```

Algorithme 7.5 : Pseudo-code d'assignation d'un nouveau travail sur les producteurs du cluster

7.1.6 Génération et assignation dynamique des tâches

Cette section présente les mécanismes de génération et d'assignation des tâches aux producteurs. La transmission de ces tâches de l'ordonnanceur jusqu'aux producteurs, ainsi que les messages de retours, fait usage des protocoles de signalisation présentés dans le chapitre 6.1.7 : *Organisation et protocoles de communication* et sont spécifiquement illustrés par la figure 6.4. Pour chaque producteur nouvellement assigné à un travail, une tâche de *Preload* lui est envoyée. La complétion de cette tâche permet alors à l'ordonnanceur d'utiliser ce producteur pour participer à la production des données du travail. Notre solution propose également une génération dynamique (*online*) des tâches pour les différents producteurs afin d'assurer la flexibilité du système.

Alors que l'assignation dynamique des travaux suit une approche par réservation de ressources, celle de l'assignation dynamique des tâches est organisée autour de l'approche *événementielle* [Kop91 ; Tab07]. Par conséquent, c'est la complétion d'une tâche qui provoque la création éventuelle d'une ou de plusieurs nouvelles tâches.

Cette approche permet de décomposer progressivement le graphe orienté de la figure 7.4 en un graphe orienté acyclique réalisant l'exécution complète du travail au travers des différents producteurs. La création et l'assignation dynamique des tâches pour chaque producteur permettent de répondre au besoin de génération progressive des données du travail tout en assurant une répartition de charge automatique. En effet, pour deux travaux d'impression de même volumétrie, mais à destination d'imprimantes ayant des vitesses différentes, il est inutile de générer leurs données à la même vitesse puisque : premièrement, chaque imprimante consommera les données uniquement lorsqu'elle en aura besoin ; secondement, au vu des volumétries possibles des travaux (cf. tableau 3.3), il serait contraignant de devoir stocker temporairement une telle quantité de données. Par conséquent, il est essentiel d'ordonnancer les travaux lents sur le long terme et les travaux rapides au rythme nécessaire pour ne pas créer de famine. Notre architecture permet de lier le délai d'exécution d'une tâche *Expel* à la consommation des données par l'utilisateur grâce au mécanisme de mémoire tampon détaillé dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*. Ainsi, la génération dynamique des tâches *Compute* à partir des événements de complétion des tâches *Expel* permet de synchroniser la production des données avec leur consommation par l'utilisateur.

Les figures 7.6 (a), (b), (c), (d) et (e) illustrent cinq graphes acycliques orientés générés à partir du graphe orienté de la figure 7.4. Chacun de ces cinq graphes présente l'exécution de travaux aux profils différents :

— la figure 7.6 (a) présente une génération séquentielle des tâches : les tâches sont générées une à une lorsque la précédente est terminée, en alternant successivement une tâche *Compute* avec une tâche *Expel*. Ce graphe est adapté pour l'exécution d'un travail

nécessitant une faible bande passante qui peut être amplement satisfaite par l'exécution et le transfert de données sans anticipation ;

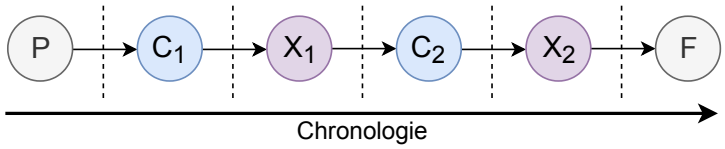
— la figure 7.6 (b) illustre également une génération séquentielle, mais dans une topologie *one-to-many* qui implique la création de plusieurs tâches *Expel* pour diffuser les données produites sur les différents consommateurs ;

— la figure 7.6 (c) illustre une situation où la capacité de calcul est inférieure à la bande passante d'échange avec le consommateur. La complétion d'une tâche *Compute* provoque la génération de deux tâches : la tâche *Expel* correspondante et une nouvelle tâche *Compute*. De ce fait les deux nouvelles tâches seront exécutées parallèlement afin d'offrir un recouvrement partiel des calculs par les communications. Cette stratégie permet de créer un *pipeline* de tâches *Compute* et d'optimiser le taux d'utilisation des capacités de calcul ;

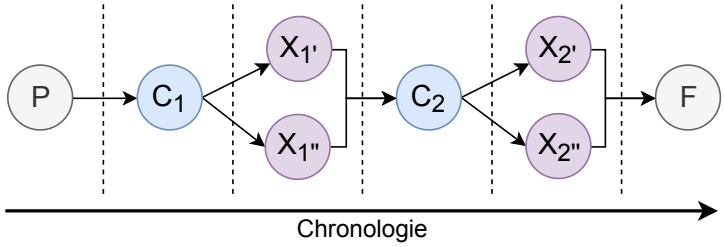
— la figure 7.6 (d) propose une amélioration du graphe de la figure 7.6 (b) en assignant au départ deux tâches *Compute*. Conformément à la section 7.2, leur exécution par le producteur sera séquentielle. L'intérêt est d'améliorer encore davantage le taux d'utilisation des capacités de calcul du producteur en recouvrant le temps nécessaire à la création et l'assignation d'une nouvelle tâche par l'exécution d'une tâche de même type ;

— la figure 7.6 (e) illustre le scénario opposé de la figure 7.6 (c), c'est-à-dire présentant une bande passante de communication inférieure à la capacité de calcul. Dans ce scénario, outre la complétion de la première tâche *Compute* qui déclenche une tâche *Compute* et *Expel*, ce sont ensuite les tâches *Expel* qui déclenchent la création des nouvelles tâches *Compute*. Ainsi, les communications sont partiellement recouvertes par les calculs et un *pipeline* de tâches *Expel* est formé. Il est tout à fait possible d'appliquer la logique de la figure 7.6 (d) afin d'améliorer le taux d'utilisation du canal de communication. Cependant, la formation de ce graphe ne signifie pas que le producteur est totalement ralenti par les tâches *Expel*. En effet, la vitesse d'écoulement des tâches *Expel* est directement liée à la vitesse de consommation des données par l'utilisateur (cf. chapitre 8 : *Agrégation et transmission des données à l'utilisateur*). De plus, d'autres canaux de communication dédiés à d'autres travaux peuvent être disponibles pour la production de données à destination d'un autre travail, notamment à destination d'un autre consommateur local.

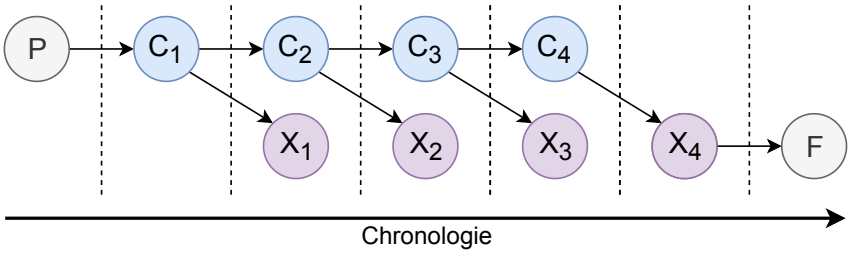
Le fonctionnement dynamique basé sur l'approche événementielle permet à notre ordonnanceur de mixer indifféremment les différents scénarios présentés ci-dessus entre les producteurs d'un travail, mais également au sein des producteurs. En effet, un producteur assigné à deux travaux différents peut présenter des localités différentes pour chacun, et donc présenter des scénarios de production et de transfert des données différents. De même, un travail assigné à plusieurs producteurs, ou bien utilisant une topologie avec plusieurs consommateurs est constitué d'un mélange de localités. La création des tâches par événement va également permettre de répartir automatiquement les charges de travail entre les producteurs les plus rapides et les plus lents : plus un producteur est rapide, plus il se verra assigner des tâches. Cette approche offre ainsi un support natif et adaptatif de l'hétérogénéité des producteurs et de leurs interconnexions. De plus, cette approche permet une intégration fine des besoins d'anticipation des famines par



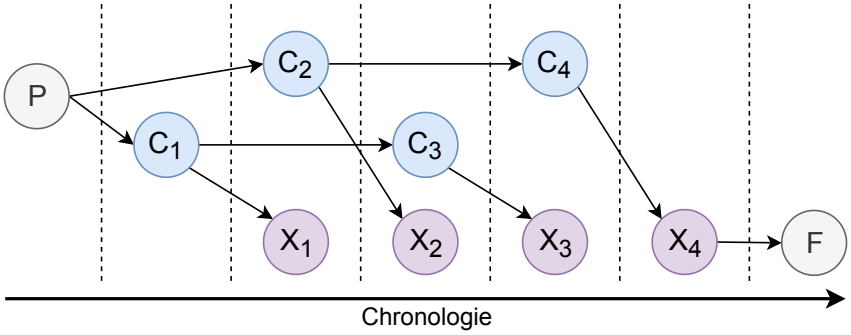
(a) : Génération séquentielle des tâches



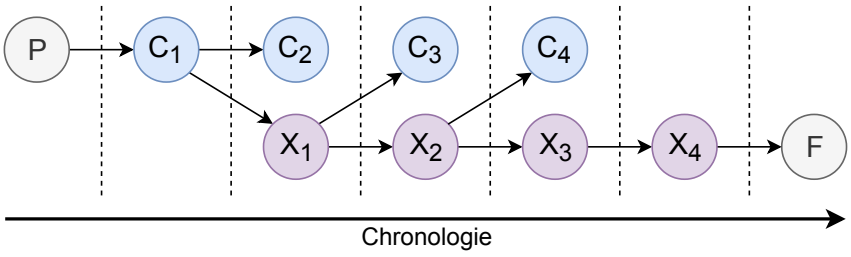
(b) : Génération séquentielle des tâches dans une topologie *one-to-many*



(c) : Génération des tâches sur un producteur limité par la capacité de calcul avec *pipelining* simple



(d) : Génération des tâches sur un producteur limité par la capacité de calcul avec *pipelining* avancé



(e) : Génération des tâches sur un producteur limité par la communication

Figure 7.6 : Graphes orientés acycliques présentant des scénarios d'enchaînements possibles des tâches sur un producteur pour un travail donné, avec **P** pour *Preload*, **C** pour *Compute*, **X** pour *Expel* et **F** pour *Cleanup*

l'utilisation des différents producteurs, telle que décrite par l'équation 7.3. Ainsi, pour créer des réserves de données, il suffit d'augmenter la profondeur du *pipeline* de tâches *Compute* afin d'augmenter la quantité de données déjà calculées. Cette profondeur peut être choisie indépendamment pour chaque travail, et pour chaque producteur.

La personnalisation de notre ordonnanceur lui permet d'optimiser la gestion et les réactions aux différents scénarios. Par exemple, dans l'implémentation utilisée pour les évaluations du chapitre 11 : *Évaluations du RIP distribué*, et illustrée par l'algorithme 7.7, une distinction est réalisée selon la localité du producteur par rapport au consommateur : si celui-ci est local, alors jusqu'à deux tâches *Compute* d'avance vont être générées contre une seule sinon. Cette distinction permet d'améliorer la profondeur du *pipeline* sur les producteurs locaux qui bénéficient d'une excellente connectivité (échange direct à travers la mémoire vive) avec le consommateur, sans pour autant anticiper négativement l'assignation de portions du travail qui seraient retardées par une connectivité moindre.

```
var task_finish  // La tâche qui vient d'être terminée
var job          // Le travail auquel appartient la tâche terminée
var producer     // Le producteur auquel appartient la tâche terminée

/* La profondeur des pipelines est définie pour chaque producteur en fonction de
 * chaque travail (de la localité entre celui-ci et du ou des consommateurs) */
var PIPELINE_COMPUTE  // Profondeur du pipeline Compute (2 en local, 1 sinon)
var PIPELINE_EXPEL    // Profondeur du pipeline Expel   (4 en local, 1 sinon)

/* Selon le type de la tâche */
switch task_type(task_finished)

    case PRELOAD :
        /* Convertit la capacité réservée en capacité allouée */
        producer_job_loaded(producer, job)

        /* Assigne une première tâche Compute au producteur
         * Création d'un pipeline simple */
        job_new_task_compute(job, producer)

    case COMPUTE :
        /* Si la profondeur du pipeline Expel n'est pas atteinte */
        if producer_task_expel_running(producer, job) < PIPELINE_EXPEL
            /* Alors assigne une nouvelle tâche Expel au producteur
             * à partir des données générées */
            job_new_task_expel(job, producer, task_finished)
        else
            /* Sinon ajoute la tâche aux tâches en attente */
            producer_task_compute_add_waiting(producer, task_finished)

        /* Si la profondeur du pipeline Compute n'est pas atteinte */
        if producer_task_compute_waiting(producer, job) < PIPELINE_COMPUTE
            /* Alors assigne une nouvelle tâche Compute au producteur */
            job_new_task_compute(job, producer)

    case EXPEL :
        /* Si des tâches Compute sont en attente */
        if producer_task_compute_waiting(producer, job) > 0
            /* Alors assigne une nouvelle tâche Expel au producteur
             * à partir d'une tâche Compute terminée et la supprime
             * des tâches en attente */
            job_new_task_expel(job, producer)
            producer_task_compute_remove_waiting(producer, job)

            /* Le pipeline Compute est libéré d'une place
             * Assignation une nouvelle tâche Compute au producteur */
            job_new_task_compute(job, producer)

