UNIVERSITÉ DE PARIS SUD
U.F.R SCIENTIFIQUE D'ORSAY

THÈSE
présentée
pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE l'UNIVERSITÉ PARIS XI ORSAY
SPÉCIALITÉ : INFORMATIQUE

Exploitation pratique et efficace du parallélisme sur processeurs multi-cœurs

par

Pierre Palatin

Soutenue le 5 septembre 2008 devant le jury composé de

M.	Lawrence	Rauchwerger	Rapporteur
M.	André	Seznec	Rapporteur
Mme	Nathalie	Drach	Examinatrice
M.	Daniel	Etiemble	Examinateur
M.	Emmanuel	Mogenet	Examinateur
Mme	Christine	Rochange	Examinatrice
M.	Olivier	Temam	Directeur de Thèse

À ma femme, Isabelle

Remerciements

Cette thèse n'a bien entendu pas été le fruit de mon unique travail. Elle a été le produit de riches interactions avec de nombreuses personnes, lors d'innombrables discussions, aussi bien informelles que dans le cadre d'accords de recherche et de conférences. Il est difficile de n'oublier personne, mais pour ne pas déroger à la tradition, j'aimerais remercier tout particulièrement :

Olivier Temam pour m'avoir supporté pendant toute la durée de cette thèse. Je le remercie de la liberté qu'il m'a toujours laissée, tout en réussissant à être toujours présent pour m'encourager et me motiver.

Lawrence Rauchwerger et André Seznec pour avoir accepté la charge de rapporteur de ce mémoire.

Nathalie Drach, Daniel Etiemble, Emmanuel Mogenet et Christine Rochange pour leur participation à mon jury de thèse.

Yves Lhuillier, pour m'avoir aidé lors du début de cette thèse et pour les heures passées à discuter de manière passionnée d'idées aussi diverses que variées.

Sylvain Girbal, avec qui j'ai passé un nombre incalculable d'heures à gérer le serveur d'équipe ainsi que l'inénarrable cluster d'Alchemy.

Toute l'équipe Alchemy, dans laquelle je me suis senti bien pendant ces 3 années de thèse.

Isabelle, avec qui je me serai marié avant la fin de cette thèse, pour son soutien sans faille, mais également pour sa relecture minutieuse de ce manuscrit[1].

Guillaume Wisniewski, qui a effectué sa thèse en même temps que moi, et avec qui j'ai pu avoir des discussions enflammées sur le comment et le pourquoi de la recherche.

Et bien sûr, toute ma famille et mes amis, toujours présents.

[1] Les fautes restantes sont de moi.

Table des matières

Chapitre 1

Introduction

Contexte

Les processeurs ont, en quelques années, changé de rythme d'évolution. Depuis les premiers micro-processeurs, l'augmentation de puissance passait essentiellement par l'exploitation de l'augmentation de la fréquence de fonctionnement. Ainsi, les PC sont passés d'une fréquence de 4.77 Mhz à une fréquence de parfois plusieurs gigahertz. Le marché de l'embarqué a suivi une évolution similaire, ayant cependant généralement une fréquence d'un ordre de magnitude inférieur par rapport à un PC de même génération.

Les différents composants d'une machine n'ont pas suivit la même évolution de fréquence. Typiquement, la mémoire a évolué selon d'autres critères, dont la taille, et n'atteint qu'une fraction de la fréquence du processeur. Cela n'a pas été sans sans conséquence sur l'architecture interne. Il a été possible de tirer parti de cette augmentation de fréquence hétérogène au niveau du processeur via diverses solutions architecturales, dont les principales sont décrites au chapitre 2 de cette thèse. L'intérêt majeur de cette approche a été de permettre d'accélérer les programmes en changeant simplement de génération de processeur. Un ordinateur de génération $N+1$ permettait de gagner en performance sur un même programme par rapport à la génération N.

Mais ces modifications architecturales sont devenues de plus en plus complexes à cause de la différence de vitesse sans cesse grandissante entre le processeur et la mémoire. À partir du début des années 2000, le gain obtenu par l'augmentation de fréquence a commencé à s'essouffler.

Intel avait parié sur cette tendance à l'augmentation de fréquence via la ligne d'architecture appelée « NetBurst ». Cette approche visait à essayer de compenser la complexité vis-à-vis d'une fréquence élevée à l'aide d'un pipeline très profond. C'est à cette période que les fréquences atteignables ont commencé à stagner, autour de 3 Ghz pour des processeurs grand public type Pentium 4.

