Migration d’un système existant vers une architecture parallèle

par

Francis Moreau

essai présenté au CeFTI
en vue de l’obtention du grade de maître en génie logiciel
(maîtrise en génie logiciel incluant un cheminement de type cours en génie logiciel)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Longueuil, Québec, Canada, avril 2014

Remerciements

Je souhaite tout d’abord remercier mon directeur académique, M. Patrice Roy, pour son aide précieuse qui a rendu possible la réalisation de cet essai. Ses conseils judicieux, sa grande disponibilité, sa rigueur et son support ont d’une part, grandement contribué à rehausser la qualité de l’essai, et d’autre part, à alimenter ma motivation à compléter ce projet.

Je tiens également à remercier M. Jean-François Ménard, qui a généreusement accepté d’être mon directeur professionnel. Ses commentaires constructifs et nos nombreux échanges ont été d’une grande aide.

Finalement, je tiens à remercier M. Claude Cardinal, Directeur adjoint du CeFTI, qui s’est montré compréhensif quant aux délais en m’accordant du temps pour finaliser cet essai.

Table des matières

Introduction 1

Chapitre 1 Parallélisme 4

1.1 Historique 5

1.2 Types de systèmes pouvant bénéficier du parallélisme 6

1.3 Concurrence et parallélisme 7

1.3.1 Ordonnanceur 8

1.4 Types d’architectures de processeurs multi-cœurs 10

1.5 Architecture SMP 11

1.5.1 Hiérarchie de la mémoire 11

1.5.2 Localité des données 13

1.5.3 Cohérence des antémémoires 13

1.5.4 Défauts de cache 14

1.5.5 Concepts généraux pour une gestion efficace d'antémémoire 16

1.6 NUMA 19

1.6.1 Gestion de la mémoire entre les différents nœuds 20

1.7 Modèles de parallélisme 21

1.7.1 Modèle basé sur les tâches 23

1.7.2 Modèle basé sur les acteurs 24

1.8 Complexité reliée au parallélisme 27

Chapitre 2 Présentation d'une bibliothèque spécialisée pour le parallélisme 29

2.1 *Concurrency Runtime* 30

2.1.1 *Resource Manager* 30

2.1.2 *Task Scheduler* 33

2.1.3 Mode coopératif 34

2.1.4 Mode préemptif 35

2.2 *Parallel Patterns Library* 35

2.2.1 Parallélisme des données 35

2.2.2 Parallélisme des tâches 36

2.2.3 Agrégation parallèle 37

Chapitre 3 Migration vers une architecture parallèle 39

3.1 Architecture initiale du système en référence 42
3.2 Précaution à prendre avec les E/S non-bloquantes 45
3.3 Modèle à un fil d'exécution par session RTP avec E/S non-bloquantes 46
3.4 Modèle à un fil d'exécution par session RTP avec E/S bloquantes 48
3.5 Modèle à un fil d'exécution par unité de traitement 49
3.6 Modèle basé sur le mécanisme *async* 50
3.7 Modèle de tâches de base 52
3.8 Modèle hybride avec fils d'exécution et tâches 54
3.8.1 Fils d'exécution et tâches en parallèle 55
3.8.2 Fils d'exécution et tâches séquentiellement 57
Chapitre 4 Analyse des résultats 62
4.1 Modèle à un seul fil d'exécution 63
4.2 Modèle à un fil d'exécution par session RTP avec E/S non-bloquantes 63
4.3 Modèle à un fil d'exécution par session RTP avec E/S bloquantes 64
4.4 Modèle à un fil d'exécution par unité de traitement 65
4.5 Modèle basé sur le mécanisme *async* 66
4.6 Modèle de tâches de base 67
4.7 Modèle hybride avec fils d'exécution et tâches 68
4.7.1 Fils d'exécution et tâches en parallèle 68
4.7.2 Fils d'exécution et tâches séquentielles 68
4.8 Analyse des résultats 69
Conclusion 71

Liste des tableaux

Tableau 1 Présentation des métriques 62
Tableau 2 Résultats pour le modèle à un fil d'exécution 63
Tableau 3 Résultats pour le modèle à un fil d'exécution par session avec E/S non-bloquantes 64
Tableau 4 Résultats pour le modèle à un fil d'exécution par session avec E/S bloquantes 65
Tableau 5 Résultats pour le modèle à un fil d'exécution par unité de traitement 65
Tableau 6 Résultats pour le modèle à deux fils d'exécution par unité de traitement 66
Tableau 7 Résultats pour le modèle à cinq fils d’exécution par unité de traitement 66
Tableau 8 Résultats pour le modèle basé sur le mécanisme *async* 67
Tableau 9 Résultats pour le modèle de tâches de base 67
Tableau 10 Résultats pour le modèle de tâches de base avec surutilisation 68
Tableau 11 Résultats pour le modèle hybride à fils d’exécution et tâches en parallèle 68
Tableau 12 Résultats pour le modèle hybride à fils d’exécution et tâches séquentielles 69

Liste des figures

Figure 1 Ordonnanceur dans son contexte 9
Figure 2 Hiérarchie de la mémoire d’un processeur SMP 12
Figure 3 Cohérence des antémémoires d’un processeur SMP 14
Figure 4 Architecture d’un processeur NUMA 19
Figure 5 Vue simplifiée d’un modèle de tâches 24
Figure 6 Vue simplifiée d'un modèle basé sur les acteurs 26
Figure 7 Bibliothèques incluses et exploitant le *Concurrency Runtime* 29
Figure 8 Resource Manager 31
Figure 9 Vue du système en référence dans son contexte 40
Figure 10 Opérations séquentielles du système en référence 42

Glossaire

Accélération (*speedup*)	Dans le contexte du parallélisme, l'accélération représente les gains obtenus en termes de vitesse d'exécution d'un système, ou algorithme, une fois parallélisé par rapport à son équivalent séquentiel.
Asynchrone	Technique qui consiste à ne pas bloquer l'exécution d'un fil d'exécution suite à l'appel d'une fonction dont la réponse peut prendre un certain délai avant d'être disponible, permettant plutôt de recevoir la réponse ultérieurement.
Antémémoire	Mémoire d'accès rapide, servant d'intermédiaire entre un processeur et la mémoire principale, qui est généralement plus lente.
Bloc d'antémémoire	Représente une partie – un bloc – de données de la mémoire principale copiées en antémémoire (en anglais : *cache line*).
Changement de contexte	Mécanisme consistant à sauvegarder puis restaurer l'état d'un fil d'exécution, pour reprendre son exécution au même point ultérieurement, permettant ainsi d'exécuter à tour de rôle plus d'un fil d'exécution sur une même unité de traitement.
Codec	Technique permettant d'encoder un signal analogique en un signal numérique, selon le type de codec. Certains codecs permettent également de compresser un signal numérisé, permettant ainsi de transmettre sensiblement le même signal mais en transmettant moins de données.
Concurrence	La concurrence et le parallélisme sont des concepts distincts, mais avec l'objectif commun de faire en sorte que l'exécution de certaines parties d'un système s'effectuent en même temps, ou du moins en donner l'impression.

Cohérence des antémémoires	Algorithmes permettant d'éviter qu'une ou plusieurs antémémoires ne contiennent une copie différente d'une même donnée de la mémoire principale. S'applique aux architectures pour lesquelles une ou plusieurs antémémoires sont présentes par cœur du processeur.
Contention	Il y a contention lorsqu'une ressource est en surutilisation. Par exemple, il peut y avoir une contention sur la mémoire principale lorsque plusieurs cœurs tentent d'y accéder simultanément.
CPU-Bound	Signifie que le temps requis à l'exécution d'une tâche repose principalement sur la vitesse du processeur, ou cœurs.
Déterministe	Un algorithme est dit déterministe lorsque le résultat qu'il produit pour un ensemble d'intrants donné est toujours le même, indépendamment du contexte.
Fil d'exécution	Fil d'exécution est la traduction française pour *thread*, permettant l'exécution d'une partie d'un système en parallèle avec d'autres fils d'exécution selon la disponibilité des unités de traitement.
Framework	Un ensemble de bibliothèques ou de classes réutilisable.
Gain en efficacité	Un gain en efficacité dans le contexte du parallélisme signifie qu'un traitement se fait en moins de temps que pour un équivalent séquentiel, ou encore que plus de traitements peuvent être faits dans un même laps de temps une fois le système parallélisé.
IO-Bound	Signifie que le temps requis à l'exécution d'une tâche repose principalement sur l'attente d'opérations d'entrées/sorties.
Mécanisme de synchronisation	Mécanisme permettant de protéger une ressource – comme une instance d'une classe – partagée entre plusieurs fils d'exécution contre les accès concurrents.

Modèle d'acteurs	Modèle permettant la mise en place du parallélisme sans que des données ne soient partagées. Les acteurs communiquent plutôt en échangeant des messages asynchrones.
Ordonnanceur coopératif	Un ordonnanceur coopératif n'interrompt pas l'exécution d'un fil d'exécution; il les laisse plutôt procéder jusqu'à ce qu'ils soient complétés, ou jusqu'à ce qu'ils tentent d'accéder une ressource système non disponible.
Ordonnanceur préemptif	Un ordonnanceur préemptif initie lui-même les changements de contexte, dans le but de donner à tour de rôle du temps d'exécution aux différents fils d'exécution sur les différentes unités de traitement.
Parallélisation	Action de rendre parallèle, voir parallélisme.
Parallélisme	Consiste à répartir le traitement des opérations d'un système sur plusieurs unités de traitement afin qu'elles puissent être exécutées en parallèle.
Pseudo temps réel	Contrairement aux systèmes temps réel, pour lesquels des délais stricts doivent être respectés, les systèmes pseudo temps réel tentent le plus possible de respecter leurs délais, sans toutefois qu'un non-respect de ceux-ci n'entraîne une défaillance du système. Les systèmes de transmission d'audio-vidéo en sont un exemple : bien qu'une cadence – délai – doive être respectée, le non-respect de celle-ci entraîne une perte de qualité, sans plus.
Tâche	Unité de code exécutable responsable d'un travail précis et dont l'exécution ne dépend généralement pas d'une autre tâche, permettant ainsi de paralléliser l'exécution de plusieurs tâches.
Transcoder	Le passage d'un codec vers un autre.

Unité de traitement	Ce qui permet l’exécution de code, par exemple un cœur d’un processeur multi-cœurs.

Liste des sigles, symboles et acronymes

ASMP	*Asymmetric Multi-Processing* : multiprocesseur asymétrique
CMP	*Chip Multi-Processing* : circuits composés de plusieurs processeurs
GPU	*Graphics Processing Unit* : unité de traitement graphique
IPC	*Inter-Process Communication* : communication interprocessus
NUMA	*Non-Uniform Memory Access* : accès mémoire non uniforme
SMP	*Symmetric Multi-Processing* : multiprocesseur symétrique
RTP	*Real-Time Transport Protocol* : protocole de communication permettant le transport de données en temps réel

Introduction

Depuis que la vitesse maximale possible de l'horloge des processeurs a cessé d'augmenter de façon significative [1], les fabricants de processeurs se sont tournés vers la conception de circuits composés de plusieurs processeurs, nommés les processeurs multi-cœurs, de sorte qu'en 2014, la majorité des ordinateurs conçus sont constitués de tels processeurs. Par conséquent, l'amélioration de la vitesse d'exécution des systèmes ne se fait plus simplement en remplaçant un processeur par un autre plus rapide, mais plutôt en tirant profit de cette nouvelle génération de processeurs [2]. Les processeurs multi-cœurs sont maintenant la norme, permettant de contourner les limites de la vitesse de leur horloge interne.

De plus, la tendance depuis 2011, et probablement pour la prochaine décennie [22], ne repose plus simplement sur l'exploitation des processeurs multi-cœurs, mais également sur l'exploitation de toutes les unités de traitement offertes par un ordinateur : processeurs multi-cœurs et GPU (*Graphics Processing Unit*) [23], et bien au-delà de cela lorsque connecté à un réseau. Par exemple, une fois connecté à Internet ou à un réseau privé, un système peut utiliser les unités de traitement des autres périphériques (téléphones intelligents, tablettes numériques) mis à sa disposition. Les téléphones intelligents et les tablettes numériques sont munis, en 2014, de deux à quatre cœurs, et un simple ordinateur portable a entre quatre et huit cœurs; il est ainsi fort probable que, d'ici 2020, à travers quelques ordinateurs et périphériques, des centaines voire des milliers d'unités de traitement soient disponibles pour un système capable de les exploiter efficacement. Sans oublier les environnements infonuagiques, dans lesquels un système peut avoir accès sur demande à une très grande quantité d'unités de traitement, à condition qu'il puisse les exploiter convenablement.

Toutefois, la plupart des systèmes existants n'ont pas été conçus pour exploiter efficacement plusieurs unités de traitement, et ne voient que peu ou pas d'amélioration significative lorsqu'ils s'exécutent sur plusieurs unités de traitement, à l'exception de systèmes développés avec des langages de programmation pour lesquels le parallélisme est omniprésent (p. ex. : les langages fonctionnels tels que Erlang et Haskell). Se familiariser avec les nouvelles techniques

et bibliothèques de parallélisme devient attrayant [2] pour la mise en place du parallélisme dans les systèmes ne l'exploitant pas, ou l'exploitant peu.

De façon générale, pour tirer profit des processeurs multi-cœurs, il faut mettre en valeur le parallélisme dans les systèmes. Cet essai met l'accent sur la mise en place du parallélisme à travers l'exploitation de plusieurs unités de traitement, et non sur les techniques de répartition des parties d'un système sur des unités de traitement (p. ex. : mécanismes de détection de telles unités ou de transfert de parties d'un système sur ces unités). L'exploitation de plusieurs unités de traitement est souvent obtenue avec l'utilisation de fils d'exécution, appelés *threads* en anglais. Les fils d'exécution sont au cœur des nouvelles techniques et bibliothèques en place pour faciliter le parallélisme, comme les tâches §1.7.1 et le modèle d'acteurs §1.7.2.

Conséquemment, cet essai fait l'exercice de prendre un système en référence et d'analyser quels aspects de ce système pourraient être parallélisés pour exploiter efficacement les unités de traitement disponibles selon l'environnement sur lequel le système s'exécute. Un système qui tire profit du parallélisme est un système qui s'adapte à son environnement, à la disponibilité des unités de traitement, disponibilité qui peut varier d'un environnement à l'autre ou même pendant l'exécution du système sur un seul et même environnement (p. ex. : qu'un système ne soit pas seul à s'exécuter sur une machine fait en sorte que plusieurs systèmes devront se partager et s'adapter à la disponibilité des unités de traitement).

Le système en référence tout au long de cet essai s'introduit au milieu de conversations audio transmises sur un réseau IP, tel qu'Internet, afin de transcoder l'audio d'un codec vers un autre codec. Ce type de système a été choisi pour ses aspects intéressants à la mise en place du parallélisme : il requiert une charge de traitement élevée, avec des contraintes de pseudo temps réel, tout en sollicitant de manière importante les entrées/sorties. La mise en place du parallélisme dans un tel système permet une utilisation efficace des différentes unités de traitement, augmentant le nombre de conversations concurrentes pouvant être traitées par le système en fonction du nombre d'unités de traitement sur lequel le système s'exécute. Les techniques présentées dans cet essai s'appliquent à d'autres types de systèmes, comme les

systèmes infonuagiques et les systèmes répartis [41] pour lesquels une quantité élevée de requêtes doivent être desservies simultanément.

Le premier chapitre se veut une mise à niveau présentant les concepts, techniques et outils servant à la mise en place d'une architecture logicielle capable de profiter du parallélisme.

Le deuxième chapitre présente et analyse en détail les outils nouvellement mis en place par plusieurs bibliothèques pour faciliter l'introduction du parallélisme. Ces bibliothèques spécialisées sont disponibles pour une variété de plateformes et de langages de programmation, et les concepts de bases se recoupent d'une bibliothèque à l'autre. Ainsi, le choix de la bibliothèque présentée dans ce chapitre, le *Concurrency Runtime* de Microsoft, n'est qu'un accessoire permettant de présenter certains concepts prédominants du parallélisme, dont l'utilisation d'un ordonnanceur coopératif §1.3.1 par-dessus l'ordonnanceur préemptif du système d'exploitation, la mise en place du parallélisme à travers les tâches §1.7.1, ainsi que des façons d'intégrer à un système les techniques et concepts présentés au chapitre 1.

Le troisième chapitre porte sur les différentes techniques de migration des systèmes existants vers une architecture permettant de tirer avantage de plusieurs unités de traitement. Ces techniques de mise en place du parallélisme sont démontrées à l'aide du système en référence, dans l'optique de présenter l'impact des différentes techniques sur le système. Ce chapitre démontre que connaître les techniques et bibliothèques appropriées permet d'exploiter efficacement plusieurs unités de traitement.

Le quatrième et dernier chapitre présente une analyse comparative des résultats obtenus avant la mise en place du parallélisme et suite à sa mise en place dans le système en référence; selon les approches présentées au troisième chapitre. Les principales métriques utilisées pour cette analyse comparative sont : la latence introduite entre la réception d'un paquet sur le réseau et la transmission de ce dernier une fois transcodé, l'utilisation moyenne du CPU, ainsi que la quantité de défauts de cache (*Cache Misses*) et de changements de contexte générés.

Chapitre 1
Parallélisme

Le parallélisme consiste à répartir le traitement des opérations d'un système sur plusieurs unités de traitement (p. ex. : cœurs d'un processeur multi-cœurs) afin qu'elles puissent être exécutées en parallèle, et ainsi accroître les performances du système. Accroître les performances d'un système signifie, dans un contexte de parallélisme, utiliser efficacement plusieurs unités de traitement pour faire un traitement en moins de temps que sur une seule d'entre elles, ou encore faire plus de traitements dans un même laps de temps. Ainsi, l'accroissement des performances suite à l'ajout de parallélisme peut se mesurer selon les gains obtenus en terme de vitesse d'exécution (ou *speedup* en anglais).

Les gains obtenus en terme de vitesse d'exécution du système est en fait le ratio du système parallèle par rapport à sa version séquentielle lors du passage d'une unité de traitement à n unités de traitement. Par exemple, si un système séquentiel s'exécute en 1000 secondes alors qu'il s'exécute en 200 secondes une fois parallélisé sur huit unités de traitement, alors le gain obtenu en vitesse d'exécution est de 5 :

$$S(P) = \frac{R(1)}{R(P)}$$

Où $S(P)$ est l'accélération avec P unités de traitement et $R(P)$ est le temps d'exécution toujours pour P unités de traitement [25].

Une seconde équation sert à calculer l'efficacité du système parallèle en comparaison avec sa version séquentielle en fonction du nombre d'unités de traitement. Ainsi, pour calculer l'efficacité d'un système parallèle, il suffit de diviser le gain obtenu en vitesse d'exécution ci-haut par le nombre d'unités traitement, soit 5 divisé par 8 cœurs dans l'exemple ci-haut, qui donne une efficacité de 63 % :

$$E(P) = \frac{S(P)}{P} \times 100$$

Où $E(P)$ est le pourcentage d'efficacité en fonction de l'accélération $S(P)$ pour P unités de traitement.

Toutefois, la plupart des systèmes sont composés en partie d'opérations qui doivent absolument être exécutées dans un ordre séquentiel strict [14][25][25][14], ce qui implique qu'il puisse y avoir une dépendance entre certaines opérations d'un système :

$$R(P) = Seq + Par(P)$$

Où Seq est la partie fondamentalement séquentielle d'un système et $Par(P)$ la partie parallélisable sur P unités de traitement.

La suite de ce chapitre présente les différentes architectures les plus répandues de processeurs multi-cœurs qui se retrouvent dans les ordinateurs en 2014, et fait état de pièges à éviter afin d'être en mesure d'exploiter pleinement le parallélisme sur ces architectures.