    case CLEANUP :
        /* Libère la capacité allouée */
        producer_job_unloaded(producer, job)
```

Algorithme 7.7 : Pseudo-code d'assignation dynamique des tâches aux producteurs assignés à un travail

7.1.7 Tolérance aux pannes

La nature même des systèmes distribués les rend particulièrement sujets aux pannes. La fréquence de ces pannes est corrélée à la qualité du matériel et du logiciel utilisé, à son environnement, ainsi qu'au nombre de nœuds composant le système. Si les risques de panne dans les systèmes distribués que nous visons sont relativement faibles en raison du faible nombre de nœuds, il est néanmoins nécessaire de disposer d'un mécanisme de gestion de ces pannes afin d'assurer une continuité de service substantielle. En effet, au-delà de la perte du temps de calcul utilisé, l'arrêt d'un travail d'impression en cours provoque également le gaspillage du support imprimé ainsi que des encres utilisées puisqu'il n'est actuellement pas possible pour les imprimantes d'atteindre une bonne qualité d'impression sur une impression réalisée en plusieurs fois (difficulté de positionnement, différence de temps d'exposition des encres au séchage, etc.).

Le principe de fonctionnement de notre ordonnanceur par assignation dynamique des travaux lui permet de fournir nativement une tolérance efficace aux pannes des producteurs. En effet, la déconnexion d'un producteur provoque la réassignation automatique des travaux aux autres producteurs en reprenant la politique de localité présentée précédemment. Simultanément à cette réassignation, l'ordonnanceur utilise sa connaissance des tâches en cours sur les différents nœuds pour réassigner les tâches en défaut sur l'ensemble des producteurs assignés au travail. Dans le cas où la défaillance concernerait une tâche *Expel*, l'ordonnanceur va recréer la tâche *Compute* correspondant aux données transmises par la tâche *Expel*. La fin de cette nouvelle tâche *Compute* provoquera d'elle-même la tâche *Expel* de remplacement.

De plus, grâce aux possibilités de personnalisation de notre ordonnanceur, les actions menées en réaction aux pannes peuvent être adaptées de nombreuses manières selon les préférences des utilisateurs. Il est par exemple possible d'assigner un ou plusieurs producteurs de secours aux travaux, ces producteurs auront alors la possibilité de récupérer de façon préventive les fichiers d'entrée par l'exécution d'une tâche *Preload* afin d'assurer le relai immédiat d'un producteur défaillant.

7.2 Production des données

L'architecture de notre solution assure un passage à l'échelle efficace en augmentant le nombre de producteurs pour augmenter les performances du système. Néanmoins, l'architecture interne des producteurs doit également être conçue avec attention afin de maximiser l'exploitation de leurs unités de calcul. Une exploitation optimale des unités de calcul permet de minimiser le nombre de nœuds nécessaires, et donc le coût global du système.

7.2.1 Exécution parallèle des tâches

L'objectif de chaque producteur est simple : exécuter en un minimum de temps les tâches que l'ordonnanceur lui a assignées. Notre implémentation spécifique à une chaîne de traitement graphique contient quatre tâches différentes mettant chacune en œuvre des mécanismes particuliers. En effet, les tâches *Preload* et *Expel* réalisent une utilisation inverse du réseau ; *Compute* repose sur une exploitation efficace des unités de calcul ; et *Cleanup* se comporte comme une routine interne.

Afin d'obtenir des performances idéales, il est essentiel de permettre l'exécution parallèle de certaines tâches. D'un côté, certaines tâches assignées à un producteur peuvent appartenir à des travaux différents et leur exécution séquentielle engendrerait des dépendances entre les travaux. C'est le cas des tâches *Expel* du fait de leur vitesse d'exécution qui est indirectement liée à celle de lecture des données par le pilote d'impression au travers du mécanisme de mémoire tampon détaillé dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*. De l'autre côté, l'exécution séquentielle de certaines tâches permet d'optimiser leurs temps d'exécution, par exemple les tâches *Compute* qui doivent bénéficier d'un usage maximal des mémoires cache du processeur. Pour permettre le contrôle de la parallélisation des tâches, l'architecture des producteurs repose sur l'utilisation de moteurs d'exécution des tâches. Sur chaque producteur, un moteur d'exécution est dédié à chaque type de tâche. Bien que générique, chaque moteur est configuré selon le type de tâche auquel il est associé. Cette configuration précise les traitements à effectuer pour compléter chaque type de tâche, et le nombre de tâches qu'il est autorisé à exécuter en parallèle.

Ainsi, cette architecture assure un double niveau de parallélisation. D'une part entre les différents types de tâches :

— exécution parallèle de tâches de types différents afin de permettre le recouvrement des calculs par les communications, ainsi que les communications unidirectionnelles entre elles ;

— exécution parallèle de plusieurs tâches du type *Preload*, *Expel* et *Cleanup*

— exécution séquentielle des tâches de type *Compute* (les unes par rapport aux autres).

7.2.2 Traitements graphiques

L'implémentation des traitements des tâches *Compute* consiste à réaliser l'ensemble des traitements graphiques permettant de produire des données compréhensibles par le pilote d'impression à partir des fichiers sources récupérés préalablement par la tâche *Preload*. L'ordonnanceur indique dans la tâche *Compute* quelle portion des données de sortie le producteur doit générer. Ainsi, à l'exécution d'une tâche *Compute*, le producteur produit une bande horizontale de l'image de sortie. Chaque tâche *Compute* applique sur l'image d'entrée l'ensemble des traite-

ments (mise à l'échelle, colorimétrie, tramage, etc.) présenté dans le chapitre I : *Environnement*. L'exécution de ces traitements est optimisée en appliquant diverses techniques, dont le découpage par tuile afin de tirer parti des localités spatiales et temporelles des données, et la parallélisation sur les différents cœurs du processeur par l'utilisation de primitives *OpenMP*.

7.2.3 Transfert des données produites au consommateur

Consécutives aux tâches *Compute*, les tâches *Expel* participent activement aux performances du producteur. Les tâches *Expel* sont initiées par l'ordonnanceur de façon à ce que leur réception par le producteur signifie le début de leur exécution. Premièrement, le moteur d'exécution des tâches *Expel* s'occupe d'établir la connexion avec le consommateur, de transférer les données et d'informer l'ordonnanceur de leur bonne réception effective. Pour organiser cet échange de données entre le producteur et le consommateur, un protocole minimal constitué de 4 étapes est utilisé. Premièrement, le producteur transmet un en-tête indiquant au consommateur quelle portion du travail concerne les données qu'il souhaite lui envoyer. Deuxièmement, lorsque le consommateur est en mesure de recevoir ces données, il donne le signal au producteur de débuter leur envoi en accusant réception de l'en-tête. Cette étape de *poignée de main* permet au consommateur de temporiser l'envoi des données si celui-ci doit en priorité recevoir des données antérieures au travail. L'attente explicite résultante permet au producteur de ne pas mettre sous pression son interface de sortie ni l'interface d'entrée du consommateur, ce qui privilégie l'exécution de flux prioritaires en limitant les congestions [Luo+12]. Troisièmement, à la réception du signal du consommateur, le producteur démarre l'envoi effectif des données. Quatrièmement, lorsque toutes les données ont été reçues par le consommateur, celui-ci envoie un acquittement qui permettra au producteur d'indiquer que la tâche *Expel* est terminée avec l'assurance qu'aucune donnée n'est encore « en vol » entre les mémoires tampons des deux cartes réseau et ne risquerait d'être perdue dans une panne de celui-ci.

Contrairement à l'exécution des tâches *Compute*, l'exécution des tâches *Expel* est conçue pour être au maximum parallèle afin d'assurer une utilisation maximale des liens réseau. L'écoulement de chaque communication, et donc son utilisation des ressources réseau, est dicté par l'utilisateur au travers des mécanismes de mise à disposition des données décrits dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*. La maximisation de l'utilisation des liens, ainsi les détails et optimisations de la couche communication sont présentés dans le chapitre 9 : *Aspects techniques de la gestion des communications*.

7.3 Limites et extensions

Le vaste domaine de l'ordonnancement et de la production de flux de données donne la possibilité à notre solution d'être étendue de multiples manières. Ces évolutions constituent des améliorations secondaires qui n'ont pas été explorées dans nos travaux. Néanmoins, elles

permettraient de consolider les performances de notre solution dans des scénarios et cas d'application plus complexes.

Nous présentons et discutons ici quelques-unes des évolutions susceptibles de fournir les améliorations de performance et d'utilisation les plus notables :

— L'implémentation actuelle des routines des prises de décision de notre ordonnanceur est *monothread* afin de supprimer toute concurrence d'accès aux structures de données. Cependant, certaines portions de ces routines pourraient aisément bénéficier d'une exécution parallèle afin de rendre l'ordonnancement encore plus réactif. Pour aller plus loin, certains algorithmes pourraient être modifiés afin de minimiser leur complexité et ainsi garantir des temps d'exécution constants [DL89 ; THW02]. Cependant, nous n'avons pas exploré cette optimisation puisque l'ordonnancement n'est pas le goulot d'étranglement dans notre cas d'étude ;

— La découverte automatique des nœuds du cluster, de leurs ressources (producteurs et consommateurs) et de leur connectivité pourrait être automatisée [All+01 ; DFC05]. Ceci simplifierait la configuration de notre solution et par conséquent améliorerait sensiblement l'expérience utilisateur ;

— De même, les capacités de calcul annoncées par les producteurs à leur connexion pourraient être automatiquement évaluées par l'exécution préalable de tâches factices [Aug+11 ; Kal+14]. De façon complémentaire, leurs capacités pourraient être ajustées au fur et à mesure de l'exécution de tâches réelles [Aug+11 ; SCC15] afin de combiner les approches *profile-based* et *history-based* ;

— L'exécution des tâches *Preload* pourrait bénéficier des travaux menés sur les échanges *pair-à-pair* (P2P) [Cho+11] ou *multicast* [LP96 ; RV97] afin d'optimiser la bande passante nécessaire lors de l'assignation d'un travail sur plusieurs producteurs ;

— De même, selon le format des fichiers sources, seule une partie de leur contenu pourrait être récupéré. Mais cela contraint alors l'ordonnanceur dans ses assignations des portions de travail ;

— Dans notre implémentation actuelle, l'ordonnanceur génère linéairement les tâches *Compute* des différents producteurs. De ce fait, chaque nouvelle portion d'un travail est attribuée par rapport à la date de création de la tâche *Compute*. L'hétérogénéité est gérée par l'assignation dynamique qui assigne un nombre de portions de travail proportionnel à la vitesse des producteurs. Or, si un producteur est particulièrement lent à produire les données, alors celui-ci risque de retarder la mise à disposition à l'utilisateur des autres portions calculées parallèlement par d'autres producteurs plus rapides. Pour corriger cette limitation, nous proposons deux techniques différentes qui peuvent être combinées : soit générer des tailles de portion de travail selon les capacités de calcul des producteurs ; soit assigner les portions de travail selon les dates de fin estimées de ces tâches. La seconde technique permet de conserver un volume de données important, ce qui permet notamment de conserver l'efficacité des processeurs de type *SIMD* ;

— De même, la possibilité de générer des portions discontinues permettrait d'améliorer la précision des *pipelines* avancés présentés dans la figure 7.6 (d) où deux tâches *Compute* qui seront exécutées séquentiellement par un seul producteur sont à générer au même instant. Dans l'idéal, il faudrait espacer ces deux tâches *Compute* d'autant de portions qu'il y a de producteurs utilisés pour ce travail ;

— Enfin, dans les topologies utilisant plusieurs producteurs et plusieurs consommateurs, il est possible d'appliquer une stratégie de réplication d'une partie (ou de la totalité) des calculs sur les différents producteurs afin de réduire (voire supprimer si ceux-ci proposent une localité hôte) la consommation réseau des tâches *Expel*. Cette stratégie permet notamment de pallier à une capacité réseau entre les producteurs et les consommateurs qui serait trop faible, mais elle nécessite de disposer des puissances de calcul suffisantes pour assurer le débit demandé sur chaque consommateur sans la pleine assistance des autres producteurs.

7.4 Conclusion

La génération des données de notre solution de système distribué adapté aux besoins et aux contraintes de l'impression numérique repose sur un couple ordonnanceur-producteur dont la conception permet une grande flexibilité afin d'atteindre les performances maximales du cluster, comme nous le montrons en section 11.2 : *Débits maximaux de l'architecture*. L'ordonnanceur est conçu pour superviser efficacement les travaux des utilisateurs grâce à une vision omnisciente et centralisée de la gestion des ressources du cluster. Son fonctionnement nativement dynamique lui assure une adaptabilité à de nombreuses configurations et situations, dont font partie l'hétérogénéité et les pannes. Nous sommes également confiants des bénéfices apportés par sa politique personnalisable qui lui permet de s'étendre facilement à des usages particuliers ou domaines supplémentaires. La liberté d'implémentation des producteurs permet à notre solution de bénéficier d'architectures et de technologies différentes qui sauront maximiser les performances de calcul et de communication de chaque nœud. L'efficacité d'organisation et de fonctionnement de ce couple est vérifiée par les multiples évaluations réalisées sur des topologies et scénarios variés. Ces évaluations sont présentées dans le chapitre 11 : *Évaluations du RIP distribué*.

8 Agrégation et transmission des données à l'utilisateur

La distribution d'un travail sur plusieurs producteurs entraîne la dispersion des données générées et nécessite de les réunir pour les transmettre à l'utilisateur sous forme de flux de données. Dans notre solution, la réunification de ces données est effectuée sur le ou les consommateurs choisis par l'utilisateur comme points logiques de sortie des flux. Les données de ces flux sont séquentielles. Le respect de cette contrainte est assuré par le consommateur qui doit alors ordonner les données générées par les différents producteurs afin de les transmettre à l'utilisateur dans le strict respect de l'ordre demandé.

La position centrale des consommateurs dans cette double fonction d'agrégation et de mise en ordre des données pose évidemment des questions de performances dans une solution visant la génération de flux de données à haut débit. C'est pour cette raison que nous apportons un soin particulier à cet aspect de notre architecture. La topologie des éléments producteurs et consommateurs de notre solution amène à la problématique générale des échanges non bloquants de données ordonnées entre des producteurs multiples et un consommateur unique.

Dans ce chapitre, nous présentons notre solution originale permettant de répondre efficacement à ce besoin spécifique [God19]. Nous organisons la réception des données et leur stockage au sein d'une mémoire tampon unique qui est organisée sous la forme d'une mémoire circulaire et associée à un mécanisme de synchronisation *lock-free* des accès en écriture et en lecture. Notre solution se définit sous la forme d'une bibliothèque exposant aux pilotes d'impression une interface de programmation permettant un accès direct aux données ordonnées. Notre bibliothèque intègre trois mécanismes de synchronisation des processus pour les cas inévitables de blocage (tel que l'accès en lecture à une donnée manquante) dont nous évaluons les performances et analysons les avantages respectifs.

8.1 Interface consommateur

Présenté dans le chapitre 6 : *Approche générale pour la génération distribuée de flux de données*, la fonction du consommateur est d'abstraire à l'utilisateur la complexité du système distribué qu'il a mandaté pour produire les flux de données. Cette abstraction est réalisée en utilisant les consommateurs comme une interface entre l'utilisateur et les producteurs, ce qui permet

d'unifier le support des différentes topologies : *one-to-one*, *one-to-many*, *many-to-one* et *many-to-many*.

Chaque consommateur est chargé de collecter les données produites par un ou plusieurs producteurs. L'objectif est de pouvoir additionner les capacités des producteurs pour offrir en sortie un débit égal à la somme des puissances des unités de calcul impliquées dans l'exécution du travail. Simultanément à cette collecte des données, le consommateur les restitue en assurant le respect des contraintes d'ordre. Enfin, en raison de sa position critique le consommateur ne doit pas induire de résistance à la constance de débit des flux qui a été supervisée par l'ordonnanceur.

8.2 Producteurs multiples et consommateur unique

Du point de vue d'un consommateur, les topologies *one-to-one* et *one-to-many* consistent à récupérer les données provenant d'un seul producteur. Bien que ces topologies ne présentent en apparence pas de difficultés particulières de synchronisation des données produites, il n'est pas exclu que le producteur transmette simultanément plusieurs blocs de données au consommateur, générant ainsi un besoin de synchronisation des accès à la mémoire tampon et de mise en ordre de ces données. Ce scénario apparaît également lorsqu'on exploite pleinement toutes les capacités d'interconnexion entre un producteur et un consommateur, telles que la présence de plusieurs liens réseaux ou bien pour maximiser la bande passante d'une connexion locale. Plus de détails sur l'optimisation des communications sont fournis dans le chapitre 9 : *Aspects techniques de la gestion des communications*. Quant aux topologies *many-to-one* et *many-to-many*, elles incluent de par leur nature un besoin explicite de synchronisation des données produites. De plus, pour simplifier les schémas de communication de notre solution distribuée et conserver sa flexibilité, les consommateurs n'ont pas connaissance de la répartition des portions de travail sur les producteurs. Ainsi, les consommateurs ignorent d'où et quand les données vont arriver.

La réception des données s'effectue conformément aux explications de la section 7.2.3 : *Transfert des données produites au consommateur* : le producteur signale la portion de données qu'il vient de calculer ; le consommateur signale en retour lorsqu'il est prêt à recevoir ces données. Notre implémentation pour le domaine de l'impression numérique repose sur le découpage en portions horizontales de la sortie à produire qui correspondent à des lignes de l'image d'entrée et donc de sortie. Ainsi, le consommateur utilise cette unité de ligne de sortie pour organiser le stockage et la mise en ordre de ces données. La réception par le consommateur de ces lignes provenant d'un ou plusieurs producteurs doit pouvoir se faire de façon simultanée sous peine de rendre séquentiels les transferts des données et donc de brider le potentiel de parallélisation du système. Par conséquent, les consommateurs doivent impérativement supporter la réception des données dans le désordre.

Présentées en détail dans le chapitre 9 : *Aspects techniques de la gestion des communications*, nos communications exploitent des canaux de communications (appelées *Berkeley sockets*). Ainsi, chaque transfert de données s'effectue par leur transmission explicite (par les appels systèmes `send(3)` et `recv(3)`) et est opérée par deux processus : l'un pour l'envoi, l'un pour la réception. Du point de vue du consommateur, ce processus local chargé de la réception des données produites est le prolongement du producteur sur son nœud. Par conséquent, nous désignons ce processus sous le nom de producteur dans la suite de ce chapitre, et par symétrie, le pilote d'impression devient le consommateur. Ainsi, l'élément *Consumer*, présenté jusqu'ici sous le nom du consommateur, s'efface pour présenter plus précisément ses mécanismes internes.

La suite de ce chapitre décompose ces mécanismes de fonctionnement en présentant successivement les différentes couches techniques : l'organisation de la mémoire ; la synchronisation des accès à cette mémoire ; l'interface de programmation permettant son utilisation par les multiples producteurs (processus de réception des données) d'un côté et l'unique consommateur (le pilote d'impression) de l'autre côté.

8.2.1 Mémoire tampon circulaire

L'architecture actuelle des systèmes informatiques implique des mécanismes de protection des données entre différents processus. En conséquence, hors processus s'exécutant en mode noyau, les espaces d'adressage de chaque processus ne sont accessibles que par leurs *threads* et non par les autres processus du système. Cette sécurité empêche l'écriture directe de données de la part d'un producteur dans l'espace mémoire d'un consommateur et inversement pour une lecture directe depuis ce dernier dans l'espace mémoire d'un producteur si ceux-ci sont des processus indépendants. Il convient donc d'utiliser un mécanisme explicite de partage de données entre processus tout en évitant leur recopie entre le producteur et le consommateur, sous peine d'affecter fortement les performances de la solution. La solution doit également tenir compte de l'environnement dans lequel elle s'exécute afin de respecter ses contraintes. Il s'agit en particulier de ne pas requérir davantage de mémoire que le système ne peut en fournir, en incluant celle nécessaire à l'exécution simultanée du ou des producteurs et consommateurs. De plus, dans le cadre d'échanges indépendants de données ordonnées, il doit être possible de prendre en charge des données temporairement manquantes afin de ne pas linéariser les échanges. La complexité de cette condition est assouplie dans notre cas d'étude puisque nous nous intéressons à l'échange de données de taille fixe, ce qui permet de faciliter la prédiction de la quantité de mémoire nécessaire à chaque donnée ainsi que son emplacement futur en mémoire. Dans un cas d'utilisation plus général de notre solution, les flux seraient découpés en blocs de données numérotés de taille fixe.

Afin de satisfaire ces différentes contraintes, notre solution s'appuie sur l'utilisation de *mémoire partagée* comme zone de transfert des données d'un processus producteur au processus consommateur puisqu'elle permet des accès en écriture et en lecture à des processus indé-

pendants. Bien que le noyau Linux propose trois interfaces différentes à la création de ces mémoires partagées (System V, POSIX et memfd), nos expérimentations préalables ont montré des performances strictement identiques entre les trois, suggérant l'utilisation d'un même sous-système de partage de mémoire inter processus. Notre solution utilise l'interface POSIX de façon à maximiser sa portabilité.

Nous organisons cette mémoire partagée sous la forme d'un tampon circulaire (*circular buffer*) permettant de stocker de manière cyclique des données dans une plage d'adresses fixes. La quantité de mémoire partagée utilisée par notre solution est paramétrable par l'utilisateur lors de son initialisation en indiquant la taille d'une donnée et le nombre de données qu'il souhaite pouvoir y stocker simultanément. Si l'utilisateur sélectionne une capacité de stockage supérieure à une seule donnée, alors notre solution permettra l'écriture parallèle et non bloquante d'autant de données, à la condition que leur emplacement (calculé à partir du numéro de la donnée et de leur taille) soit libre, c'est-à-dire ne contienne pas une donnée non lue par le consommateur. Dans un *Consumer*, chaque point de sortie logique dispose de son tampon circulaire dédié afin et organiser indépendamment la réception et la mise en ordre des données issues des différents travaux.

L'utilisation de mémoire partagée nous permet également de bénéficier d'un mécanisme de *zero-copy* entre les producteurs et le consommateur. Pour ce faire, l'interface de programmation (API) de notre solution retourne directement l'adresse mémoire des données, ainsi le programme peut directement utiliser cette adresse dans ses opérations d'écriture et de lecture. Une fois l'opération effectuée il indique, via l'API, que cette donnée est désormais lisible ou bien réutilisable pour une autre écriture. L'obtention de ces adresses, qui déclenche la possibilité d'y écrire ou d'y lire, est assurée par la bibliothèque de synchronisation.

La taille de cette mémoire tampon circulaire définit le nombre de lignes pouvant y être stockées simultanément, et donc le nombre de portions de calcul que les producteurs peuvent y écrire parallèlement. La taille nécessaire de cette mémoire tampon et ses conséquences sont présentées dans le chapitre 7 : *Génération distribuée des données de travaux concurrents*. Dans notre implémentation, la taille de la mémoire tampon est définie en fonction de la configuration de l'ordonnanceur et du nombre d'éléments *Producer* impliqués au démarrage du travail en suivant la formule **8.1**. Le nombre d'éléments *Producer* nécessaires est connu par l'ordonnanceur grâce au débit requis indiqué dans la configuration du travail et la capacité des *Producers* indiqués à leur connexion au système. Le support d'un nombre dynamique de *Producers* au cours de l'exécution du travail est débattu dans la section 8.4 : *Limites et extensions*. L'objectif de cette formule est de fournir suffisamment d'espace pour pouvoir stocker simultanément quatre portions du travail pour chaque élément *Producer*. Dans notre étude empirique, ce parallélisme maximal de quatre portions n'est en pratique jamais saturé, mais il permet une marge d'absorption qui a montré ses avantages. Cependant, nous limitons la taille de la mémoire tampon à la valeur maximale de 1 Gio afin de ne pas phagocyter les

ressources mémoire du nœud, permettant ainsi une utilisation équitable des ressources du nœud, notamment si un élément *Producer* s'exécute localement.

Soient :

— $T_{portion}$ taille d'une portion de données générées ;

— N nombre de producteurs initialement alloués à l'exécution du travail ;

alors :

$$\text{Taille mémoire tampon} = T_{portion} \times N \times 4 \tag{8.1}$$

8.2.2 Synchronisation

Une fois la taille et l'organisation de la mémoire tampon définies, il est essentiel de concevoir la stratégie avec laquelle les processus vont y accéder efficacement. La synchronisation des producteurs et du consommateur est nécessaire pour deux opérations :

— pour qu'un producteur obtienne l'adresse de la ligne qu'il veut écrire ;

— pour que le consommateur obtienne l'adresse de la ligne suivante à lire.

L'échange de données étant dépendant de l'exécution du ou des producteurs, il existe deux scénarios empêchant d'honorer immédiatement une requête :

— la lecture d'une ligne encore non écrite ;

— l'écriture d'une ligne alors que l'emplacement est utilisé par une ligne non lue.

De façon à simplifier son utilisation, notre solution propose de bloquer le processus appelant afin de le mettre en attente, avec pour objectif complémentaire de le débloquer avec une latence minimale.

Une approche naïve consiste à assurer l'intégrité de l'état (lu/non lu) des données de la mémoire circulaire en assurant l'accès à une section critique par des mutex puis si nécessaire de suspendre/réveiller les processus par l'utilisation de conditions et de signaux. Ces opérations sont classiquement réalisées via l'utilisation de la bibliothèque `pthreads(7)`. Cette approche naïve que nous évaluons dans le chapitre 8.3 : *Évaluations* y est désignée sous le nom de *mutex/cond*.

Pour accélérer cette synchronisation, notre solution propose de supprimer le recours à des sections critiques. Ainsi, pour connaître l'état d'utilisation de la mémoire circulaire notre solution place en en-tête de la mémoire circulaire deux informations : un entier désignant le prochain numéro de données que le consommateur doit lire (initialement 1) ; et un tableau d'entiers dont chaque élément indique le numéro (initialement 0) de la ligne de données actuellement stockée à l'adresse correspondante dans le buffer circulaire. L'état (c'est-à-dire le numéro) de chaque ligne peut être interrogé et modifié sans jamais être corrompu. Si la

valeur du prochain élément à lire et l'état de la ligne correspondante tombent dans un des deux scénarios bloquants présentés précédemment, alors notre solution propose deux stratégies pour maintenir efficacement le processus en attente. La première consiste en une attente active sur la valeur de l'état, jusqu'à ce que celui-ci soit positionné sur la valeur attendue. La seconde ajoute à cette attente active l'appel système `sched_yield(2)` indiquant au noyau le souhait d'être préempté par un autre processus. Dans le chapitre 8.3 : *Évaluations* ces deux stratégies sont respectivement évaluées sous les noms d'*active* et de *yield*.

8.2.3 Utilisation de l'interface de programmation

Les opérations de création, de gestion et de synchronisation des accès à cette mémoire tampon sont abstraites dans une bibliothèque commune afin de simplifier la complexité algorithmique des producteurs et des consommateurs. Cette bibliothèque expose une interface de programmation explicite qui est présentée dans les algorithmes **8.2** et **8.3** (où ses fonctions y sont indiquées par le préfixe `cbuf_` et soulignées) :

— Le premier décrit la réception depuis la connexion `fd_socket` d'une portion débutant à la ligne `pos` du travail et contenant `size` lignes. Le processus producteur récupère l'adresse de cette portion de données dans la mémoire tampon `cbuf` ; y organise directement la réception les données ; et lorsque toute la portion est écrite, il le signale à la mémoire tampon ;

— Le second présente les opérations réalisées par le processus consommateur pour lire ligne par ligne les données du travail qui ont été générées par notre solution distribuée : demande l'adresse mémoire de la prochaine ligne à la mémoire tampon `cbuf` ; exécute les traitements spécifiques qui, dans notre cas, dépendent du fabricant de l'imprimante et du modèle, tels que l'ajout d'un court en-tête ou caractères d'encapsulation ; signale à la mémoire tampon que la ligne a été lue.

8.3 Évaluations

Nous évaluons indépendamment la mémoire tampon circulaire du consommateur et ses mécanismes de synchronisation qui organisent l'accès en écriture et en lecture aux données qu'il contient. Les performances de cette mémoire tampon sont déterminantes pour l'efficacité des *Consumers* de notre solution de chaîne de traitement graphique distribuée.

8.3.1 Protocole d'évaluation

Afin d'évaluer les performances de notre solution, nos évaluations mesurent le temps d'exécution du mécanisme de synchronisation en incluant l'accès aux données échangées. Afin d'isoler au mieux le temps pris par ces mécanismes, la taille des éléments est définie à 1 octet. Le fonctionnement de la synchronisation étant dépendante du niveau de remplissage de la

```c
/* Tant que des portions sont a recevoir */
while (data_to_receive)
{
    /* En attente d'une portion de données d'un travail à recevoir */
    recv(fd_socket, &portion_header, portion_header_size);

    /* Décompose l'en-tête de la portion */
    cbuf = get_cbuf_from_job(portion_header.id); // Mémoire tampon concernée
    portion_pos  = portion_header->pos;          // Position de la portion
    portion_size = portion_header->size;         // Taille de la portion

    /* Récupère l'adresse à laquelle écrire la portion de données
     * Attente bloquante si besoin */
    void *portion_addr = cbuf_wait_space(cbuf, portion_pos, portion_size);

    /* Réception directe de la portion dans le buffer circulaire */
    recv(fd_socket, portion_addr, portion_size);

    /* Informe le buffer de la portion de données écrite */
    cbuf_fill_space(cbuf, portion_pos, portion_size);

    /* Réponse au producteur que les données ont bien été
     * reçues et écrites */
    write(fd_socket, &portion_header, portion_header_size);
}
```

Algorithme 8.2 : Réception d'une portion de données par le processus producteur d'un *Consumer*

```c
/* Tant que le travail contient des lignes à imprimer */
while (line_in_job--)
{
    /* Demande l'adresse de la prochaine ligne du travail
     * Attente bloquante si besoin */
    void *line_addr = cbuf_get_next_line(cbuf);

    /* Traitement de la prochaine ligne du travail */
    process_line(line_addr); // Opérations spécifiques à chaque imprimante