Simultanément, le marché de l'embarqué manipulait déjà des puces multi-coeurs ou des SoC (« System on Chip », systèmes sur puce). Il est effet souvent nécessaire d'avoir plusieurs types d'unités très spécialisées, comme par exemple un coeur de décodage GSM et un de gestion bluetooth, afin d'avoir une consommation réellement minimale. Cela ne pose de plus pas particulièrement de problème à exploiter puisque les différentes tâches à paralléliser sont claires, bien séparées et généralement adaptées à la plate-forme. Cette simplicité d'utilisation et les avantages associés ont fait que ces processeurs multi-coeurs ou SoC ont commencé à être utilisés pour l'embarqué alors même qu'il était toujours possible d'augmenter les fréquences pour augmenter la performance si nécessaire.

À l'inverse, dans le domaine de l'informatique généraliste, le parallélisme offert par les multi-

coeurs ou les multi-processeurs est difficile à exploiter, surtout face à l'évolution de fréquence qui ne demande aucun effort pour le programmeur. Bien que les systèmes multi-processeurs existaient depuis longtemps pour des machines type x86, leur présence sur le marché restait anecdotique.

Cependant, le ralentissement de la course à la fréquence mentionné plus haut a fait qu'il n'était plus possible de gagner aisément en puissance de calcul. La seule alternative possible était d'arrêter de cacher le parallélisme offert par le nombre de transistors disponibles. Cela a donc conduit à l'apparition de processeurs offrant un parallélisme explicite, laissant son exploitation à charge du programmeur et du système.

Tout comme pour le domaine de l'embarqué, ce phénomène a bien sûr commencé là où ce parallélisme était aisément exploitable, voire un avantage. En l'occurrence, les cartes graphiques ont commencé à introduire des GPU [1] qui, en plus d'une architecture permettant plus d'effets, avaient un niveau de parallélisme élevé en dupliquant simplement les pipelines de traitement. Cela est bien sûr relativement facile à exploiter grâce à la nature généralement hautement parallèle du rendu graphique.

Intel, pour les processeurs x86 grand public, a commencé à proposer du parallélisme explicite via sa technologie HyperThreading, dénomination commerciale du SMT (voir section 2.1.2). L'intérêt initial de cette approche était de proposer du multi-threading rapidement et à moindre coût puisque cela n'a nécessité que peu de modifications par rapport au Pentium 4 standard.

L'évolution de la course à la fréquence vers la course au parallélisme a été rapide. De manière assez intéressante, au début de cette thèse en 2004, aucun processeur grand public disponible chez Intel ou AMD n'était multi-coeur ; seul l'HyperThreading d'Intel commençait à être un peu disponible. Maintenant, en 2008, quasiment toutes les machines grand public vendues sont multi-coeurs. Pour pouvoir faire face à l'évolution des besoins, il est nécessaire de tirer parti d'une forme ou d'une autre de parallélisme.

Il est à noter également qu'un autre paramètre est apparu quasiment simultanément. Si la consommation n'était guère un souci jusqu'à il y a quelques années, il devient un critère prépondérant. Les processeurs comme le Pentium 4 arrivaient à une consommation extrêmement élevée, rendant difficile son alimentation. D'autre part, les périphériques portables prennent également le pas sur les machines sédentaires classiques. Comme la technologie des batteries n'a pas suivi l'évolution de la consommation des processeurs, la consommation devient critique pour l'autonomie. Enfin, les datacenters nécessaires pour suivre l'explosion d'Internet et des sites fournissant toujours plus de fonctionnalités concentrent de nombreuses machines dans des espaces réduits. L'alimentation et le refroidissement deviennent alors un problème majeur pour leur construction. La convergence des intérêts aussi bien du grand public que de l'industrie fait que le problème de la consommation est devenu tout comme le parallélisme un problème majeur en très peu de temps.

Contributions

Cette thèse se place dans ce contexte ; comment exploiter ce parallélisme croissant de manière efficace ?

La recherche de parallélisme dans un problème ou un dans programme donné constitue souvent un problème majeur, il s'agit en soi d'un sujet à part entière. Cette thèse ne traite pas de comment trouver du parallélisme, mais de comment le gérer. Bien sûr, obtenir du parallélisme et le gérer sont des problèmes proches et parfois liés.

[1] « Graphique Processing Unit », processeur d'effets graphiques implémentant matériellement des primitives 22 et 3D afin d'accélerer le rendu graphique.