1.1 Historique

Jusqu'au début du 21e siècle, les gains en terme de vitesse d'exécution des processeurs, jumelés aux raffinements des optimisations réalisées par les compilateurs, ont fait en sorte que la vitesse d'exécution des systèmes existants puisse augmenter constamment, et ce, peu importe si les systèmes exploitaient ou non le parallélisme. Depuis, l'augmentation de la vitesse d'exécution des processeurs a atteint un mur, nommé le *Power Wall* [4]. Le terme *Power Wall* vient du fait suivant : lorsque la vitesse d'exécution d'un processeur atteint un certain seuil, la quantité d'énergie requise pour le faire fonctionner à cette vitesse croît de façon exponentielle plutôt que linéaire. Dû à ce problème de consommation énergétique, les fabricants de processeurs se sont tournés vers des circuits capables de traiter des instructions parallèlement sur plusieurs cœurs, d'où l'apparition des processeurs multi-cœurs [1], aussi nommés les CMP (*Chip Multi-Processing*).

Cette situation fait en sorte que, pour la première fois, l'amélioration des performances des systèmes existants ne passe plus par une simple substitution d'un processeur par un autre, plus rapide, mais plutôt par l'exploitation de plusieurs cœurs.

1.2 Types de systèmes pouvant bénéficier du parallélisme

Introduire du parallélisme dans un système existant n'est pas forcément une tâche simple : si le parallélisme est mal implanté, les performances du système peuvent se dégrader [6]. De plus, le parallélisme peut introduire de nouvelles catégories de problèmes [24], nuisant à la robustesse et à la stabilité du système. Les difficultés et les problèmes pouvant découler de la mise en place du parallélisme sont couverts à partir de la section 1.3 de ce chapitre.

Il faut également noter que ce ne sont pas tous les systèmes qui ont des besoins élevés en performance, tout comme ce ne sont pas tous les systèmes qui peuvent offrir des gains intéressants en terme de vitesse d'exécution sans une refonte importante. Voici, à titre d'exemple, des types de systèmes pour lesquels la mise en place du parallélisme est dorénavant presque indispensable :

- Les systèmes qui agissent en tant que serveurs, voués à desservir des centaines, voire des milliers de requêtes simultanément.

- Les systèmes en temps réel, qui doivent faire différents traitements tout en garantissant une cadence régulière.

- Les systèmes qui ont besoin de traiter une quantité importante de données, et ce, le plus rapidement possible. Par exemple, les systèmes de forage de données, les systèmes neuronaux, les systèmes de reconnaissance de la parole, et bien d'autres.

Il serait tentant de croire qu'un système qui exploite efficacement le parallélisme verra sa vitesse d'exécution s'accroître presque de façon linéaire en fonction du nombre de cœurs du processeur. Cependant, il est rare que la totalité d'un système puisse être parallélisée. Tel que mentionné en début de chapitre, tout système a une proportion séquentielle Seq incompressible. Ainsi, de façon très simplifiée, plus petite est Seq, plus nombreuses sont les

opérations indépendantes d'un système et plus ce système tend à offrir des gains de performances linéaires en fonction du nombre de cœurs du processeur lorsque le parallélisme y est bien implanté. Différents facteurs peuvent nuire à un tel accroissement de la performance; ces facteurs sont principalement reliés aux accès concurrents par les différents cœurs aux ressources partagées telles que la mémoire et les entrées/sorties, une problématique discutée en détail à §1.3.

Le paragraphe précédent se résume bien par la loi d'Amdahl [14], c'est-à-dire que les gains en terme de vitesse d'exécution qu'un système peut obtenir en s'exécutant sur un processeur multi-cœurs sont limités par la proportion Seq du système. Par exemple, un système s'exécutant sur huit cœurs et dont 20 % des opérations s'exécutent séquentiellement aura, selon Amdahl, au mieux un gain en terme de vitesse d'exécution de 1 / (0,20 + (0,80/8)) = 3,33 :

$$M(P) = \frac{1}{Seq + (\frac{Par(P)}{P})}$$

Où $M(P)$ est le pourcentage de gain maximal en terme de vitesse d'exécution pouvant être obtenu sur P unités de traitement.

Ainsi, toujours selon Amdahl, si un système prend dix heures à s'exécuter sur un processeur à un seul cœur et si 20 % de ses opérations doivent s'exécuter séquentiellement, alors le système ne pourra pas s'exécuter en moins de deux heures, et ce, peu importe le nombre de cœurs utilisés.

Plusieurs critiques ont été faites au sujet de la loi d'Amdahl, dont Gustafson [29], qui propose sa propre loi; [25] présente d'ailleurs quelques façons d'outrepasser la loi d'Amdahl.

1.3 Concurrence et parallélisme

La concurrence et le parallélisme sont des concepts distincts, mais avec l'objectif commun de faire en sorte que l'exécution de certaines parties d'un système s'effectuent en même temps,

ou du moins en donner l'impression. L'une des principales différences entre concurrence et parallélisme repose sur le fait que la concurrence peut être réalisée sur une seule unité de traitement, tandis que le parallélisme requiert plusieurs unités de traitement (soit plusieurs processeurs ou un processeur à multi-cœurs). La concurrence peut donc être vue comme le partage de ressources communes entre différents systèmes ou différentes parties d'un même système, et ainsi donner une impression de parallélisme, tandis que le parallélisme est l'exécution en parallèle de différentes parties d'un même système sur plusieurs unités de traitement.

Les systèmes d'exploitation tels que Windows, Linux et Mac OS X supportent la concurrence depuis longtemps, soit bien avant l'apparition des processeurs multi-cœurs. La concurrence dans les systèmes d'exploitation repose en grande partie sur les changements de contexte [5], mécanisme consistant à sauvegarder et restaurer l'état d'un processus, ou fil d'exécution, pour reprendre son exécution au même point ultérieurement. De même, puisqu'en 2014, le nombre de fils d'exécution de tous les processus qui s'exécutent sur un ordinateur est en général largement plus élevé que le nombre de cœurs et de processeurs présents dans l'ordinateur, le parallélisme repose également sur une gestion des changements de contexte.

1.3.1 Ordonnanceur

Différents types d'algorithmes sont utilisés par les systèmes d'exploitation pour gérer les changements de contexte. Ces algorithmes sont généralement implantés dans l'ordonnanceur (*scheduler*) du système d'exploitation, et sont soit préemptifs, soit coopératifs. La majorité des systèmes d'exploitation « grand public » d'aujourd'hui utilisent un ordonnanceur basé sur un type d'algorithme préemptif. La figure suivante présente un ordonnanceur de type préemptif dans son contexte :

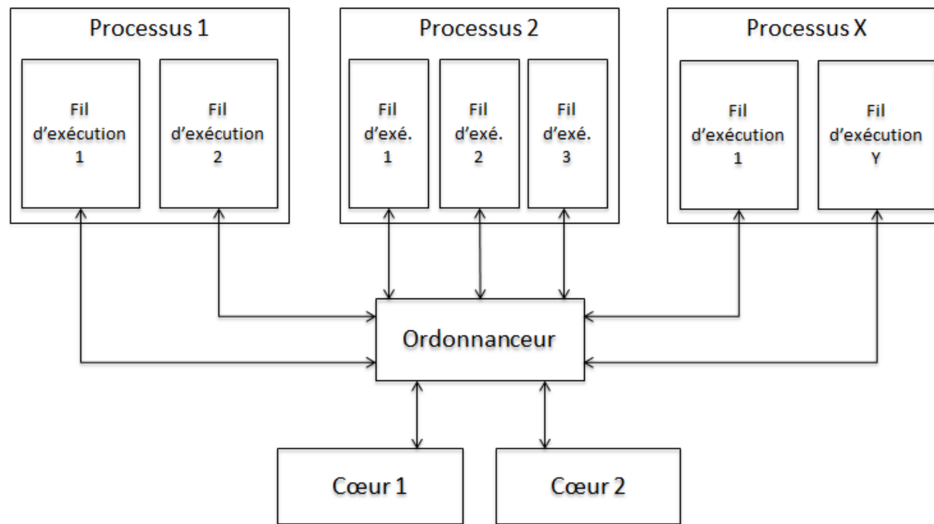

Figure 1 Ordonnanceur dans son contexte

Un ordonnanceur préemptif initie lui-même les changements de contexte, dans le but de donner à tour de rôle du temps d'exécution aux différents fils d'exécution sur un processeur ou sur les différents cœurs d'un processeur. La probabilité qu'un fil d'exécution soit choisi est généralement en fonction de sa priorité.

Contrairement à un ordonnanceur préemptif, un ordonnanceur coopératif n'interrompt pas l'exécution d'un fil d'exécution; il laisse plutôt les fils d'exécution procéder jusqu'à ce qu'ils soient complétés, ou jusqu'à ce qu'ils tentent d'accéder une ressource système non disponible (p. ex. : tenter d'acquérir un *mutex* déjà acquis par un autre fil d'exécution).

À travers les changements de contexte, les ordonnanceurs préemptifs ont l'avantage de donner l'impression d'exécuter plusieurs fils d'exécution simultanément. Cependant, la gestion des changements de contexte impacte le temps d'exécution sous deux aspects. Le premier tient à la permutation des fils d'exécution sur les différents cœurs du processeur. Cette permutation implique plusieurs étapes, dont la sauvegarde et le remplacement des valeurs des registres d'un fil d'exécution par les valeurs d'un autre fil d'exécution, le nettoyage du pipeline du processeur, la sauvegarde et le remplacement de la pile d'exécution d'un fil d'exécution par celle d'un autre fil d'exécution. Ces étapes accèdent à la mémoire principale pour la sauvegarde et le chargement des données relatives aux fils d'exécution. Conséquemment,

pendant ces accès, le processeur n'exécute pas les instructions des différents fils d'exécution; §1.5.1 présente la hiérarchie de la mémoire tout en donnant une idée des coûts reliés au passage d'un niveau à l'autre d'antémémoire jusqu'à la mémoire principale.

Le second aspect est en fait un effet de bord du premier : il s'agit de la perturbation d'antémémoire (*cache*) des cœurs du processeur suite aux permutations des fils d'exécution. À chaque fois qu'un fil d'exécution est permuté, l'antémémoire du cœur sur lequel le fil d'exécution s'exécute risque d'être vidée afin de faire place aux données accédées en mémoire par le fil d'exécution qui le remplacera. Plusieurs recherches démontrent que cet aspect a même un plus grand impact sur la performance que le premier [15]. La section 1.5 présente plus en détail le fonctionnement de l'antémémoire, ainsi que l'impact sur les performances que peut avoir leur perturbation.

Le caractère onéreux des changements de contexte rend intéressant, dans certaines situations, le recours à un ordonnanceur coopératif. En effet, ce type d'ordonnanceur permet à l'application de décider du moment où son exécution doit être suspendue, réduisant ainsi le nombre d'occurrences des changements de contexte. La bibliothèque présentée au chapitre 2 permet entre autre d'émuler le comportement d'un ordonnanceur coopératif sur un système d'exploitation dont l'ordonnanceur est préemptif. De son côté, le chapitre 3 donne des exemples concrets pour lesquels l'utilisation d'un ordonnanceur coopératif donne de bons résultats.

1.4 Types d'architectures de processeurs multi-cœurs

Comprendre les architectures des processeurs multi-cœurs aide à faire de meilleurs choix lors de la mise en place du parallélisme dans un système s'exécutant sur ces architectures. Conséquemment, cette section décrit certaines de ces architectures, tout comme §1.5 et §1.6.

Les grandes familles d'architectures de processeurs multi-cœurs sont : ASMP (*Asymmetric Multi-Processing*) et SMP (*Symmetric Multi-Processing*). L'architecture ASMP est principalement utilisée pour le matériel spécialisé tel que les cartes graphiques, et est de moins en moins présente en 2014 dans les processeurs multi-cœurs des ordinateurs personnels grand

public. Pour cette raison, l'architecture ASMP n'est pas couverte par cet essai. Par contre, l'architecture SMP est utilisée par la majorité des ordinateurs à processeur multi-cœurs grand public contemporains (voir §1.5 pour plus de détails). Bien que très répandue, l'architecture SMP présente des lacunes quant au support d'un nombre toujours grandissant de cœurs, d'où l'apparition de l'architecture NUMA (*Non-Uniform Memory Access*), qui est en quelque sorte une évolution de l'architecture SMP. L'exploitation d'une architecture NUMA ajoute cependant des contraintes supplémentaires, présentées à §1.6.

1.5 Architecture SMP

Les cœurs d'un processeur SMP ont la caractéristique principale d'être symétriques, c'est-à-dire que chacun des cœurs est équivalent aux autres, donc ils sont interchangeables. Puisque les différents cœurs d'un processeur SMP sont symétriques, ils peuvent tous :

- travailler sur le même espace mémoire adressable, et sur la même mémoire virtuelle §1.5.1;
- exécuter n'importe quel fil d'exécution du système;
- traiter les interruptions du système;
- initier des opérations d'entrée et de sortie (E/S).

1.5.1 Hiérarchie de la mémoire

Chaque processeur SMP est composé d'au moins deux cœurs, d'un ou plusieurs niveaux d'antémémoire associés à chaque cœur, et d'une mémoire principale partagée par tous les cœurs de ce processeur. Certains processeurs contiennent également un niveau d'antémémoire partagé par tous les cœurs, comme le niveau 3 présenté à la figure suivante.

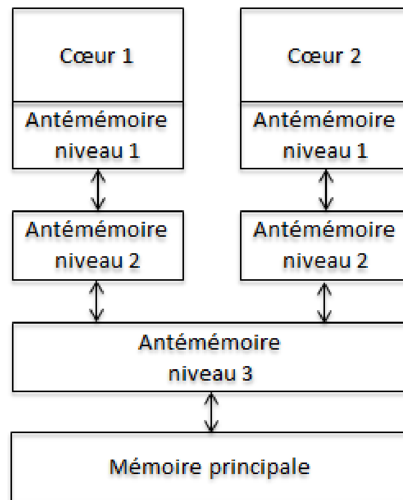

Figure 2 Hiérarchie de la mémoire d'un processeur SMP

Chaque niveau d'antémémoire se distingue principalement par le temps requis à un cœur pour y accéder, par ses capacités, ainsi que par sa complexité. L'antémémoire la plus près du cœur est celle qui offre les meilleurs temps d'accès; par contre, c'est également celle qui offre le moins de capacité. Conséquemment, l'antémémoire a pour rôle de pallier le fait que les opérations sur la mémoire principale sont très lentes; elle repose entre le ou les cœurs, qui sont très rapides, et la mémoire principale. À titre d'exemple, le processeur Intel i7-8xx [28] offre 32 Ko d'antémémoire de niveau 1 par cœur, 256 Ko de niveau 2 par cœur, et 8 Mo au niveau 3, partagé entre tous les cœurs. Pour mettre en contexte la lenteur des opérations sur la mémoire principale, il faut savoir qu'une opération sur une antémémoire de niveau 1 prend approximativement quatre cycles à un cœur, onze pour le niveau 2 et 39 pour le niveau 3, tandis qu'une opération sur la mémoire principale requiert approximativement 107, toujours pour un processeur Intel i7-8xx.

Sans entrer dans le détail de la gestion des différents niveaux d'antémémoires, il est important de présenter le format des entrées en antémémoire (*cache line*). Dû à la taille d'une page mémoire, qui est généralement de 4 Ko (p. ex. : sous Windows 7 et Linux), il serait inefficace de copier une page mémoire entière en antémémoire, tout comme il serait inefficace de copier un mot à la fois (soit 32 ou 64 bits selon l'architecture). Conséquemment, les données sont généralement copiées de la mémoire principale vers l'antémémoire en bloc de taille fixe, p.

ex : 64 octets – la taille exacte dépend du processeur – permettant ainsi de tirer avantage de l'aspect « localité » des données §1.5.2.

1.5.2 Localité des données

L'antémémoire fonctionne selon les suppositions suivantes :

- Localité temporelle : une donnée accédée de la mémoire a une forte probabilité d'être accédée de nouveau dans un avenir rapproché.

- Localité spatiale : une donnée localisée en mémoire près d'une donnée venant d'être accédée a aussi de fortes probabilités d'être accédée dans un avenir rapproché.

L'importance pour un système d'avoir une bonne utilisation d'antémémoire, qui passe forcément par une bonne gestion de la localité de ses données, est présentée dans les sections suivantes. Certaines techniques pour aider à y parvenir sont également présentées.

1.5.3 Cohérence des antémémoires

Puisque chaque cœur a son antémémoire, et puisque celle-ci reflète les états de différentes adresses de la mémoire principale, il est possible qu'au même moment, plusieurs antémémoires contiennent chacune une copie d'une même donnée de la mémoire principale.

Conséquemment, si deux fils d'exécution s'exécutent en parallèle sur deux cœurs différents, puis si un premier fil d'exécution modifie une donnée de son antémémoire, alors qu'immédiatement après, le second fil d'exécution accède cette même donnée, mais de sa propre antémémoire, il y a un risque que le second fil d'exécution obtienne une valeur désuète de la donnée puisque l'état de celle-ci dans son antémémoire pourrait ne pas avoir encore été mise à jour par le processeur.

Heureusement, pour contrer ce problème, les nouvelles générations de processeurs multi-cœurs SMP implémentent différents algorithmes assurant la cohérence des antémémoires. Cette cohérence des antémémoires entraîne évidemment un coût puisqu'elle augmente le trafic entre les différentes antémémoires et la mémoire principale afin de synchroniser les

antémémoires. La figure suivante présente de façon simplifiée l'architecture d'un processeur multi-cœurs SMP :

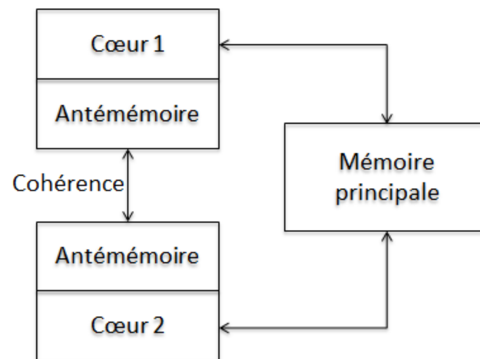

Figure 3 Cohérence des antémémoires d'un processeur SMP

Il existe plusieurs algorithmes pour qu'un processeur multi-cœurs assure la cohérence des données présentes dans les différentes antémémoires [9]. La plupart de ces algorithmes fonctionnent de la façon suivante : lorsqu'une entrée est modifiée dans une des antémémoires d'un processeur multi-cœurs, l'algorithme invalide les entrées dupliquées de celle-ci dans les autres antémémoires du processeur. Ainsi, la prochaine fois que cette entrée sera lue dans les autres antémémoires, elle sera d'abord copiée de la mémoire principale vers l'antémémoire; il sera donc impossible d'avoir une incohérence des données entre les différentes antémémoires.

1.5.4 Défauts de cache

À chaque fois qu'une entrée doit être lue par un processeur et qu'elle ne se trouve pas dans l'antémémoire, ou encore lorsqu'une telle entrée a été invalidée, par exemple, suite à un algorithme de cohérence des antémémoires §1.5.3, le processeur copie d'abord l'entrée de la mémoire principale vers l'antémémoire (voir *cache line* §1.5.1). Cette étape est relativement coûteuse du point de vue de la performance et est connue sous le terme défaut de cache (*Cache Miss*). Le coût tient au fait qu'une lecture dans la mémoire principale est beaucoup plus lente qu'une lecture dans une antémémoire, mais aussi au fait que cette étape augmente le trafic sur le bus d'accès à la mémoire principale, ce qui crée un engorgement de ce bus lorsque les défauts de cache sont fréquents. Un engorgement du bus d'accès à la mémoire principale peut ralentir tous les cœurs du processeur, car il n'est pas rare que les instructions à exécuter

par les différents cœurs doivent accéder de l'information en mémoire principale, et l'accès à cette mémoire est beaucoup plus lent lorsque bus principal est engorgé §1.5.1.