    /* Indique à la mémoire tampon que cette ligne du travail a
     * fini d'être lue */
    cbuf_free_next_line(cbuf);
}
```

Algorithme 8.3 : Lecture dans l'ordre des lignes d'un travail par le processus consommateur d'un *Consumer*

mémoire tampon (un faible remplissage augmente le risque d'attente pour la lecture tandis qu'un fort remplissage celui d'attente pour l'écriture) nous évaluons notre solution en fonction de la capacité de la mémoire tampon. La notion de circularité impliquant des mécanismes de bordure, chaque évaluation réalise l'échange d'une quantité d'éléments supérieure à la capacité de la mémoire tampon.

Notre programme d'évaluation exécute un *thread* consommateur qui lit en continu la mémoire tampon circulaire : dès qu'il reçoit l'adresse de l'élément suivant, il y accède puis le signale comme lu ; parallèlement, un ou plusieurs *threads* producteurs demandent en continu d'écrire dans la mémoire tampon circulaire en entrelaçant leurs écritures selon le nombre de producteurs.

8.3.2 Plateforme d'évaluation

Nous réalisons nos mesures sur deux processeurs différents : un Intel Xeon D-1521 ayant 4 cœurs physiques **avec** *hyper-threading* cadencés à **2,40 GHz** ; un Intel Core i5-3470 ayant 4 cœurs physiques **sans** *hyper-threading* cadencés à **3,20 GHz**. Ils sont tous deux équipés de $2\times$ 4 Gio de mémoire vive, respectivement en DDR4 à 2133 MT/s et en DDR3 à 1600 MT/s. Nos programmes sont compilés avec `gcc` en version 8.2.0 et exécutés sous Ubuntu 18.04.1 LTS avec le noyau Linux 4.15.0-43-generic.

8.3.3 Analyse des résultats

Les résultats de nos évaluations sont présentés dans les figures 8.4 à 8.7. Sur chacune, les marqueurs verticaux représentent le nombre d'éléments à partir duquel la taille de la mémoire tampon dépasse celle du cache : respectivement de gauche à droite celle du cache L1, L2 et L3.

La figure 8.4 montre une relation entre le nombre de lectures par seconde et la fréquence du processeur, quelle que soit la méthode de synchronisation. En effet, le processeur Intel Xeon D-1521 affiche une fréquence de fonctionnement inférieure de 25 % à celle du processeur Intel Core i5-3470, ce qui correspond à la différence globale de performance entre les deux figures 8.4 (a) et 8.4 (b). De plus, cette figure met en avant les faibles performances d'un système de synchronisation *mutex/cond* pthread par rapport à notre solution, en effet, avec la stratégie de synchronisation *yield* notre solution exécute jusqu'à 6,6 fois plus de lectures par seconde. Il est intéressant de remarquer la meilleure performance de la solution *active* par rapport à la solution *yield* pour les mémoires tampons de petites capacités (< 8 éléments). En effet, la faible capacité de la mémoire tampon provoquant de nombreux blocages, celle-ci se montre plus réactive.

Les figures 8.5, 8.6 et 8.7 comparent les performances des méthodes de synchronisation en fonction du nombre de producteurs et de l'assignation explicite ou non des producteurs et du consommateur sur les cœurs du CPU. La numérotation des cœurs correspond à celle du système (via `/proc/cpuinfo`), le cœur 4 est ainsi le cœur *hyper-threadé* complémentaire du cœur 0. La notation dans les légendes des figures indique à quel numéro de cœur le consommateur (C) et les producteurs (P) sont assignés. Une assignation explicite des producteurs et du consom-

mateur sur les cœurs du CPU permet d'influer fortement sur les performances par rapport à l'assignation dynamique faite par le noyau par défaut (annotée $*$). Nos évaluations n'ayant pas montré d'améliorations notables des performances pour la synchronisation *mutex/cond* en fonction de l'assignation sur les cœurs, nous ne la présentons pas dans ces figures.

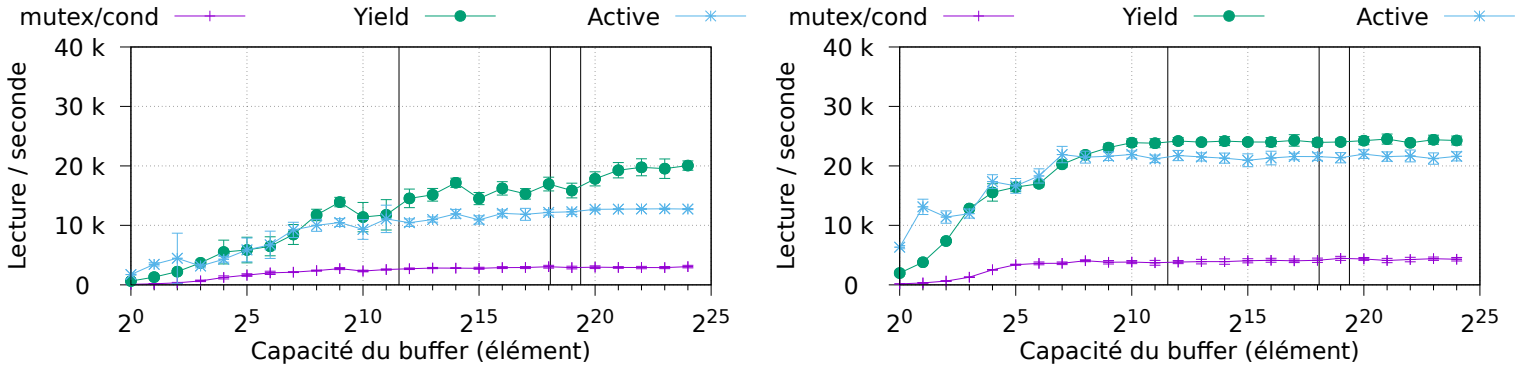
Le seul choix d'assignation clairement préjudiciable consiste à assigner tous les processus sur le même cœur. Cela est particulièrement visible pour la synchronisation *active* (courbes C0 P0), figures **8.5 (a)**, **8.6 (a)** et **8.7 (a)**, qui montrent des résultats catastrophiques quand la mémoire tampon circulaire tient dans le cache L1 ou L2. Au-delà, la latence imposée par le cache L3 et la mémoire vive force la préemption des processus, ce qui libère du temps d'exécution pour les autres processus et vient débloquer le processus en attente. À l'opposé, les meilleures performances sont obtenues lorsque les processus sont assignés deux à deux sur les cœurs hyper-threadés, en privilégiant le regroupement en premier lieu d'un producteur avec le consommateur. La figure **8.5 (a)** présente ainsi une performance doublée pour la configuration *C0 P4* par rapport à l'assignation par défaut *C* P**. De façon générale, la stratégie *yield* présente un profil équilibré, peu importe les assignations, ce qui suggère une meilleure robustesse sur des systèmes variés. De plus, ces résultats sous-entendent une utilisation plus équitable des ressources de la machine que celle de la stratégie *active*, ce qui devrait être un atout sur une machine dont les cœurs sont fortement sollicités par d'autres tâches, comme la génération des données.

Pour conclure l'analyse de ces évaluations, il est intéressant de situer les résultats dans notre cas d'application. Premièrement, la stabilité des performances, et surtout, l'absence de ralentissement lorsque la taille de la mémoire tampon dépasse celle des caches nous permet d'utiliser des mémoires tampons de taille importante permettant de prévenir les famines, comme prévu dans les chapitres précédents. Secondement, il est à noter que plus il y a de processus producteurs, plus les performances du processus consommateur sont bonnes.

8.4 Limites et extensions

Cette architecture de mémoire tampon circulaire permet de gérer efficacement le problème de réception désordonnée des portions de travail envoyées par les différents éléments *Producer*. Cependant, des améliorations sont possibles :

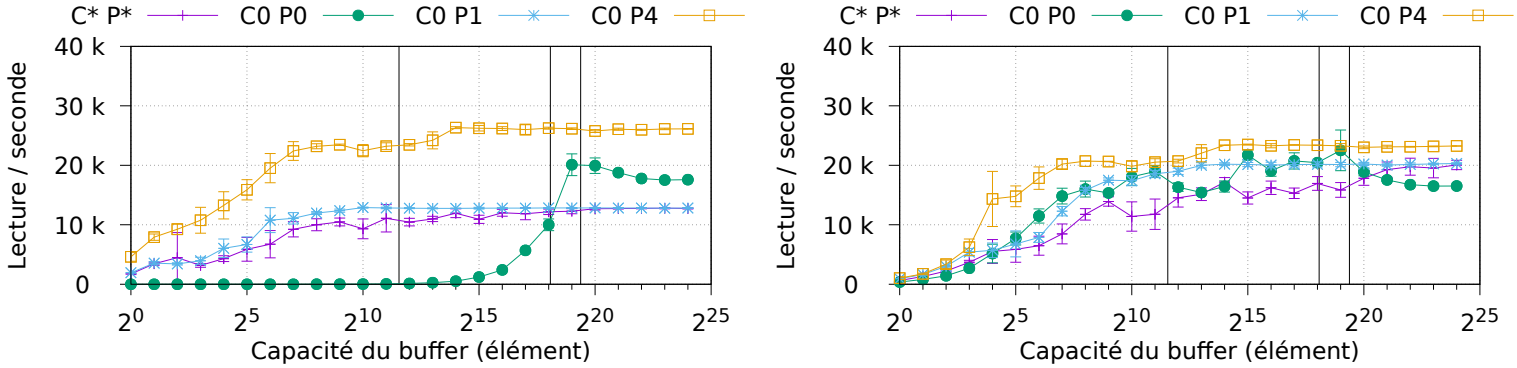
— Actuellement, la taille de la mémoire tampon est définie par l'équation **8.1** au lancement du travail en fonction de la capacité annoncée des *Producers* et du débit du travail requis. Or, si l'une de ces deux métriques est erronée, alors le nombre de *Producers* associé est susceptible de varier durant son exécution. Ainsi, il serait intéressant de pouvoir redéfinir la taille de la mémoire tampon « à chaud ». Cependant, selon le système d'exploitation, cette opération de redimensionnement d'une zone mémoire peut invalider les adresses en cours d'utilisation par les producteurs ou le pilote d'impression. Il est



(a) : Intel Xeon D-1521

(b) : Intel Core i5-3470

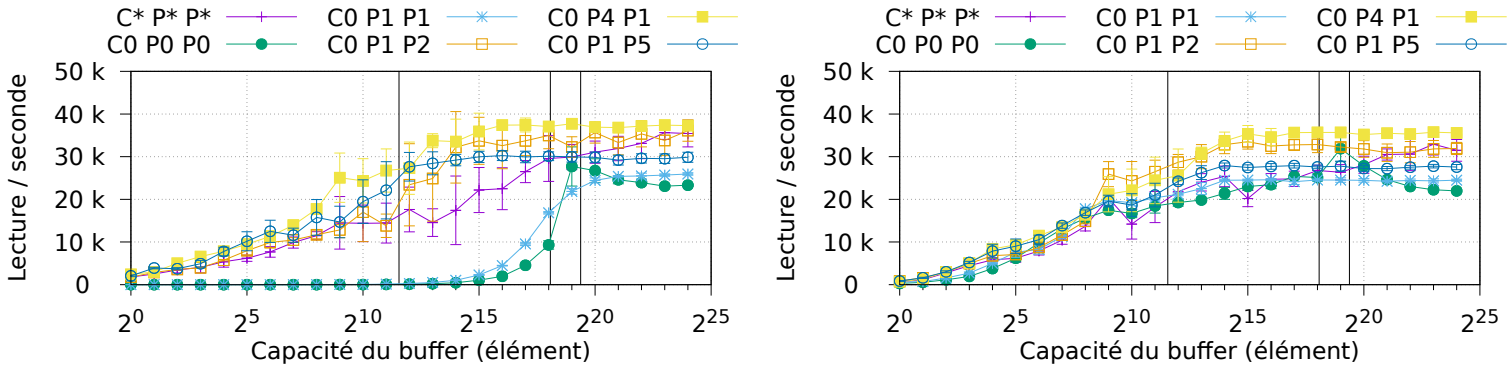
Figure 8.4 : Nombre de lectures par seconde pour **un** producteur en fonction de la capacité de la mémoire tampon et de la méthode d'attente utilisée sur différents processeurs



(a) : Attente active

(b) : Attente yield

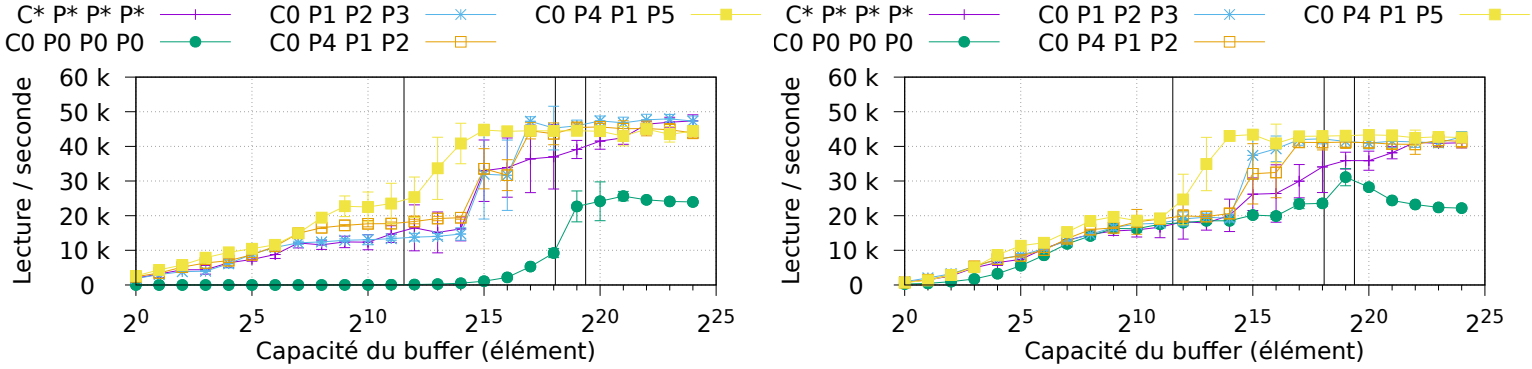
Figure 8.5 : Nombre de lectures par seconde pour **un** producteur en fonction de la capacité de la mémoire tampon et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521



(a) : Attente active

(b) : Attente yield

Figure 8.6 : Nombre de lectures par seconde pour **deux** producteurs en fonction de la capacité de la mémoire tampon et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521



(a) : Attente active

(b) : Attente yield

Figure 8.7 : Nombre de lectures par seconde pour **trois** producteurs en fonction de la capacité de la mémoire tampon et du placement des processus selon la méthode d'attente utilisée sur Intel Xeon D-1521

donc nécessaire de réaliser cette opération en lui ajoutant les mécanismes suffisants pour garantir la cohérence des données sans pour autant entraîner une famine auprès du pilote d'impression ;

— Dans le cas d'une génération des données par un *Producer* situé sur le même nœud physique que le *Consumer*, il serait possible d'éviter la recopie des données qui a lieu lors de la communication locale, améliorant ainsi les performances. Cependant, cette amélioration introduit une forte dépendance entre l'état de la mémoire tampon et les tâches *Compute*. En effet, le *Producer* doit alors disposer des adresses mémoires de la mémoire tampon dans un état libre pour pouvoir exécuter les calculs et y écrire directement les données générées.

8.5 Conclusion

Ce chapitre a permis de présenter les différents mécanismes qui composent les *Consumers* et leur permettent de délivrer efficacement les données générées à l'utilisateur tout en garantissant leur ordre. Nos évaluations démontrent l'efficacité de notre solution pour la transmission non bloquante de données de taille fixe entre un ou plusieurs producteurs vers un unique pilote d'impression tout en assurant leur mise en ordre. En effet, notre solution apporte deux indicateurs de passage à l'échelle essentiels. Premièrement, une augmentation des performances avec l'ajout de producteurs : de 20k lectures/s pour un seul producteur à 40k lectures/s pour trois producteurs. Secondement, une stabilité remarquable des débits de lecture une fois la taille des caches du processeur dépassés. Ces deux indicateurs donnent à notre solution la capacité de gérer efficacement des travaux toujours plus rapides et volumineux.

Notre solution, qui dispose de deux stratégies de synchronisation différentes, est jusqu'à 6,6 fois plus performante qu'une solution avec *mutex/cond* pthread et peut être améliorée davantage en assignant les processus sur les cœurs appropriés. De plus, le design généraliste de notre solution la rend applicable à de nombreux cas d'usage autres que l'impression numérique. Nos prochains travaux visent d'une part à permettre le redimensionnement à chaud de la mémoire tampon, et d'autre part à lever la contrainte sur la taille fixe des données pour permettre l'échange de données de taille variable, telles que des données compressées.

9 Aspects techniques de la gestion des communications

Les bonnes performances d'un système distribué sont intimement dépendantes de son utilisation du réseau. Au-delà du protocole des messages échangés que nous avons présenté dans le chapitre 6, son utilisation découle également des technologies employées pour échanger efficacement ces messages et leurs données. Ce chapitre est dédié à présenter les technologies mises en œuvre dans notre solution ainsi que le travail réalisé pour en exploiter leurs performances maximales.

Nous avons implémenté une bibliothèque de communication spécifique aux besoins de notre solution en matière de faibles latences, haut débit et flexibilité. Notre implémentation est utilisée pour toutes les communications de notre solution, et donc essentiellement pour l'interconnexion de l'ordonnanceur, des producteurs et des consommateurs, entre eux, et avec l'environnement dont font partie l'utilisateur et le stockage.

Dépendante des configurations des travaux soumis durant l'exécution du système distribué, les connexions, dédiées à chaque travail, entre les producteurs et les consommateurs ne peuvent pas être déterminées à l'avance alors que celles-ci sont à même de générer de nombreuses communications concurrentes pouvant congestionner les équipements réseau. Nous abordons cette problématique par l'emploi d'une stratégie globale de type *best-effort delivery* dans laquelle nous prônons une équité entre les différents messages et flux de données circulant sur le réseau. Néanmoins, leur répartition est supervisée par l'ordonnanceur qui, en accord avec sa politique, est en mesure de les répartir dynamiquement sur le cluster en répartissant, dans la mesure du possible, les travaux eux-mêmes.

9.1 Communications locales et distantes

Notre bibliothèque de communication est en charge d'ouvrir et de gérer les différentes connexions du système avec les célèbres *Berkeley sockets* au travers de leur déclinaison `UNIX` pour les communications locales et `INET` pour les communications distantes [SFR04]. Dans ces deux déclinaisons, nous utilisons le mode connecté `SOCK_STREAM`. Le choix des *Berkeley sockets* en mode *connecté* permet d'unifier les communications locales et distantes et participe

ainsi à la gestion flexible et homogène des différents éléments du cluster tout en assurant une portabilité maximale sur d'autres systèmes [1].

Le choix de la déclinaison `INET` associée au mode connecté `SOCK_STREAM` permet d'utiliser les protocoles IPv4 et TCP pour le transport des données. L'utilisation de ces deux protocoles assure une intégrité des données transmises sur une variété de topologies physiques ainsi qu'un support quasi universel par les équipements réseau actuels. Enfin, nos expérimentations ont confirmé que les implémentations actuelles de TCP permettent une application efficace de la politique globale d'équité des flux souhaitée. En effet, le démarrage d'un flux concurrent à un flux déjà présent résulte en un partage équitable de la bande passante du lien, et une récupération de la totalité de celle-ci lorsqu'un des deux flux s'arrête.

Enfin, l'interface bas niveau accessible par l'utilisation des *Berkeley sockets* permet également une gestion fine des interfaces réseau des différents nœuds. Les informations de ces interfaces sont remontées à l'ordonnanceur qui peut ainsi organiser finement des communications *point à point* en s'appuyant sur sa vue globale des travaux en cours.

9.1.1 Persistance des connexions et échanges de données

Bien que très répandue, l'utilisation du protocole TCP n'est pas sans contrainte et inconvénient. En effet, le fonctionnement connecté du protocole repose sur son *three-way handshake* préalable à toute communication entre les deux interfaces réseau. Or l'établissement de cette poignée de main comporte deux inconvénients :

1. Premièrement, cette connexion impose une latence avant de pouvoir échanger des données : d'un *round-trip time* (RTT) pour envoyer des données ; de deux RTT pour en recevoir (*TCP Fast Open* permet de diminuer ces délais d'un RTT entre deux interfaces réseau éligibles) ;
2. Secondement, l'ouverture de cette connexion est dépendante de la réception des premiers paquets `SYN`, qui, en cas de perte ne sont retransmis qu'après un délai supérieur à 1 seconde [RFC6298]. Sur les systèmes UNIX, ce délai est classiquement de 1 seconde les 5 premières tentatives, puis est doublé à chaque tentative supplémentaire jusqu'à atteindre un temps d'attente de plusieurs minutes, auquel cas la connexion est abandonnée.

Face à ces inconvénients, nous privilégions l'ouverture des connexions le plus tôt possible, et surtout la réutilisation de connexions précédemment ouvertes entre deux processus du système. Cette politique de connexions persistantes permet d'économiser des RTT de latence qui sont particulièrement sensibles pour l'échange de courts messages de synchronisation, mais également de supprimer des délais d'attente démesurés de retransmission des paquets `SYN` qui

1. Afin d'unifier complètement la gestion des connexions, nous aurions également pu utiliser la déclinaison `INET` pour les connexions locales grâce aux interfaces *loopback* des différents nœuds, mais nos expérimentations ont montré une perte de débits de l'ordre de 15 à 20 % par rapport à la déclinaison `UNIX`. Cette sous-utilisation des capacités de communication s'explique par le surcoût d'exécution nécessaire à la construction et la gestion des paquets TCP.

sont enclins à se produire régulièrement sur des liens congestionnés tels que présents dans notre système distribué orienté haut débit. De plus, conserver des connexions persistantes permet de conserver les acquis de celle-ci, tels que la taille des fenêtres d'envoi.

L'utilisation des *Berkeley sockets* comporte également des points sensibles qu'il est important de maîtriser pour assurer la cohérence d'un système distribué complexe :

1. Premièrement, lors de la connexion, l'interface distante est perçue comme fonctionnelle dès la phase de connexion réalisée (*TCP handshake* pour les connexions distantes) et non lors de l'appel à la fonction `accept(2)` par le destinataire. Il est nécessaire d'envoyer un message applicatif (au sens du modèle OSI) pour assurer l'acceptation effective par le destinataire, ce qui vient ajouter un RTT supplémentaire et renforce l'intérêt des connexions persistantes :
2. Secondement, le retour au programme de la fonction `send(2)` ne donne aucune garantie de l'envoi effectif des données au destinataire et encore moins de leur bonne réception par celui-ci. En effet, cette fonction n'assure que la prise en charge des données correspondantes par le noyau après quoi l'utilisateur perd toute information sur l'état de ces données. Ainsi, si nécessaire, il convient d'assurer la réception et la prise en compte de ces données par une confirmation au niveau applicatif.

Il est évident qu'exploiter la bande passante maximale des liens entre les différents éléments est essentiel pour optimiser la bande passante de l'ensemble du système. Cependant, bien qu'il soit relativement facile d'exploiter la bande passante maximale d'un lien réseau en utilisant une *Berkeley socket* distante de type `INET`, il en est autrement pour les communications inter processus utilisant une *Berkeley socket* locale de type `UNIX`. En effet, si les liens réseau distants sont saturés à la seule condition de fournir suffisamment rapidement des données à l'interface, en revanche, fournir rapidement des données à une connexion inter processus n'assure pas une utilisation maximale des capacités de communication inter processus d'un nœud. Nos expérimentations ont montré qu'exploiter parallèlement plusieurs connexions `UNIX` décuple le débit total obtenu. En ce sens, il semblerait que l'utilisation d'une connexion `UNIX` par cœur physique du processeur donne les meilleurs résultats, permettant ainsi de multiplier le débit par autant de cœurs physiques utilisés. Cette observation semble indiquer un goulot d'étranglement au niveau de l'interconnexion entre les canaux mémoires de la mémoire vive et chacun des cœurs physiques du processeur. Les moteurs d'exécution des tâches présentés dans le chapitre 7 : *Génération distribuée des données de travaux concurrents* sont à même d'exploiter cette observation afin d'améliorer les échanges de données entre les processus d'un même nœud.

De la même manière, minimiser la latence des opérations de lecture et d'écriture des communications permet d'améliorer la réactivité globale du système. Il est par exemple essentiel de réaliser parallèlement les opérations de lecture lorsqu'elles concernent des connexions différentes. Nos expérimentations ont montré l'avantage décisif de dédier un *thread* à la lecture permanente de chaque connexion plutôt que d'employer un `select(2)` global surveillant les

connexions et de créer dynamiquement un *thread* (avec la bibliothèque `pthread`) pour lire les connexions actives. En effet, la seconde méthode bien que minimisant le nombre de *threads* utilisés provoque un délai important évalué à environ 8 ms par lecture. Par conséquent, nous utilisons dans notre solution la méthode des *threads* dédiés qui permet une réactivité quasi instantanée au prix d'une consommation mémoire légèrement supérieure.