En fait, en l'état actuel, exprimer simplement le parallélisme d'un code est insuffisant pour l'exécuter réellement efficacement sur une architecture à parallélisme explicite, même sans nécessairement viser la performance maximum théorique. De nombreux frameworks permettant d'exprimer de manière plus ou moins générique le parallélisme d'un code existent, mais peu permettent d'exprimer les informations pratiques permettant d'extraire de la performance de ce parallélisme. En effet, une fois un code parallélisé, la quantité de parallélisme dépasse rapidement les capacités d'un multi-coeur. Par exemple, une fois une boucle complètement parallélisée, toutes les itérations peuvent être exécutées simultanément. Mais cela ne veut pas pour autant dire qu'il est intéressant de lancer chaque itération simultanément sur un processeur avec quelques coeurs. Cela dépasse sa capacité, au risque de faire s'écrouler ses performances. Dès lors, comment choisir quoi exécuter à quel moment ? La plupart des frameworks parallèles offrent des solutions ad-hoc permettant d'éviter ce type d'explosion de parallélisme. Cependant, peu généralisent vraiment cela.

Cette problématique a donné naissance à Capsule. Capsule permet d'avoir des programmes parallèles pouvant adapter dynamiquement leurs degrés de parallélisme afin d'optimiser des contraintes diverses, dont par exemple la performance. Le chapitre 3 décrit les principes généraux de Capsule.

L'approche choisie pour Capsule prend la liberté de modifier architecture et code ; cela a été un des principes directeurs. L'idée sous-jacente est que, pour exploiter du parallélisme, il est nécessaire d'apporter plus d'informations sur l'exécution lors de la conception d'un programme que ce qui est fourni pour un code séquentiel. Or, il est possible d'extraire de l'information à plusieurs niveaux de création et d'exécution d'un programme. Par exemple, si il est souvent facile pour un programmeur de savoir si une boucle peut être parallélisée, cela est difficile à voir au niveau du processeur. À l'inverse, le processeur est capable de réagir directement à la charge de travail afin d'adapter le parallélisme, alors qu'il est souvent difficile pour le programmeur d'estimer la répartition d'une tâche complexe.

Capsule, conformément à cela, comporte plusieurs aspects. La partie matérielle, présentée au chapitre 4, fut la première partie implémentée. Elle apporte des modifications matérielles permettant au processeur de fournir un retour au programme sur le niveau de parallélisme actuel.

La partie logicielle de Capsule est décrite au chapitre 5. Elle fournit une API permettant de manipuler aisément les instructions introduites par la version matérielle. Notons que les parties matérielles et logicielles de Capsule ne sont cependant pas strictement complémentaires. En effet, lors de la construction de la partie logicielle de Capsule, il a été nécessaire d'émuler logiciellement les parties matérielles de Capsule afin de faciliter le test et le débuggage. Ce chapitre montre qu'il est ainsi possible de concevoir du code adaptant dynamiquement son niveau de parallélisme en fonction des conditions à l'exécution, et ce sans support matériel.

Enfin, le chapitre 6 présente un modèle mémoire possible pour Capsule. Ce modèle mémoire est adapté au parallélisme dynamique introduit par Capsule et vise notamment des architectures mémoire irrégulières. Il s'agit avant tout d'un chapitre prospectif.

Chapitre 2

Le parallélisme

2.1 Architectures parallèles

Bien que de nombreuses architectures de processeurs essaient de présenter un modèle simple et séquentiel, le substrat de transistors composant une puce reste intrinsèquement parallèle. De plus, l'évolution des technologies de gravure a provoqué un changement de contraintes, passant d'une limitation vis-à-vis du nombre de transistors disponibles à une limitation de la fréquence atteignable.

Face à ces différentes contraintes, plusieurs évolutions et directions existent afin de fournir à l'utilisateur la possibilité d'opérer toujours plus de traitements à des latences inférieures. La première sous-section traite des architectures qui offrent un modèle type von Neumann, séquentiel ; la seconde sous-section traite d'architectures qui, bien qu'offrant un parallélisme explicite, conservent localement un modèle von Neumann. La troisième sous-section présente quelques architectures qui s'éloignent de la notion de processeur séquentiel simple.

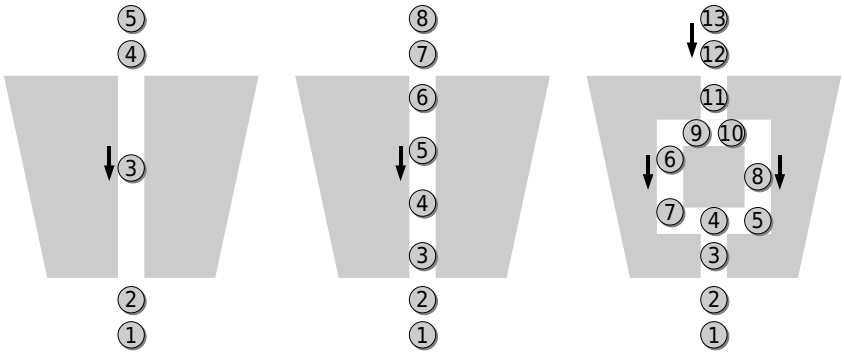
2.1.1 Parallélisme implicite sur architecture von Neumann

Afin de profiter de l'augmentation du nombre de transistors, plusieurs techniques se basant sur du parallélisme implicite ont été créées. Le principe en est de continuer à offrir un modèle conventionnel de von Neumann au programmeur, imposant ainsi une exécution séquentielle ordonnée vue de l'extérieur. Cependant, en interne, le processeur est libre de travailler en parallèle.