Conséquemment, un système qui exploite moins bien le parallélisme, mais qui génère moins de défauts de cache peut offrir de meilleures performances qu'un système exploitant mieux le parallélisme, mais causant plus de défauts de cache. De plus, un système qui génère une quantité importante de défauts de cache peut avoir de la difficulté à exploiter pleinement plusieurs cœurs d'un processeur, car les différents cœurs pourraient être trop souvent en attente d'informations provenant de la mémoire principale. Ainsi, lorsqu'un système est développé en fonction d'une bonne utilisation de l'antémémoire, les différents cœurs du processeur peuvent traiter plus d'opérations dans un temps donné car ils sont moins souvent en attente sur des données se trouvant dans la mémoire principale.

Pour s'efforcer de diminuer le nombre de défauts de cache, les ordonnanceurs d'aujourd'hui implémentent des algorithmes qui ont pour but d'utiliser le plus efficacement possible les différents cœurs du processeur. Un de ces algorithmes se nomme *High Cache Affinity*, ou *CPU Pinning* [49], et consiste à exécuter, lorsque possible, un fil d'exécution sur un cœur récemment utilisé par ce fil d'exécution. Conséquemment, une partie des données utilisées par ce fil d'exécution pourraient être toujours présentes dans l'antémémoire du cœur lorsque le même fil d'exécution y sera à nouveau exécuté.

Bien que les algorithmes des ordonnanceurs aident à diminuer la quantité de défauts de cache, il n'en demeure pas moins que certaines précautions peuvent être prises afin qu'un système génère le moins de défauts de cache possible §1.5.5. Il peut évidemment être ardu de développer un système complet avec une utilisation optimale de l'antémémoire en tout temps, surtout lorsque le système est complexe. Cependant, en connaissant l'impact sur la performance des défauts de cache, il peut être avantageux de trouver les endroits spécifiques dans un système où une gestion optimale de l'antémémoire peut donner des gains en terme de vitesse d'exécution.

Pour ce faire, il faut connaître les principes de base utilisés par les systèmes d'exploitation pour la gestion d'antémémoire. Il existe plusieurs algorithmes de gestion (insertion et suppression) des données dans une antémémoire [13]. La majorité de ces algorithmes fonctionnent de la façon suivante (vision simplifiée) : les données sont insérées par blocs (souvent de 64 octets §1.5.1) de la mémoire principale vers l'antémémoire; pour faire place à un nouveau bloc lorsque l'antémémoire est remplie, le bloc le plus vieux (soit celui qui a été accédé il y a le plus longtemps) est retiré. Il existe différentes variations de cet algorithme, qui prennent par exemple aussi en compte la fréquence d'utilisation des données d'un bloc en antémémoire. Il faut retenir qu'en règle générale, lorsque les données d'un bloc sont souvent lues en antémémoire, ce bloc devrait rester plus longtemps en antémémoire que les blocs moins lus.

1.5.5 Concepts généraux pour une gestion efficace d'antémémoire

Une technique pour réduire les défauts de cache est d'avoir des algorithmes qui travaillent de façon à réutiliser les données déjà en antémémoire. En analysant les boucles imbriquées d'un système, il peut y avoir des cas apparents où une simple réorganisation du travail fait dans une série de boucles imbriquées, de manière à mieux réutiliser les données déjà en antémémoire, réduit considérablement la quantité de défauts de cache générés; [10] en présente d'ailleurs un exemple intéressant.

Une autre méthode à considérer pour limiter les défauts de cache est d'éviter d'avoir plus d'un fil d'exécution qui travaille sur des données susceptibles de se trouver dans un même bloc une fois en antémémoire si au moins une de ces données est accédée en écriture. Lorsque deux fils d'exécution s'exécutent en parallèle sur deux cœurs et manipulent des données situées dans un même bloc en antémémoire, la copie de ce bloc de l'antémémoire d'un cœur est invalidée par chaque modification apportée à ce bloc par le fil d'exécution qui s'exécute sur l'autre cœur, et vice versa. Ceci est très pénalisant en terme de vitesse d'exécution car l'antémémoire de chacun des cœurs doit constamment être rafraîchie avec les données de la mémoire principale.

Le paragraphe précédent se résume bien par l'exemple suivant. En supposant que les deux répétitives de cet exemple s'exécutent en parallèle sur deux cœurs, et qu'elles mettent chacune

à jour une variable globale différente, il serait tentant de croire que le temps d'exécution requis soit sensiblement deux fois plus court sur un processeur à double cœurs que sur un processeur à simple cœur. Cela peut effectivement être le cas, à condition que les variables countModulo3 et countModulo7 soient assez éloignées l'une de l'autre en mémoire principale (p. ex. : 64 octets, voir 1.5.1) pour ne pas qu'elles se retrouvent dans un même bloc une fois dans l'antémémoire de chacun des cœurs.

```
long countModulo3 {};
long countModulo7 {};
Concurrency::parallel_invoke(
  [&countModulo3, number] {
    for(int i = 0; i < number; ++i)
      if ((i % 3) == 0) {
        countModulo3 += 1;
      }
  },
  [&countModulo7, number] {
    for(int i = 0; i < number; ++i)
      if ((i % 7) == 0) {
        countModulo7 += 1;
      }
  }
);
```

Cependant, si les variables countModulo3 et countModulo7 se retrouvent dans un même bloc en antémémoire (comme cela risque d'être le cas dans l'exemple ci-haut), les modifications faites en parallèle (à travers l'utilisation de parallel_invoke §2.2.2) aux variables countModulo3 et countModulo7 invalident l'antémémoire de l'autre cœur. Ceci réduit la vitesse d'exécution en générant du trafic supplémentaire sur le bus d'accès à la mémoire principale. Cette dernière problématique est largement documentée [27] et connue sous le nom des faux partage (en anglais : *false sharing*).

Il existe plusieurs solutions qui permettent de contrer ce type de problème, dont [16] :

- Faire en sorte que les variables susceptibles d'être utilisées en parallèle par plus d'un fil d'exécution soient assez distancées les unes des autres en mémoire pour ne pas qu'elles se retrouvent dans un même bloc une fois en antémémoire, du moins si au moins l'une d'elles sera modifiée fréquemment §1.5.1. Par exemple, pour les variables countModulo3 et countModulo7 de l'exemple ci-haut, il serait possible d'ajouter

délibérément du rembourrage (*padding*) entre celles-ci afin qu'elles soient allouées dans différents blocs une fois en antémémoire. Cette technique est un exemple où il peut être avantageux de consommer plus de mémoire que nécessaire dans l'optique d'améliorer les performances.

- Garder les données fréquemment utilisées par un fil d'exécution le plus près possibles les unes des autres en mémoire, de sorte que ces données soient plus concentrées dans chacun des blocs de l'antémémoire lorsqu'elles sont copiées de la mémoire vers l'antémémoire.

- Garder les données souvent utilisées éloignées des données peu utilisées, car la taille de l'antémémoire est habituellement restreinte. Cela permet d'avoir plus de données fréquemment utilisées dans chacun des blocs en antémémoire.

L'exemple suivant reprend l'exemple précédent, à la différence que les variables countModulo3 et countModulo7 sont alignées sur des blocs de 64 octets. Conséquemment, une fois en antémémoire, ces variables se retrouvent dans des blocs séparés, à condition que la taille des blocs en antémémoire (*cache line*), qui varie d'un processeur à l'autre, n'excède pas 64 octets. Cette technique permet d'éviter les effets négatifs du faux partage sur l'antémémoire.

```
alignas(64) long countModulo3 {};
alignas(64) long countModulo7 {};
Concurrency::parallel_invoke(
  [&countModulo3, number {
    for(int i = 0; i < number; ++i)
      if ((i % 3) == 0) {
        countModulo3 += 1;
      }
  },
  [&countModulo7, number] {
    for(int i = 0; i < number; ++i)
      if ((i % 7) == 0) {
        countModulo7 += 1;
      }
  }
);
```

1.6 NUMA

Tel que présenté à la section 1.5, une architecture SMP est composée d'une seule mémoire principale, partagée par les différents cœurs du processeur, et d'une antémémoire pour chaque cœur. Chaque fois qu'un cœur du processeur accède une donnée qui ne se trouve pas dans son antémémoire, il doit interroger la mémoire principale pour y retrouver cette donnée. Conséquemment, l'architecture SMP compose plutôt mal avec un nombre élevé de cœurs (un processeur SMP dépasse rarement huit cœurs), car avec tous ces accès à la mémoire principale, le bus pour y accéder deviendrait rapidement un goulot d'étranglement et les cœurs seraient souvent en attente d'accès à la mémoire principale, plutôt que d'exécuter des instructions appartenant aux différents fils d'exécution.

L'architecture NUMA (*Non-Uniform Memory Access*) est conçue pour pallier ce problème [17], [39]. Elle y parvient en regroupant un nombre limité de cœurs par nœud, où chaque nœud contient sa propre mémoire. La figure suivante présente à haut niveau l'architecture NUMA :

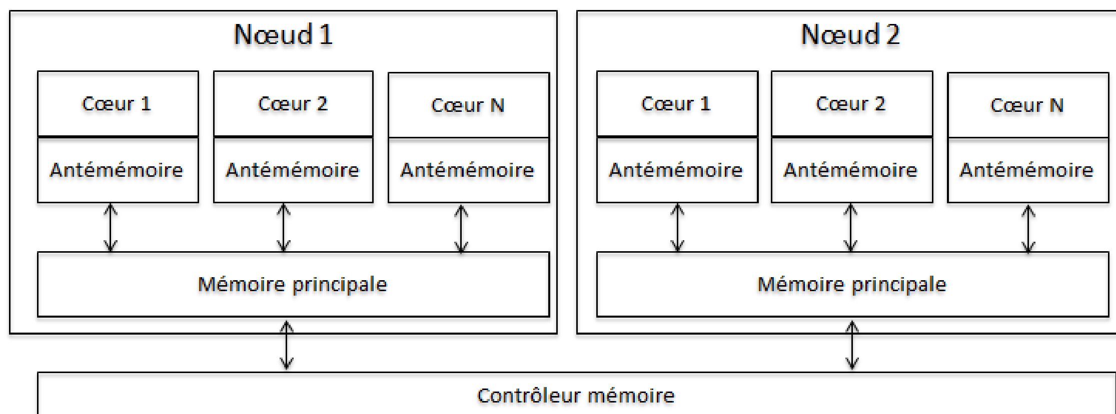

Figure 4 Architecture d'un processeur NUMA

L'architecture à l'intérieur d'un nœud NUMA ressemble à ce qui est présent dans une architecture SMP, mais puisqu'une architecture NUMA contient une mémoire principale pour chaque nœud, un cœur d'un nœud peut au besoin accéder la mémoire principale d'un autre nœud. Cependant, l'accès par un fil d'exécution à des données dans la mémoire d'un autre nœud entraîne un délai. Lorsque les accès inter-nœuds sont fréquents, le bus du contrôleur

mémoire qui se charge de la gestion de ces accès mémoire inter-nœuds peut devenir lui-même un goulot d'étranglement. Conséquemment, un système voulant atteindre de bonnes performances sur une architecture NUMA doit éviter le plus possible ces accès mémoire inter-nœuds discutés plus en détail à §1.6.1.

1.6.1 Gestion de la mémoire entre les différents nœuds

Comme l'architecture NUMA à l'intérieur d'un nœud est similaire à une architecture SMP, tout ce qui est mentionné sur la gestion de la mémoire à §1.5 sur l'architecture SMP s'applique aussi à une architecture NUMA. Ainsi, tout comme pour une architecture SMP, pour exploiter efficacement l'architecture NUMA, un système doit éviter que ses fils d'exécution ne génèrent trop de défauts de cache.

De plus, un système cherchant à exploiter pleinement une architecture NUMA doit éviter que ses fils d'exécution n'accèdent des données susceptibles de se trouver en mémoire principale des autres nœuds. Pour ce faire, il existe plusieurs techniques, qui requièrent un support de l'architecture NUMA par le système d'exploitation. Un système d'exploitation récent, comme Windows 7, offre ce type de support [19]. Les points suivants présentent donc quelques-unes de ces techniques [18] :

- Affinité du processeur (*Processor Affinity*) : cette technique consiste à indiquer au système d'exploitation sur lequel des cœurs du processeur un fil d'exécution doit s'exécuter, ou sur lequel des nœuds dans le cas d'une architecture NUMA. Pour qu'un système soit efficace sur une architecture NUMA, chacun de ses fils d'exécution doit autant que possible accéder la mémoire locale du nœud sur lequel il s'exécute, et éviter d'accéder la mémoire des autres nœuds, d'où l'importance de pouvoir mettre en relation un fil d'exécution avec un nœud NUMA. Cette technique doit cependant être utilisée avec précaution, car il faut éviter de se retrouver dans une situation où une quantité importante de fils d'exécution sont en compétition afin d'avoir du temps d'exécution parmi les différents cœurs d'un nœud NUMA pendant que d'autres nœuds sont inutilisés. Cette technique peut être mise en place par différents moyens, incluant

l'utilisation d'une bibliothèque spécialisée; la bibliothèque *Concurrency Runtime* présentée au chapitre 2 offre entre autre cette possibilité.

- Pour appliquer la technique discutée au point précédent, il faut avoir un certain contrôle sur les allocations de mémoire faites par les fils d'exécution lorsqu'ils s'exécutent. Il faut entre autre qu'un fil d'exécution puisse décider sur lequel des nœuds NUMA ses allocations dynamiques de mémoire doivent être faites, de façon implicite ou explicite. D'une façon implicite, lorsqu'un fil d'exécution fait une allocation mémoire, le système d'exploitation (avec un support minimum de l'architecture NUMA) fait les allocations dynamiques de mémoire dans la mémoire physique du nœud NUMA sur lequel s'exécute le fil d'exécution, à condition qu'il y reste de la mémoire disponible, sinon l'allocation se fera sur la mémoire d'un autre nœud. La façon implicite est intéressante lorsque les fils d'exécution d'un système font leurs propres allocations dynamiques de mémoire qu'ils réutilisent par la suite, ce qui n'est évidemment pas le cas pour tous les systèmes, d'où l'utilité de pouvoir réaliser cette tâche aussi de façon explicite. La façon explicite permet à un fil d'exécution d'allouer de la mémoire sur un autre nœud que celui sur lequel il est exécuté, ce qui se fait généralement à travers une bibliothèque spécialisée, ou directement à travers des appels systèmes (de tels appels sont offerts sur la plupart des systèmes d'exploitation modernes).

1.7 Modèles de parallélisme

Un système qui gère directement la création et la destruction de ses fils d'exécution n'est généralement plus un choix idéal en 2014 pour une mise en place efficace du parallélisme. Le principal problème avec cette approche vient du fait que ce type de système s'adapte souvent mal à son environnement, car la responsabilité d'avoir un nombre optimal de fils d'exécution, selon le nombre d'unités de traitement disponible pour le système, repose sur le programmeur. Un système dont le nombre de fils d'exécution n'est pas assez élevé sous-utilisera les ressources disponibles, et inversement, un système dont le nombre de fils d'exécution est trop élevé fera une surutilisation des ressources disponibles. Dans un cas comme dans l'autre, il y

aura une réduction du taux d'efficacité : avec une sous-utilisation, des unités de traitement sont sous-utilisées; dans le cas d'une surutilisation, les changements de contexte trop fréquents ralentissent l'exécution du système.

Pour contrer les problèmes associés à la gestion des fils d'exécution, les fournisseurs de bibliothèques spécialisées pour le parallélisme proposent des modèles de plus haut niveau qui encapsulent les fils d'exécution; pour une majorité de situations, le programmeur n'a plus à gérer lui-même les fils d'exécution. Ces modèles utilisent généralement un groupe de fils d'exécution (*thread pool*), typiquement créé au démarrage du système et disponible pour la durée de vie de ce système. Un tel groupe gère un nombre de fils d'exécution en fonction du nombre d'unités de traitement disponible de la machine (p. ex. : selon le nombre de cœurs), permettant ainsi d'avoir la quantité optimale de fils d'exécution selon les d'unités de traitement disponibles.

Les bibliothèques spécialisées pour le parallélisme offrent différents modèles simplifiant la mise en place du parallélisme. Une caractéristique importante de ces modèles est l'encapsulation des fils d'exécution par quelque chose de plus simple à utiliser et à gérer. Deux de ces modèles prédominent en 2014, soit le modèle basé sur les tâches §1.7.1, et celui basé sur les acteurs §1.7.2. Des implémentations du modèle de tâches sont présentées au chapitre 2, et des exemples de son utilisation apparaissent au chapitre 3.

Le parallélisme à l'intérieur d'un même programme (SPMD, *Single Program Multiple Data*) repose généralement sur l'une ou l'autre des catégories suivantes : parallélisme guidé par les données (*data-driven*) ou guidé par le contrôle (*control-driven*) [40]. Le parallélisme guidé par les données (*data-driven*) consiste à découper un ensemble de données en sous-ensembles, afin que les données des sous-ensembles puissent être traitées parallèlement par plusieurs fils d'exécution. Inversement, si les données à traiter ne sont pas un ensemble mais bien des données indépendantes les unes des autres, alors le parallélisme est guidé par le contrôle (*control-driven*). Le système en référence utilisé au chapitre 3 est un exemple de parallélisme guidé par le contrôle.

1.7.1 Modèle basé sur les tâches

Une tâche est une unité de code exécutable responsable d'un travail précis. Une tâche peut, selon la disponibilité des unités de traitement, être exécutée ou non en parallèle avec d'autres tâches. Toujours selon la disponibilité des unités de traitement, et selon le nombre de tâches en cours ou en attente d'exécution, une tâche peut ne pas être exécutée immédiatement après sa création. La durée de vie d'une tâche peut être courte (p. ex. : compression d'une image), tout comme elle peut s'apparenter à celle du système l'ayant créée (p. ex. : lire les paquets provenant du réseau).

Le modèle de tâches fait partie d'une approche dite de partage de données (*shared memory*) : une même donnée en mémoire peut être partagée par plusieurs tâches, comme c'est le cas, d'ailleurs, à travers l'utilisation de fils d'exécution. Avec le partage de données, les tâches doivent parfois se synchroniser entre elles; par exemple, le patron producteur/consommateur peut être implanté avec plusieurs tâches, certaines agissant en tant que productrices, et d'autres en tant que consommatrices.

Avec une approche dite de partage de données, il est possible qu'une même donnée soit accessible par plusieurs tâches – ou fils d'exécution – et qu'elle puisse ainsi être modifiée simultanément par ces tâches. Ainsi, lorsque nécessaire, il faut avoir recours à des mécanismes de synchronisation (p. ex. : *mutex*) pour protéger la donnée contre les accès simultanés et éviter qu'une donnée se retrouve dans un état incohérent. Une fonction est alors dite réentrante lorsqu'elle permet d'être accédée par plusieurs tâches sans risque que les données qu'elle manipule ne se retrouvent dans un état incohérent. La réentrance se réalise à travers l'utilisation de différentes pratiques telles que le recours à des mécanismes de synchronisation.