9.1.2 Utilisation optimisée des protocoles TCP et IPv4

Les implémentations actuelles des protocoles TCP et IPv4 proposent aux développeurs un large panel d'options pour ajuster leur comportement [RFC7323 ; Lin17]. Nous employons trois ajustements pour optimiser les performances du protocole TCP dans notre système :

1. Premièrement, nous activons l'option `TCP_NODELAY` qui désactive l'algorithme de Nagle [RFC896] et supprime ainsi la rétention (pouvant aller jusqu'à 200 ms sur les systèmes Linux) des paquets inférieurs à la taille du *maximum transmission unit (MTU)* en exécutant leur transmission immédiatement. Cette option réduit significativement la latence des messages de synchronisation utilisés dans notre solution en échange de quelques kilobits de bande passante inutilisés ;
2. Deuxièmement, nous activons l'option `TCP_QUICKACK` qui supprime la rétention des messages d'acquittement et l'utilisation du *piggybacking* qui fusionne les messages d'acquittement avec des messages de données éventuellement émis dans les prochaines millisecondes [RFC1644]. Ici aussi, la latence en est réduite en échange d'un MTU sous-utilisé ;
3. Troisièmement, nos expérimentations ont confirmé l'importance du choix de l'algorithme de contrôle des congestions pour maximiser la bande passante des liens réseau [GM04]. Ainsi, nous utilisons l'algorithme de congestion TCP *NewReno* qui, grâce à sa relative agressivité, accélère la retransmission des paquets perdus sur les réseaux ayant un faible RTT, tel que le réseau d'un cluster. Il offre sur notre plateforme d'évaluation une accélération des débits allant jusqu'à 1,21× pour la topologie *many-to-one* (qui est génératrice d'une congestion importante sur l'interface du switch desservant le consommateur) par rapport à l'algorithme de congestion TCP *CUBIC* utilisé par défaut, ce qui nous permet d'exploiter la bande passante maximale du lien.

Les systèmes Linux cherchent à offrir par défaut une connectivité maximale d'un nœud en fusionnant les différentes interfaces réseau si celles-ci appartiennent à un même sous-réseau IP. Les interfaces sont toutes fonctionnelles, mais une seule est utilisée pour l'émission des paquets à destination de ce sous-réseau, ce qui par le jeu des requêtes ARP redirige tous les paquets entrants sur cette unique interface. Cette politique empêche d'exploiter les capacités de réception de plusieurs cartes réseau installées dans un même nœud. Pour lever cette contrainte technique, nous configurons le noyau Linux pour dédier une table de routage à chacune des interfaces du nœud, permettant ainsi à notre ordonnanceur de superviser indépendamment l'utilisation de chaque interface de chaque nœud du cluster.

9.2 Limites et extensions

L'intégration bas niveau de notre bibliothèque de communication lui permet de bénéficier de nombreuses extensions pouvant améliorer ses limitations actuelles. Ainsi, l'utilisation des *Berkeley sockets* offre un support natif de nombreux autres protocoles de communication, tel que l'IPv6 au moyen de la déclinaison `INET6`. De même, de nombreuses options de ces *sockets* et de l'implémentation des protocoles réseau offrent de nombreux axes d'amélioration. Citons par exemple :

— L'utilisation des mécanismes *keepalive* au travers des nombreuses options disponibles dans les implémentations réseau des systèmes d'exploitation actuels (`SO_KEEPALIVE`, `TCP_USER_TIMEOUT`, etc.) [SFR04] afin de pouvoir détecter plus rapidement certaines pannes de nœuds ;

— L'exploitation des mécanismes *zero copy* au travers des stratégies *RDMA* présentées dans la section 5.2.4 : *Partage de données inter-processus* pour diminuer la charge de travail du processeur lors de l'envoi et de la réception des données ;

— Sous réserve que l'ensemble des équipements réseau traversés soient compatibles et configurés en conséquence, l'utilisation des *jumbos frames* du protocole Ethernet permet une optimisation de la bande passante des liens en augmentant le MTU [CGY01], ce qui réduit alors la quantité d'en-têtes nécessaires pour transmettre de grandes quantités de données ;

— L'agrégation d'interfaces et de liens réseau [RFC7424] ou, plus vastement, l'exploitation de plusieurs chemins [RFC6897] afin de bénéficier de la totalité des bandes passantes existant entre les nœuds pour accélérer les transferts.

De plus, de nombreux travaux et technologies permettent d'améliorer la qualité de service des différents types de communications pour certaines applications [LYF98 ; SC99 ; FRS00]. L'utilisation d'une approche incluant une qualité de service au niveau des communications pourrait venir assister l'ordonnanceur pour garantir le respect de certaines contraintes voulues par l'utilisateur du système en échange d'une configuration plus poussée des nœuds et équipements du cluster.

9.3 Conclusion

La conception d'une bibliothèque de communication spécifique aux besoins de notre solution distribuée nous permet de maîtriser l'ensemble des communications du système. Les technologies et méthodes employées offrent une gestion fine des différentes connexions qu'elle requiert : une faible latence, un haut débit, ou les deux. De plus, l'utilisation de technologies standards et éprouvées assure à notre bibliothèque, et donc à notre système, une évolutivité ainsi qu'une capacité d'adaptation à des cas d'application variés. Enfin, la maîtrise efficace des commu-

nications fournie par notre bibliothèque est une des clés qui permet à notre ordonnanceur d'exploiter efficacement l'ensemble des ressources du cluster.

Quatrième partie

Évaluations

10 Simulation d'un pilote d'impression

La chaîne de traitement graphique ne serait pas complète sans la présence à son extrémité d'un pilote d'impression nécessaire à l'évaluation de ses performances. Dans notre référentiel, il représente l'utilisateur des données générées par notre système distribué. Dans les conditions réelles, il est en charge d'assurer le pilotage d'une imprimante et d'ajouter ces données dans son flux de communication en respectant les spécificités de chaque modèle d'imprimante. Devant la grande variété des imprimantes et de leurs évolutions au fil des années, il existe autant de pilotes d'impression que de modèles d'imprimantes jet d'encre professionnelles.

Par conséquent, pour disposer d'un pilote d'impression qui ne soit pas bridé par les contraintes de son modèle associé, nous simulons un pilote d'impression en isolant la composante qui leur sera universelle : la récupération des données générées par notre chaîne de traitement graphique distribué. Nous disposons de deux simulations de pilote d'impression capables de tester et d'évaluer l'efficacité de notre solution dans de nombreuses configurations.

10.1 Pilotes standards d'imprimante numérique

Le premier pilote d'impression que nous simulons est celui d'une imprimante numérique classique représentant l'usage fait au sein des fermes d'impression : ce pilote d'impression n'accepte qu'un seul travail à la fois et ne pilote qu'une seule imprimante. Conformément à notre architecture, ce pilote d'impression est physiquement placé sur le même nœud qu'un *Consumer* qui lui servira de point logique de sortie des données. Il n'y a donc qu'un seul point logique de sortie pour toutes les données du travail. Ainsi, peu importe le nombre de canaux chromatiques de l'image de sortie, ils sont placés de façon contiguë dans chaque ligne de la mémoire tampon, si bien qu'une ligne de donnée contient tous les canaux. Cette organisation des données est très classique pour les pilotes d'impression des imprimantes nécessitant un débit raisonnable.

10.2 Pilotes distribués pour presse numérique

Le second pilote d'impression que nous simulons est implémenté sur la base des contraintes particulières des presses numériques. Ainsi, il se décompose en plusieurs processus répartis sur plusieurs nœuds afin d'atteindre les débits importants requis par ces presses numériques.

Chaque processus est un petit pilote d'impression ayant à sa charge de transmettre par un lien réseau dédié un seul canal chromatique. Par conséquent, chaque processus a besoin en local d'un *Consumer* formant le point logique de sortie des données du canal chromatique lui correspondant. Enfin, par souci de gestion de ces processus et de synchronisation, ils sont synchronisés par un processus supplémentaire appelé *processus maître* qui s'occupe de leur transmettre les informations nécessaires pour chaque travail à traiter.

10.3 Simulation de consommation des données

Nos implémentations de pilote d'impression récupèrent les données générées ligne par ligne au travers de la mémoire tampon située dans le *Consumer* et détaillée dans le chapitre 8 : *Agrégation et transmission des données à l'utilisateur*. Grâce au *Consumer*, les données lui sont fournies dans l'ordre, dès qu'elles sont disponibles et sans connaissance du *Producer* qui les a calculées.

Nous proposons deux méthodes d'accès à ces données : dans la première, le pilote d'impression va tenter d'écouler ces lignes le plus rapidement possible afin d'évaluer le débit maximal auquel le système peut fournir les données ; dans la seconde, il impose un délai d'attente avant la récupération de la ligne suivante afin de simuler la présence d'un traitement spécifique à une imprimante. La longueur de ce délai d'attente est définie en fonction de la vitesse d'impression de l'imprimante simulée et se détermine par l'équation 10.1.

Soient :

— $D_{imprimante}$ débit de l'imprimante (en octets par seconde) ;

— $T_{travail}$ taille du travail (en octets) ;

— Nb_{lignes} nombre de lignes dans le travail ;

alors :

$$\text{Délai d'attente pour chaque ligne} = \frac{(T_{travail} \ / \ D_{imprimante})}{Nb_{lignes}} \qquad (10.1)$$

Pour représenter un comportement probant, cette attente doit être réalisée avec une grande précision. En effet, ce traitement effectué à chaque ligne peut être bref, par exemple un simple transfert des données vers une interface de communication dédiée à très haut débit. Dans ces scénarios, le délai d'attente à produire est compris entre quelques centaines de nanosecondes et quelques millisecondes. Or, les mécanismes d'attente classique proposés par les systèmes d'exploitation actuels (tels que `usleep(3)`, `nanosleep(2)`, `clock_nanosleep(2)`, `select(2)`) n'offrent pas une précision suffisante. En effet, lors de nos expérimentations préalables, leur temps d'attente effectif minimal est aux alentours de 55 ms. Ce délai élevé s'explique par les mécanismes de signaux et de changement de contexte qui sont utilisés par le noyau. Par

conséquent, nous utilisons une attente active sur la valeur de l'horloge monotonique qui utilise l'horloge interne nanométrique du processeur. Cette attente active nous permet de réaliser des délais d'attente avec un surcoût moyen observé de seulement 50 ns par rapport au temps demandé, soit une précision 1100× plus importante.

Malheureusement, comme toute attente active, elle présente le défaut de fortement solliciter des cycles d'exécution du processeur, ce qui n'est pas représentatif d'un envoi de données par une carte réseau. En effet, celle-ci serait davantage consommatrice de bande passante mémoire que de cycle de calcul. Cependant, au vu de la complexité et de la spécificité de ces échanges mémoires (taille, alignement des données, etc.) spécifiques à chaque carte réseau, nous considérons comme hors d'atteinte de réaliser une simulation probante de ces mécanismes à cette échelle de temps. La simulation des temps de traitement des pilotes d'impression est donc réalisée par une utilisation du processeur, qui est une simulation de qualité permettant de respecter le critère principal : une grande précision dans le temps d'attente. [1]

10.4 Limites et extensions

L'application des délais d'attente, exécutés par les pilotes d'impression entre chaque ligne de données récupérée, pourrait être développée pour approfondir les contraintes qu'ils représentent. En effet, les imprimantes sont généralement dotées d'une mémoire tampon interne de petite capacité permettant notamment de lisser les éventuels aléas des communications. C'est également grâce à cette mémoire tampon qu'elles ne nécessitent pas un pilotage temps réel strict. Par effet de bord, ces mémoires tampon génèrent de légères variations de débit autour du débit moyen que nous utilisons comme référence pour définir les délais d'attente. Ainsi, pour améliorer le mimétisme des conditions réelles, nos délais d'attente pourraient intégrer aléatoirement des variations similaires. Il serait également possible de définir la survenue de ces variations en analysant les exécutions de différents pilotes d'impression, et ainsi de reproduire plus fidèlement l'exécution de pilotes d'impression réels.

10.5 Conclusion

Enfin, les deux simulations de pilotes d'impression que nous avons développées permettent d'évaluer de manière relativement réaliste les performances de notre solution de génération distribuée de flux de données ordonnées sous différents scénarios imposant des contraintes

1. Il serait possible d'améliorer la précision de ces délais d'attente en demandant au noyau d'effectuer un ordonnancement temps réel du processus du pilote d'impression. Cette approche lui fournirait la garantie d'une vérification régulière de l'horloge monotonique. Cependant, dans les systèmes réels, les pilotes d'impression ne sont pas exécutés de cette manière et sont donc soumis à la contrainte du partage d'accès aux cœurs du processeur et peuvent notamment être perturbés par l'exécution concurrente d'un *Producer* local. Ainsi, nous n'intégrons pas d'exécution temps réel de nos simulations de pilote d'impression pour ne pas dévier des conditions réelles de leur exécution.

variées. Ces évaluations sont présentées et commentées dans le chapitre 11 : *Évaluations du RIP distribué*.

11 Évaluations du RIP distribué

Ce second chapitre est dédié aux évaluations, il présente l'analyse des performances de notre solution de chaîne de traitement graphique distribuée. Nous évaluons notre solution sur les différentes topologies possibles et analysons ses performances sur trois aspects complémentaires : les débits maximaux atteignables par notre architecture par rapport aux débits théoriques de la plateforme d'évaluation ; les débits effectifs de notre solution distribuée avec implémentation des traitements de RIP ; et la capacité à produire des débits constants nécessaires pour le pilotage des imprimantes.

11.1 Plateforme d'évaluation

La même plateforme est utilisée pour les différentes évaluations, elle se compose de cinq nœuds identiques équipés d'un processeur Intel Xeon D-1521, de 2× 4 Gio de mémoire vive DDR4 à 2133 MT/s, de deux interfaces réseau 10 GbE, sur une carte mère Supermicro X10SDV-4C-TLN2F. Leur système d'exploitation est Ubuntu 18.04.1 LTS doté du noyau Linux 4.15.0-43-generic. Chacune des interfaces réseau est interconnectée aux autres par un unique switch Netgear XS512EM qui est capable d'alimenter simultanément ses douze interfaces avec le débit maximum de 10 GbE. Le réseau utilise des trames Ethernet avec un *maximum transmission unit* (MTU) par défaut à 1 500 octets.

La présence de deux interfaces réseau sur chaque nœud nous permet de simuler jusqu'à huit nœuds distants en dédiant une interface réseau à chaque nœud simulé. Nous n'avons observé aucune perturbation des performances en utilisant cette méthode de simulation. Elle nous fournit un moyen réaliste d'augmenter artificiellement le nombre de nœuds dans nos expériences en évitant toute intrusion dans la pile réseau du noyau.

11.2 Débits maximaux de l'architecture

Le premier aspect que nous évaluons est la capacité de notre solution à exploiter le potentiel de communication du cluster sur lequel elle s'exécute. Pour ce faire, nous confrontons les vitesses de communication atteintes par notre solution aux vitesses théoriques offertes par notre plateforme d'évaluation. Ainsi, dans cette évaluation, nous désactivons l'exécution des

traitements graphiques du RIP pour ne conserver que l'exécution des échanges de données entre les différents éléments et de cette façon tester le débit maximal de notre architecture.

11.2.1 Protocole d'évaluation

Le protocole utilisé pour mener cette évaluation consiste à exécuter trois travaux d'impression ayant des caractéristiques variées sur les quatre topologies présentées dans le chapitre 6 : *one-to-one*, *one-to-many*, *many-to-one* et *many-to-many*. Pour chacune de ces topologies, le *Scheduler* s'exécute sur le premier nœud de notre cluster et n'en change pas quand on ajoute des nœuds supplémentaires à la topologie.

Les travaux d'impression sont présentés dans le tableau 11.1, ils reflètent respectivement trois cas d'utilisation qui couvrent des volumétries de quelques mégaoctets à plusieurs gigaoctets permettant d'évaluer la performance de notre solution face à la diversité de volumétrie des travaux d'impression. Ce tableau indique pour chaque travail d'impression les caractéristiques de dimension, de résolution, de précision, et de volumétrie lors des calculs et en sortie. Les volumétries sont exprimées pour un seul canal chromatique et sont présentées en octets et en bits afin de quantifier respectivement l'espace mémoire nécessaire dans les nœuds et la bande passante réseau utilisée. Pour chaque travail, trois fichiers d'entrée (pour un total de 10 Mo représentatif d'un travail d'impression classique) sont à récupérer par chaque producteur impliqué dans son exécution.

Tableau 11.1 : Caractéristiques des travaux et volumétrie pour un canal chromatique

	Travail A	**Travail B**	**Travail C**
Cas d'utilisation	Packaging Basse qualité	Poster Qualité standard	Affiche Haute qualité
Dimension (in.)	10 × 10	40 × 40	100 × 100
Résolution (dpi)	300 × 300	900 × 900	1 200 × 1 200
Précision des calculs	1 octet / canal	1 octet / canal	1 octet / canal
Précision des gouttes	1 taille (1 bit)	3 tailles (2 bits)	10 tailles (4 bits)
Largeur calculs (octets)	3 000	36 000	120 000
Hauteur calculs (lignes)	3 000	36 000	200 000
Volumétrie calculs / canal	**9,0 Mo**	**1,3 Go**	**14,4 Go**
Largeur sortie (bits)	3 000	72 000	480 000
Hauteur sortie (lignes)	3 000	36 000	200 000
Volumétrie sortie / canal	**9,0 Mbit**	**2,6 Gbit**	**57,6 Gbit**

Comme expliqué au fil des chapitres précédents, notre solution dispose de nombreux points de personnalisation. Ainsi, nous définissons dans l'ordonnanceur la taille maximale des portions générées à 24 Mio. Cette valeur a été déterminée dans nos expérimentations comme offrant

la granularité optimale pour notre plateforme [1]. Un seul producteur est exécuté par nœud, il dispose de l'ensemble des ressources du nœud pour exécuter les tâches qui lui sont assignées. De même, un seul consommateur est exécuté par nœud, et la taille des mémoires tampon suit les règles établies par l'équation **8.1** avec une taille maximale de 1 Gio.

Pour supprimer les interférences issues d'un mélange de localités hôtes et racks entre les producteurs et les consommateurs, nous réalisons les mesures des topologies *one-to-many* et *many-to-one* avec une relation exclusivement distante entre, respectivement, le producteur avec les consommateurs et les producteurs avec le consommateur. Pour la topologie *many-to-many*, nous exécutons sur chaque nœud un producteur et un consommateur afin de conserver un ratio stable de communications hôtes et racks entre les nœuds. De plus, pour maintenir le respect de cette règle, nous n'utilisons pas de nœud simulé dans cette topologie.