Pipelining

La notion de pipeline permet à un processeur d'exécuter plusieurs instructions simultanément en ordre décalé (voir figure 2.1b). Cela se base sur le fait que l'exécution d'une instruction complète est décomposable en plusieurs étapes distinctes [40] : Fetch, Decode, Execute, Memory, Commit. Ainsi, pendant qu'une instruction est à l'étape n, l'étape n-1 est libre pour une autre instruction. On peut ainsi avoir au maximum un nombre d'instructions en vol égal au nombre d'étages ; la vitesse du processeur est alors limitée par l'étape la plus lente du pipeline.

En pratique, les étapes ne sont pas indépendantes, ce qui limite l'ILP (Instruction Level Parallelism) possible. Ainsi, si une instruction dépend du résultat de la précédente pour effectuer son propre calcul, cela introduit un délai dans le pipeline, de même qu'un saut qui peut empêcher de récupérer les instructions suivantes.



(a) Intel 4004 *(b)* Pipelined processor *(c)* Superscalar / Out of Order processor

Figure. 2.1 – Architectures classiques sans parallélisme explicite.

Exécution dans le désordre

Puisqu'il est de toute façon nécessaire de tenir compte des dépendances afin de pouvoir profiter du pipeline, plus rien n'oblige à exécuter la plupart des instructions dans l'ordre. La seule contrainte à respecter est de conserver le modèle séquentiel présenté à l'utilisateur ; il est donc nécessaire de valider le résultat des instructions dans l'ordre.

L'exécution « out of order » (voir figure 2.1c) ou « exécution dans le désordre » se base sur une analyse de dépendances des prochaines instructions afin de pouvoir lancer leur exécution en parallèle lorsque cela est possible. Par exemple, cela permet de lancer la récupération d'une valeur depuis la mémoire avant qu'elle ne soit réellement nécessaire, ce qui a pour effet de cacher la latence d'accès mémoire.

Pour cela, une fenêtre d'instructions sera utilisée. Ainsi, il est facile de faire l'analyse de dépendances et de commencer l'exécution des instructions présentes dans la fenêtre s'il n'y a pas de problème de dépendance apparent.

Superscalaire

Pour que l'exécution dans le désordre puisse être réellement efficace, il faut d'une part avoir des instructions suffisamment indépendantes à disposition, mais également être capable d'en exécuter plusieurs simultanément. Il est possible de profiter du fait que les instructions n'utilisent pas toutes les mêmes zones du processeur ; par exemple, il est possible dans une certaine mesure d'exécuter simultanément un « store » vers la mémoire et une opération arithmétique simple. Cependant, pour profiter effectivement du parallélisme potentiel offert par l'exécution dans le désordre, il est également intéressant de pouvoir exécuter plusieurs instructions similaires simultanément.

Pour cela, les processeurs « superscalaires » multiplient les unités fonctionnelles. Par exemple, le processeur Alpha 21164 dispose ainsi de deux pipelines entiers complets parallèles ainsi que deux pipelines flottant [31].

La notion de super-scalaire n'impose pas nécessaire l'utilisation d'exécution dans le désordre. Le processeur Pentium d'Intel, premier processeur superscalaire grand public, a ainsi un pipeline flottant et deux pipelines entiers, dont l'un capable uniquement de traiter des instructions simples [7].

Unités vectorielles

Enfin, les processeurs récents fournissent désormais souvent des unités vectorielles, en plus des unités entières et flottantes. Le principe en est d'appliquer la même opération (soustraction, mise au carré, ...) sur plusieurs valeurs (formant un vecteur) simultanément. L'utilisateur voit cela comme une série d'instructions dédiées à la manipulation des vecteurs.

Bien que cela ne soit pas vraiment du parallélisme implicite, puisque l'utilisateur doit générer lui-même des instructions vectorielles, le processeur conserve un modèle séquentiel de von Neumann.

2.1.2 Parallélisme explicite avec modèle von Neumann local

Les techniques présentées dans la section précédente permettent dans de nombreux cas d'exploiter efficacement une certaine augmentation du nombre de transistors. Cependant, ces méthodes ont souvent des limites au passage à l'échelle empêchant d'exploiter efficacement encore plus de transistors, comme c'est le cas pour les pipelines [68] ou les fenêtres d'instructions [59]. Le moyen le plus simple pour d'exploiter un nombre quelconque de transistors reste de présenter à l'utilisateur non pas un seul fil d'exécution séquentiel, mais plusieurs. Ainsi, il est possible de profiter aisément de la surface disponible en exploitant la régularité du modèle d'architecture considéré.