Certaines bibliothèques spécialisées pour le parallélisme – comme celle présentée au chapitre 2 –permettent de traiter des tâches en mode coopératif, par opposition au mode préemptif, ce qui permet de mieux appliquer le principe de localité des données §1.5.2 en réduisant les changements de contexte. La figure suivante montre les principaux éléments en place dans une implémentation simplifiée d'un modèle de tâches reposant sur un groupe de fils d'exécution (*thread pool*) :

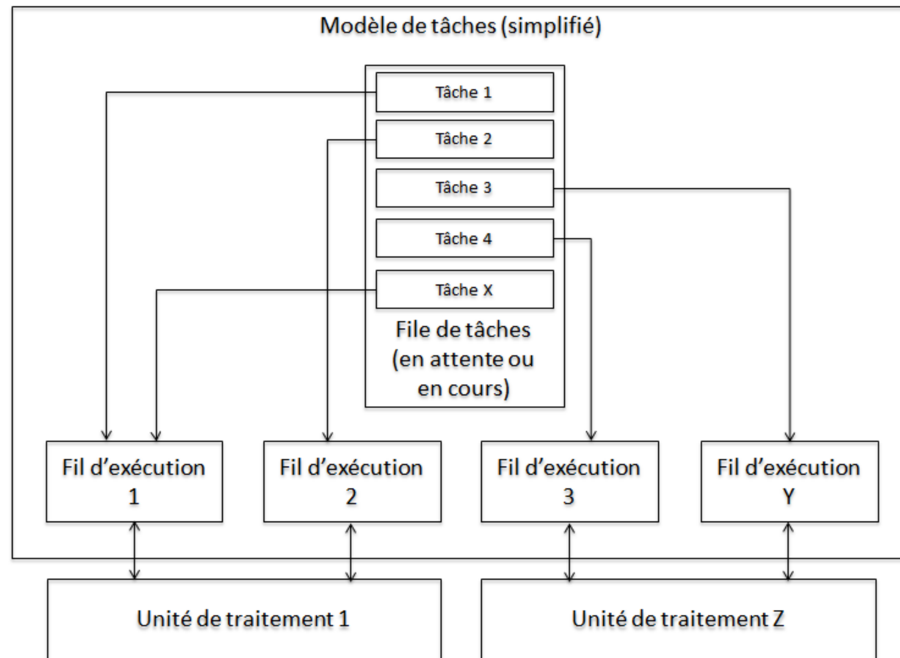

Figure 5 Vue simplifiée d'un modèle de tâches

1.7.2 Modèle basé sur les acteurs

Le modèle basé sur les acteurs est une alternative aux approches dites de partage de données telles que le modèle basé sur les tâches §1.7.1. Il permet d'avoir un ensemble d'acteurs s'exécutant en parallèle et communiquant entre eux à travers des messages asynchrones, selon une approche d'échange de données, par opposition à une approche de partage de données.

Comme présenté à la Figure 6, bien que ce ne soit pas exploré dans cet essai, il est intéressant de souligner que ce modèle permet l'exploitation d'unités de traitement réparties avec des acteurs répartis sur plusieurs machines, pour qu'un système soit en mesure de supporter une charge de traitement plus grande que celle permise par les unités traitements locale à une seule machine. Toutefois, le transfert de messages au-delà d'une machine requiert une charge supplémentaire en termes de traitements (p. ex. : la sérialisation des messages, et possiblement de réponses) et de temps, dû au délai de propagation d'un message à travers le réseau. Il faut donc que le traitement demandé à un acteur par un autre acteur soit suffisamment important pour que cette charge supplémentaire soit négligeable en perspective. Toutefois, cette charge

supplémentaire est réduite lorsque les acteurs s’exécutent tous sur des unités de traitement locales à une même machine.

Généralement, pour éviter tout partage de données, lorsqu’un acteur récupère un message d’une file d’attente, c’est une copie du message qu’il obtient, car l’approche reposant sur les acteurs sous-tend le principe du *Share Nothing*, qui signifie que les messages échangés entre les acteurs sont en fait des copies. Le fait qu’un acteur opère sur sa propre copie d’un message accroît l’efficacité de la gestion d'antémémoire §1.5.4, puisqu’une même donnée ne risque pas d’être modifiée parallèlement par plusieurs acteurs, et améliore la gestion des nœuds sur une architecture NUMA §1.6.1 : la mémoire allouée pour la copie du message se fait normalement par le fil d’exécution responsable de l’instance d’un acteur, et est donc allouée à même son nœud.

Comme présenté à la Figure 6, plusieurs acteurs peuvent soumettre des messages vers une même file d’attente, suivant le principe d'une boîte postale par destinataire. Par exemple, pour qu'un message soit diffusé à plusieurs destinataires, il suffit d'en envoyer une copie à chacun d’eux. Il existe plusieurs types de patrons de stockage de messages, qui varient d’une implémentation à l’autre. Les plus communs sont :

- Une liste de type premier arrivé, premier servi (FIFO). Lorsqu’un acteur consomme un message dans une telle liste, le message en est retiré automatiquement. Ce type de stockage permet, par exemple, l’exécution parallèle de plusieurs acteurs consommant des messages d’une même liste, afin de partager équitablement le travail à faire entre eux. La synchronisation des messages entres les acteurs consommateurs d'une même liste est généralement gérée directement au niveau de la bibliothèque spécialisée offrant une implémentation d'acteurs.

- Un stockage qui contient seulement le dernier message reçu. Lorsqu’un acteur consomme un message dans ce type de stockage, le message n’est pas retiré du stockage, qui reste présent jusqu’à ce qu’un nouveau message le remplace. Cela permet, par exemple, à plusieurs acteurs de consommer les mêmes messages. Ce type

de stockage a pour effet, dans le cas où un acteur A transmettrait des messages à un acteur B plus rapidement que ce dernier pourrait les traiter, que le dernier message envoyé invalide les messages précédents, puisque seul le dernier message y est conservé. Tout comme au point précédent, la synchronisation des messages se fait directement au niveau de la bibliothèque spécialisée.

- Permettre à un acteur d'être informé d'un nouveau message seulement lorsqu'au moins un message sera présent dans plusieurs sources (ou stockage), par exemple si un acteur A veut être notifié d'un nouveau message seulement lorsque l'acteur B et l'acteur C lui auront tous deux transmis un message.

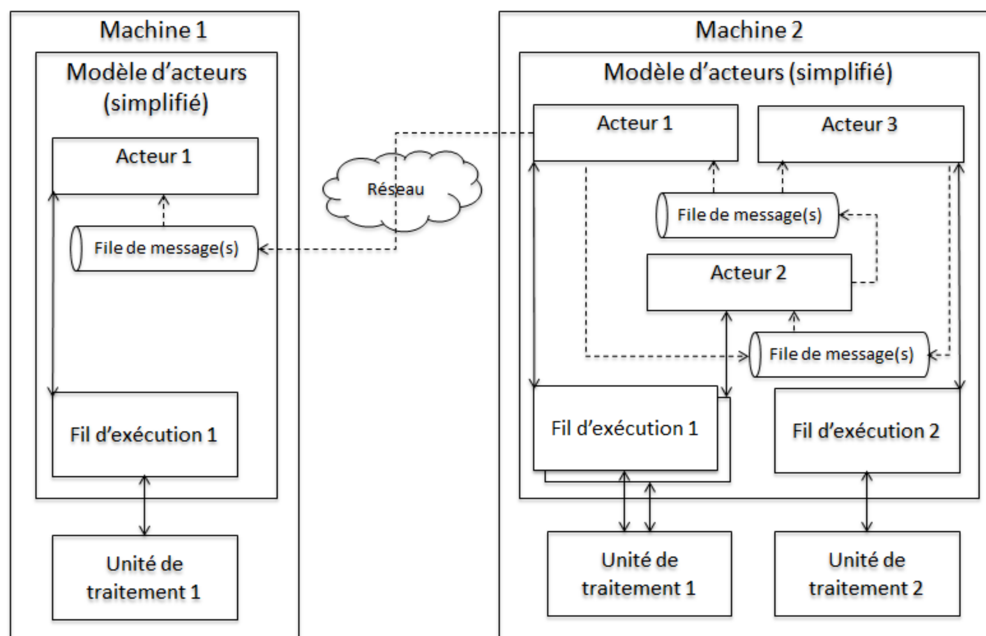

Figure 6 Vue simplifiée d'un modèle basé sur les acteurs

Bien que les modèles reposant sur les échanges de messages asynchrones – tel le modèle basé sur les acteurs – soient largement utilisés, Bartosz Milewski, dans un article [40] sur sa vision du parallélisme, en fait une critique sévère :

> « *Message passing's major flaw is the inversion of control–it is a moral equivalent of gotos in un-structured programming (it's about time somebody said that message*

passing is considered harmful). Message passing still has its applications and, used in moderation, can be quite handy; but PGAS (Partitioned Global Address Space) offers a much more straightforward programming model–its essence being the separation of implementation from algorithm [...] »[1] ([40]).

1.8 Complexité reliée au parallélisme

Le parallélisme a pour objectif d'exploiter les différentes unités de traitement de disponibles. Toutefois, il peut aussi contribuer à complexifier un système. Cette complexité est principalement reliée à la difficulté d’implémentation et de débogage d’un système parallèle [6]. En voici quelques exemples :

- Il est souvent complexe de localiser les endroits dans un système où le parallélisme peut être implanté. Tel que discuté en début de chapitre, il peut y avoir une dépendance des données entre les différentes opérations d’un système, et une mauvaise compréhension de ces dépendances lors de la mise en place du parallélisme peut compromettre la stabilité du système. Ce type de problématique se retrouve principalement dans les modèles où le partage de données est possible entre les différentes unités d'exécution concurrentes d’un système (p. ex. : fils d’exécution, tâches). D’autres modèles font en sorte que cette problématique ne puisse pas se produire (p. ex. : modèle d’acteurs), car les données sont échangées, et non partagées, entre les différentes unités d'exécution concurrentes d’un système.

- L’aspect indéterministe d’un système parallèle a pour effet de rendre imprévisible la séquence d’exécution des différentes parties parallèle du système puisqu’elles s’exécutent selon les disponibilités des unités de traitement. Il devient alors difficile, voire impossible de prédire à quel moment les différents fils d’exécution du système accèderont les données qu’ils partagent.

[1] La principale problématique avec les échanges de messages asynchrone est l’inversion de contrôle qui en résulte, équivalent aux *gotos* dans la programmation non-structuré (un jour viendra où quelqu’un dira que cela est nuisible). Les échanges de messages ont toujours leurs applications et, lorsqu’utilisés avec modération, peuvent être bien utiles; cependant, le partage de données offre un modèle de programmation plus simple dû à la séparation entre l’implémentation et l’algorithme. (Traduction libre)

- L'aspect indéterministe d'un système parallèle a également pour effet de compliquer les tests et le débogage. Cet aspect indéterministe fait naître une nouvelle catégorie de bogues, principalement reliés aux problèmes circonstanciels entre les différents fils d'exécution résultant de conditions de courses ([46] §1.10). Une condition de course se présente de façon sporadique, ce qui rend le problème difficilement reproductible, et donc difficile à déboguer.

Le prochain chapitre présente la bibliothèque *Concurrency Runtime* de Microsoft, bibliothèque spécialisée en vue de la mise en place du parallélisme dans un système. Plusieurs concepts présentés au chapitre 1 peuvent être mis en place à travers l'utilisation de cette bibliothèque.

Chapitre 2

Présentation d'une bibliothèque spécialisée pour le parallélisme

Plusieurs bibliothèques ont pour objectif de simplifier la mise en place du parallélisme dans un système afin que ce dernier puisse exploiter efficacement les différents cœurs d'un processeur. Dans le cadre de cet essai, la bibliothèque *Concurrency Runtime* ([20],[21]) sert à titre d'exemple. Cette bibliothèque permet de présenter différents concepts utilisés par la suite au chapitre 3, incluant les tâches, les acteurs, ainsi que le support des ordonnanceurs préemptifs et coopératifs.

Puisque l'objectif de cet essai n'est pas de présenter en détail cette bibliothèque, ce chapitre ne couvre que les aspects réutilisés au chapitre 3.

La figure suivante présente les différents composants des bibliothèques incluses et exploitant le *Concurrency Runtime* :

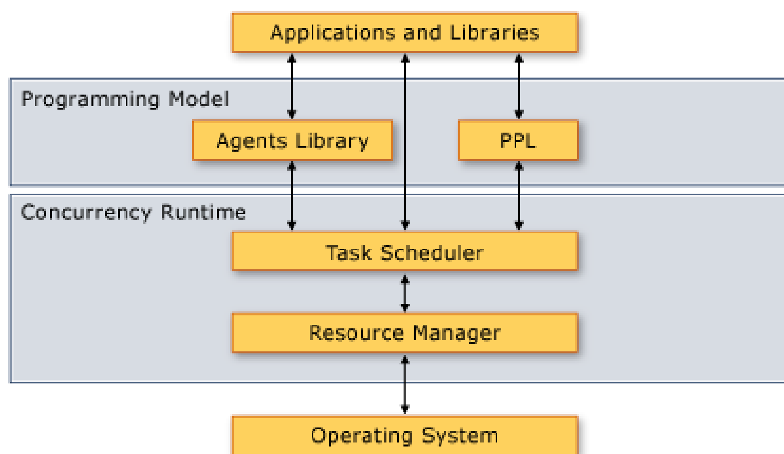

Figure 7 Bibliothèques incluses et exploitant le *Concurrency Runtime*

Source : Microsoft, http://msdn.microsoft.com/en-us/library/ee207192.aspx, mars 2014

La figure précédente montre que le *Resource Manager* et le *Task Scheduler* font partie du *Concurrency Runtime*, tandis qu'*Agents Library* (ou plus précisément *Asynchronous Agents*

Library) et PPL (*Parallel Patterns Library*) font partie des bibliothèques qui utilisent le *Concurrency Runtime*.

Les sections suivantes donnent une vue d'ensemble des fonctionnalités du *Resource Manager* et du *Task Manager*; elles couvrent également PPL et *Agents Library*. *Agents Library* est d'ailleurs utilisée au chapitre 3 pour mettre en place le parallélisme basé sur les acteurs.

2.1 *Concurrency Runtime*

Le *Concurrency Runtime* est un *framework* C++ qui a pour objectif de faciliter le développement d'applications parallèles. Il contient le *Resource Manager* §2.1.1 et le *Task Scheduler* §2.1.2, qui servent respectivement à faire abstraction des ressources matérielles de la machine et à fournir différents types d'ordonnanceurs par-dessus l'ordonnanceur préemptif du système d'exploitation Windows.

2.1.1 *Resource Manager*

En plus d'offrir une abstraction des ressources matérielles de la machine, le *Resource Manager* est responsable de détecter l'architecture du processeur de la machine (p. ex. : SMP §1.5, NUMA §1.6, le nombre de nœuds, de cœurs), puis d'affecter les différentes unités de traitement à une ou plusieurs instances du *Task Scheduler* de l'application. Par exemple, si une application a deux instances du *Task Scheduler*, alors le *Resource Manager* divisera équitablement le nombre de cœurs entre ces deux instances, et ce, en respectant les limites des nœuds NUMA lorsque l'application est exécutée sur une telle architecture. L'affectation des cœurs par le *Resource Manager* se fait de façon à respecter le plus possible l'aspect « local » des données §1.6.1, pour utiliser efficacement la mémoire et l'antémémoire, et ce, tant pour une architecture SMP que NUMA.

La figure suivante présente un exemple dans lequel le *Resource Manager* affecte de façon équitable les différents cœurs des nœuds NUMA à deux instances de *Task Scheduler* :

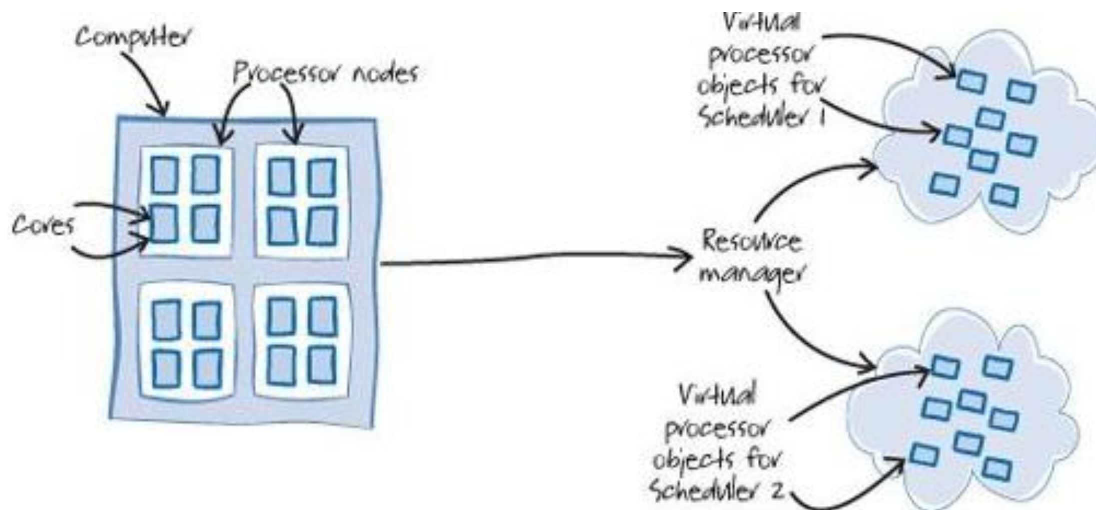

Figure 8 Resource Manager

Source : Microsoft, http://msdn.microsoft.com/en-us/library/gg663535.aspx, mars 2014

La Figure 8 présente le concept de processeur virtuel. Pour le *Resource Manager*, un processeur virtuel est en quelque sorte une abstraction d'un cœur physique, soit un objet qui permet à une instance de *Task Scheduler* §2.1.2 de démarrer ou de suspendre un fil d'exécution. Par défaut, le *Resource Manager* alloue autant de processeurs virtuels qu'il y a de cœurs. Cependant, un processeur virtuel n'est pas lié à un cœur spécifique; il est plutôt associé à un regroupement, tel un nœud NUMA sur ce type d'architecture, de sorte qu'une fois un processeur virtuel associé à un nœud, il peut être exécuté sur n'importe quel cœur de ce nœud §1.6. Cette association entre un processeur virtuel et un nœud se nomme affinité du processeur (*Processor Affinity Mask*). Lorsque le *Task Scheduler* choisit un processeur virtuel disponible pour exécuter un fil d'exécution, le *Resource Manager* choisit un cœur selon l'affinité du processeur virtuel, ce qui permet de respecter les limites d'un nœud §1.6.1 sur une architecture NUMA.

Une instance de *Task Scheduler* peut créer plus de fils d'exécution que de processeurs virtuels lui ayant été affectés; il ne peut toutefois pas y avoir plus de fils d'exécution qui s'exécutent en parallèle qu'il n'y a de processeurs virtuels. Un processeur virtuel exécute un fil d'exécution jusqu'à ce que ce dernier soit complété, jusqu'à ce qu'il y ait un changement de

contexte demandé par une instance de *Task Scheduler* configurée en mode préemptif, ou jusqu'à ce que le fil d'exécution fasse une opération dite coopérative bloquante (*Cooperative Context Switch*) à l'intérieur d'une instance de *Task Scheduler* configurée en mode coopératif. À l'un ou l'autre de ces moments, le fil d'exécution est désassocié du processeur virtuel, et ce dernier devient disponible pour un autre fil d'exécution.

Le *Resource Manager* supervise constamment l'utilisation des ressources. Il peut ainsi détecter une sous-utilisation de certaines ressources (p. ex. : des cœurs) par une instance de *Task Scheduler*. Lorsque cette situation se produit, le *Resource Manager* peut réaffecter de façon dynamique des cœurs d'une instance de *Task Scheduler* sous-utilisée vers une autre instance de *Task Scheduler*.

Une surutilisation (*Oversubscription)* des cœurs dans le *Resource Manager* survient lorsqu'une instance de *Task Scheduler* se fait affecter plus de processeurs virtuels qu'il n'y a de cœurs physiques. Tel que mentionné précédemment, par défaut, le *Resource Manager* attribue les cœurs et les processeurs virtuels de façon à ne pas créer une surutilisation, car lorsqu'il y a trop de processeurs virtuels pour un même cœur, des problèmes de contention sur l'antémémoire de ce cœur s'ensuivent. Cependant, il existe des situations pouvant entraîner une surutilisation :

- au moment de la création d'une instance de *Task Scheduler*, il est possible de demander qu'il y ait une surutilisation. Lorsque par exemple les tâches à exécuter par l'instance du *Task Scheduler* sont souvent en attente sur des opérations d'entrées/sorties, une surutilisation maintiendra les cœurs sous sa responsabilité plus occupés, car lorsqu'une tâche sera en attente sur une opération d'E/S une autre tâche pourra être exécutée;

- le *Resource Manager* peut détecter une sous-utilisation d'un cœur par une instance de *Task Scheduler,* créer temporairement une nouvelle instance de processeur virtuel pour ce cœur, et l'associer à une autre instance de *Task Scheduler,* sans toutefois retirer l'instance de processeur virtuel de ce cœur à l'instance du *Task Scheduler* sous-utilisé.