Les topologies *one-to-one*, *one-to-many* et *many-to-one* exécutent les travaux d'impression du tableau **11.1** avec quatre canaux chromatiques. Dans la topologie *many-to-many*, les travaux d'impression sont configurés avec autant de canaux chromatiques que de nœuds présents dans le scénario. Ceci permet de conserver une charge de travail et de communication équilibrée entre les nœuds puisque chaque consommateur gère un canal chromatique.

11.2.2 Limites théoriques

Chacune des quatre topologies présente une limite théorique spécifique à la nature et l'organisation des communications entre le ou les producteurs et le ou les consommateurs. Ces limites théoriques peuvent être définies par la bande passante de la mémoire, du réseau, ou bien un mélange des deux.

La topologie *one-to-one* existe en deux configurations impliquant deux limites théoriques différentes. La première configuration place le producteur et le consommateur sur le même nœud, offrant ainsi une localité hôte ne faisant pas intervenir le réseau pour l'échange des données. Ainsi, la limite théorique correspond à la bande passante de la mémoire vive. Nos nœuds exploitent une mémoire vive *dual-channel* dont les caractéristiques techniques lui associent un débit maximal de 34 Gio/s. Le transfert entre le producteur et le consommateur nécessite une copie des données de l'espace mémoire du producteur vers celui de la mémoire tampon du consommateur. Ainsi, la limite théorique vaut la moitié de la bande passante de la mémoire vive : 17 Gio/s, que nous convertissons en bits afin d'unifier les valeurs des limites théoriques avec celle des autres topologies utilisant le réseau : 136 Gbit/s. La seconde configuration repose sur une localité rack avec le producteur et le consommateur placés chacun sur un nœud différent. Ainsi, la limite théorique est celle du lien réseau les reliant, qui est de

1. Cette valeur est influencée par les différentes tailles de cache des systèmes et la configuration du réseau. Toutefois, une valeur de 24 Mio est apte à couvrir les configurations types des clusters ciblés par le domaine de l'impression numérique qui sont composés de machines aux configurations matérielles standards. Cependant, plus les caches sont grands et le réseau rapide, plus la taille des portions devrait être grande.

10 GbE sur notre plateforme. Cependant, comme nous utilisons les protocoles TCP/IP sur une liaison Ethernet avec un MTU de 1 500 octets, la bande passante maximale disponible pour nos données est de 9,466 Gbit/s.

Nous considérons les topologies *one-to-many* et *many-to-one* comme ayant une relation distante entre l'unique producteur ou l'unique consommateur et les autres nœuds consommateurs ou producteurs. De ce fait, ces topologies sont toutes deux limitées par le lien réseau partant de, ou allant vers, l'élément unique. Ainsi, comme pour la topologie *one-to-one* distante, la limite théorique est de 9,466 Gbit/s.

La limite théorique de la topologie *many-to-many* est plus complexe à définir à cause du mélange de communications de localité hôte et rack entre les producteurs et les consommateurs. Cependant, notre protocole d'évaluation définit une quantité identique de données circulant vers chaque consommateur et la présence sur chaque nœud d'un seul producteur et d'un seul consommateur. Ainsi, la limite théorique peut être calculée par l'équation **11.2** qui prend uniquement en compte la quantité de données échangées sur le réseau, c'est-à-dire en excluant la communication sur la localité hôte puisque les communications internes sont recouvertes par les communications distantes qui sont plus lentes, multiplié par le nombre de liens, c'est-à-dire le nombre de nœuds et leur capacité. Toutefois, la validité de cette équation est restreinte à la présence de bandes passantes homogènes entre les nœuds, comme c'est le cas sur notre plateforme.

Soient :

— C nombre de consommateurs dans la topologie ;

— H nombre de nœuds dans la topologie [2] ;

— R bande passante d'un lien réseau ;

alors :

$$\text{Limite théorique } \textit{many-to-many} = \frac{C}{C-1} \times H \times R \tag{11.2}$$

11.2.3 Analyse des résultats

La figure **11.3** présente les résultats de cette évaluation où nous désactivons les traitements graphiques pour isoler la capacité de communication de notre solution. Les débits obtenus pour chaque type de travail d'impression sont présentés face aux limites théoriques (barre la plus à droite de chaque groupe) de la topologie. Pour les topologies *one-to-one*, *many-to-one* et *many-to-many*, le débit de chaque travail d'impression est obtenu en divisant la volumétrie en sortie par le temps d'exécution du travail d'impression qui s'étend de la soumission du

2. Dans notre scénario, nous avons $C = H$ car chaque nœud possède un unique producteur et un unique consommateur.

travail jusqu'à sa terminaison. Les mesures sont indiquées avec la déviation standard du temps d'exécution de chaque travail. De par les conditions particulières de la topologie *one-to-many*, les débits y sont calculés en divisant la volumétrie totale de tous les travaux d'impression par le temps total de l'exécution du scénario, excluant ainsi la mesure de l'écart type du temps d'exécution des travaux.

Tel qu'expliqué dans la section 11.1, nous tirons bénéfice des deux interfaces 10 GbE des nœuds pour simuler jusqu'à deux producteurs et deux consommateurs par nœud. Les scénarios utilisant des nœuds simulés sont indiqués par la présence d'un astérisque ∗ à côté du nombre de nœuds dans les figures **11.3 (b)** et **11.3 (c)**.

Les figures **11.3 (a)**, **11.3 (b)** and **11.3 (c)** montrent les capacités de notre architecture à atteindre les limites théoriques de notre plateforme d'évaluation pour les travaux B et C dans les topologies *one-to-one* distante, *one-to-many* et *many-to-one*. Plus le nombre de producteurs et de consommateurs est élevé, meilleurs sont les résultats. Pour le travail A, les faibles performances sont dues à la faible volumétrie en sortie du travail. En effet, sa faible volumétrie ne nous permet pas de compenser le surcoût de sa soumission au système. Cependant, dans la figure **11.3 (b)**, ce surcoût disparaît grâce aux recouvrements des soumissions par l'exécution des travaux concurrents, permettant ainsi aux trois types de travail d'impression d'atteindre le débit théorique dès l'utilisation de 4 consommateurs. Pour l'ensemble des travaux et des topologies, la valeur des écarts types est relativement faible. De plus, celui-ci devient négligeable plus les travaux sont long à exécuter, ce qui démontre une stabilité des performances de notre solution dans le temps.

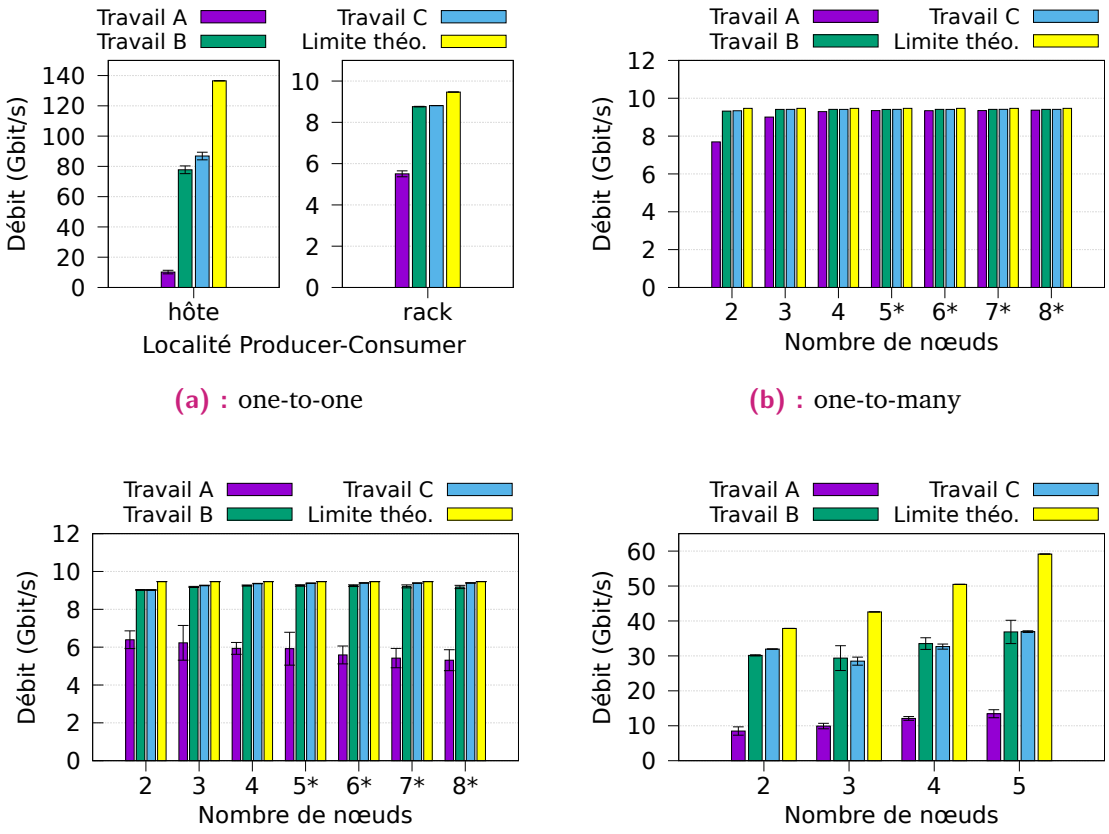
Dans la figure **11.3 (a)**, la topologie exploitant la localité hôte reste en retrait du débit théorique offert par la mémoire vive. Ce résultat s'explique par la présence de nombreuses opérations mémoire concurrentes réalisées par les autres programmes s'exécutant sur le nœud. Néanmoins, nous atteignons un débit satisfaisant de 90 Gbit/s au vu de la complexité du système pilotant l'exécution de ces communications hôtes. Enfin, seule la topologie *many-to-many* présentée dans la figure **11.3 (d)** n'atteint pas les limites théoriques de la plateforme d'évaluation. Nos premières analyses de ce résultat laissent envisager l'apparition régulière de courtes congestions réseau. En effet, chaque pilote des cartes réseau gère davantage de connexions concurrentes, ce qui augmente le risque de collisions de paquets sur les différentes interfaces réseau du switch. L'ajout d'une meilleure supervision des tâches *Expel* concurrentes par l'ordonnanceur devrait améliorer ces résultats.

11.2.4 Conclusion

Ces différentes mesures montrent la grande flexibilité de notre solution qui se révèle fonctionnelle sur une variété de topologies tout en offrant de bonnes performances de communication pour une large gamme de volumétrie de travaux. Les débits de sortie obtenus permettent à

de nombreuses applications de génération de flux d'exploiter notre solution pour distribuer efficacement leurs traitements. Les résultats de cette évaluation nous amènent à attendre de notre solution qu'elle fournisse des débits supérieurs, et des évolutions de débits similaires en fonction du nombre de nœuds dans le système, sur un réseau avec des liens de plus grande capacité, telle que le 20, 25, 40 et 50 GbE.

Concernant notre cas d'application de l'impression numérique professionnelle, nous pouvons observer avec cette première évaluation que notre architecture répond aux objectifs de l'environnement que nous avons présenté dans le chapitre 3 : *Contraintes et objectifs*, à la fois pour les fermes d'impression et pour les presses numériques. En effet, dans la topologie *one-to-many* correspondante aux fermes d'impression, quelle que soit la volumétrie des travaux, plus il y a de consommateurs (c'est-à-dire plus il y a de pilotes d'impression) dans le système, plus notre solution maximise l'utilisation du réseau. De même, dans la topologie *many-to-many* exécutant des travaux de volumétrie moyenne et très grande, correspondant aux cas d'usage des presses numériques, notre solution fournit un débit supérieur à celui des presses numériques actuelles présentées dans le tableau 3.4 avec un cluster composé de seulement 5 nœuds interconnectés en 10 GbE.



(a) : one-to-one

(b) : one-to-many

(c) : many-to-one

(d) : many-to-many

Figure 11.3 : Mesures des débits de trois travaux différents exécutés sur quatre topologies différentes par rapport aux débits maximaux (limites théoriques) de la plateforme d'évaluation

11.3 Passage à l'échelle des traitements graphiques

Après l'évaluation des performances maximales de notre solution de génération de flux distribués, nous continuons son évaluation en mesurant ses performances avec l'ajout des traitements graphiques lors de la génération des données. Cette seconde évaluation permet de mesurer les performances effectives de notre solution lors de l'exécution des traitements graphiques d'un RIP et ainsi d'apprécier sa capacité de passage à l'échelle.

11.3.1 Protocole d'évaluation

Pour réaliser cette évaluation, nous reprenons le même protocole d'évaluation que celui de l'expérience précédente en section 11.2 : *Débits maximaux de l'architecture*. De cette manière, notre évaluation couvre également les différentes topologies et la variété des volumétries de travaux. De plus, l'utilisation de la même plateforme d'évaluation nous permet de comparer les performances maximales obtenues aux performances effectives de cette évaluation.

La différence entre les deux évaluations se situe au sein des producteurs. Ceux-ci sont désormais chargés d'exécuter l'ensemble des traitements graphiques sur les portions de données assignés avant de réaliser leur transmission aux consommateurs. Les traitements effectués sont ceux présentés dans la section 7.2 : *Production des données*. Les traitements faisant un usage total des unités de calcul des producteurs, nous n'utilisons pas de nœuds simulés puisque ceux-ci ne seraient pas en mesure d'exécuter indépendamment les traitements graphiques.

11.3.2 Limites théoriques

Les limites théoriques de cette évaluation sont doubles : premièrement, les débits établis lors de l'évaluation précédente forment les débits maximaux de notre solution sur cette plateforme ; secondement, l'ajout des traitements graphiques implique les limites des capacités des unités de calcul. Dans notre cas, la charge de calcul des traitements graphiques d'un RIP est telle que la limite de calcul devrait s'exprimer en premier jusqu'à ce que l'accélération permise par la distribution des calculs permette d'atteindre la limite des débits maximaux de cette plateforme.

11.3.3 Analyse des résultats

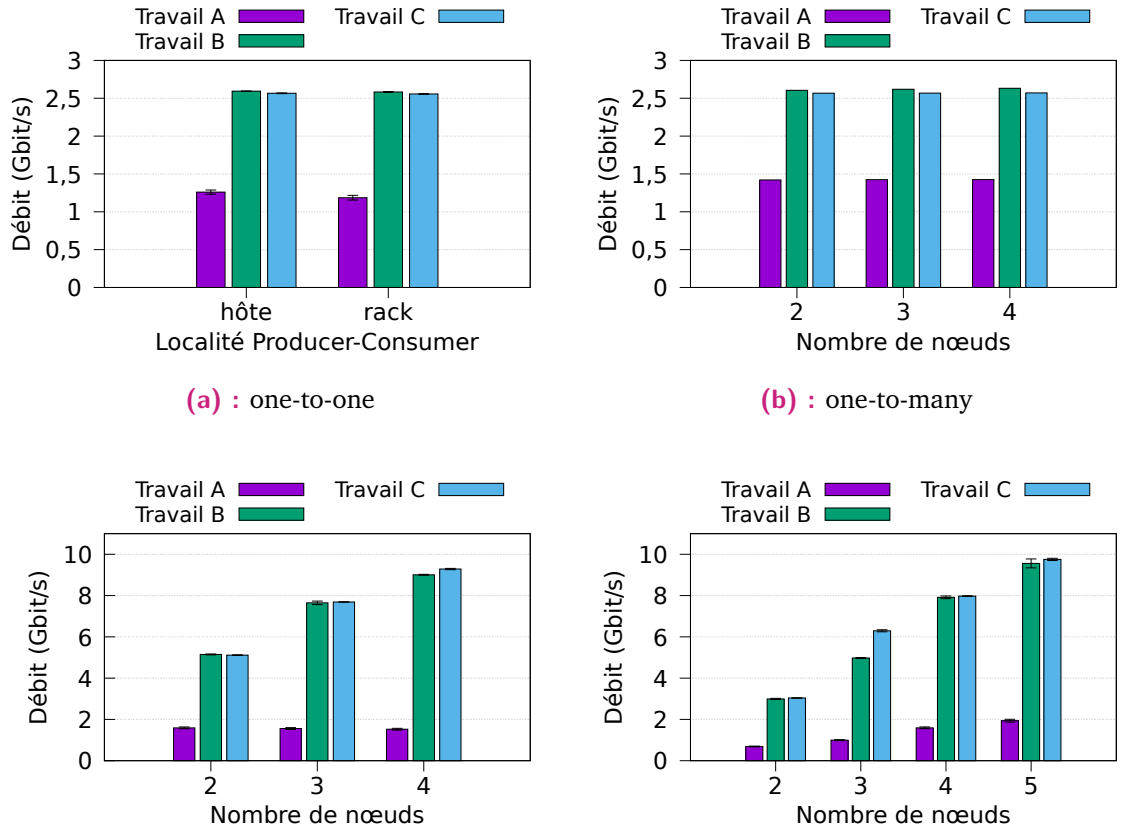
La figure 11.4 présente les résultats de cette évaluation qui exécute les traitements d'un RIP sur notre solution distribuée afin de mesurer sa capacité à passer à l'échelle. Les débits de sortie sont calculés de la même façon que pour l'évaluation précédente et les écarts types sont également affichés pour les topologies *one-to-one*, *many-to-one* et *many-to-many*.

Les résultats présentés dans la figure 11.4 affichent une excellente stabilité et un passage à l'échelle optimal sur l'ensemble des topologies. Les écarts types sont également négligeables sur l'ensemble des travaux, quelle que soit la topologie, démontrant ainsi la stabilité du système même lors de l'exécution d'une charge de calcul élevée.

La figure 11.4 (a) montre des résultats identiques, quelle que soit la localité du producteur et du consommateur. Ainsi, notre solution permet de distribuer leurs emplacements sur le cluster (et donc celui des pilotes d'impression) sans perte de performance si le réseau est dimensionné pour supporter le débit généré. De la même manière, la figure 11.4 (b) illustre la possibilité d'alimenter plusieurs consommateurs depuis un même producteur sans générer de perte de performances. Inversement, la figure 11.4 (c) présente une agrégation optimale de la capacité de calcul de plusieurs producteurs desservant un unique consommateur. Les débits obtenus avec 4 producteurs flirtent avec la limite maximale du lien réseau rencontré dans la figure 11.3 (c) de l'évaluation précédente. Seul le travail A ne bénéficie pas de ce gain à cause de sa trop faible volumétrie qui provoque la génération de la totalité de ses données en une seule tâche *Compute* exécutée sur un seul producteur. Enfin, les résultats de la topologie *many-to-many* présentée dans la figure 11.4 (d) démontrent un passage à l'échelle idéal sur tous les travaux sans incidence des limitations observées pour la même topologie de l'évaluation précédente (figure 11.3 (d)).

11.3.4 Conclusion

Les résultats obtenus dans cette seconde évaluation démontrent l'efficacité de notre solution pour la distribution de l'exécution des traitements graphiques d'un RIP quelle que soit la topologie pour l'ensemble des travaux pouvant être distribués. De plus, nous atteignons dans les topologies *many-to-one* et *many-to-many* les débits maximaux de notre plateforme d'évaluation, la première limite étant celle du lien réseau, et la seconde celle de la somme des capacités de calcul. Par conséquent, nous sommes confiants dans les capacités de notre solution à proposer des performances supérieures avec un réseau et des unités de calcul plus performantes.



(a) : one-to-one

(b) : one-to-many

(c) : many-to-one

(d) : many-to-many

Figure 11.4 : Mesures des débits de trois travaux différents exécutés sur quatre topologies différentes avec exécution des traitements graphiques d'un RIP

11.4 Qualité de service des traitements graphiques

Au-delà des besoins de flexibilité et de performances brutes, notre solution doit aussi assurer la constance des flux. Nous poursuivons son évaluation en abordant cette dernière contrainte de constance dans les débits des flux fournis aux pilotes d'impression afin qu'ils puissent transmettre continuellement les données suivantes à l'imprimante.

11.4.1 Protocole d'évaluation

Pour évaluer la capacité de notre solution à produire des flux de données avec un débit constant, nous utilisons les mécanismes de simulation de pilote d'impression détaillés dans la section 10 : *Simulation d'un pilote d'impression*. À chaque fois que notre pilote d'impression accède à une ligne du flux de données du travail, il marque un temps d'arrêt correspondant à l'équation **10.1**. Ainsi, ce temps d'attente rythme la lecture des données par le pilote d'impression. Par conséquent, si les données suivantes ne sont pas disponibles à l'issue de l'attente, alors nous observerons des temps d'exécution des travaux supérieurs aux temps relevés dans l'expérience précédente (cf. section 11.3) et donc un débit d'exécution du travail plus faible.

Pour réaliser cette évaluation, nous utilisons le travail B qui présente une volumétrie moyenne et affiche les écarts types les plus importants dans les évaluations précédentes. Le travail B est exécuté avec un temps d'attente constant pour chaque exécution. Différents temps d'attente sont utilisés, chacun correspondant à un pourcentage du débit de l'expérience précédente (section 11.3). Le pourcentage de ce débit est appelé « débit réclamé » et est comparé au « débit obtenu » lors de l'exécution.

11.4.2 Limites théoriques

Une génération fournissant une constance idéale des flux correspond à un débit obtenu identique au débit réclamé. Par construction, le débit obtenu ne peut pas être meilleur que le débit réclamé. Pour modéliser cette limite théorique, nous utilisons un modèle *roofline*.

11.4.3 Analyse des résultats

Les résultats de cette troisième évaluation, présentés dans la figure **11.5**, montrent le débit de sortie obtenu en fonction du débit réclamé. Ces débits sont indiqués en pourcentage des débits de l'expérience précédente pour les mêmes topologies et mis en comparaison avec la *roofline* formant la limite théorique.

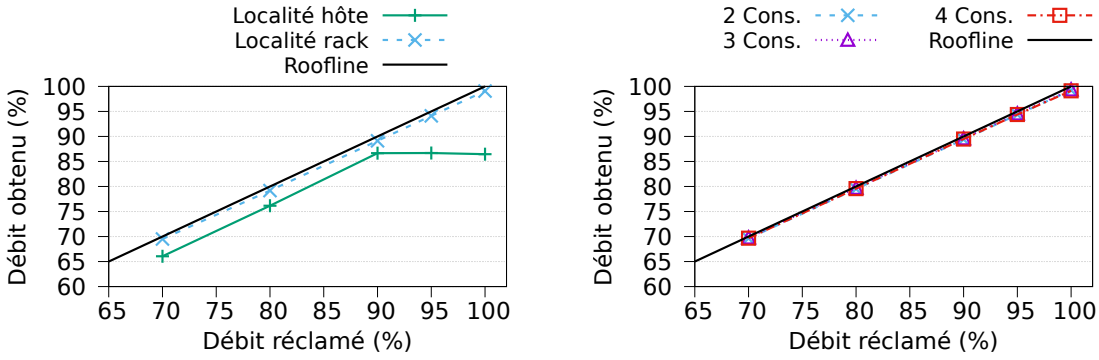
Les topologies *one-to-one* et *many-to-many* affichent dans leurs figures, respectivement **11.5 (a)** et **11.5 (d)**, une stagnation du débit obtenu à partir de 90 % du débit réclamé. Cette stagnation est un effet de bord issu de la méthode utilisée pour exécuter un temps d'attente avec une précision suffisante, dont les raisons sont expliquées en détail dans la section 10 : *Simulation d'un pilote d'impression*. En effet, l'attente repose sur un mécanisme d'attente active qui sollicite fortement un cœur du processeur et vient donc priver le producteur présent sur le même nœud que le driver d'impression d'une partie des capacités de calcul dont il disposait dans l'évaluation précédente. Cette affirmation se démontre en comparant les résultats obtenus dans la figure **11.5 (a)** où la topologie *one-to-one* donne une constance optimale lorsque le producteur et le pilote d'impression sont exécutés sur deux nœuds distants.

En excluant les portions des courbes présentant cet artefact de mesure, nous pouvons constater une évolution linéaire du débit obtenu en fonction du débit réclamé. Les évolutions linéaires présentes pour toutes les topologies démontrent la capacité de notre solution à fournir des flux de données à un débit parfaitement constant tout au long de l'exécution du travail B. Toutes topologies confondues, la différence entre le débit réclamé et le débit obtenu est de 3,2 % avec un maximum à 13,5 % dû à l'artefact de mesure dans la topologie *one-to-one*. De même, l'écart type moyen s'établit à 0,5 %, avec un maximum de seulement 2,4 % pour 100 % de débit réclamé dans la topologie *many-to-many*.

La topologie *one-to-many*, figure **11.5 (b)**, présente une constance idéale quel que soit le nombre de consommateurs. Ce résultat s'explique par l'absence de perte de paquet sur le réseau puisque la congestion est directement régulée par la carte réseau du producteur qui n'émet les paquets que lorsque le lien réseau est inoccupé.