Le Transputer

Le processeur Transputer d'INMOS [82] a été un des premiers processeurs conçu pour le parallélisme. Le principe était de faire un processeur simple, peu onéreux mais conçu dans le but de coopérer avec plusieurs autres unités.

Les processeurs Transputer avaient des systèmes de communication conçus de manière à pouvoir en utiliser aisément plusieurs simultanément. Pour ce faire, ils étaient équipés de 4 liens séries en interne ; ainsi, s'il était nécessaire d'en connecter plus de 4, il était possible d'avoir une architecture complexe telle qu'une grille. Les communications nécessitent alors de gérer plusieurs «hops» de routage. En fait, il était même envisageable de connecter des transputers au travers de liens longs de plusieurs mètres, chose qui n'est pas sans rappeler les clusters d'aujourd'hui.

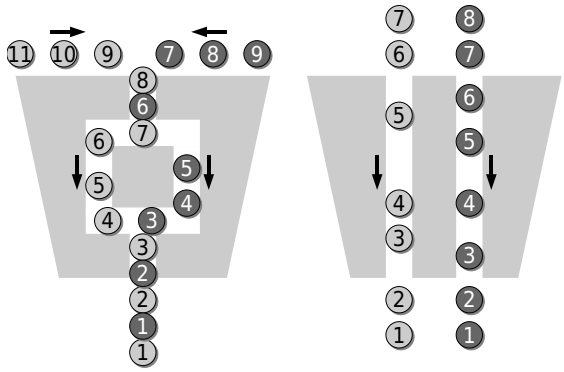
Le transputer intégrait de quoi gérer une exécution multi-tâche simple. Un ordonnanceur léger était capable de prendre en compte les capacités réseau intégrées. Par exemple, lorsque qu'un processus attendait pour une lecture ou une écriture sur le réseau, le Transputer était capable d'ordonnancer un autre processus en attendant le résultat.

Simultaneous Multi-Threading (SMT)

L'idée sous-jacente de l'approche SMT [78] est d'offrir explicitement les ressources d'un processeur superscalaire au programmeur (voir figure 2.2a). Plutôt que le processeur profite d'une large fenêtre d'instructions extraite d'un même pointeur d'instruction dans laquelle il essaye d'extraire un maximum de concurrence, il récupère en permanence plusieurs fils d'instructions ayant chacun leur pointeur d'instruction.

Ainsi, à partir du moment où le programmeur est capable de fournir un nombre de threads suffisant pour occuper les différents threads matériels proposé par le SMT, les unités fonctionnelles peuvent être utilisées simultanément, ce qui n'aurait pas été nécessairement possible avec un seul fil d'exécution ayant des dépendances inter-instructions.

L'intérêt de cette approche est, dans le cas d'un processeur à exécution dans le désordre, de ne nécessiter que peu de modifications. Il faut d'une part modifier l'étape de « fetch » du



(a) SMT processor *(b)* CMP processor

Figure. 2.2 – Architectures à parallélisme de threads explicite.

processeur afin qu'il puisse gérer plusieurs flux d'instructions. Il faut d'autre part que chaque instruction en vol soit identifiée afin de savoir à quel thread elle appartient lors de la phase de « commit ». Puisque, dans le cas d'un processeur à exécution dans le désordre, il est déjà nécessaire de gérer plusieurs versions des registres présentés à l'utilisateur, la gestion des différents threads ne nécessite pas d'augmentation notable des ressources à ce niveau.

Intel a introduit une technologie similaire dans son processeur grand public Pentium 4, appelée commercialement « HyperThreading » [49]. Sun, pour sa gamme serveur, a également crée un processeur faisant appel massivement au SMT, l'UltraSpace T1 (connu également sous le nom de code Niagara) [48]. Chacun de ses 8 coeurs est capable de gérer matériellement 4 threads ; cela permet en partie de compenser l'absence d'exécution dans le désordre et donc la perte d'une grande partie du parallélisme d'instruction.

Chip MultiProcessor (CMP) et System on Chip (SoC)

L'étape suivante pour proposer du parallélisme explicite depuis une seule puce est de simplement dupliquer les coeurs : il s'agit alors de « CMP » pour « Chip Multi Processor ». Toutes les ressources sont alors dupliquées, il n'y a pas de conflit d'unités fonctionnelles entre les différents threads (voir figure 2.2b).