Deux processeurs virtuels dans deux instances de *Task Scheduler* se partageront alors un cœur.

2.1.2 *Task Scheduler*

Le *Task Scheduler* a pour responsabilité de planifier et de gérer l'exécution des fils d'exécution qui lui ont été affectés, avec comme objectif de garder occupés le plus possible les cœurs placés sous sa responsabilité par le *Resource Manager*. Les fils d'exécution affectés à une instance de *Task Scheduler* servent principalement à l'exécution de tâches §1.7.1 et d'acteurs §1.7.2 sous sa responsabilité.

Afin d'exploiter efficacement les différentes architectures de processeurs multi-cœurs, le *Task Scheduler* gère un *Schedule Group*, permettant de regrouper des tâches qui auraient un avantage à être exécutées sur un même nœud NUMA. Les tâches qu'il est avantageux d'exécuter sur un même nœud sont celles travaillant sur sensiblement les mêmes objets en mémoire, permettant ainsi de réduire les accès mémoire inter-nœuds (très pénalisants §1.6.1). Une application peut alors créer plusieurs *Schedule Groups*, et choisir les tâches à regrouper en les associant à une instance spécifique d'un *Schedule Group*; ainsi, le *Task Scheduler* garantit que les tâches d'un *Schedule Group* seront exécutées par des processeurs d'un même nœud (voir l'affinité de processeur virtuel §2.1.1).

Tout comme le *Resource Manager* gère une affinité entre un processeur virtuel et un nœud, les systèmes d'exploitation modernes tels que Windows ou Linux gèrent aussi une telle affinité, cette fois entre un fil d'exécution et un cœur (*Processor Affinity* ou *CPU Pinning*, §1.5.4). Ce mécanisme consiste à exécuter, lorsque possible, un fil d'exécution sur un cœur récemment utilisé par ce fil d'exécution. Cette différence d'approche quant à la gestion d'affinité fait en sorte que dans certaines situations (§3.8 et §4.7), l'utilisation des fils d'exécution peut s'avérer plus avantageuse que l'utilisation du *Resource Manager* à travers le *Concurrency Runtime*, et ainsi entraîner une meilleure utilisation de l'antémémoire lors de changements de contexte §1.5.4.

La granularité des tâches, soit la quantité d'opérations d'une tâche dans un intervalle de temps choisi, d'un regroupement est importante afin d'optimiser l'utilisation des ressources de la machine. Par exemple, un regroupement avec une granularité trop fine risquera d'avoir une charge élevée seulement pour la gestion de toutes ses tâches, tandis qu'une granularité trop grossière réduira les opportunités de parallélisme. Comme pour toute technique de parallélisation, décomposer un problème en unités d'exécution concurrentes demande une bonne compréhension du problème et des algorithmes appliqués pour le résoudre. Dans le contexte du *Concurrency Runtime*, la création de tâches se fait généralement à travers le *Parallel Patterns Library*; plus précisément à travers des méthodes telles que parallel_for, parallel_invoke, task_group::run, présentées plus en détail à §2.2.1 et §2.2.2.

Une instance de *Task Scheduler* peut opérer en deux modes distincts, soit les modes coopératif (*Enhanced Locality Mode* §2.1.3) et préemptif (*Forward Progress Mode* §2.1.4). Il faut noter que différentes instances de *Task Scheduler* configurées dans des modes distincts peuvent coexister dans une même application.

2.1.3 Mode coopératif

Le mode coopératif est le mode par défaut du *Task Scheduler*, qui met l'accent sur l'aspect localité des données dans l'optique d'utiliser efficacement l'antémémoire. L'idée derrière ce mode est de regrouper des tâches à l'intérieur d'une instance de *Schedule Group*, et de les exécuter entièrement avant de passer au regroupement de tâches suivant.

Les tâches d'un regroupement doivent être indépendantes les unes des autres, pour être exécutées en parallèle ou en série, en fonction de la quantité de tâches du regroupement et de la disponibilité des cœurs (ou processeurs virtuels) associés à l'instance du *Schedule Group* responsable du regroupement. Par conséquent, lorsqu'une tâche d'un regroupement commence son exécution, celle-ci n'est pas interrompue par le *Task Scheduler* avant d'être complétée, à moins qu'elle n'exécute une opération bloquante. Ceci permet de générer moins de changements de contexte qu'avec le mode préemptif.

2.1.4 Mode préemptif

Contrairement au mode coopératif, le mode préemptif préconise une équité en terme de temps de processeur alloué aux tâches, plutôt qu’une utilisation maximale de l'antémémoire. Pour réaliser cette équité, une instance de *Task Scheduler* en mode préemptif initie continuellement des changements de contexte entre les tâches sous sa responsabilité. Ce mode de fonctionnement est donc similaire à celui des ordonnanceurs des systèmes d’exploitation modernes.

2.2 *Parallel Patterns Library*

Tel que présenté à la Figure 7, différentes bibliothèques permettent de tirer profit du *Task Scheduler* et du *Resource Manager* offert par le *Concurrency Runtime*, en particulier la bibliothèque *Parallel Patterns Library* (PPL) qui offre une variété de fonctions permettant de paralléliser différentes parties d’un système. Ces fonctions se retrouvent dans deux catégories distinctes : le parallélisme des données (*data parallelism* §2.2.1) et celui des tâches (*task parallelism* §2.2.2).

Cette section ne fait qu’un survol très rapide de PPL afin d’introduire des concepts utilisés au Chapitre 3.

2.2.1 Parallélisme des données

Le parallélisme sur des données permet d’appliquer un traitement en parallèle sur des données indépendantes les unes des autres. Les principales fonctions de PPL pour ceci sont parallel_for et parallel_for_each. L’exemple de code suivant provient de [20] et présente un cas simple d’une migration d’une boucle exécutée séquentiellement vers une boucle exécutée en parallèle.

```
// Exemple d’une boucle séquentielle
vector<double> results = …
int workload = …
auto size = results.size();
for (vector<double>::size_type i=0; i<size; ++i)
{
  results[i] = DoWork(i, workLoad);
}
```

```
// Maintenant le même traitement mais en parallèle
vector<double> results = …
int workload = …
vector<double>::size_type begin=0
auto size = results.size();
parallel_for (begin, size, [&](vector<double>::size_type i)
{
  results[i] = DoWork(i, workLoad);
});
```

Au premier coup d'œil, il paraît simple de paralléliser une boucle séquentielle. Il n'en demeure pas moins que cette transformation demande une grande vigilance afin de s'assurer de ne pas introduire de nouveaux problèmes. Par exemple, avec une boucle parallèle aussi simple que celle de l'exemple précédent, il faut s'assurer que les opérations de la méthode DoWork soient indépendantes les unes des autres, donc réentrantes. Cependant, si la méthode DoWork est réentrante à travers l'utilisation d'un mécanisme de synchronisation, le gain de performance obtenu pourrait être pratiquement nul, puisque les fils d'exécution seraient constamment en attente de la primitive de synchronisation, ce qui pourrait être équivalent à la version séquentielle. Et si la ligne de code « results[i] = DoWork(i, workLoad) » était remplacée par « result += DoWork(i, workLoad) », le résultat obtenu ne serait probablement pas celui attendu, car la valeur de la variable results serait mise à jour en parallèle par les différents fils d'exécution. §2.2.3 présente une solution à ce dernier problème.

2.2.2 Parallélisme des tâches

La bibliothèque PPL permet également le traitement de tâches en parallèle, qui se fait principalement par la fonction parallel_invoke et la classe task_group. L'exemple de code suivant présente un comparatif simple des traitements séquentiels et de traitements parallèles, avec et sans tâches.

```
// Exemple de deux méthodes indépendantes exécutées séquentiellement
DoLeft();
DoRight();

// Les mêmes deux méthodes exécutées en parallèle avec parallel_invoke
parallel_invoke(
  []() { DoLeft(); },
  []() { DoRight(); }
);

// parallel_invoke est en fait un raccourci à ce qui suit
task_group tg;
```

```
tg.run([](){ DoLeft(); });
tg.run([](){ DoRight(); });
tg.wait();
```

Comme discuté à §2.1.4, l'ordonnanceur des systèmes d'exploitation contemporains est généralement préemptif. L'ordonnanceur coopératif utilisé par PPL tente au contraire de maximiser les performances globales du système §2.1.3 en limitant les changements de contexte, de sorte qu'un task_group peut ne pas commencer son exécution immédiatement après sa création. Le task_group est plutôt mis dans une file d'attente jusqu'à ce que l'ordonnanceur coopératif en retire les tâches pour exécution, ce qui dépend principalement de la disponibilité des cœurs et des autres tâches en attente.

2.2.3 Agrégation parallèle

Tel que mentionné à §2.2.1, un traitement sur des données peut être fait efficacement en parallèle lorsque les données sont indépendantes les unes des autres. Cependant, il existe des cas où les données sont effectivement indépendantes, mais où le résultat ne l'est pas. Le code suivant en présente un exemple, soit compter les nombres premiers.

```
// Compter les nombres premiers de façon séquentielle
vector<int> sequence = …
int count = 0;
for (vector<int>::size_type i=0; i<sequence.size(); ++i)
{
  count += IsPrime(sequence[i]) ? 1 : 0;
}
return count;
```

Dans cet exemple, paralléliser l'expression « count += IsPrime(sequence[i]) ? 1 : 0; » pose problème, car la variable count risque d'être modifiée en parallèle, entraînant un résultat erroné. C'est pourquoi PPL propose l'agrégation parallèle, utilisant des variables non partagées qui sont fusionnées à la fin de la boucle en parallèle. Ceci rend indépendants des traitements qui, autrement, ne le seraient pas. L'exemple de code suivant présente la même méthode que ci-haut, mais en parallèle et avec l'utilisation de l'agrégation parallèle.

```
// Compter les nombres premiers en parallèle
vector<int> sequence = …
combinable<int> count([]() { return 0; });
vector<int>::size_type n = sequence.size();
parallel_for (0, n, [&](vector<int>::size_type i)
```

```
{
  count.local() += IsPrime(i) ? 1 : 0;
});
return count.combine(plus<int>());
```

Le prochain chapitre présente différentes techniques de mise en place du parallélisme, dont certaines reposent sur la bibliothèque *Concurrency Runtime*.

Chapitre 3
Migration vers une architecture parallèle

Le chapitre 1 présente différentes architecture de processeurs multi-cœurs, dont les architectures SMP et NUMA, ainsi que des techniques pour une exploitation optimale de celles-ci. Les effets néfastes des changements de contexte trop fréquents et d'une mauvaise utilisation de l'antémémoire y sont également exposés, tout comme les avantages et inconvénients des ordonnanceurs préemptifs et coopératifs.

Quant à lui, le chapitre 2 présente une bibliothèque facilitant l'exploitation des architectures parallèles. Ce chapitre montre qu'une telle bibliothèque simplifie la mise en place du parallélisme dans un système, avec par exemple l'utilisation de tâches. Une telle bibliothèque permet également d'exécuter des tâches sur un ordonnanceur de type coopératif §1.3.1, ce qui aide généralement un système à limiter les changements de contexte générés, et conséquemment réduit les défauts de cache.

Le présent chapitre fait l'exercice de prendre un système exploitant peu le parallélisme et d'y intégrer des techniques de parallélisme afin d'exploiter les différents cœurs, tout en s'assurant d'éviter le plus possible les contraintes et les pièges décrits au chapitre 1. Les techniques de parallélisme mises en place dans ce chapitre reposent sur différents modèles, soit :

- en manipulant directement les fils d'exécution (§3.3, §3.4 et §3.5);
- en utilisant les techniques nouvellement mises en place depuis l'avènement de C++ 11 (*async* et *future* [43]) (§3.6);
- en utilisant les tâches exécutées sur un ordonnanceur coopératif offert par la bibliothèque *Concurrency Runtime* présentée au chapitre 2 (§3.7);
- en exploitant un modèle hybride avec fils d'exécution et tâches (§3.8.1 et §3.8.2).

L'analyse des résultats obtenus à travers les différentes techniques de mises en place du parallélisme de ce chapitre se retrouve au chapitre 4.

Le système en référence dans ce chapitre s'introduit au milieu de conversations transmises sur un réseau IP (*Internet Protocol*) tel qu'Internet, afin de transformer l'audio des conversations d'un codec vers un autre codec. Ce type de système permet ainsi à des entités ne supportant pas de codec commun de converser.

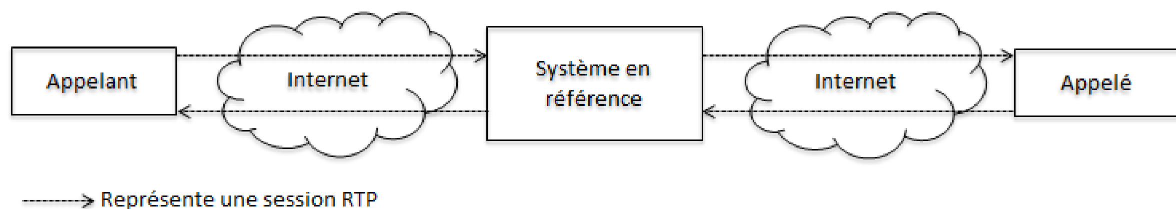

Figure 9 Vue du système en référence dans son contexte

Tel que présenté à la Figure 9, les conversations sont reçues et transmises à travers le protocole RTP (*Real-time Transport Protocol* [45]), protocole utilisé pour transmettre des données audio ou vidéo entre une source et une destination. Les paquets RTP transitent habituellement sur le protocole UDP (*User Datagram Protocol*), qui repose sur le protocole IP (*Internet Protocol*). Les données transmises par RTP doivent généralement l'être à intervalles réguliers stricts, sans quoi la qualité de la conversation risque de se dégrader rapidement lorsque les intervalles entre les paquets de données reçues sont trop distancés.

Dans le vocabulaire RTP, une conversation correspond à une session; une session est ainsi responsable de traiter l'ensemble de paquets RTP d'une conversation.

Le système en référence a été choisi parce qu'il partage plusieurs caractéristiques avec d'autres types de systèmes répartis :

- Les sessions sont indépendantes les unes des autres.
- Les opérations d'E/S sont très sollicitées.

- Les paquets reçus doivent être traités dans les plus brefs délais : il doit y avoir le moins de latence possible.

Plusieurs techniques de mise en place du parallélisme présentées tout au long de ce chapitre s'appliquent donc à d'autres systèmes partageant sensiblement les mêmes caractéristiques.

En plus des caractéristiques décrites ci-haut, le système en référence doit composer avec une forme de pseudo temps réel, car la plupart des codecs utilisés pour convertir un signal audio analogique en un flux de données numériques prêt à être transmis sur Internet à travers RTP ont une fréquence d'échantillonnage de 8000 Hz encodée sur huit bits, de sorte qu'à chaque seconde, 64 kbits de données audio encodées doivent être transmis par session. Ce 64 kbits de données par seconde est souvent séparé en cinquante paquets RTP contenant chacun vingt ms de données audio, envoyés les uns à la suite des autres. La mise en place du parallélisme aux endroits adéquats aide ainsi le système à respecter une cadence régulière, et limite la latence ajoutée par celui-ci.

Ce chapitre présente la méthodologie appliquée pour repérer des endroits où l'ajout de parallélisme au système permet de traiter un plus grand nombre de session RTP, tout en respectant la cadence de vingt ms, et en minimisant les changements à apporter au code existant. Les modifications apportées au système se font aussi dans l'optique de limiter le plus possible les changements de contexte, pour une meilleure utilisation de l'antémémoire, et d'exploiter efficacement les architectures SMP et NUMA. Puisque la tendance est à l'augmentation du nombre de cœurs par ordinateur, un des principaux objectifs de cet exercice est de faire en sorte que les modifications apportées au système permettent des gains proportionnels au nombre de cœurs sur lesquels le système sera exécuté.

Afin de mesurer les gains obtenus suite à l'ajout du parallélisme au système, l'unité de mesure principalement utilisée dans cet essai est le nombre de conversations simultanées que le système peut maintenir sans perte de qualité des conversations. Normalement, tant que le temps maximum requis au traitement des paquets RTP associés à l'ensemble des conversions simultanées se fait à l'intérieur de la cadence de vingt ms, il n'y a pas de perte de qualité. Par

contre, lorsque le temps moyen requis pour ce traitement est supérieur à cette cadence, alors il y aura une forcément dégradation, car des paquets RTP ne pourront être traités.

3.1 Architecture initiale du système en référence

Le système en référence gère des sessions RTP indépendantes les unes des autres, ce qui est d'ailleurs le cas pour plusieurs types de systèmes répartis desservant des clients. Cette indépendance des sessions simplifie généralement la mise en place du parallélisme, car les sessions n'échangent pas de données entre elles, ce qui a pour effet d'offrir plusieurs possibilités quant à la granularité des opérations à paralléliser.

Ce système intègre une bibliothèque à source ouvert, JRTPLIB [31], implémentant le protocole RTP. Les opérations faites par le système pour chacune des sessions RTP sont présentées à haut niveau à la Figure 10, et peuvent être faites séquentiellement ou non pour des raisons d'optimisation §3.8.

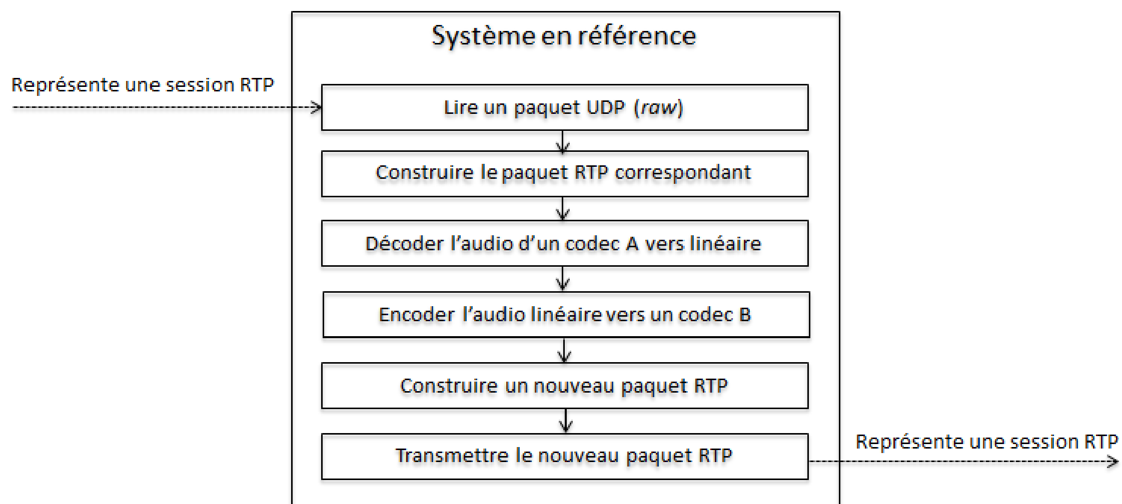

Figure 10 Opérations séquentielles du système en référence

L'architecture courante du système en référence n'exploite pas le parallélisme. Le système s'exécute entièrement sur le fil d'exécution principal, et utilise les E/S non-bloquantes offertes à travers la bibliothèque JRTPLIB. Ce modèle à un seul fil d'exécution utilisant les E/S non-bloquantes ne tire donc pas avantage des multiples unités de traitements pouvant être

disponibles; cependant, il est très répandu encore en 2014, car il s'agit d'une solution traditionnelle à des modèles semblables mais avec des architectures mono-cœur. L'exemple de code suivant présente d'une manière simplifiée la façon dont ce modèle est intégré au système en référence, dans lequel périodiquement (soit à chaque vingt milliseconde) les paquets RTP de chacune des sessions sont traités.

```
...

for (auto pSession : rtpSessions)
{
  // Lire le ou les paquets de pSession, si disponible, de façon
  // non-bloquante
  pSession->Poll();

  // Vérifier si des paquets sont prêts à être traités
  if (pSession->GotoFirstSourceWithData())
  {
    do
    {
      RTPPacket *packet;
      while ((packet = pSession->GetNextPacket()) != nullptr)
      {
        // Traiter les paquets préalablement lus
        vector<unsigned char> audioConverted =
          ConvertAudio(packet->GetPayloadData(),
                       packet->GetPayloadLength();

        // Envoyer le paquet avec l'audio converti vers la destination
        pSession->SendPacket(audioConverted.data(),
                             audioConverted.size(),
                             0, false, CInterval);

        pSession->DeletePacket(packet);
      }
    } while (pSession->GotoNextSourceWithData());
  }
}

...
```

Le nombre de conversations simultanées que le système est capable de maintenir est ainsi équivalent au nombre de sessions RTP pouvant être traitées à l'intérieur de ce vingt ms, autrement des paquets ne pourront être traités et seront ainsi perdus.