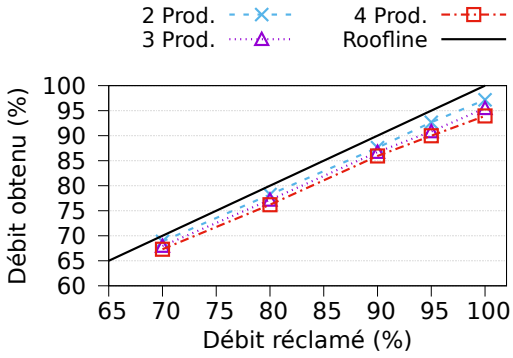
11.4.4 Conclusion

Pour conclure cette dernière évaluation, nous observons que notre solution est apte à produire des flux de données constants à destination des pilotes d'impression indépendamment de la topologie reliant les producteurs aux consommateurs. De plus, quel que soit le débit réclamé, celui-ci peut être obtenu avec confiance par une surallocation des capacités des producteurs de seulement quelques pourcents.

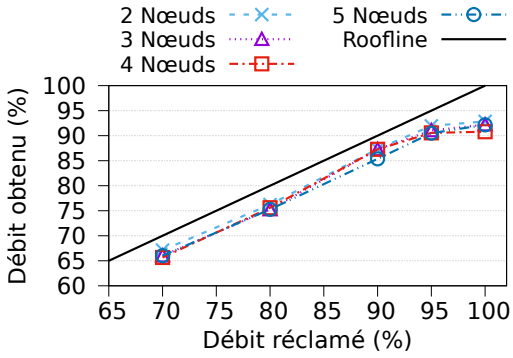


(a) : one-to-one

(b) : one-to-many



(c) : many-to-one



(d) : many-to-many

Figure 11.5 : Mesures des débits (débit obtenu) du travail B exécuté sur quatre topologies différentes en fonction d'une lecture à vitesse constante des données produites (débit réclamé). Les débits sont exprimés en pourcentage des débits obtenus pour la configuration correspondante dans la figure 11.4

Cinquième partie

Conclusion

12 Conclusion

Dans cette thèse, nous avons exploré le domaine de l'impression numérique qui fait face à de nombreuses transformations de ses usages et à des évolutions technologiques atteignant les limites des solutions de traitement graphique existantes. Notre analyse des spécificités de ces traitements graphiques fait apparaître une complexité qui repose sur un éventail de caractéristiques et de contraintes en lien avec la nature des périphériques d'impression et leur diversité, tels que la volumétrie des travaux d'impression, la variété des débits nécessaires (de quelques Mio/s à plusieurs Gio/s), la production constante des données ainsi que leurs emplacements de sortie imposés. Si les contraintes mises en cause sont individuellement connues de la littérature scientifique, leur association implique de nouveaux défis tant scientifiques que techniques.

Ces contraintes s'appliquent à deux cas d'utilisation qui s'opposent l'un à l'autre dans leurs besoins. D'un côté, les nouveaux usages issus de l'utilisation de fermes d'impression, réunissant plusieurs dizaines d'imprimantes à faible cadence pour produire davantage à moindre coût, posent un défi de mutualisation des ressources de calcul afin d'optimiser la quantité de ressources informatiques nécessaires pour piloter simultanément ces imprimantes. De l'autre côté, l'apparition des presses numériques, capables d'atteindre des cadences de production proches de celles des presses traditionnelles, requiert la mise en commun des ressources informatiques afin de satisfaire les besoins de ces imprimantes en débit de données.

Le pilotage d'une imprimante numérique, quelle que soit sa taille, repose sur la création d'un flux de données ordonnées dont la transmission doit se faire de manière continue et contrôlée. Durant les dernières décennies, les solutions de traitement graphique se sont concentrées sur la production et la transmission effective de ces flux de données et se sont ancrées dans des architectures monolithiques qui sont incompatibles avec une distribution efficace des traitements.

Nous proposons à travers nos travaux une approche innovante permettant de fournir au domaine de l'impression numérique : de nouvelles méthodes pour améliorer les performances par l'intermédiaire de l'approche moderne de la parallélisation et des mécanismes offerts par les systèmes d'exploitation ; la flexibilité nécessaire pour aborder sereinement les récentes et les prochaines évolutions du domaine de l'impression numérique qui requièrent la création de topologies dynamiques entre les nœuds d'un cluster afin d'acheminer les flux de données au plus proche des pilotes d'impression indépendamment du ou des nœuds qui les ont produites ; et la distribution efficace des traitements graphiques sur les nœuds afin de proposer un passage

à l'échelle durable capable de repousser les limitations en débit actuelles et d'unifier au sein d'une même solution les besoins de mutualisation des ressources informatiques et de distribution des traitements.

12.1 Contributions

Dans la partie II : *Prétraitements graphiques*, nous avons établi une nouvelle approche pour exécuter les rotations et transpositions de matrices rectangulaires présentant les caractéristiques combinées d'*out-of-core* et d'*out-of-place*. Notre approche basée sur une parallélisation des traitements associée à une organisation des accès aux données coordonnée avec le système d'exploitation assure des transformations performantes sur tous les types de mémoire de masse actuels : mécaniques à plateaux ou bien à mémoire flash. Nos travaux permettent une accélération allant jusqu'à un facteur 10× par rapport aux précédents travaux de l'entreprise Caldera sur cette problématique qui représentait alors un goulot d'étranglement dans la chaîne de traitement des grandes images. Sur la nouvelle génération de mémoire de masse utilisant de la mémoire flash (SSD), nos performances sont proches des valeurs optimales et permettent ainsi de réaliser de façon transparente les transformations de rotation et de transposition lors d'une copie de l'image.

Dans la partie III : *dRIP : Distributed Raster Image Processor*, nous avons présenté notre architecture de calcul distribué permettant de générer des flux de données respectant les contraintes de l'impression numérique. Cette architecture flexible permet l'acheminement des flux de données jusqu'aux pilotes d'impression, indépendamment du nœud, ou des nœuds du cluster les ayant produits. Sa flexibilité permet également la création dynamique et l'utilisation concurrente de topologies adaptées à chaque travail d'impression. Notre architecture exploite un ordonnanceur centralisé capable d'organiser dynamiquement la production des données en accord avec leur vitesse de transmission aux pilotes d'impression tout en offrant des protections ajustables au risque de famine. Nous avons également présenté l'implémentation de cette architecture qui se démarque par une gestion fine des communications et laisse une grande capacité de personnalisation dans l'ordonnanceur et les producteurs. De même, un travail important a été consacré à concevoir une transmission efficace des données entre notre système et les pilotes d'impression. Les différentes évaluations de notre système distribué ont permis de démontrer ses capacités sous trois aspects essentiels dans toutes les topologies : des débits de communication proches des débits théoriques maximaux de notre plateforme d'évaluation ; un passage à l'échelle permettant une mutualisation et une agrégation des ressources importante ; et une capacité à produire des flux de données constants qui est essentielle au bon fonctionnement des pilotes d'impression.

Les résultats obtenus sur notre plateforme d'évaluation sont satisfaisants par rapport aux besoins actuels et à venir des fermes d'impression et des presses numériques, alors que celle-ci

est seulement représentative des clusters de taille moyenne pour l'impression professionnelle. Bien que la puissance de calcul de nos nœuds soit limitante, la très bonne capacité de passage à l'échelle qui a été obtenue permet d'apercevoir le potentiel de notre solution avec des nœuds plus puissants en analysant les débits maximaux atteints par notre architecture. D'un côté, nos évaluations ont montré qu'un seul nœud producteur de données est capable de saturer aisément un lien 10 GbE en desservant simultanément plusieurs pilotes d'impression. Ce débit de 10 GbE correspond au pilotage simultané de 14 imprimantes requérant 79 Mio/s chacune (débit typique de celles utilisées dans les fermes d'impression actuelles). De l'autre côté, nos évaluations sur une topologie de nœuds propre à celle des presses numériques nous permet d'atteindre un débit en sortie de 4,42 Gio/s avec seulement 5 nœuds interconnectés en 10 GbE. Ce débit permet d'alimenter les presses numériques existantes les plus importantes à notre connaissance (3,6 Gio/s) avec une marge confortable de 1,15 Gio/s. Dans ce scénario également, les résultats de passage à l'échelle montrent que le débit maximal en sortie augmente progressivement avec le nombre de nœuds, ce qui laisse entrevoir le pilotage de presses numériques plus gourmandes par l'ajout de nœuds supplémentaires tout en restant dans des clusters de petite à moyenne taille, acceptables pour cette industrie.

Enfin, bien que les différentes projections indiquent que les imprimantes et les presses numériques vont voir leur débit augmenter progressivement dans la prochaine décennie, il en est de même pour celui des équipements informatiques de calcul et de communication. Nos travaux axés sur la distribution des traitements sont à même de bénéficier de ces évolutions technologiques, ce qui donne l'assurance de supporter les prochaines évolutions de débit de données du domaine de l'impression numérique.

12.2 Travaux futurs et perspectives

À court terme, la suite immédiate de nos travaux de recherche consiste à éprouver notre architecture distribuée dans un environnement réel (et non plus seulement réaliste) de production avec le pilotage effectif de fermes d'impression et de presses numériques en bout de chaîne sur des clusters aux configurations matérielles variées. Parallèlement, le prolongement de cette thèse consiste à réaliser l'intégration complète de nos travaux au sein des logiciels de l'entreprise Caldera. Ainsi, une première version de notre nouvel algorithme de transformation de matrice *out-of-core* a d'ores et déjà été implémenté dans le logiciel *CalderaRIP V12.1* [Cal19] pour l'exécution des rotations. Il s'agit désormais de l'appliquer également aux transpositions et d'intégrer l'utilisation des SSD pour accélérer sensiblement les rotations sur les HDD encore très utilisés par les clients de l'entreprise. Il en est de même pour notre solution de traitement graphique distribuée qui devrait être intégrée progressivement aux produits de l'entreprise Caldera au travers de la prochaine refonte de son moteur d'exécution des traitements graphiques.

Dans cette thèse, nous avons présenté les possibilités d'optimisation de certaines étapes et la faisabilité de la distribution efficace des traitements graphiques, mais nous sommes convaincus que nos travaux ne constituent qu'une première phase.

Dans chaque chapitre nous avons analysé les perspectives de nos propres travaux et nous y proposons plusieurs améliorations pouvant être apportées à notre solution pour diversifier ses capacités, augmenter ses performances et améliorer sa tolérance aux pannes :

— Premièrement, la possibilité d'empiler et de chaîner virtuellement les producteurs entre eux serait une approche originale pour permettre l'exécution de tâches formant un graphe acyclique orienté contenant des dépendances au sein des tâches de production des données tout en conservant la couche unique d'agrégation et de mise en ordre des données fournie par les consommateurs ;

— Deuxièmement, de nombreuses imprimantes permettent la réception des données à un format compressé, afin d'optimiser la bande passante utilisée, souvent commun à celui utilisé par d'autres modèles ou d'autres constructeurs. Par conséquent, il serait intéressant de réaliser ces traitements de compression au sein des producteurs et d'ajouter au consommateur les mécanismes de gestion de taille de portions de données variables ;

— Troisièmement, la mise en place automatique d'un mécanisme d'un ou de plusieurs ordonnanceurs de secours capables de prendre le relais en cas de défaillance de celui-ci serait un pas important pour assurer la continuité de service, et donc de production, en cas de défaillance de ce nœud ;

— Quatrièmement, automatiser la découverte des nœuds et mettre en place la mesure dynamique de leur capacité de production de données formerait une amélioration bienvenue dans la configuration de notre solution et lui assurerait l'exactitude des valeurs communiquées, en lieu et place de celles actuellement renseignées par l'utilisateur.

Nos travaux soulignent également les nombreuses améliorations possibles des chaînes de traitement graphique actuellement utilisées dans l'impression numérique professionnelle. Ainsi, un axe d'innovation complémentaire se situe au sein même de l'exécution des différents traitements graphiques et du *pipeline* graphique qui en découle. Améliorer par exemple la parallélisation des traitements ainsi que la localité des données dans les caches des processeurs permettrait de démultiplier la puissance de calcul de chaque nœud. Pour cela, les outils d'optimisation et de parallélisation automatique de code, tels que le modèle polyédrique et les DSL, fournissent des solutions efficaces pouvant s'intégrer parfaitement à l'intérieur des éléments producteurs de notre architecture. Ces optimisations permettront, sans remplacer les avantages de la distribution des travaux, d'une part d'augmenter le nombre de pilotes d'impression pilotés par nœud et d'autre part de réduire le nombre de nœuds nécessaires pour atteindre les débits nécessaires au pilotage des presses numériques.

À plus long terme, les algorithmes utilisés par l'impression numérique sont susceptibles d'évoluer pour améliorer la qualité des impressions. Ces évolutions sont à même de modifier les dépendances actuelles entre les pixels des images traités et entre les canaux chromatiques. Pour

supporter ces nouveaux algorithmes, notre solution devra intégrer l'échange de données entre les producteurs sans trahir sa capacité à générer parallèlement des flux de données constants pour des travaux concurrents. De plus, il est probable de voir émerger de nouvelles technologies, matérielles et logicielles, qui sauront influencer les algorithmes de traitement graphique en proposant de nouvelles approches, telle que l'intelligence artificielle pour améliorer le rendu des étapes de mise à l'échelle et de correction colorimétrique. Leur intégration efficace dans les logiciels de traitement graphique présage l'exploration de nouvelles problématiques à travers des travaux de recherche passionnants !

Enfin, nos travaux conservent une approche suffisamment générique pour être appliqués à d'autres domaines nécessitant également de générer des flux de données contraints. Ce besoin, qui s'annonce grandissant avec l'émergence de l'industrie 4.0, permet l'extension de nos travaux à de nouveaux domaines industriels ou scientifiques qui s'avèrent être également une source de problématiques et de défis stimulants.

Bibliographie personnelle

En soumission

P. GODARD, V. LOECHNER et C. BASTOUL. « Efficient Out-of-core and Out-of-place Rectangular Matrix Transposition and Rotation ». En cours de soumission (IEEE Transactions on Computers)

Conférence internationale

P. GODARD, V. LOECHNER, C. BASTOUL, F. SOULIER et G. MULLER. « A Flexible and Distributed Runtime System for High-Throughput Constrained Data Streams Generation ». Dans : *20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) in conjunction with IPDPS*. Rio de Janeiro, Brazil, mai 2019

Conférence nationale

P. GODARD. « Échanges non bloquants de données ordonnées entre producteurs multiples et consommateur unique ». Dans : *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*. Bayonne, France, juin 2019

Posters

P. GODARD, V. LOECHNER, C. BASTOUL et F. SOULIER. « Distributed Raster Image Processing ». Dans : *14th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. Fiuggi, Italie, juil. 2018

P. GODARD, V. LOECHNER, C. BASTOUL et F. SOULIER. « Distributed Raster Image Processing ». Dans : *Journée Poster de l'École Doctorale de Mathématiques, Sciences de l'Information et de l'Ingénieur (MSII)*. Strasbourg, France, oct. 2017

Bibliographie

[Aba+16] Martin ABADI, Paul BARHAM, Jianmin CHEN et al. « Tensorflow : A system for large-scale machine learning ». Dans : *12th Symposium on Operating Systems Design and Implementation (USENIX)*. 2016, p. 265–283 (cf. p. 61).

[ABB00] Umut A ACAR, Guy E BLELLOCH et Robert D BLUMOFE. « The data locality of work stealing ». Dans : *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM. 2000, p. 1–12 (cf. p. 62).

[Ado19] ADOBE. *Adobe PDF Print Engine*. `https://www.adobe.com/products/pdfprintengine.html` Accédée le : 22/08/2019. 2019 (cf. p. 15).

[All+01] Gabrielle ALLEN, David ANGULO, Ian FOSTER et al. « The Cactus Worm : Experiments with dynamic resource discovery and allocation in a grid environment ». Dans : *The International Journal of High Performance Computing Applications* 15.4 (2001), p. 345–358 (cf. p. 99).

[AM97] William J ALLEN et Steven O MILLER. *Ink limiting in ink jet printing systems*. US Patent 5,633,662. Mai 1997 (cf. p. 16).

[Amd67] Gene M AMDAHL. « Validity of the single processor approach to achieving large scale computing capabilities ». Dans : *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, p. 483–485 (cf. p. 83).

[Ant+16] Samuel F ANTAO, Alexey BATAEV, Arpith C JACOB et al. « Offloading support for OpenMP in Clang and LLVM ». Dans : *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*. IEEE Press. 2016, p. 1–11 (cf. p. 57).

[Aok+11] Ryo AOKI, Shuichi OIKAWA, Takashi NAKAMURA et Satoshi MIKI. « Hybrid opencl : Enhancing opencl for distributed processing ». Dans : *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2011, p. 149–154 (cf. p. 57).

[ApApex] THE APACHE SOFTWARE FOUNDATION. *Apache Apex*. `https://apex.apache.org/` Accédée le : 13/07/2019 (cf. p. 61).

[ApFlink] THE APACHE SOFTWARE FOUNDATION. *Apache Flink*. `https://flink.apache.org/` Accédée le : 10/01/2019 (cf. p. 61).

[ApHado] THE APACHE SOFTWARE FOUNDATION. *Apache Hadoop*. `https://hadoop.apache.org/` Accédée le : 17/01/2019. 2019 (cf. p. 60).

[ApHBase] THE APACHE SOFTWARE FOUNDATION. *Apache HBase*. `https://hbase.apache.org/` Accédée le : 13/07/2019 (cf. p. 60).

[ApHIVE] THE APACHE SOFTWARE FOUNDATION. *Apache HIVE*. `https://hive.apache.org/` Accédée le : 13/07/2019 (cf. p. 60).

[ApSpark] THE APACHE SOFTWARE FOUNDATION. *Apache Spark*. `https://spark.apache.org/` Accédée le : 17/01/2019 (cf. p. 60).

[ApStorm] THE APACHE SOFTWARE FOUNDATION. *Apache Storm*. `https://storm.apache.org/` Accédée le : 10/01/2019 (cf. p. 61).

[Aug+11] Cédric AUGONNET, Samuel THIBAULT, Raymond NAMYST et Pierre-André WACRENIER. « StarPU : a unified platform for task scheduling on heterogeneous multicore architectures ». Dans : *Concurrency and Computation : Practice and Experience* 23.2 (2011), p. 187–198 (cf. p. 57, 99).

[Aug+12] Cédric AUGONNET, Olivier AUMAGE, Nathalie FURMENTO, Raymond NAMYST et Samuel THIBAULT. « StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators ». Dans : *Recent Advances in the Message Passing Interface*. Sous la dir. de Jesper Larsson TRÄFF, Siegfried BENKNER et Jack J. DONGARRA. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 298–299 (cf. p. 57).

[AV88] Alok AGGARWAL et S. VITTER Jeffrey. « The Input/Output Complexity of Sorting and Related Problems ». Dans : *Communications of the ACM* 31.9 (sept. 1988), p. 1116–1127 (cf. p. 32).

[AW96] Jan ALLEBACH et Ping Wah WONG. « Edge-directed interpolation ». Dans : *Proceedings of 3rd IEEE International Conference on Image Processing*. T. 3. IEEE. 1996, p. 707–710 (cf. p. 15).