Cependant, certaines parties peuvent être partagées, comme par exemple le cache de niveau 2 dans les Intel Core Duo [22] ; dans ce cas précis, cela évite la charge de la gestion de cohérence, permet une répartition dynamique du cache entre les threads et favorise des threads travaillant sur des zones similaires et pouvant ainsi partager des lignes de cache [44]. AMD fit le choix dès le départ de caches de niveau 2 séparés par coeurs. Cette approche profite directement de l'augmentation de la densité possible de transistors.

Une variante du CMP beaucoup utilisé pour les systèmes embarqués est le « SoC » pour « System on Chip ». Il y a toujours plusieurs coeurs sur une même puce, mais, au lieu de dupliquer un même coeur, plusieurs types de coeurs coexistent sur la puce, reliés par des systèmes de bus, comme pour le processeur OMAP [20]. Le problème de la communication peut prendre ainsi un aspect décisif [62] puisque toutes les interactions entre les coeurs doivent passer par ce bus.

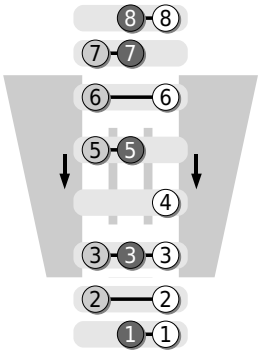


Figure. 2.3 – «Very Large Instruction Word», architecture à parallélisme d'instruction explicite.

Very Large Instruction Word (VLIW)

La possibilité d'avoir un grand nombre d'unités fonctionnelles est également exploitée par les processeur dit « VLIW », pour « Very Long Instruction Word » [34] . Le principe est de regrouper les instructions au sein d'un même thread en paquets de taille fixe, contenant plusieurs instructions exécutables simultanément (voir figure 2.3). Par rapport à l'exécution dans le désordre, le travail de détection et de gestion des dépendances est ici relégué au compilateur, qui doit générer les paquets d'instructions.

L'intérêt de cette approche par rapport au SMT est de fournir des performances plus stables et plus prédictibles. Ainsi, les processeurs VLIW sont adaptés pour des traitements de type flux, où il est relativement simple d'extraire les paquets d'instructions. Cependant, sur du code généraliste, la difficulté de trouver des instructions indépendantes au moment de la compilation rend le gain de performance plus hasardeux que la création de threads indépendants lorsque cela est possible.

Graphic Processing Units (GPU)

Les processeurs de type GPU, pour « Graphic Processing Unit », souvent présents sur les machines grand public pour gérer l'affichage 3D dans les jeux, bien que non généralistes, proposent un parallélisme massif via des traitements spécifiques et adaptés. En effet, il est possible sur des circuits graphiques modernes d'avoir des programmes exécutés sur chaque « vertex » (noeud d'un modèle 3D) et sur chaque pixel de l'image, le tout en soutenant des fréquences de plusieurs images par seconde. Sur les générations récentes, ces programmes, appelés « shaders », sont capables de gérer des instructions complexes, dont des sauts conditionnels.

Le parallélisme à exploiter est simple, puisque chaque traitement est indépendant. Il est donc possible au sein de l'architecture de dupliquer les pipelines de traitement afin d'être capable d'assurer la bande passante nécessaire.

Les possibilités offertes par l'architecture des GPU sont de plus en plus utilisées en tant qu'unité de calcul parallèle générique [57], appelée « GPGPU », pour « General Purpose Graphic Processing Unit ». Par exemple, le calcul de rendu 3D non temps-réel, comme par exemple pour les films d'animation, ne peut généralement pas profiter des GPU classiques, les algorithmes de rendus n'étant pas les mêmes que ceux du rendu temps réel. Avec cette approche des « GPGPU », il devient possible de les utiliser pour ce type de tâches [61].

Nvidia propose cette notion de «GPGPU» au travers de Cuda [15]. Le choix de NVidia n'est pas de compiler des codes génériques directement pour ses processeurs graphiques, mais de

fournir une spécification de machine virtuelle générique. Cette machine virtuelle reste d'assez bas niveau, et surtout conçue pour permettre d'exploiter efficacement et sans perte de performance le GPU sous-jacent.

Ainsi, ce modèle d'architecture virtuelle présente un processeur capable d'exécuter des milliers de threads non persistants simultanément, et dont le nombre peut varier dynamiquement. Les unités d'exécution ont un accès normal à la mémoire, mais des caches de données parallèles proches des threads est disponible afin d'assurer une performance correcte ; ces caches sont gérés explicitement par l'utilisateur et non pas cachés à l'utilisateur comme dans le cas de machine virtuelles plus classiques. Ils sont partagés entre plusieurs threads, ce qui permet de communiquer rapidement entre les unités d'exécution et d'éviter de devoir récupérer les mêmes données plusieurs fois depuis la mémoire. Typiquement, dans le cas de rendus 3D, cela permet de stocker une partie des textures des modèles 3D.