Les principales étapes du système en référence présentées à la Figure 10 doivent être exécutées séquentiellement. Toutefois, il est possible de cerner les opérations pouvant être parallélisées à l'intérieur d'une étape, comme par exemple les étapes de décodage de l'audio et d'encodage vers un autre codec, qui sont plus exigeantes en terme de temps de traitement

requis. Par contre, puisque l'objectif est d'exploiter l'ensemble des cœurs – le plus équitablement possible – les modèles de parallélisme mis en place dans les prochaines sections exploitent le parallélisme en parallélisant les sessions RTP entre elles, et non en parallélisant les opérations à l'intérieur des étapes de chaque session. Étant donné que le nombre maximum de sessions simultanées pouvant être supportées est largement supérieur aux nombre de cœurs, cette technique permet d'exploiter l'ensemble des cœurs.

À première vue, le système en référence – avec ses quelques étapes séquentielles – semble être un bon candidat pour être parallélisé avec le patron pipeline ([20] §7); patron qui repose sur des acteurs §1.7.2, où par exemple chaque étape de la Figure 10 pourrait être exécutée par un acteur, qui une fois l'étape complétée l'acteur envoie un message à un autre acteur responsable d'exécuter la prochaine étape, et ainsi de suite pour l'ensemble des étapes. Cependant, ce modèle n'a pas été retenu dans cet essai pour les raisons suivantes :

- Pour exploiter les cœurs efficacement, le temps de traitement requis par chacune des étapes du pipeline doit être sensiblement le même, autrement, les étapes plus rapide du pipeline risquent d'être sous-utilisées. Il est tout de même possible de remédier à cette problématique en ayant plusieurs acteurs responsables d'une même étape, soit l'étape qui requiert le temps de traitement le plus élevé. Cependant, cette duplication de certaines étapes afin de mieux balancer l'ensemble des étapes vient complexifier grandement la mise en place de ce patron.

- Puisque que le système en référence ne contient que six principales étapes, un pipeline avec ces d'étapes ne pourrait pas exploiter un nombre toujours grandissant de cœurs; soit au-delà de six cœurs. Tout comme le point précédent, il serait possible de dupliquer des étapes en fonction du nombre de cœurs de la machine, mais cela complexifierait également sa mise en place.

Les sections suivantes présentent en détail différentes techniques de mise en place du parallélisme, permettant ainsi de maintenir un plus grand nombre de conversations simultanées.

3.2 Précaution à prendre avec les E/S non-bloquantes

Les modèles utilisant les E/S non-bloquantes, incluant l'architecture initiale du système en référence §3.1 et plusieurs autres modèles décrits dans les sections suivantes (§3.3, §3.5, §3.7 et §3.8), doivent prendre certaines précautions pour ne pas boucler sans répit sur l'ensemble des sessions, et ainsi consommer la totalité du temps de traitement des unités de traitement sur lesquelles ils s'exécutent. Pour ce faire, le système doit suspendre ses fils d'exécution lorsque le traitement de l'ensemble des sessions RTP se fait à l'intérieur de la cadence de vingt ms imposée par le protocole RTP.

```
...

static const uint32_t CInterval = 20; // 20ms

...

while (executing)
{
  auto cycleStartedTime = chrono::system_clock::now();

  for (auto pSession : rtpSessions)
  {
    ...
  }

  // Calculer le temps requis pour le traitement des sessions
  auto cycleEndedTime = chrono::system_clock::now();
  auto elapsedTimeMs = chrono::duration_cast<chrono::milliseconds>(
    cycleEndedTime-cycleStartedTime).count();
  if (elapsedTimeMs < CInterval)
  {
    // Dormir le temps restant, soit 20ms moins le temps requis pour le
    // traitement de toutes les sessions
    chrono::milliseconds sleepForMs(CInterval - elapsedTimeMs);
    this_thread::sleep_for(sleepForMs);
  }
}
```

Cependant, cette technique pour ne pas consommer tout le temps de traitement cause problème sous le système d'exploitation Windows 7 (soit celui utilisé pour comparer les différents modèles), car la méthode sleep_for(x) peut avoir une résolution approximative de 15 ms [34]. Ainsi, un appel à sleep_for(x) où *x* aurait par exemple une valeur de 5 peut revenir approximativement 15 ms plus tard, au lieu de 5 ms, qui vient ainsi amputer du temps du prochain cycle de vingt ms. Il est possible de changer cette résolution par défaut du système d'exploitation Windows 7 à une valeur plus petite; cette résolution a donc été changée à une

milliseconde pour les modèles utilisant les E/S non-bloquantes, tel que présenté dans l'exemple de code suivant.

```
timeBeginPeriod(1);

// Traitement des sessions.
...

timeEndPeriod(1);
```

3.3 Modèle à un fil d'exécution par session RTP avec E/S non-bloquantes

Une autre technique assez répandue pour exploiter le parallélisme dans un système qui doit desservir plusieurs sessions est d'avoir un fil d'exécution par session; technique qui s'agence bien avec le système en référence puisqu'une session à une durée de vie relativement longue – soit la durée de la conversation – ce qui évite de constamment créer et détruire des fils d'exécution. Sur certains systèmes, la création d'un fil d'exécution peut prendre jusqu'à quelques ms [44].

Cette technique donne des résultats satisfaisants lorsque le nombre de sessions est limité; par contre, les performances sont inversement proportionnelles au nombre de fils d'exécution quand ce nombre est suffisamment grand [48]. Cette détérioration est principalement reliée au fait qu'à partir d'un certain seuil, le système d'exploitation passe plus de temps à gérer les différents fils d'exécution qu'à exécuter leurs instructions (voir changements de contexte et défauts de cache §1.5.3). Ce seuil varie en fonction de différents critères tels que le nombre d'unités de traitement ainsi que la quantité de défauts de cache générés suite aux changements de contexte.

```
...

auto pThread = make_shared<thread>(thread([pSession]
{
  while (executing)
  {
    // Lire le ou les paquets de pSession, si disponible, de façon
    // non-bloquante
    pSession->Poll();

    // Vérifier si des paquets sont prêts à être traités
    if (pSession->GotoFirstSourceWithData())
    {
      do
```

```
      {
        RTPPacket *packet;
        while ((packet = pSession->GetNextPacket()) != nullptr)
        {
          // Traiter les paquets préalablement lus
          vector<unsigned char> audioConverted =
            ConvertAudio(packet->GetPayloadData(),
                         packet->GetPayloadLength()

          // Envoyer le paquet avec l'audio convertis vers la destination
          pSession->SendPacket(audioConverted.data(),
                               audioConverted.size(),
                               0, false, CInterval);

          pSession->DeletePacket(packet);
        }
      } while (pSession->GotoNextSourceWithData());
    }
  }
  ...
}));

...
```

Comme présenté dans l'exemple de code précédent, chaque fil d'exécution exécute séquentiellement les opérations présentées à la Figure 10, mais pour une seule session RTP. Ainsi, lorsqu'une session est affectée à un fil d'exécution, ce dernier est responsable d'exécuter les opérations de la session, et ce, pour la durée de vie de la session. Pour respecter la cadence de vingt ms, chaque fil d'exécution doit être en mesure d'exécuter dans cet intervalle les opérations sur la session lui ayant été assignée.

Plus le nombre d'unités de traitement disponibles pour le système est élevé, plus cette technique supporte un nombre élevé de conversations simultanées en comparaison avec la version initiale du système avec un seul fil d'exécution §4.2, car les fils d'exécution pourront s'exécuter sur l'ensemble des unités de traitement disponibles. Cependant, tel que mentionné au début de cette section, plus l'écart est grand entre le nombre de fils d'exécution et le nombre d'unités de traitement disponible et plus les changements de contexte sont fréquents, ce qui entraîne un impact non-négligeable sur le nombre de conversations simultanées pouvant être supportées par le système. L'analyse des résultats obtenus avec cette technique présentée à §4.2 met en évidence des données sur l'impact des changements de contexte fréquents dû à un nombre élevé de fils d'exécution.

3.4 Modèle à un fil d'exécution par session RTP avec E/S bloquantes

Tout comme le modèle présenté à §3.3, le modèle présenté ici utilise un fil d'exécution par session, mais avec des E/S bloquantes. Il est possible, avec un modèle à un fil d'exécution par session, d'utiliser les E/S bloquantes, puisque l'attente sur les E/S d'une session n'impacte pas les fils d'exécution des autres sessions, contrairement aux modèles pour lesquels un même fil d'exécution gère plusieurs sessions.

Tel que présenté dans l'exemple de code suivant, ce modèle bloque (méthode WaitForIncomingData) l'exécution des fils d'exécution jusqu'à l'arrivée du prochain paquet RTP sur la session affectée au fil d'exécution.

```
...

auto pThread = make_shared<thread>(thread([pSession]
{
  while (executing)
  {
    // Lire le ou les paquets de pSession, si disponible, de façon
    // bloquante. Attendre un maximum d'une seconde
    bool dataAvailable = false;
    pSession->WaitForIncomingData(CMaxWaitTime, &dataAvailable);
    if (dataAvailable)
    {
      pSession->Poll();

      // Vérifier si des paquets sont prêts à être traités
      if (pSession->GotoFirstSourceWithData())
      {
        do
        {
          RTPPacket *packet;
          while ((packet = pSession->GetNextPacket()) != nullptr)
          {
            ...
          }
        } while (pSession->GotoNextSourceWithData());
      }
    }
  }
  ...
}));

...
```

Ce modèle donne des résultats un peu meilleurs (voir chapitre 4) que son équivalent avec E/S non-bloquantes §3.3. Plus l'écart est grand entre le nombre de fils d'exécution et le nombre d'unités de traitement disponibles et plus les changements de contexte sont fréquents. Aussi,

ce modèle est généralement plus facile à mettre en place, car il ne requiert pas de prendre certaines précautions pour ne pas consommer tout le temps de traitement disponible §3.2.

3.5 Modèle à un fil d'exécution par unité de traitement

Le modèle à un fil d'exécution par unité de traitement est en quelque sorte un hybride entre le modèle à un seul fil d'exécution pour l'ensemble des sessions §3.1 et celui à un fil d'exécution par session (§3.3 et §3.4). Pour que ce modèle fonctionne bien, les sessions à traiter doivent être équitablement partagées entre les fils d'exécution, par conséquent entre les différentes unités de traitement.

Tout comme les modèles à un fil d'exécution par session, le modèle à un fil d'exécution par unité de traitement tend à exploiter équitablement l'ensemble des unités de traitement disponibles. Contrairement aux modèles à un fil d'exécution par session, toutefois, celui-ci génère moins de changements de contexte; puisque le nombre de fils d'exécution ne croît pas avec le nombre de sessions. Conséquemment, ce modèle permet de pallier un problème présent avec les modèles à un fil d'exécution par session pour lesquels les performances sont inversement proportionnelles au nombre de fils d'exécution quand ce nombre est suffisamment grand [48].

La première étape pour mettre en place ce modèle consiste à connaître le nombre d'unités de traitement disponible sur la machine, car la quantité de fils d'exécution à créer est directement basée sur ce nombre.

```
auto coreCount = thread::hardware_concurrency();
```

Ensuite, un fil d'exécution est créé pour chaque unité de traitement, et chaque session à traiter est affectée à un seul fil d'exécution, responsable de la session. Puisqu'une certaine portion des opérations du système en référence est constituée d'opérations d'E/S, la création de plus d'un fil d'exécution par unité de traitement pourrait donner de bons résultats (§4.4 en fait la comparaison).

Tel que présenté dans l'exemple de code suivant, chaque fil d'exécution reçoit un pointeur sur une liste de sessions lui ayant été affectée, et itère sur ces sessions à une période de vingt ms.

```
...

// Un fil d'exécution est créé par unité de traitement, et reçoit un
// pointeur sur une liste de sessions à traiter
auto pThread = make_shared<thread>(thread([pVectorOfSessions]
{
  while (executing)
  {
    // Itérer sur chaque session affectée au fil d'exécution.
    for (auto pSession : pVectorOfSessions)
    {
      // Lire le ou les paquets de pSession, si disponible, et de façon
      // non-bloquante.
      pSession->Poll();

      // Vérifier si des paquets sont prêts à être traités.
      if (pSession->GotoFirstSourceWithData())
      {
        do
        {
          RTPPacket *packet;

          while ((packet = pSession->GetNextPacket()) != nullptr)
          {
            ...
          }
        } while (pSession->GotoNextSourceWithData());
      }
    });

    ...
  }
}));

...
```

Parmi les modèles de cet essai pour lesquels la mise en place du parallélisme se fait strictement à travers l'utilisation directe de fils d'exécution (§3.3, §3.4 et le modèle courant), celui-ci offre les meilleurs gains, c'est-à-dire qu'il est celui pouvant maintenir le plus grand nombre de conversations simultanées.

3.6 Modèle basé sur le mécanisme *async*

Bartosz Milewski fait une critique intéressante du mécanisme *async/future* tel que spécifié dans le standard C++ 11 [42]. Il reproche entre autres au standard C++ de ne pas imposer l'utilisation d'un regroupement de fils d'exécution (*thread pool*) pour l'exécution des tâches

créées par le mécanisme *async*. Cela aurait permis d'atténuer l'impact coûteux relié à la création d'un fil d'exécution [44].

Bien que le regroupement de fils d'exécution ne soit pas imposé par le standard C++ 11, l'analyse des résultats §4.5 suite à la mise en place du parallélisme à travers le mécanisme *async* montre que le compilateur C++ (Microsoft Visual Studio 2012) utilisé pour cet essai exécute les tâches créées avec *async* sur des fils d'exécution provenant d'un regroupement, tel que suggéré par Bartosz Milewski. Il faut ainsi noter que la création d'une tâche n'implique pas nécessairement la création d'un fil d'exécution.

Contrairement aux modèles présentés précédemment (§3.3, §3.4 et §3.5), le modèle basé sur le mécanisme *async* met en place du parallélisme sans manipulation directe de fils d'exécution. L'utilisation de ce mécanisme se fait d'une façon très simple, c'est-à-dire qu'à une période de vingt ms, une nouvelle tâche est créée pour chacune des sessions à traiter. Chaque tâche est ainsi responsable de traiter séquentiellement les opérations d'une seule session, pour laquelle en moyenne un seul paquet RTP sera en attente de traitement.

```
...

while (executing)
{
  ...

  vector<future<int>> futures;

  for_each(vectorSessions.begin(), vectorSessions.end(),
          [&]
          (shared_ptr<RTPSession> & pSession)
  {
    // Un fil d'exécution du regroupement sera utilisé pour la session
    futures.push_back(async(launch::async, [pSession]
    {
      // Lire le ou les paquets de pSession, si disponible, de façon
      // non-bloquante
      pSession->Poll();

      // Vérifier si des paquets sont prêts à être traités.
      if (pSession->GotoFirstSourceWithData())
      {
        do
        {
          RTPPacket *packet;

          while ((packet = pSession->GetNextPacket()) != nullptr)
          {
```

```
            ...
          }
        } while (pSession->GotoNextSourceWithData());
      }
    }));
  });

  // Attendre le résultat de chacun des appels à async() avant de continuer
  for (auto &f : futures)
  {
    f.wait();
  }

  ...
}
```

Contrairement aux modèles à un fil d'exécution par session (§3.3 et §3.4), pour lesquels un fil d'exécution est responsable d'une session pour la durée de vie de celle-ci, la durée de vie des tâches dans le contexte présent est brève.

En comparaison avec les modèles à un fil d'exécution par session, le modèle présent exploite un regroupement de fils d'exécution, reposant ainsi sur considérablement moins de fils d'exécution, et génère donc moins de changements de contexte. Ceci devrait résulter en une moins grande perturbation de l'antémémoire §1.5.5. Par le fait même, ce modèle permet de maintenir un plus grand nombre de conversations simultanées que le modèle à un fil d'exécution par session.

3.7 Modèle de tâches de base

Le modèle de tâches de base que propose la bibliothèque PPL (*Parallel Patterns Library*, §2.2) du *Concurrency Runtime* est similaire au modèle précédent exploitant le parallélisme à travers l'utilisation du mécanisme *async*. La différence ici est que les tâches créées à travers la bibliothèque PPL sont exécutées sur un ordonnanceur coopératif.

Le système en référence se prête bien à l'utilisation d'un ordonnanceur coopératif puisque le traitement de l'ensemble des sessions doit se faire à l'intérieur d'un cycle de 20 ms et ce, les unes à la suite des autres, sans dépendre d'une équité entre les tâches comme sur un ordonnanceur préemptif.

Comme le présente l'exemple de code suivant, la mise en place de ce modèle se fait simplement en appelant la méthode parallel_for_each de PPL.

```
...

while (executing)
{
  ...

  // Une tâche sera créée pour chaque session, tâche responsable d'exécuter
  // séquentiellement toutes les opérations de sa session assignée
  parallel_for_each(vectorSessions.begin(), vectorSessions.end(),
                    [&]
                    (shared_ptr<RTPSession> & pSession)
  {
    // Lire le ou les paquets de pSession, si disponible, de façon
    // non-bloquante
    pSession->Poll();

    // Vérifier si des paquets sont prêts à être traités.
    if (pSession->GotoFirstSourceWithData())
    {
      do
      {
        RTPPacket *packet;

        while ((packet = pSession->GetNextPacket()) != nullptr)
        {
          ...
        }
      } while (pSession->GotoNextSourceWithData());
    }
  });
}

...
```

Tel que décrit à §2.1, le *Concurrency Runtime* exécute les tâches de façon coopérative §2.1.3, et selon la disponibilité des fils d'exécution créés initialement par l'ordonnanceur (*Task Scheduler* §2.1.2) du *Concurrency Runtime* en fonction du nombre d'unités de traitement disponibles sur la machine. Conséquemment, en comparaison avec les modèles à un fil d'exécution par session, ce modèle de tâches repose sur considérablement moins de fils d'exécution, et génère ainsi considérablement moins de changements de contexte; ce qui devrait entraîner une moins grande perturbation de l'antémémoire §1.5.5. Ce modèle permet de maintenir un plus grand nombre de conversations simultanées que le modèle à un fil d'exécution par session.