[B+13] Upendra BHOI, Purvi N RAMANUJ et al. « Enhanced max-min task scheduling algorithm in cloud computing ». Dans : *International Journal of Application or Innovation in Engineering and Management (IJAIEM)* 2.4 (2013), p. 259–264 (cf. p. 62).

[BAB12] Anton BELOGLAZOV, Jemal ABAWAJY et Rajkumar BUYYA. « Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing ». Dans : *Future generation computer systems* 28.5 (2012), p. 755–768 (cf. p. 62).

[Ban+99] Guruduth BANAVAR, Tushar CHANDRA, Bodhi MUKHERJEE et al. « An efficient multicast protocol for content-based publish-subscribe systems ». Dans : *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No. 99CB37003)*. IEEE. 1999, p. 262–272 (cf. p. 77).

[Bar94] Greg BARNES. « A method for implementing lock-free shared data structures ». Dans : (1994) (cf. p. 64).

[Bas04] Cedric BASTOUL. « Code generation in the polyhedral model is easier than you think ». Dans : *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society. 2004, p. 7–16 (cf. p. 58).

[BC05] Daniel P BOVET et Marco CESATI. *Understanding the Linux Kernel : from I/O ports to process management*. O'Reilly Media, Inc., 2005 (cf. p. 30, 40).

[Bee85] Michael BEERING. « The determination of metameric mismatch limits of industrial colorant sets ». Thèse de doct. Rochester Institute of Technology, 1985 (cf. p. 14).

[BF99] Gregory T BYRD et Michael J FLYNN. « Producer-consumer communication in distributed shared memory multiprocessors ». Dans : *Proceedings of the IEEE* 87.3 (1999), p. 456–466 (cf. p. 64).

[BH15] Roberto BELLI et Torsten HOEFLER. « Notified access : Extending remote memory access programming models for producer-consumer synchronization ». Dans : *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, p. 871–881 (cf. p. 65).

[Bin+16] Timo BINGMANN, Michael AXTMANN, Emanuel JÖBSTL et al. « Thrill : High-performance algorithmic distributed batch data processing with C++ ». Dans : *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, p. 172–183 (cf. p. 60).

[BK11] Motti BECK et Michael KAGAN. « Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure ». Dans : *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*. International Teletraffic Congress. 2011, p. 9–15 (cf. p. 65).

[BL99] Robert D BLUMOFE et Charles E LEISERSON. « Scheduling multithreaded computations by work stealing ». Dans : *Journal of the ACM (JACM)* 46.5 (1999), p. 720–748 (cf. p. 62).

[Bon+08] Uday BONDHUGULA, Albert HARTONO, Jagannathan RAMANUJAM et Ponnuswamy SADAYAPPAN. « A practical automatic polyhedral parallelizer and locality optimizer ». Dans : *Acm Sigplan Notices*. T. 43. 6. ACM. 2008, p. 101–113 (cf. p. 59).

[Bor+08] Dhruba BORTHAKUR et al. « HDFS architecture guide ». Dans : *Hadoop Apache Project* 53.1-13 (2008), p. 2 (cf. p. 60).

[Bos+12] George BOSILCA, Aurelien BOUTEILLER, Anthony DANALIS et al. « DAGuE : A generic distributed DAG engine for high performance computing ». Dans : *Parallel Computing* 38.1-2 (2012), p. 37–51 (cf. p. 63).

[BRS10] Muthu Manikandan BASKARAN, Jj RAMANUJAM et P SADAYAPPAN. « Automatic C-to-CUDA code generation for affine programs ». Dans : *International Conference on Compiler Construction*. Springer. 2010, p. 244–263 (cf. p. 59).

[CAK16] Ignacio CANO, Srinivas AIYAR et Arvind KRISHNAMURTHY. « Characterizing private clouds : A large-scale empirical analysis of enterprise clusters ». Dans : *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM. 2016, p. 29–41 (cf. p. 70).

[Cal+13] Irina CALCIU, Dave DICE, Yossi LEV et al. « NUMA-aware reader-writer locks ». Dans : *ACM SIGPLAN Notices*. T. 48. 8. ACM. 2013, p. 157–166 (cf. p. 64).

[Cal18] CALDERA. *Caldera RIP*. Version 12.0. `https://www.caldera.com/software-workflow/version-12/` Accédée le : 08/07/2019. 3 oct. 2018 (cf. p. 41).

[Cal19] CALDERA. *Caldera RIP*. Version 12.1. `https://www.caldera.com/software-workflow/version-12/` Accédée le : 07/08/2019. 20 juin 2019 (cf. p. 143).

[Car+15] Paris CARBONE, Asterios KATSIFODIMOS, Stephan EWEN et al. « Apache flink : Stream and batch processing in a single engine ». Dans : *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cf. p. 61).

[CCD07] Liqun CHENG, John B CARTER et Donglai DAI. « An adaptive cache coherence protocol optimized for producer-consumer sharing ». Dans : *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, p. 328–339 (cf. p. 64).

[CCH08] Chun CHEN, Jacqueline CHAME et Mary HALL. *CHiLL : A framework for composing high-level loop transformations*. Rapp. tech. Citeseer, 2008 (cf. p. 59).

[CCZ04] David CALLAHAN, Bradford L CHAMBERLAIN et Hans P ZIMA. « The cascade high productivity language ». Dans : *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE. 2004, p. 52–60 (cf. p. 58).

[CGY01] Jeffrey S CHASE, Andrew J GALLATIN et Kenneth G YOCUM. « End system optimizations for high-speed TCP ». Dans : *IEEE Communications Magazine* 39.4 (2001), p. 68–74 (cf. p. 117).

[Cha+05] Philippe CHARLES, Christian GROTHOFF, Vijay SARASWAT et al. « X10 : an object-oriented approach to non-uniform cluster computing ». Dans : *Acm Sigplan Notices*. T. 40. 10. ACM. 2005, p. 519–538 (cf. p. 58).

[Che+13] Quan CHEN, Minyi GUO, Qianni DENG et al. « HAT : history-based auto-tuning MapReduce in heterogeneous environments ». Dans : *The Journal of Supercomputing* 64.3 (2013), p. 1038–1054 (cf. p. 63).

[Che12] Chun CHEN. « Polyhedra scanning revisited ». Dans : *Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2012, p. 499–508 (cf. p. 58).

[Chi+12a] Bobby CHIN, Carl MINCHEW, Patrick Tak Fu CHONG et Anthony Joseph CALABRIA. *Method for managing metamerism of color merchandise*. US Patent 8,330,991. Déc. 2012 (cf. p. 14).

[Chi+12b] Charisee CHIW, Gordon KINDLMANN, John REPPY, Lamont SAMUELS et Nick SELTZER. « Diderot : a parallel DSL for image analysis and visualization ». Dans : *Acm sigplan notices*. T. 47. 6. ACM. 2012, p. 111–120 (cf. p. 59).

[Cho+11] Mosharaf CHOWDHURY, Matei ZAHARIA, Justin MA, Michael I JORDAN et Ion STOICA. « Managing data transfers in computer clusters with orchestra ». Dans : *ACM SIGCOMM Computer Communication Review* 41.4 (2011), p. 98–109 (cf. p. 99).

[Com] COMMISSION INTERNATIONALE DE L'ÉCLAIRAGE. *ISO/CIE 10527-1991 : Colorimetric Observers*.
`http://www.cie.co.at/publications/colorimetric-observers` Accédée le : 22/08/2019 (cf. p. 14).

[Coo14] Simon COOKE. *The Bip Buffer - The Circular Buffer with a Twist*. 2014. URL : `https://www.codeproject.com/Articles/3479/The-Bip-Buffer-The-Circular-Buffer-with-a-Twist` (visité le 4 mar. 2019) (cf. p. 64).

[CT90] Richard J CLOUTIER et Donald E THOMAS. « The combination of scheduling, allocation, and mapping in a single algorithm ». Dans : *27th ACM/IEEE design automation conference*. IEEE. 1990, p. 71–76 (cf. p. 89).

[DFC05] Benoit DONNET, Timur FRIEDMAN et Mark CROVELLA. « Improved algorithms for network topology discovery ». Dans : *International Workshop on Passive and Active Network Measurement*. Springer. 2005, p. 149–162 (cf. p. 99).

[DG08] Jeffrey DEAN et Sanjay GHEMAWAT. « MapReduce : Simplified Data Processing on Large Clusters ». Dans : *Communications of the ACM* 51.1 (jan. 2008), p. 107–113 (cf. p. 60).

[DG10] Jeffrey DEAN et Sanjay GHEMAWAT. *System and method for efficient large-scale data processing*. US Patent 7,650,331. Jan. 2010 (cf. p. 59).

[Din+09] James DINAN, D Brian LARKINS, Ponnuswamy SADAYAPPAN, Sriram KRISHNAMOORTHY et Jarek NIEPLOCHA. « Scalable work stealing ». Dans : *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE. 2009, p. 1–11 (cf. p. 62).

[DL89] Jianzhong DU et Joseph Y-T LEUNG. « Complexity of scheduling parallel task systems ». Dans : *SIAM Journal on Discrete Mathematics* 2.4 (1989), p. 473–487 (cf. p. 99).

[Dre05] Ulrich DREPPER. *Futexes are tricky*. Rapp. tech. Citeseer, 2005 (cf. p. 65).

[Ekl72] J. O. EKLUNDH. « A Fast Computer Method for Matrix Transposing ». Dans : *IEEE Transactions on Computers* 21.7 (juil. 1972), p. 801–803 (cf. p. 32).

[EN07] Kobra ETMINANI et M NAGHIBZADEH. « A min-min max-min selective algorihtm for grid task scheduling ». Dans : *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*. IEEE. 2007, p. 1–7 (cf. p. 62).

[ERE12] OM ELZEKI, MZ RESHAD et MA ELSOUD. « Improved max-min algorithm in cloud computing ». Dans : *International Journal of Computer Applications* 50.12 (2012) (cf. p. 62).

[ES06] Tarek EL-GHAZAWI et Lauren SMITH. « UPC : unified parallel C ». Dans : *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 27 (cf. p. 58).

[Eug+03] Patrick Th EUGSTER, Pascal A FELBER, Rachid GUERRAOUI et Anne-Marie KERMARREC. « The many faces of publish/subscribe ». Dans : *ACM computing surveys (CSUR)* 35.2 (2003), p. 114–131 (cf. p. 72).

[Fei+97] Dror G FEITELSON, Larry RUDOLPH, Uwe SCHWIEGELSHOHN, Kenneth C SEVCIK et Parkson WONG. « Theory and practice in parallel job scheduling ». Dans : *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, p. 1–34 (cf. p. 61).

[FH07] Keir FRASER et Tim HARRIS. « Concurrent programming without locks ». Dans : *ACM Transactions on Computer Systems (TOCS)* 25.2 (2007), p. 5 (cf. p. 64).

[FK98] Ian FOSTER et Nicholas T KARONIS. « A grid-enabled MPI : Message passing in heterogeneous distributed computing systems ». Dans : *SC'98 : Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE. 1998, p. 46–46 (cf. p. 57).

[FL11] Paul FEAUTRIER et Christian LENGAUER. « Polyhedron model ». Dans : *Encyclopedia of parallel computing* (2011), p. 1581–1592 (cf. p. 58).

[FR96] Dror G FEITELSON et Larry RUDOLPH. « Toward convergence in job schedulers for parallel supercomputers ». Dans : *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, p. 1–26 (cf. p. 61).

[FR98] Dror G FEITELSON et Larry RUDOLPH. « Metrics and benchmarking for parallel job scheduling ». Dans : *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1998, p. 1–24 (cf. p. 61).

[FRS00] Ian FOSTER, Alain ROY et Volker SANDER. « A quality of service architecture that combines resource reservation and application adaptation ». Dans : *2000 Eighth International Workshop on Quality of Service. IWQoS 2000 (Cat. No. 00EX400)*. IEEE. 2000, p. 181–188 (cf. p. 117).

[Gab+04] Edgar GABRIEL, Graham E FAGG, George BOSILCA et al. « Open MPI : Goals, concept, and design of a next generation MPI implementation ». Dans : *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer. 2004, p. 97–104 (cf. p. 57).

[Gau+13] Thierry GAUTIER, Joao VF LIMA, Nicolas MAILLARD et Bruno RAFFIN. « Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures ». Dans : *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, p. 1299–1308 (cf. p. 63).

[GKK12] Fred GUSTAVSON, Lars KARLSSON et Bo KÅGSTRÖM. « Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion ». Dans : *ACM Trans. Math. Softw.* 38.3 (avr. 2012), 17 :1–17 :32 (cf. p. 32).

[GM04] Luigi A GRIECO et Saverio MASCOLO. « Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control ». Dans : *ACM SIGCOMM Computer Communication Review* 34.2 (2004), p. 25–38 (cf. p. 116).

[God+19] P. GODARD, V. LOECHNER, C. BASTOUL, F. SOULIER et G. MULLER. « A Flexible and Distributed Runtime System for High-Throughput Constrained Data Streams Generation ». Dans : *20th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) in conjunction with IPDPS*. Rio de Janeiro, Brazil, mai 2019 (cf. p. 67, 147).

[God19] P. GODARD. « Échanges non bloquants de données ordonnées entre producteurs multiples et consommateur unique ». Dans : *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*. Bayonne, France, juin 2019 (cf. p. 101, 147).

[Goi+10] Inigo GOIRI, Ferran JULIA, Ramon NOU et al. « Energy-aware scheduling in virtualized datacenters ». Dans : *2010 IEEE International Conference on Cluster Computing*. IEEE. 2010, p. 58–67 (cf. p. 62).

[GRF06] Pedro GAMA, Carlos RIBEIRO et Paulo FERREIRA. « Heimdhal : A history-based policy engine for grids ». Dans : *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. T. 1. IEEE. 2006, 8–pp (cf. p. 63).

[GST10] Anders GIDENSTAM, Håkan SUNDELL et Philippas TSIGAS. « Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency ». Dans : *International Conference On Principles Of Distributed Systems*. Springer. 2010, p. 302–317 (cf. p. 64).

[Gus88] John L GUSTAFSON. « Reevaluating Amdahl's law ». Dans : *Communications of the ACM* 31.5 (1988), p. 532–533 (cf. p. 83).

[Guy19] GUY ERIC SCHALNAT, ANDREAS DILGER, JOHN BOWLER, GLENN RANDERS-PEHRSON, COSMIN TRUTA, AND OTHERS. *libpng Home Page*. `http://www.libpng.org/pub/png/libpng.html` Accédée le : 22/08/2019. 2019 (cf. p. 15).

[Har+07] Mark HARRIS et al. « Optimizing parallel reduction in CUDA ». Dans : *Nvidia developer technology* 2.4 (2007), p. 70 (cf. p. 57).

[Hov+03] Matthias HOVESTADT, Odej KAO, Axel KELLER et Achim STREIT. « Scheduling in HPC resource management systems : Queuing vs. planning ». Dans : *Workshop on Job Scheduling Strategies for Parallel Processing*. 2003, p. 1–20 (cf. p. 63).

[HS11] Mohammad HAMMOUD et Majd F SAKR. « Locality-aware reduce task scheduling for MapReduce ». Dans : *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, p. 570–576 (cf. p. 62).

[HSV03] XiaoShan HE, XianHe SUN et Gregor VON LASZEWSKI. « QoS guided min-min heuristic for grid task scheduling ». Dans : *Journal of Computer Science and Technology* 18.4 (2003), p. 442–451 (cf. p. 62).

[HW92] Wilson C HSIEH et William E WEIHL. « Scalable reader-writer locks for parallel systems ». Dans : *Proceedings Sixth International Parallel Processing Symposium*. IEEE. 1992, p. 656–659 (cf. p. 64).

[Ili61] JK ILIFFE. « The use of the genie system in numerical calculation ». Dans : *International Tracts in Computer Science and Technology and Their Application*. T. 2. Elsevier, 1961, p. 1–28 (cf. p. 29).

[Jim+14] Alexandra JIMBOREAN, Philippe CLAUSS, Jean-François DOLLINGER, Vincent LOECHNER et Juan Manuel MARTINEZ CAAMAÑO. « Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons ». Dans : *International Journal of Parallel Programming* 42.4 (2014), p. 529–545 (cf. p. 59).

[JPA99] R. JIMENEZ-PERIS, M. PATINO-MARTINEZ et S. AREVALO. « Multithreaded rendezvous : a design pattern for distributed rendezvous ». Dans : (1999) (cf. p. 64).

[Kal+14] Rashid KALEEM, Rajkishore BARIK, Tatiana SHPEISMAN et al. « Adaptive heterogeneous scheduling for integrated GPUs ». Dans : *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE. 2014, p. 151–162 (cf. p. 62, 99).

[Kau+93] S. D. KAUSHIK, C.-H. HUANG, R. W. JOHNSON, P. SADAYAPPAN et J. R. JOHNSON. « Efficient Transposition Algorithms for Large Matrices ». Dans : *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA : ACM, 1993, p. 656–665 (cf. p. 32).

[KBK13] Dzmitry KLIAZOVICH, Pascal BOUVRY et Samee Ullah KHAN. « DENS : data center energy-efficient network-aware scheduling ». Dans : *Cluster computing* 16.1 (2013), p. 65–75 (cf. p. 62).

[KF90] Alan H KARP et Horace P FLATT. « Measuring parallel processor performance ». Dans : *Communications of the ACM* 33.5 (1990), p. 539–543 (cf. p. 83).

[Koe+94] Charles H KOELBEL, David B LOVEMAN, Robert S SCHREIBER, Guy Lewis STEELE JR et Mary ZOSEL. *The high performance Fortran handbook*. MIT press, 1994 (cf. p. 58).

[Kop91] Hermann KOPETZ. « Event-triggered versus time-triggered real-time systems ». Dans : *Operating Systems of the 90s and Beyond*. Springer, 1991, p. 86–101 (cf. p. 91).

[Kri+03] S. KRISHNAMOORTHY, G. BAUMGARTNER, D. COCIORVA, C. LAM et P. SADAYAPPAN. « Efficient parallel out-of-core matrix transposition ». Dans : *2003 Proceedings IEEE International Conference on Cluster Computing*. Hong Kong, China, déc. 2003, p. 300–307 (cf. p. 32).

[Kri+06] Sriram KRISHNAMOORTHY, Gerald BAUMGARTNER, Daniel COCIORVA, Chi-Chung LAM et Ponnuswamy SADAYAPPAN. « Efficient parallel out-of-core matrix transposition ». Dans : *International Journal on High Performance Computing and Networking* 2.4 (jan. 2006), p. 110–119 (cf. p. 32).

[KS09] Jacek KOBUS et Rafal SZKLARSKI. *Completely Fair Scheduler and its tuning*. Rapp. tech. 2009 (cf. p. 62).

[KTF03] Nicholas T KARONIS, Brian TOONEN et Ian FOSTER. « MPICH-G2 : A grid-enabled implementation of the message passing interface ». Dans : *Journal of Parallel and Distributed Computing* 63.5 (2003), p. 551–563 (cf. p. 57).

[LaC+13] Katrina LACURTS, Shuo DENG, Ameesh GOYAL et Hari BALAKRISHNAN. « Choreo : Network-aware task placement for cloud applications ». Dans : *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, p. 191–204 (cf. p. 63).

[Lam87] Leslie LAMPORT. « A fast mutual exclusion algorithm ». Dans : *ACM Trans. Comput. Syst.* 5.1 (1987), p. 1–11 (cf. p. 64).

[LBC09] Patrick PC LEE, Tian BU et Girish CHANDRANMENON. « A lock-free, cache-efficient shared ring buffer for multi-core architectures ». Dans : *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM. 2009, p. 78–79 (cf. p. 64).

[Lê+13] Nhat Minh LÊ, Adrien GUATTO, Albert COHEN et Antoniu POP. « Correct and efficient bounded FIFO queues ». Dans : *2013 25th International Symposium on Computer Architecture and High Performance Computing*. IEEE. 2013, p. 144–151 (cf. p. 64).

[LGS99] Thomas Martin LEHMANN, Claudia GONNER et Klaus SPITZER. « Survey : Interpolation methods in medical image processing ». Dans : *IEEE transactions on medical imaging* 18.11 (1999), p. 1049–1075 (cf. p. 15).

[Lin17] LINUX MAN-PAGES. *Linux Programmer's Manual - TCP protocol - tcp(7)*. `http://man7.org/linux/man-pages/man7/tcp.7.html` Accédée le : 17/08/2018. 2017 (cf. p. 116).

[Liu+19] Ying LIU, Lei HUANG, Mingchuan WU et al. « PPOpenCL : a performance-portable OpenCL compiler with host and kernel thread code fusion ». Dans : *Proceedings of the 28th International Conference on Compiler Construction*. ACM. 2019, p. 2–16 (cf. p. 57).

[LKB77] Jan Karel LENSTRA, AHG Rinnooy KAN et Peter BRUCKER. « Complexity of machine scheduling problems ». Dans : *Annals of discrete mathematics*. T. 1. Elsevier, 1977, p. 343–362 (cf. p. 54).

[LMA15] S. LI, M. A. MADDAH-ALI et A. S. AVESTIMEHR. « Coded MapReduce ». Dans : *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. Sept. 2015, p. 964–971 (cf. p. 60).

[LMD94] Birger LANDWEHR, Peter MARWEDEL et Rainer DÖMER. *OSCAR : Optimum simultaneous scheduling, allocation and resource binding based on integer programming*. Dekanat Informatik, Univ., 1994 (cf. p. 89).

[LNR09] Karthik LAKSHMANAN, Dionisio de NIZ et Ragunathan RAJKUMAR. « Coordinated task scheduling, allocation and synchronization on multiprocessors ». Dans : *2009 30th IEEE Real-Time Systems Symposium*. IEEE. 2009, p. 469–478 (cf. p. 89).

[LO01] Xin LI et Michael T ORCHARD. « New edge-directed interpolation ». Dans : *IEEE transactions on image processing* 10.10 (2001), p. 1521–1527 (cf. p. 15).

[Loe99] Vincent LOECHNER. *PolyLib : A library for manipulating parameterized polyhedra*. `https://icps.u-strasbg.fr/polylib/`. 1999 (cf. p. 58).

[LP96] John C LIN et Sanjoy PAUL. « RMTP : A reliable multicast transport protocol ». Dans : *Proceedings of IEEE INFOCOM'96. Conference on Computer Communications*. T. 3. IEEE. 1996, p. 1414–1424 (cf. p. 99).

[LSD89] John LEHOCZKY, Lui SHA et Yuqin DING. « The rate monotonic scheduling algorithm : Exact characterization and average case behavior ». Dans : *[1989] Proceedings. Real-Time Systems Symposium*. IEEE. 1989, p. 166–171 (cf. p. 62).