The Cell

The Cell [47], processeur crée par IBM est, en quelque sorte, à mi-chemin entre le processeur généraliste classique et le GPU. Il a été parmi les premiers processeurs généralistes (il est notamment utilisé dans la Playstation 3) ayant une architecture multi-coeurs hétérogène, nécessitant plusieurs jeux d'instructions différents pour le programmer.

L'architecture du Cell est constituée d'un coeur de calcul générique nommé PPE pour « Power Processor Element » et de plusieurs coeurs vectoriels nommés SPE pour « Synergistic Processor Element », le tout étant relié via un bus commun.

Le PPE est un coeur relativement classique, de type Power, in-order. L'aspect in-order a permis de simplifier le coeur et par conséquent de réduire le nombre de transistors nécessaires ; pour éviter une trop grande perte de performance en calcul généraliste par rapport à un processeur dans le désordre, le PPE est en fait un processeur SMT, gérant 2 threads dans la version classique du Cell.

Les SPE, entre 6 et 8 sur les versions standard, proposent un jeu d'instructions spécialisé dans le traitement vectoriel de données. Ces coeurs sont très simples, ce qui permet d'une part d'en mettre beaucoup mais également de les faire fonctionner à haute fréquence.

Afin de permettre aux SPE de fonctionner efficacement et de traiter de gros flux de données parallèles, chaque SPE a sa propre mémoire privée. Le SPE n'ayant pas de cache transparent, cette mémoire est gérée par l'utilisateur. Il donc doit être capable de programmer le transfert des données à traiter par les SPE entre la mémoire centrale et les mémoires locales. À l'inverse, cela permet une optimisation fine dans le cas d'application de streaming par exemple. De plus, à partir du moment où les données sont présentes dans les mémoires locales, les traitements peuvent être effectués efficacement, sans problème de goulot d'étranglement vers la mémoire, puisque chaque SPE travaille sur sa propre zone mémoire.

2.1.3 Au delà des threads von Neumann

RAW

L'approche RAW [74] propose une architecture matérielle parallèle permettant le passage à l'échelle avec l'augmentation de la surface disponible. Elle présente pour cela explicitement à l'utilisateur les problèmes sous-jacent de latence. Le processeur est conçu dès le départ pour gérer efficacement une grille de coeurs, dont le nombre peut être augmenté si nécessaire sans toucher les caractéristiques locales présentées à l'utilisateur. En effet, RAW présente un modèle où les coeurs ne peuvent, de toute façon, communiquer que de voisin à voisin ; ainsi, si des

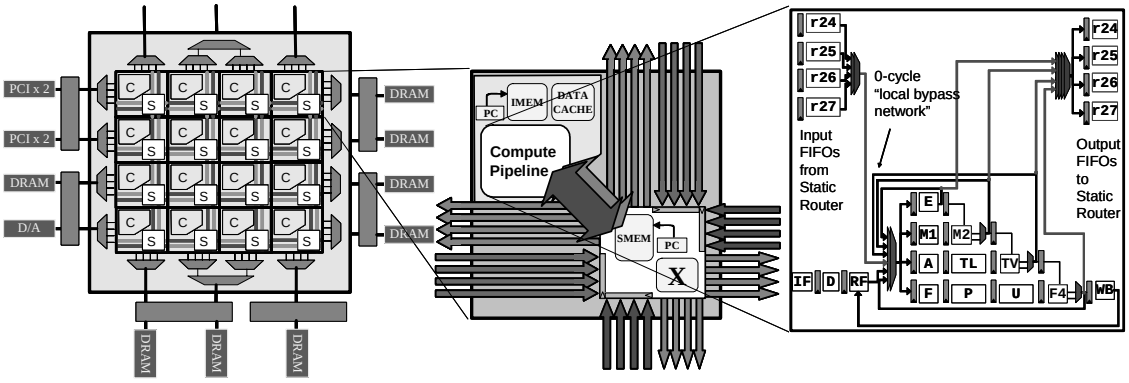


Figure. 2.4 – Représentation en bloc des différents niveaux de l'architecture RAW. (extrait de [74])

coeurs sont ajoutés, les temps de communication locaux ne sont pas affectés et le programme peut néanmoins profiter de la puissance de calcul supplémentaire disponible.

Puisque les communications ne sont que locales, il est nécessaire d'avoir un réseau efficace. RAW propose au sein de chaque coeur un système de routage (voire figure 2.4), gérant 4 réseaux d'interconnexions différents. Chaque routeur de chaque coeur est programmable et a sa propre mémoire d'instructions gérable par l'utilisateur. Il est donc possible d'adapter les politiques de routage à chaque code et programmer des configurations ad-hoc efficaces pour le problème donné. On notera que le fait de programmer le réseau est au coeur même du modèle RAW ; les latences de communication entre les coeurs ne sont pas cachées mais reflètent directement au niveau de l'utilisateur les distances sur la puce. La latence dépend directement du nombre de « hops » de routage.