Ce modèle avec tâches utilise les E/S de façon non-bloquante, il n'en demeure pas moins que chaque opération d'E/S est relativement longue, principalement dû à une communication avec le noyau du système d'exploitation. Conséquemment, une partie du temps d'exécution d'une tâche se trouve suspendue en attente de la complétion d'opérations d'E/S. Dans ce cas, il est possible d'indiquer au *Task Scheduler* qu'il peut opérer en mode de surutilisation, de sorte que le *Resource Manager* lui affectera plus de processeurs virtuels qu'il n'y a de cœurs physiques §2.1.1. Le code suivant présente une façon simple de l'indiquer au *Task Scheduler*.

```
Context::Oversubscribe(true);
...
Context::Oversubscribe(false);
```

Il est aussi possible d'indiquer au *Task Scheduler* un minimum et un maximum de processeurs virtuels qu'il peut demander au *Resource Manager*, tel que présenté dans l'exemple suivant pour lequel un minimum de deux et un maximum de cinq processeurs virtuels sont demandés; le 2 passé en paramètre au *SchedulerPolicy* indique que deux propriétés suivent.

```
auto coreCount = thread::hardware_concurrency();

SchedulerPolicy schedulerPolicy(
  2,
  MinConcurrency, coreCount * 2,
  MaxConcurrency, coreCount * 5);
Scheduler::Create(schedulerPolicy)->Attach();
```

De cette façon, il est possible de trouver de façon empirique le minimum et le maximum de processeurs virtuels à utiliser afin d'obtenir les meilleurs résultats sur une machine donnée.

Les modèles de §3.8 essaient d'exploiter le meilleur des deux mondes, c'est-à-dire une utilisation mixte de fils d'exécution et de tâches.

3.8 Modèle hybride avec fils d'exécution et tâches

Les modèles hybrides avec fils d'exécution et tâches tentent de trouver une façon encore plus efficace d'exploiter l'ensemble des unités de traitement disponibles. Pour ce faire, les fils d'exécution sont utilisés seulement pour les E/S, c'est-à-dire pour lire les paquets RTP, selon le modèle à un fil d'exécution par unité de traitement qui, pour les modèles basés sur les fils

d'exécution, est celui offrant les meilleurs résultats (voir le chapitre 4). La conversion d'un codec vers un autre de l'audio contenu dans les paquets RTP, plus exigeante en terme de temps de traitement, se fait à travers l'utilisation de tâches, selon l'hypothèse qu'en limitant les changements de contexte par l'utilisation d'un ordonnancement coopératif permettra de maintenir un plus grand nombre de sessions concurrentes.

3.8.1 Fils d'exécution et tâches en parallèle

Ce modèle exécute en parallèle les fils d'exécution responsables de lire les paquets RTP ainsi que les tâches responsables de convertir d'un codec à un autre l'audio contenu dans les paquets RTP. Cependant, puisque la bibliothèque JRTPLIB n'est pas réentrante [35] pour une session donnée, il faut s'assurer de bien protéger les accès à la session pour éviter qu'une tâche – responsable de la conversion d'un codec vers un autre – et un fil d'exécution – responsable des E/S – y accèdent simultanément.

Tout comme le modèle à un fil d'exécution par unité de traitement §3.5, chaque fil d'exécution responsable de lire et construire les paquets RTP reçoit un pointeur sur une liste de sessions lui ayant été affectée, et itère à travers l'ensemble de ces sessions à une période de vingt ms.

```
...

// Un fil d'exécution est créé par unité de traitement, et reçoit un
// pointeur sur une liste de sessions à traiter
auto pThread = make_shared<thread>(thread([pVectorOfSessions]()
{
  while (executing)
  {
    // Itérer sur chaque session affectée au fil d'exécution
    for (auto pSession : pVectorOfSessions)
    {
      // Protéger l'accès à la session, sinon la même session
      // pourrait être manipulée simultanément par une tâche
      lock_guard<mutex> lock(pSession->mutex);

      // Lire le ou les paquets de pSession, si disponible, de façon
      // non-bloquante
      pSession->Poll();
    });

    ...
  }
}));
```

```
...
```

Tout comme le modèle de tâches de base §3.7, l'utilisation des tâches dans ce modèle se fait d'une façon très simple, c'est-à-dire qu'à une période de vingt ms, une nouvelle tâche est créée pour chacune des sessions à traiter. Chaque tâche est ainsi responsable de traiter séquentiellement les opérations d'une seule session, pour laquelle en moyenne un seul paquet RTP sera en attente de traitement.

```
...

while (executing)
{
  // Une tâche est créée pour chaque session. Cette tâche est responsable
  // d'exécuter séquentiellement certaines opérations de sa session
  parallel_for_each(vectorSessions.begin(), vectorSessions.end(),
           [&]
           (shared_ptr<RTPSession> & pSession)
  {
    // Protéger l'accès à la session, sinon la même session
    // pourrait être manipulée simultanément par un fil d'exécution
    lock_guard<mutex> lock(pSession->mutex);

    // Vérifier si des paquets sont prêts à être traités
    if (pSession->GotoFirstSourceWithData())
    {
      do
      {
        RTPPacket *packet;
        while ((packet = pSession->GetNextPacket()) != nullptr)
        {
          // Traiter les paquets préalablement lus.
          vector<unsigned char> audioConverted =
            ConvertAudio(packet->GetPayloadData(),
                         packet->GetPayloadLength()

          // Envoyer le paquet avec l'audio convertis vers
          // la destination.
          pSession->SendPacket(audioConverted.data(),
                               audioConverted.size(),
                               0, false, CInterval);

          pSession->DeletePacket(packet);
        }
      } while (pSession->GotoNextSourceWithData());
    }
  });

  ...
}

...
```

3.8.2 Fils d'exécution et tâches séquentiellement

Kenny Kerr, dans un de ses articles sur le parallélisme [30], mentionne que le simple fait de regrouper les opérations d'E/S peut grandement améliorer les performances :

> « *The ability to have multiple I/O requests active at any given time has the potential to improve performance dramatically. Storage and network drivers are designed to scale well as more I/O requests are in flight. […]In the case of network drivers, more requests mean larger network packets, optimized sliding window operations and more.* »[2] ([30]).

Le modèle de parallélisme présenté dans cette section met en pratique cette recommandation. Contrairement au modèle précédent §3.8.1, pour lequel les fils d'exécution responsables des E/S s'exécutent simultanément avec les tâches, ce modèle-ci exécute d'abord les fils d'exécution responsables des E/S, ensuite les tâches. Ceci permet de concentrer les E/S pour toutes les sessions au début de chaque cycle de vingt ms et ce, sans que les fils d'exécution ne soient perturbés par l'exécution des tâches, et vice versa.

Tel que présenté dans l'exemple de code suivant, à la fin de chaque cycle, la méthode threadsSynchonizer.WorkerThreadCycleDoneAndBlocked() est appelée afin de suspendre l'exécution des fils d'exécution pour faire place aux tâches chargées de traiter les paquets RTP lus par les fils d'exécution.

```
...

auto coreCount = thread::hardware_concurrency();

...

// Entité responsable de la synchronisation entre les fils d'exécution et
// les tâches
ThreadsSynchronizer threadsSynchonizer(coreCount);

...
```

[2] La possibilité d'avoir plusieurs requêtes d'E/S actives au même moment a le potentiel d'améliorer de façon substantielle les performances. Les pilotes de stockage et de réseau sont conçus pour mieux performer lorsque plusieurs requêtes d'E/S sont en cours. […] Dans le cas des pilotes de réseau, plus il y a de requêtes en attente et plus il y a de possibilités de les regrouper afin de les gérer en lots, permettant ainsi d'optimiser les opérations. (Traduction libre)

```
// Un fil d'exécution d'E/S est créé par unité de traitement, et reçoit un
// pointeur sur une liste de sessions à traiter
auto pThread = make_shared<thread>(thread([pVectorOfSessions,
                                    &threadsSynchonizer]
{
  while (executing)
  {
    // Itérer sur chaque session affectée au fil d'exécution
    for (auto pSession : pVectorOfSessions)
    {
      // Lire le ou les paquets de pSession, si disponible, de façon
      // non-bloquante
      pSession->Poll();
    });

    // Indiquer que le fil d'exécution courant a complété un cycle, et
    // attendre un signal avant de reprendre un prochain cycle
    threadsSynchonizer.WorkerThreadCycleDoneAndBlocked();
  }
}));

...
```

Tout comme le modèle de tâches de base §3.7, l'utilisation des tâches dans ce modèle se fait d'une façon très simple, c'est-à-dire qu'une nouvelle tâche est créée pour chacune des sessions à traiter. Chaque tâche est ainsi responsable de traiter séquentiellement les opérations d'une seule session, pour laquelle en moyenne un seul paquet RTP sera en attente de traitement. Tel que le montre l'exemple de code suivant, la méthode threadsSynchonizer.WaitForAllWorkerThreadsCycleDone() permet de suspendre le fil d'exécution principal, responsable de la création des tâches, pendant que les fils d'exécution d'E/S sont en exécution. Une fois les tâches complétées pour un cycle, la méthode threadsSynchonizer.UnblockAllWorkerThreads() réveille les fils d'exécution d'E/S pour reprendre un nouveau cycle.

```
...

while (executing)
{
  // Le fil d'exécution principal responsable de la création des tâches
  // attend que les fils d'exécution aient terminé leur cycle
  threadsSynchonizer.WaitForAllWorkerThreadsCycleDone();

  // Une tâche sera créée pour chaque session. Cette tâche est responsable
  // d'exécuter séquentiellement certaines opérations de sa session
  parallel_for_each(vectorSessions.begin(), vectorSessions.end(),
           [&]
           (shared_ptr<RTPSession> & pSession)
  {
```

```
    // Vérifier si des paquets sont prêts à être traités
    if (pSession->GotoFirstSourceWithData())
    {
      do
      {
        RTPPacket *packet;
        while ((packet = pSession->GetNextPacket()) != nullptr)
        {
          // Traiter les paquets préalablement lus.
          vector<unsigned char> audioConverted =
            ConvertAudio(packet->GetPayloadData(),
                         packet->GetPayloadLength()

          // Envoyer le paquet avec l'audio converti vers
          // la destination
          pSession->SendPacket(audioConverted.data(),
                               audioConverted.size(),
                               0, false, CInterval);

          pSession->DeletePacket(packet);
        }
      } while (pSession->GotoNextSourceWithData());
    }
  });

  // Dormir le temps restant, soit 20ms moins le temps requis pour les
  // opérations faites par les fils d'exécution ainsi que le temps requis
  // pour les opérations faites par les tâches, et ce sur l'ensemble
  // des sessions
  ...

  // Indiquer aux fils d'exécution qu'ils peuvent reprendre un nouveau
  // cycle
  threadsSynchonizer.UnblockAllWorkerThreads();
}

...
```

La classe suivante sert à synchroniser un fil d'exécution, typiquement le fil d'exécution principal, avec N fils d'exécution de travail, où N signifie par exemple un fil d'exécution par unité de traitement. Le fil d'exécution principal attend alors que les N fils d'exécution de travail terminent leur traitement, puis les met en attente tant qu'il n'a pas à son tour complété son traitement.

```
class ThreadsSynchronizer
{
private:
  mutex m_mutexMainThreadWaiting;
  condition_variable m_cvMainThreadWaiting;
  mutex m_mutexWorkerThreadsWaiting;
  condition_variable m_cvWorkerThreadsWaiting;
  unsigned int m_countNotify;
  unsigned int m_waitForNotifyCount;
```

```
public:
  ThreadsSynchronizer(int waitForNotifyCount) :
    m_countNotify(0),
    m_waitForNotifyCount(waitForNotifyCount)
  {
  }

  // Méthode appelée par les fils d’exécution de travail, dont l’exécution
  // sera suspendue jusqu’à ce que la méthode UnblockAllWorkerThreads()
  // soit appelée par le fil d’exécution principal
  void WorkerThreadCycleDoneAndBlocked()
  {
    unique_lock<mutex>
      lockWorkerThreadsWaiting(m_mutexWorkerThreadsWaiting);
    {
      lock_guard<mutex> lockMainThreadWaiting(m_mutexMainThreadWaiting);
      ++m_countNotify;
      m_cvMainThreadWaiting.notify_one();
    }
    m_cvWorkerThreadsWaiting.wait(lockWorkerThreadsWaiting);
  }

  // Méthode appelée par le fil d’exécution principal, dont l’exécution
  // sera suspendu jusqu’à ce que tous les fils d’exécution de travail
  // aient complété leur traitement et soient suspendus sur la méthode
  // WorkerThreadCycleDoneAndBlocked()
  void WaitForAllWorkerThreadsCycleDone()
  {
    unique_lock<mutex> lock(m_mutexMainThreadWaiting);

    // La condition suivante gère le cas où la méthode
    // WorkerThreadCycleDoneAndBlocked a déjà été appelée par tous les
    // les fils d’exécution de travail avant que la méthode
    // WaitForAllWorkerThreadsCycleDone soit appelée par le fil
    // d’exécution principal
    if (m_countNotify >= m_waitForNotifyCount)
    {
      m_countNotify = 0;
      lock.unlock();
      return;
    }
    m_cvMainThreadWaiting.wait(lock, [&]
    {
      if (m_countNotify >= m_waitForNotifyCount)
      {
        m_countNotify = 0;
        return true;
      }
      return false;
    });
  }

  // Méthode appelée par le fil d’exécution principal permettant de
  // débloquer tous les fils d’exécution de travail préalablement
  // suspendus
  void UnblockAllWorkerThreads()
  {
    lock_guard<mutex> lock(m_mutexWorkerThreadsWaiting);
    m_cvWorkerThreadsWaiting.notify_all();
  }
};
```

Il est facile de constater que la mise en place de ce modèle requiert un effort de travail de loin supérieur aux modèles précédents; il faut donc que cet effort supplémentaire permette de supporter un nombre plus élevé de conversations simultanément. Le chapitre 4 présente les résultats obtenus pour chacun des modèles présentés au chapitre 3.

Chapitre 4
Analyse des résultats

Ce chapitre fait l'analyse des résultats obtenus suite à l'intégration au système en référence des modèles de parallélisme présentés au chapitre 3. L'exécution du système avec les différents modèles s'est faite sous le système d'exploitation Windows 7 avec un processeur Intel Core i7 2635QM [36] à quatre cœurs physiques - dont deux threads d'exécution (*threads hardware*) par cœur - pour un total de huit unités de traitement.

En plus du système en référence, deux autres systèmes ont été développés : un premier pour l'envoi de paquets RTP au système en référence, et un second pour recevoir les paquets transcodés par le système en référence. Le premier système est responsable de transmettre 10000 paquets RTP par session vers le système en référence, tandis que le second vérifie que les paquets ont bel et bien été transcodés puis retransmis par le système en référence.

Les différents modèles de parallélisme sont comparés en fonction de métriques communes, définies dans le Tableau 1. Les métriques obtenues proviennent exclusivement du processus exécutant le système en référence, modifié afin d'intégrer chacun des modèles de parallélisme présenté au chapitre 3. L'outil *Process Explorer* [37] a été utilisé pour l'obtention des métriques telles que l'utilisation du CPU et le nombre de changements de contexte. Une métrique importante qui ne se retrouve pas dans ce tableau est la quantité de conversations simultanées que le système peut maintenir à l'intérieur d'un cycle de vingt ms, qui est calculé en fonction de la perte ou non de paquets RTP entre le programme responsable de transmettre les paquets et celui responsable de les recevoir.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
Le nombre de sessions RTP simultanées utilisées pour l'obtention des autres métriques	L'utilisation moyenne du CPU par seconde	Nombre moyen de changements de contexte par seconde pour le traitement de l'ensemble des paquets, pour l'ensemble des sessions

Tableau 1 Présentation des métriques

La section 1.3.1 présente deux types de coûts causés par les changements de contexte, le premier causé par la gestion même des changements de contexte, et le second relié à la perturbation de l'antémémoire suite aux changements de contexte. L'analyse des résultats des différents modèles montre que le système en référence génère très peu de défauts de cache, de sorte que le second type de coût a peu d'impact sur sa performance; par contre, pour d'autres types de systèmes répartis gérant aussi des sessions mais générant plus de défauts de cache, l'impact de ce second type de coût pourrait être plus grand impact. Dans un tel cas, l'utilisation d'un modèle basé sur des tâches (avec ordonnanceur coopératif) pourrait aider à limiter les impacts de ce second type de coût, donc offrir encore de meilleurs gains sur une machine à plusieurs cœurs.

4.1 Modèle à un seul fil d'exécution

L'analyse des résultats du tableau suivant montre qu'avec 750 conversations concurrentes, 11% du temps processeur disponible est utilisé en moyenne, ce qui représente pratiquement 100% d'une des huit unités de traitement disponibles. Ainsi, ce modèle peut gérer un maximum de 750 conversations simultanément et ce, indépendamment du nombre d'unités de traitement. Au-delà de 750, il commence à y avoir des pertes de paquets puisque le système est incapable de traiter l'ensemble des paquets à l'intérieur de l'intervalle de vingt ms.

750 conversations simultanées est donc la référence du modèle initial de base à un seul fil d'exécution. Les autres modèles ont ainsi comme objectif d'exploiter l'ensemble des unités de traitement disponibles afin de supporter un nombre plus élevé de conversations simultanément.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
500	~7%	~375
750	~11%	~450

Tableau 2 Résultats pour le modèle à un fil d'exécution

4.2 Modèle à un fil d'exécution par session RTP avec E/S non-bloquantes

Dû à un nombre élevé de changements de contexte causé par une quantité importante de fils d'exécution (soit un fil d'exécution associé à chacune des sessions RTP, donc 1000 fils

d'exécution pour 1000 sessions), le modèle à un fil d'exécution par session RTP avec E/S non-bloquantes maintient à peine plus de conversations concurrentes que le modèle initial à un seul fil d'exécution. Le tableau suivant montre que ce modèle peut maintenir jusqu'à 1000 conversations concurrentes et ce, avec presque 100000 changements de contexte par seconde. Au-delà de 1000 conversations simultanées, le système n'est plus en mesure de traiter l'ensemble des paquets des sessions à l'intérieur de l'intervalle de vingt ms.

Il est probable que la quantité élevée de changements de contexte avec ce modèle serait encore plus pénalisante en terme de performance si le système en référence n'avait pas la particularité de générer peu de défauts de cache. Conséquemment, le nombre de conversations simultanées pouvant être maintenues pourrait être bien moins élevé, voire même moins élevé qu'avec le modèle à un seul fil d'exécution.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~27%	~65000
1000	~36%	~93000
1250	~46%	~117000

Tableau 3 Résultats pour le modèle à un fil d'exécution par session avec E/S non-bloquantes

4.3 Modèle à un fil d'exécution par session RTP avec E/S bloquantes

Le modèle à un fil d'exécution par session RTP avec E/S bloquantes génère considérablement moins de changements de contexte que son homologue avec E/S non-bloquantes. Avec les E/S bloquantes, il n'est plus nécessaire de suspendre l'exécution des fils d'exécution entre chacun des cycles de vingt ms (voir 3.2 à propos des précautions à prendre avec les E/S non-bloquantes). Chaque fois qu'un fil d'exécution suspend son exécution, un changement de contexte est fait.

Le tableau suivant montre que le temps d'exécution est en moyenne 10% inférieur à celui obtenu avec la version avec à E/S non-bloquantes. Par contre, à 1500 conversations simultanées, le système cesse brusquement de s'exécuter, faute d'espace disponible sur la pile (*stack*) de l'exécutable. En effet, chaque fil d'exécution utilise une certaine quantité d'espace de la pile (p. ex. : un Mo au minimum sous Windows), et lorsqu'un nouveau fil d'exécution

est créé et qu'il ne reste plus assez d'espace de disponible sur la pile pour lui offrir cet espace, le système cesse brusquement de fonctionner [32].

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~21%	~48000
1000	~29%	~64000
1250	~36%	~76000

Tableau 4 Résultats pour le modèle à un fil d'exécution par session avec E/S bloquantes

4.4 Modèle à un fil d'exécution par unité de traitement

Pour les modèles exploitant le parallélisme seulement à travers l'utilisation des fils d'exécution, celui avec un seul fil d'exécution par unité de traitement offre les meilleurs gains de performances : c'est lui qui maintient le plus grand nombre de conversations simultanées tout en respectant la cadence de vingt ms.