[Luo+12] Miao LUO, Dhabaleswar K PANDA, Khaled Z IBRAHIM et Costin IANCU. « Congestion avoidance on manycore high performance computing systems ». Dans : *Proceedings of the 26th ACM international conference on Supercomputing*. ACM. 2012, p. 121–132 (cf. p. 98).

[Luo16] Ronnier LUO. *Encyclopedia of color science and technology*. Springer Publishing Company, Incorporated, 2016 (cf. p. 14).

[LWP04] Jiuxing LIU, Jiesheng WU et Dhabaleswar K PANDA. « High performance RDMA-based MPI implementation over InfiniBand ». Dans : *International Journal of Parallel Programming* 32.3 (2004), p. 167–198 (cf. p. 65).

[LYF98] K LAKSHMAN, Raj YAVATKAR et Raphael FINKEL. « Integrated CPU and network-I/O QoS management in an endsystem ». Dans : *Computer Communications* 21.4 (1998), p. 325–333 (cf. p. 117).

[Mae+01] Rafael MAESTRE, Fadi J KURDAHI, Milagros FERNÁNDEZ et al. « A framework for reconfigurable computing : task scheduling and context management ». Dans : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.6 (2001), p. 858–873 (cf. p. 63).

[McL86] Keith MCLAREN. *The colour science of dyes and pigments*. Adam Hilger Ltd., 1986 (cf. p. 14).

[Men+16] Xiangrui MENG, Joseph BRADLEY, Burak YAVUZ et al. « Mllib : Machine learning in apache spark ». Dans : *The Journal of Machine Learning Research* 17.1 (2016), p. 1235–1241 (cf. p. 60).

[Mor08] Ján MOROVIČ. *Color gamut mapping*. T. 10. John Wiley & Sons, 2008 (cf. p. 16).

[MPI15] MPI FORUM. *Message-Passing Interface (MPI) Standard, Version 3.1*. `https://www.mpi-forum.org/` Accédée le : 13/07/2019. 2015 (cf. p. 57).

[MS91] John M MELLOR-CRUMMEY et Michael L SCOTT. « Scalable reader-writer synchronization for shared-memory multiprocessors ». Dans : *ACM SIGPLAN Notices*. T. 26. 7. ACM. 1991, p. 106–113 (cf. p. 64).

[New+13] Chris J NEWBURN, Serguei DMITRIEV, Ravi NARAYANASWAMY et al. « Offload compiler runtime for the Intel® Xeon Phi coprocessor ». Dans : *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, p. 1213–1225 (cf. p. 57).

[OH99] Victor OSTROMOUKHOV et Roger David HERSCH. « Stochastic clustered-dot dithering ». Dans : *Journal of electronic imaging* 8.4 (1999), p. 439–446 (cf. p. 17).

[ON63] Gerald OSTER et Yasunori NISHIJIMA. « Moiré patterns ». Dans : *Scientific American* 208.5 (1963), p. 54–63 (cf. p. 17).

[Ope18] OPENMP ARB. *The OpenMP API 5.0 specification for parallel programming*. `https://www.openmp.org/` Accédée le : 13/07/2019. 2018 (cf. p. 35, 56).

[Ope19] OPENACC. *OpenACC official website*. `https://www.openacc.org/` Accédée le : 30/08/2019. 2019 (cf. p. 56).

[Pai+98] Vivek S PAI, Mohit ARON, Gaurov BANGA et al. « Locality-aware request distribution in cluster-based network servers ». Dans : *ACM Sigplan Notices* 33.11 (1998), p. 205–216 (cf. p. 63).

[Pal+11] Balaji PALANISAMY, Aameek SINGH, Ling LIU et Bhushan JAIN. « Purlieus : locality-aware resource allocation for MapReduce in a cloud ». Dans : *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2011, p. 58 (cf. p. 62).

[Pat+90] John F PATTERSON, Ralph D HILL, Steven L ROHALL et Scott W MEEKS. « Rendezvous : An architecture for synchronous multi-user applications ». Dans : *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*. ACM. 1990, p. 317–328 (cf. p. 64).

[Per+15] Jonathan PERRY, Amy OUSTERHOUT, Hari BALAKRISHNAN, Devavrat SHAH et Hans FUGAL. « Fastpass : A centralized zero-queue datacenter network ». Dans : *ACM SIGCOMM Computer Communication Review* 44.4 (2015), p. 307–318 (cf. p. 62).

[PKT83] J Anthony PARKER, Robert V KENYON et Donald E TROXEL. « Comparison of interpolating methods for image resampling ». Dans : *IEEE Transactions on medical imaging* 2.1 (1983), p. 31–39 (cf. p. 15).

[Pol+11] Jorda POLO, Claris CASTILLO, David CARRERA et al. « Resource-aware adaptive scheduling for mapreduce clusters ». Dans : *Proceedings of the 12th International Middleware Conference*. International Federation for Information Processing. 2011, p. 180–199 (cf. p. 89).

[Pop07] Ioan POP. *Method for generating customized ink/media transforms*. US Patent 7,251,058. Juil. 2007 (cf. p. 16).

[PRP16] Harsh PATHAK, Manas RATHI et Aniket PAREKH. « Introduction to real-time processing in Apache Apex ». Dans : *Int. J. Res. Advent Technol.* (2016), p. 19 (cf. p. 61).

[Qia13] Yue QIAO. *Color conversion with toner/ink limitations*. US Patent 8,395,831. Mar. 2013 (cf. p. 16).

[RA07] Mohammad J RASHTI et Ahmad AFSAHI. « 10-Gigabit iWARP Ethernet : comparative performance analysis with InfiniBand and Myrinet-10G ». Dans : *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, p. 1–8 (cf. p. 65).

[Rag+13] Jonathan RAGAN-KELLEY, Connelly BARNES, Andrew ADAMS et al. « Halide : a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines ». Dans : *Acm Sigplan Notices*. T. 48. 6. ACM. 2013, p. 519–530 (cf. p. 59).

[Rei+01] Erik REINHARD, Michael ADHIKHMIN, Bruce GOOCH et Peter SHIRLEY. « Color transfer between images ». Dans : *IEEE Computer graphics and applications* 21.5 (2001), p. 34–41 (cf. p. 16).

[RFC1644] R. BRADEN. *T/TCP – TCP Extensions for Transactions Functional Specification*. `https://tools.ietf.org/html/rfc1644` Accédée le : 13/07/2019. 1994 (cf. p. 116).

[RFC6298] J. Chu V. PAXSON M. Allman et M. SARGENT. *Computing TCP's Retransmission Timer*. `https://tools.ietf.org/html/rfc6298` Accédée le : 13/07/2019. 2011 (cf. p. 114).

[RFC6897] M. SCHARF et A. FORD. *Multipath TCP (MPTCP) Application Interface Considerations*. `https://tools.ietf.org/html/rfc6897` Accédée le : 17/08/2018. 2013 (cf. p. 117).

[RFC7323] V. Jacobson D. BORMAN B. Braden et Ed. R. SCHEFFENEGGER. *TCP Extensions for High Performance*. `https://tools.ietf.org/html/rfc7323` Accédée le : 17/08/2018. 2014 (cf. p. 116).

[RFC7424] A. Ghanwani R. KRISHNAN L. Yong. *Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks*. `https://tools.ietf.org/html/rfc7424` Accédée le : 20/08/2018. 2015 (cf. p. 117).

[RFC896] John NAGLE. *Congestion Control in IP/TCP Internetworks*. `https://tools.ietf.org/html/rfc896` Accédée le : 17/08/2018. 1984 (cf. p. 116).

[RHG15] Mahesh RAVISHANKAR, Justin HOLEWINSKI et Vinod GROVER. « Forma : A DSL for image processing applications to target GPUs and multi-core CPUs ». Dans : *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. ACM. 2015, p. 109–120 (cf. p. 59).

[RHJ09] Rolf RABENSEIFNER, Georg HAGER et Gabriele JOST. « Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes ». Dans : *2009 17th Euromicro international conference on parallel, distributed and network-based processing*. IEEE. 2009, p. 427–436 (cf. p. 57).

[RLS98] Rajesh RAMAN, Miron LIVNY et Marvin SOLOMON. « Matchmaking : Distributed resource management for high throughput computing ». Dans : *17th International Symposium on High Performance Distributed Computing (HPDC)*. IEEE. 1998, p. 140 (cf. p. 63, 69).

[RMZ13] Amitabha ROY, Ivo MIHAILOVIC et Willy ZWAENEPOEL. « X-stream : Edge-centric graph processing using streaming partitions ». Dans : *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, p. 472–488 (cf. p. 61).

[Ros+13] Christopher J ROSSBACH, Yuan YU, Jon CURREY, Jean-Philippe MARTIN et Dennis FETTERLY. « Dandelion : a compiler and runtime for heterogeneous systems ». Dans : *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, p. 49–68 (cf. p. 58).

[RSZ89] Krithi RAMAMRITHAM, John A. STANKOVIC et Wei ZHAO. « Distributed scheduling of tasks with deadlines and resource requirements ». Dans : *IEEE Transactions on Computers* 38.8 (1989), p. 1110–1123 (cf. p. 63, 69).

[RV97] Luigi RIZZO et Lorenzo VICISANO. « A Reliable Multicast data Distribution Protocol based on software FEC techniques ». Dans : *The Fourth IEEE Workshop on High-Performance Communication Systems*. IEEE. 1997, p. 116–125 (cf. p. 99).

[Sam03] SAM LEFFLER, AND OTHERS. *Home of the LibTIFF software*. `http://www.libtiff.org/` Accédée le : 22/08/2019. 2003 (cf. p. 15).

[SB94] Marco SPURI et Giorgio C BUTTAZZO. « Efficient Aperiodic Service Under Earliest Deadline Scheduling. » Dans : *RTSS*. 1994, p. 2–11 (cf. p. 62).

[SC99] Frank SIQUEIRA et Vinny CAHILL. « Delivering QoS in open distributed systems ». Dans : *Proceedings 7th IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE. 1999, p. 185–190 (cf. p. 117).

[SCC15] Luis SANT'ANA, Daniel CORDEIRO et Raphael CAMARGO. « Plb-hec : A profile-based load-balancing algorithm for heterogeneous cpu-gpu clusters ». Dans : *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, p. 96–105 (cf. p. 99).

[Sch00] Günter SCHMIDT. « Scheduling with limited machine availability ». Dans : *European Journal of Operational Research* 121.1 (2000), p. 1–15 (cf. p. 54).

[Seg+15] Oren SEGAL, Philip COLANGELO, Nasibeh NASIRI, Zhuo QIAN et Martin MARGALA. « Sparkcl : A unified programming framework for accelerators on heterogeneous clusters ». Dans : *arXiv preprint arXiv :1505.01120* (2015) (cf. p. 60).

[SFR04] W Richard STEVENS, Bill FENNER et Andrew M RUDOFF. *UNIX Network Programming : The Sockets Networking API*. T. 1. Addison-Wesley Professional, 2004 (cf. p. 113, 117).

[SG09] Bianca SCHROEDER et Garth GIBSON. « A large-scale study of failures in high-performance computing systems ». Dans : *IEEE transactions on Dependable and Secure Computing* 7.4 (2009), p. 337–350 (cf. p. 70).

[SGS10] John E STONE, David GOHARA et Guochun SHI. « OpenCL : A parallel programming standard for heterogeneous computing systems ». Dans : *Computing in science & engineering* 12.3 (2010), p. 66 (cf. p. 57).

[SK10] Jason SANDERS et Edward KANDROT. *CUDA by example : an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010 (cf. p. 57).

[SO95] Markus Andreas STRICKER et Markus ORENGO. « Similarity of color images ». Dans : *Storage and retrieval for image and video databases III*. T. 2420. International Society for Optics et Photonics. 1995, p. 381–392 (cf. p. 16).

[SP02] Jinwoo SUH et V. K. PRASANNA. « An efficient algorithm for out-of-core matrix transposition ». Dans : *IEEE Transactions on Computers* 51.4 (avr. 2002), p. 420–438 (cf. p. 32).

[ST03] Adam SMYK et Marek TUDRUJ. « RDMA communication based on rotating buffers for efficient parallel fine-grain computations ». Dans : *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2003, p. 50–58 (cf. p. 65).

[Ste+15] Michel STEUWER, Christian FENSCH, Sam LINDLEY et Christophe DUBACH. « Generating performance portable code using rewrite rules : from high-level functional expressions to high-performance OpenCL code ». Dans : *ACM SIGPLAN Notices* 50.9 (2015), p. 205–217 (cf. p. 57).

[Sti09] Tim STITT. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009 (cf. p. 58).

[SV96] Madhavapeddi SHREEDHAR et George VARGHESE. « Efficient fair queuing using deficit round-robin ». Dans : *IEEE/ACM Transactions on networking* 3 (1996), p. 375–385 (cf. p. 62).

[Tab07] Paulo TABUADA. « Event-triggered real-time scheduling of stabilizing control tasks ». Dans : *IEEE Transactions on Automatic Control* 52.9 (2007), p. 1680–1685 (cf. p. 91).

[TBU00] Philippe THÉVENAZ, Thierry BLU et Michael UNSER. « Image interpolation and resampling ». Dans : *Handbook of medical imaging, processing and analysis* 1.1 (2000), p. 393–420 (cf. p. 15).

[Thu+09] Ashish THUSOO, Joydeep Sen SARMA, Namit JAIN et al. « Hive : a warehousing solution over a map-reduce framework ». Dans : *Proceedings of the VLDB Endowment* 2.2 (2009), p. 1626–1629 (cf. p. 60).

[THW02] Haluk TOPCUOGLU, Salim HARIRI et Min-you WU. « Performance-effective and low-complexity task scheduling for heterogeneous computing ». Dans : *IEEE transactions on parallel and distributed systems* 13.3 (2002), p. 260–274 (cf. p. 99).

[Tor10] Massimo TORQUATI. « Single-producer/single-consumer queues on shared cache multi-core systems ». Dans : *arXiv preprint arXiv :1012.1824* (2010) (cf. p. 64).

[TS10] Ayşegül TOPTAL et Ihsan SABUNCUOGLU. « Distributed scheduling : a review of concepts and applications ». Dans : *International Journal of Production Research* 48.18 (2010), p. 5235–5262 (cf. p. 63, 69).

[TY13] Risi THONANGI et Jun YANG. « Permuting Data on Random-access Block Storage ». Dans : *Proceedings of the VLDB Endowment* 6.9 (juil. 2013), p. 721–732 (cf. p. 32).

[Ull75] Jeffrey D. ULLMAN. « NP-complete scheduling problems ». Dans : *Journal of Computer and System sciences* 10.3 (1975), p. 384–393 (cf. p. 54).

[Val95] John D VALOIS. « Lock-free linked lists using compare-and-swap ». Dans : *PODC*. T. 95. 1995, p. 214–222 (cf. p. 64).

[Vav+13] Vinod Kumar VAVILAPALLI, Arun C. MURTHY, Chris DOUGLAS et al. « Apache Hadoop YARN : Yet Another Resource Negotiator ». Dans : *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California : ACM, 2013, 5 :1–5 :16 (cf. p. 63).

[Ven+17] Shivaram VENKATARAMAN, Aurojit PANDA, Kay OUSTERHOUT et al. « Drizzle : Fast and adaptable stream processing at scale ». Dans : *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, p. 374–389 (cf. p. 61).

[Ver+13] Sven VERDOOLAEGE, Juan CARLOS JUEGA, Albert COHEN et al. « Polyhedral parallel code generation for CUDA ». Dans : *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), p. 54 (cf. p. 59).

[Ver10] Sven VERDOOLAEGE. « isl : An Integer Set Library for the Polyhedral Model ». Dans : *Mathematical Software (ICMS'10)*. Sous la dir. de Komei FUKUDA, Joris HOEVEN, Michael JOSWIG et Nobuki TAKAYAMA. LNCS 6327. Springer-Verlag, 2010, p. 299–302 (cf. p. 58).

[Vor11] Mehul Nalin VORA. « Hadoop-HBase for large-scale data ». Dans : *Proceedings of 2011 International Conference on Computer Science and Network Technology*. T. 1. IEEE. 2011, p. 601–605 (cf. p. 60).

[Wal72] David C WALDEN. « Systems for Interprocess Communication in a Resource Sharing Computer Network ». Dans : *Communications of the ACM* 15.4 (1972) (cf. p. 64).

[Wan+10] Lizhe WANG, Gregor VON LASZEWSKI, Jay DAYAL et Fugang WANG. « Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with DVFS ». Dans : *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society. 2010, p. 368–377 (cf. p. 62).

[Whi84] Mary C WHITTON. « Memory design for raster graphics displays ». Dans : *IEEE Computer Graphics and Applications* 4.3 (1984), p. 48–65 (cf. p. 64).

[Wie+14] Sandra WIENKE, Christian TERBOVEN, James C BEYER et Matthias S MÜLLER. « A pattern-based comparison of OpenACC and OpenMP for accelerator computing ». Dans : *European Conference on Parallel Processing*. Springer. 2014, p. 812–823 (cf. p. 57).

[Won+08] Chee Siang WONG, Ian TAN, Rosalind Deena KUMARI et Fun WEY. « Towards achieving fairness in the Linux scheduler ». Dans : *ACM SIGOPS Operating Systems Review* 42.5 (2008), p. 34–43 (cf. p. 62).

[WT95] Tom WAGNER et Don TOWSLEY. *Getting started with posix threads*. Rapp. tech. University of Massachusetts at Amherst, 1995 (cf. p. 65).

[YA95] Jae-Heon YANG et Jams H ANDERSON. « A fast, scalable mutual exclusion algorithm ». Dans : *Distributed Computing* 9.1 (1995), p. 51–60 (cf. p. 64).

[You02] Thomas YOUNG. « II. The Bakerian Lecture. On the theory of light and colours ». Dans : *Philosophical transactions of the Royal Society of London* 92 (1802), p. 12–48 (cf. p. 16).

[Zah+10] Matei ZAHARIA, Dhruba BORTHAKUR, Joydeep SEN SARMA et al. « Delay scheduling : a simple technique for achieving locality and fairness in cluster scheduling ». Dans : *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, p. 265–278 (cf. p. 62).

[Zah+12] Matei ZAHARIA, Mosharaf CHOWDHURY, Tathagata DAS et al. « Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing ». Dans : *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, p. 2–2 (cf. p. 60).

[Zah+16] Matei ZAHARIA, Reynold S XIN, Patrick WENDELL et al. « Apache spark : a unified engine for big data processing ». Dans : *Communications of the ACM* 59.11 (2016), p. 56–65 (cf. p. 60).

[Zek15] A. ZEKRI. « Restructuring and implementations of 2D matrix transpose algorithm using SSE4 vector instructions ». Dans : *2015 International Conference on Applied Research in Computer Science and Engineering (ICAR)*. Beirut, Lebanon, oct. 2015, p. 1–7 (cf. p. 32).

[ZK06] Christian ZINNER et Wilfried KUBINGER. « ROS-DMA : a DMA double buffering method for embedded image processing with resource optimized slicing ». Dans : *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE. 2006, p. 361–372 (cf. p. 64).

Table des figures

Liste des tableaux

Colophon

Ce manuscrit de thèse a été rédigé avec le langage LaTeX.
Son apparence se base sur le style *Clean Thesis* [1] développé par Ricardo Langner.

1. http://cleanthesis.der-ric.de/

Université de Strasbourg
Laboratoire ICube
Équipe ICPS

Paul Godard

msii ÉCOLE DOCTORALE

Parallélisation et passage à l'échelle durable d'une chaîne de traitement graphique pour l'impression professionnelle

Résumé

Les nombreuses avancées du vaste domaine de l'impression professionnelle ont permis la multiplication des objets imprimés dans nos quotidiens. Désormais, la flexibilité introduite par les procédés d'impression numérique promet d'associer les souhaits de personnalisation avec les avantages de la production de masse. La rapide évolution des usages et des technologies, caractérisée par des fermes d'impression toujours plus grandes et des presses numériques toujours plus rapides, pose des problèmes inédits aux systèmes informatiques actuels qui pilotent les imprimantes. Dans cette thèse, nous explorons de nouvelles techniques inspirées des systèmes de calcul haute performance afin d'accélérer l'exécution des traitements graphiques indispensables à l'impression numérique. Nous introduisons pour cela une architecture de calculs distribués flexible exploitant des techniques de traitement et de synchronisation optimisées. Nous détaillons les principes de fonctionnement et les subtilités de l'implémentation de nos travaux qui permettent de respecter les contraintes spécifiques des flux de données produits. Nous réalisons une évaluation complète de notre solution qui y démontre ses excellentes performances et sa viabilité.

Mots clés : HPC, systèmes distribués, flux de données contraints, ordonnancement, transposition et rotation de matrices out-of-core, impression numérique.

Abstract

The strong and continuous improvements in the professional printing field have led to the ubiquity of printed objects in our daily life. The flexibility introduced by the digital printing process promises to associate extensive customization with mass production. The quick growth of printing usages and technologies, illustrated by wider printer farms and faster digital presses, leads to original challenges for the computer system in charge of driving them. In this thesis, we explore new approaches inspired by the high performance computing field to speedup the graphics processing necessary to digital printing. To achieve this goal, we introduce a distributed system which provides the adequate flexibility and performance by exploiting and optimizing both processing and synchronization techniques. We present our architecture up to the subtle parts of its implementation which allows our solution to meet the specific constraints on generating streams for printing purpose. We perform a complete evaluation of our solution and provide experimental evidence of its great performance and viability.

Keywords: HPC, distributed systems, constrained data stream, scheduling, out-of-core matrices transposition and rotation, digital printing.