En comparaison avec le modèle de référence, soit le modèle à un seul fil d'exécution, celui à un fil d'exécution par unité de traitement supporte trois fois plus de conversations simultanément et ce, sur une machine à quatre cœurs physiques.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~11%	~2900
1500	~28%	~2100
2250	~49%	~4100

Tableau 5 Résultats pour le modèle à un fil d'exécution par unité de traitement

Le tableau suivant présente les métriques obtenues avec le même modèle, mais cette fois-ci avec deux fils d'exécution par unité de traitement, soit en mode surutilisation des unités de traitement. Ceci a pour but de vérifier s'il est possible d'obtenir plus de gains du fait que les E/S, même non-bloquantes, entraînent un certain délai dû à la communication entre le système et le noyau du système d'exploitation.

Ces métriques montrent que les performances sont sensiblement les mêmes, outre le nombre de changements de contextes qui est moyenne 50% plus élevé. Si le système en référence générait plus de défauts de cache, il y a de fortes chances que tous ces changements de contexte supplémentaires nuiraient davantage aux performances. Cependant, si le système était

encore plus *IO-Bound* – c'est-à-dire que la proportion de code relié aux E/S était encore plus élevée, par exemple sans le transcodage de codecs – une telle surutilisation donnerait probablement des résultats supérieurs à une non-surutilisation.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~12%	~4600
1500	~29%	~3200
2250	~50%	~6300

Tableau 6 Résultats pour le modèle à deux fils d'exécution par unité de traitement

Avec cinq fils d'exécution par unité de traitement, les changements de contexte augmentent de presque 100% en comparaison avec la version à deux fils d'exécution par unité de traitement.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~13%	~8000
1500	~31%	~7600
2250	~52%	~11500

Tableau 7 Résultats pour le modèle à cinq fils d'exécution par unité de traitement

Ces métriques montrent que le modèle offrant les meilleurs gains parmi tous les modèles exploitant le parallélisme uniquement avec une gestion des fils d'exécution et celui à un seul fil d'exécution par unité de traitement. L'hypothèse de cet essai est à l'effet que ce modèle offre les meilleurs gains en mode *standalone*; puisqu'il devrait continuer à offrir les meilleurs gains en mode hybride avec tâches; ainsi, ce modèle est utilisé pour les fils d'exécution des modèles hybrides avec fils d'exécution et tâches §4.7.

4.5 Modèle basé sur le mécanisme *async*

Les résultats obtenus avec ce modèle montre que le compilateur C++ 11 inclus avec Microsoft Visual Studio 2012 exécute les tâches créées à travers le mécanisme *async* par un regroupement de fils d'exécution; peu importe le nombre de sessions RTP en parallèle, il y a toujours une quarantaine de fils d'exécution créés pour le système.

Ce modèle est simple à mettre en place et offre des gains intéressants. Tout comme le modèle à un fil d'exécution par unité de traitement §4.4, il peut supporter jusqu'à 2250 sessions en

parallèle, mais avec une utilisation du CPU plus élevée et environ dix fois plus de changements de contexte.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~23%	~47000
1500	~53%	~50000
2250	~66%	~43000

Tableau 8 Résultats pour le modèle basé sur le mécanisme *async*

4.6 Modèle de tâches de base

Les résultats obtenus avec ce modèle de tâches sont similaires à ceux obtenus avec le modèle à un fil d'exécution par unité de traitement §4.4; ces deux modèles peuvent supporter jusqu'à 2250 conversations simultanément. La grande différence entre ces modèles est la quantité de changements de contexte générés par le modèle de tâches, qui est à peu près dix fois plus élevée que celle du modèle à un fil d'exécution par unité de traitement. Par contre, puisque le *Task Scheduler* du *Concurrency Runtime* exécute les tâches sur un ordonnanceur coopératif, les changements de contexte ne se font pas au milieu de l'exécution d'une tâche, étant plutôt générés par le *Task Scheduler* pour la gestion des différentes tâches. Pour cette raison, les effets de ces changements de contexte sont moins dommageables sur l'antémémoire.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~13%	~37000
1500	~36%	~20000
2250	~45%	~22000

Tableau 9 Résultats pour le modèle de tâches de base

Tout comme pour le modèle à un fil d'exécution par unité de traitement §4.4, pour lequel il a été comparé un fil d'exécution par unité de traitement avec jusqu'à cinq fils d'exécution par unité de traitement (soit une forme de surutilisation), le tableau suivant présente les résultant obtenus lorsqu'une indication est faite au *Task Scheduler* pour chaque opération d'E/S non-bloquantes qu'il peut opérer un mode de surutilisation.

Ces résultats montrent que le fait d'indiquer au *Task Scheduler* qu'il peut opérer en mode de surutilisation n'offre aucun gain de performance; au contraire, la quantité de conversations simultanées que le système peut maintenir passe de 2250 à 1500. Il se peut que cette

fonctionnalité soit intéressante lorsque les opérations d'E/S sont susceptibles d'être longues, par exemple avec des E/S bloquantes.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~15%	~42000
1500	~43%	~28000

Tableau 10 Résultats pour le modèle de tâches de base avec surutilisation

4.7 Modèle hybride avec fils d'exécution et tâches

Pour ces modèles hybrides exploitant les unités de traitement disponibles à la fois avec fils d'exécution et tâches, la partie utilisant les fils d'exécution est basée sur le modèle à un seul fil d'exécution par unité de traitement §3.5, et la partie utilisant les tâches est basée sur le modèle de base de tâches §3.7.

4.7.1 Fils d'exécution et tâches en parallèle

Tout comme le modèle à un fil d'exécution par unité de traitement §4.4 et celui de tâches de base §4.6, ce modèle maintient au maximum 2250 conversations simultanées.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~14%	~42000
1500	~37%	~23000
2250	~49%	~24000

Tableau 11 Résultats pour le modèle hybride à fils d'exécution et tâches en parallèle

4.7.2 Fils d'exécution et tâches séquentielles

Ce modèle met en pratique ce que mentionne Kenny Kerr dans un de ses articles sur le parallélisme [30], c'est-à-dire que le simple fait de regrouper les opérations d'E/S peut grandement améliorer les performances §3.8.2. Pour ce modèle, tout comme pour certains autres modèles du chapitre courant d'ailleurs, les changements de contexte générés sont moins nombreux à 2250 session en parallèle qu'à 750. Ceci laisse croire qu'effectivement à partir d'un certain seuil – soit en fonction du nombre de sessions en parallèle dans le cas présent – plus d'opérations d'E/S peuvent être regroupées pour être traitées en lot, limitant ainsi les changements de contexte nécessaires à leur gestion.

Ce modèle, tout comme celui décrit à §4.7.1, maintient au maximum 2250 conversations simultanées.

Sessions RTP en parallèle	Utilisation CPU	Changements de contexte / seconde
750	~14%	~48000
1500	~36%	~19000
2250	~48%	~21000

Tableau 12 Résultats pour le modèle hybride à fils d'exécution et tâches séquentielles

4.8 Analyse des résultats

Les résultats obtenus montrent que les modèles à un seul fil d'exécution §4.1, à un fil d'exécution par session avec E/S non-bloquantes §4.2 et bloquantes §4.3, sont rarement des choix à privilégier en 2014 pour la mise en place du parallélisme dans un système réparti reposant sur des sessions indépendantes les unes des autres.

Par contre, le fait que le système en référence gère des sessions RTP qui, de par leur nature, ont une durée de vie relativement longue, a pour conséquence que le recours à des fils d'exécution à travers le modèle à un fil d'exécution par unité de traitement §4.4 donne de très bons résultats. Avec ce modèle, l'ensemble des unités de traitement disponibles est utilisé, et les changements de contexte sont peu nombreux. Ce modèle permet de bénéficier des techniques offertes par le système d'exploitation, par exemple l'affinité du processeur, améliorant l'usage de l'antémémoire. D'autres types de systèmes répartis reposant sur des sessions pourraient également offrir de bons résultats avec ce modèle, à condition que la durée de vie des sessions soit aussi relativement longue. De plus, pour un système encore plus *IO-bound*, une surutilisation des fils d'exécution pourrait même être avantageuse.

Le modèle basé sur une utilisation simple des tâches §4.6 offre également de très bons résultats, similaires aux résultats obtenus avec le modèle à un fil d'exécution par unité de traitement. La principale force de ce modèle est sa simplicité : il est mis en place à travers l'utilisation de la fonction parallel_for_each, sans gérer de fils d'exécution ou connaître le nombre d'unité de traitement disponibles sur la machine. Étant donné que le système doit traiter l'ensemble des paquets RTP reçus à chaque vingt ms, l'équité du temps alloué aux différentes tâches n'est pas importante, dans la mesure où les tâches sont toutes complétées à

l'intérieur d'un intervalle d'au plus vingt ms, ce qui permet de tirer avantage d'un ordonnanceur coopératif. Plus il y a de traitements à faire par session, et plus ces traitements nécessitent d'accéder des données en mémoire, plus le recours à un ordonnanceur coopératif offrira des gains intéressants en terme de vitesse d'exécution.

Les modèles hybrides offrent également des résultats similaires à ceux du modèle à un fil d'exécution par unité de traitement et du modèle de tâches de base. Par contre, la mise en place du parallélisme pour ces modèles implique une plus grande charge de travail, surtout dans le cas combinant fils d'exécution et tâches s'exécutant séquentiellement.

Conclusion

L'analyse des résultats des différents modèles de parallélisme présentés au chapitre 4 montre que plusieurs de ces modèles permettent au système en référence de supporter un plus grand nombre de conversations simultanées que la version initiale, qui n'exploite pas le parallélisme. Pour certains de ces modèles, particulièrement celui basé sur les tâches, la mise en place du parallélisme s'est faite d'une façon relativement simple.

Malgré que la mise en place du parallélisme se soit grandement simplifiée – principalement à travers l'utilisation de modèles tels que les tâches et acteurs – il demeure que l'adoption du parallélisme est encore limitée à une catégorie de développeurs davantage exposés aux systèmes dits serveurs ou scientifiques [38]. D'ailleurs, toujours selon [38], un concept qui semble gagner en popularité et permettant d'exploiter plusieurs unités de traitement sans avoir à mettre en place des techniques de parallélisme à même le système, est celui du *coarse-grained parallelism*, qui consiste à exécuter parallèlement plusieurs instances d'un même système, lorsque possible.

Ce concept peut être intéressant lorsque applicable; il est cependant limité quant à la granularité du parallélisme pouvant être mis en place. Cette granularité fait en sorte qu'il soit difficile d'exploiter l'ensemble des opportunités de parallélisme d'un système. Il n'en demeure pas moins que le recours à des fils d'exécution ou à une bibliothèque spécialisée dans la mise en place du parallélisme permet généralement à un système de mieux exploiter l'ensemble des unités de traitement disponibles.

Le chapitre 4 démontre que l'ajout de parallélisme à travers l'utilisation de fils d'exécution et de tâches dans le système en référence permet d'obtenir des gains intéressants sur une architecture SMP à quatre cœurs. Pour compléter le travail entamé par cet essai, il serait intéressant de comparer les différents modèles du chapitre 3 lorsque exécutés sur une architecture NUMA. Puisque l'affinité des processeurs gérée par le système d'exploitation tente le plus possible ré-exécuter un fil d'exécution sur la même unité de traitement –

conséquemment à l’intérieur du même nœud NUMA – il est probable que les modèles basés sur l’utilisation directe des fils d’exécution exploiteraient bien cette architecture. Par contre, pour les modèles strictement basés sur les tâches à travers l’utilisation de la bibliothèque *Concurrency Runtime* présentée au chapitre 2, il faudrait s’assurer que les tâches créées pour gérer une session soient toujours créées sur une unité de traitement du même nœud NUMA. Cette bibliothèque n’offre pas une affinité par cœur, mais plutôt une affinité par nœud. Pour les modèles hybrides avec fils d’exécution et tâches, chaque session devra être manipulée par des fils d’exécution et tâches s’exécutant à l’intérieur d’un même nœud. De plus, il serait pertinent d’utiliser une seconde bibliothèque spécialisée dans la mise en place du parallélisme afin de comparer les résultats obtenus à travers différentes bibliothèques.

En plus de s’assurer de bien supporter l’architecture NUMA, pour compléter le travail entamé, il serait valable d’explorer les résultats pouvant être obtenus à travers l'exploitation du GPU (*Graphics Processing Unit*) [23] conjointement avec les cœurs du CPU. Des bibliothèques spécialisées telles que CUDA et OpenCL permettent en 2014 d'exploiter le GPU pour des opérations autres que celles reliées aux opérations graphiques. Les opérations reliées à l'encodage et le décodage de codecs dans le système en référence seraient un exemple de traitement pouvant être fait par le GPU.

Finalement, une tendance à suivre, et de plus en plus répandue, est de rendre les compilateurs assez « intelligents » pour qu’ils puissent détecter et paralléliser eux-mêmes certaines parties d’un système et ce, avec peu ou pas d’intervention d’un programmeur; soit l’auto vectorisation et « l’autoparallélisation ». Avec toutes ces avancées dans les compilateurs et les langages de programmation, la complexité reliée à la mise en place du parallélisme se déplace du programmeur vers son environnement et ses outils de développement [47].

Liste des références

[1] Olukotun K., Hammond L., The Future of Microprocessors, http://portal.acm.org/citation.cfm?id=1095418, septembre 2005.

[2] Sutter H., *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, http://www.gotw.ca/publications/concurrency-ddj.htm, mars 2005.

[3] Sutter H., Larus J., *Software and the Concurrency Revolution* http://portal.acm.org/citation.cfm?id=1095421, septembre 2005.

[4] Wikipedia, *Multi-Core*, http://en.wikipedia.org/wiki/Multi-core, 5 octobre 2012.

[5] Wikipedia, *Context switch*, http://en.wikipedia.org/wiki/Context_switch, 6 octobre 2012.

[6] Suess M., *What Makes Parallel Programming Hard*, http://www.thinkingparallel.com/2007/08/06/what-makes-parallel-programming-hard/, 6 août 2007.

[7] Suess M., *10 Ways to Reduce Lock Contention in Threaded Programs*, http://www.thinkingparallel.com/2007/07/31/10-ways-to-reduce-lock-contention-in-threaded-programs/, 31 juillet 2007.

[8] Wilson G., Oram A., *Beautiful Code*, juin 2007.

[9] Wikipedia, *Cache coherence*, http://en.wikipedia.org/wiki/Cache_coherence, 23 novembre 2012.

[10] Brandy L., *How We Made Our Face Recognizer 25x Faster*, http://lbrandy.com/blog/2008/10/how-we-made-our-face-recognizer-25-times-faster/, 10 février 2010.

[11] Herlihy M., Shavit N., *The Art of Multiprocessor Programming*, mars 2008.

[12] Chapman B., Jost G., Pas R., *Using OpenMP : Portable Shared Memory Parallel Programming*, octobre 2007.

[13] Wikipedia, *Cache algorithms*, http://en.wikipedia.org/wiki/Cache_algorithms, 8 février 2012.

[14] Wikipedia, *Amdahl's law*, http://en.wikipedia.org/wiki/Amdahl's_law, 8 novembre 2012.

[15] Liu F., Solihin Y., *Understanding the Behavior and Implications of Context Switch Misses*, http://portal.acm.org/citation.cfm?id=1880048, décembre 2010.

[16] Sutter H., *Maximize Locality, Minimize Contention*, http://drdobbs.com/architecture-and-design/208200273, 23 mai 2008.

[17] *Présentation de l'accès NUMA (Non-Uniform Memory Access)*, http://msdn.microsoft.com/fr-fr/library/ms178144.aspx, 29 mars 2011.

[18] Ott D. *Optimizing Software Application for NUMA*, http://drdobbs.com/218401502, 10 juillet 2010.

[19] Microsoft Corporation, *Supporting Systems That Have More Than 64 Processors*, http://msdn.microsoft.com/en-us/windows/hardware/gg463349, novembre 2008.

[20] Campbell C, Miller A., *Parallel Programming with Microsoft Visual C++*, mars 2011.

[21] Microsoft Corporation, *Concurrency Runtime*, http://msdn.microsoft.com/en-us/library/dd504870.aspx, février 2012.

[22] Sutter H., *Welcome to the Jungle*, http://herbsutter.com/welcome-to-the-jungle/, 9 avril 2012.

[23] Wikipedia, *Graphics Processing Unit*, http://en.wikipedia.org/wiki/Graphics_processing_unit, 9 avril 2013.

[24] Sankaralingam K., Arpaci-Dusseau R., *Get the Parallelism out of My Cloud*, http://research.cs.wisc.edu/wind/Publications/hotpar10-cloud.pdf, 2010.

[25] Sutter H., *Break Amdahl's Law!*, http://www.drdobbs.com/cpp/205900309, 17 janvier 2008.

[26] Microsoft Corporation, *Casablanca*, http://msdn.microsoft.com/en-us/devlabs/casablanca.aspx, 14 mai 2012.

[27] Sae-eung S., Arpaci-Dusseau R., *Analysis of False Cache Line Sharing Effects on Multicore CPUs*, http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1001&context=etd_projects, janvier 2010.

[28] Intel, *Intel Core i7-800 and i5-700 Desktop Processor Series – Datasheet – Volume 1*, http://download.intel.com/design/processor/datashts/322164.pdf, juillet 2010.

[29] Wikipedia, *Gustafson's law*, http://en.wikipedia.org/wiki/Gustafson's_law, juin 2012.

[30] Kerr K., *Back to the Future with Resumable Functions*, http://msdn.microsoft.com/en-us/magazine/jj658968.aspx, octobre 2012.

[31] Liesenborgs J., *JRTPLIB*, http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Jrtplib, novembre 2012.

[32] MSDN, *Does Windows have a limit of 2000 threads per process?*, http://blogs.msdn.com/b/oldnewthing/archive/2005/07/29/444912.aspx, juillet 2005.

[34] Dawson B., *Sleep Variation Investigated*, http://randomascii.wordpress.com/2013/04/02/sleep-variation-investigated/, avril 2013.

[35] Wikipedia, *Thread safety*, http://en.wikipedia.org/wiki/Thread_safety, mai 2013.

[36] Intel, *Intel Core i7-2635QM*, http://ark.intel.com/products/53463/, mai 2013.

[37] Microsoft, *Process Explorer*, http://technet.microsoft.com/en-ca/sysinternals/bb896653.aspx, mai 2013.

[38] Binstock A., *Will Parallel Code Ever Be Embraced?*, http://www.drdobbs.com/parallel/will-parallel-code-ever-be-embraced/240003926, juillet 2012.

[39] Lameter C., *NUMA (Non-Uniform Memory Access): An Overview*, http://queue.acm.org/detail.cfm?id=2513149, juillet 2013.

[40] Milewski B., *Beyond Locks and Messages: The Future of Concurrent Programming*, http://bartoszmilewski.com/2010/08/02/beyond-locks-and-messages-the-future-of-concurrent-programming/, août 2010.

[41] Coulouris G.,Dollimore J., Kindberg T., *Distributed Systems: Concepts and Design Edition 3*, 2001.

[42] Milewski B., *Async Tasks in C++11: Not Quite There Yet*, http://bartoszmilewski.com/2011/10/10/async-tasks-in-c11-not-quite-there-yet/, octobre 2011.

[43] Nelson M., *C+11 – Threading Made Easy*, http://marknelson.us/2012/05/23/c11-threading-made-easy/, mai 2012.

[44] Roy P., *Petit regroupement de threads*, http://h-deb.clg.qc.ca/Sujets/Parallelisme/thread_pool.html, novembre 2013.

[45] Wikipedia, *Real-time Transport Protocol*, http://en.wikipedia.org/wiki/Real-time_Transport_Protocol, novembre 2013.

[46] ISO/IEC, *Working Draft, Standard for Programming Language C++*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf, novembre 2012.

[47] Intel, *Auto-Parallelization Overview*, http://software.intel.com/en-us/node/459172, mars 2014.

[48] Li C., Ding C., Shen K,, *Quantifying The Cost of Context Switch*, http://www.cs.rochester.edu/u/cli/research/switch.pdf, 2007.

[49] Wikipedia, *Processor Affinity*, http://en.wikipedia.org/wiki/Processor_affinity, avril 2014.