L'intérêt d'intégrer un routeur au sein de chaque coeur est de permettre d'intégrer directement le réseau dans le fonctionnement du processeur, réduisant d'autant les latences de communication.

Enfin, les entrées/sorties trouvent naturellement leur place sur les bords de la puce, puisqu'à ces endroits il est possible de les interconnecter au réseau, comme le sont les coeurs entre eux.

TRIPS

Les coeurs TRIPS [65] proposent de gérer l'exécution de macro-instructions afin de profiter du parallélisme type dataflow obtenu via de grandes fenêtres d'instructions.

Le processeur fournit une grille de petits éléments de calcul simples, type ALU ; ces éléments sont organisés en couches successives (voir figure 2.5). Le programme est écrit en macro-instructions, mappant des opérandes sur chacun de ces éléments simples. L'exécution au sein d'une macro-instruction se fait à la manière du dataflow, l'exécution sur un élément donné ne se fait que lorsque ses opérandes sont disponibles.

Ce modèle permet d'extraire un parallélisme local, mais reste relativement caché au programmeur puisque les macro-instructions sont habituellement générées par le compilateur. Cependant, vu du compilateur / assembleur, comme il s'agit de dataflow, il est possible de considérer cela comme du parallélisme explicite puisqu'il faut gérer une partie des dépendances à la compilation pour pouvoir extraire efficacement ces macro-blocs.

Il est également possible d'utiliser la grille d'éléments simples afin d'avoir du parallélisme de

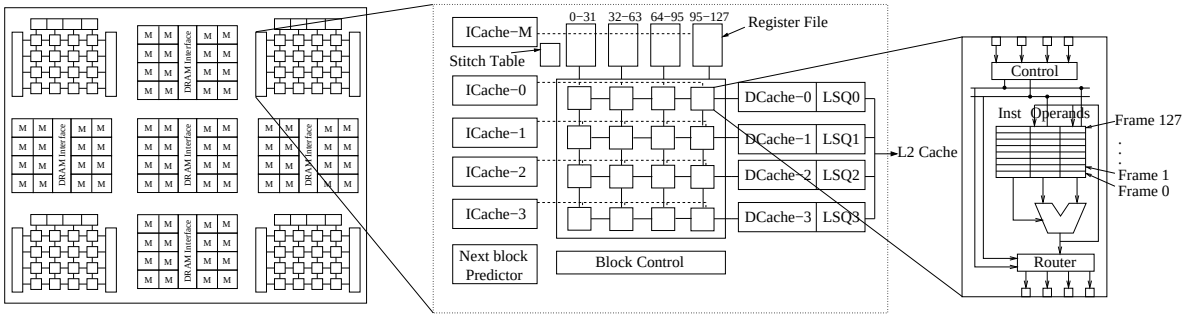


Figure. 2.5 – Représentation des différents niveaux de l'architecture TRIPS. (extrait de [65])

données simple. Il suffit pour cela d'utiliser chaque « colonne » de la grille d'éléments comme un pipeline simple, chacune traitant une partie des données.

Enfin, le processeur TRIPS est capable de changer rapidement et régulièrement le contenu des opérandes sur la grille d'éléments simples. Cela permet d'une part de gérer les multiples macro-instructions qu'un programme réaliste ne manque pas d'avoir. Cela permet d'autre part de gérer une forme de parallélisme de thread, en ordonnançant des macro-instructions tantôt d'un thread, tantôt de l'autre.

SCORE

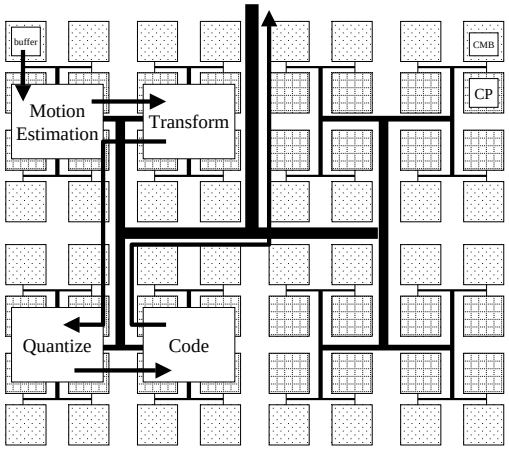


Figure. 2.6 – Exemple d'adaptation d'un encodeur vidéo sur un processeur SCORE ayant suffisamment d'espace pour un traitement parallèle. (extrait de [17])

Le modèle SCORE [17] offre un modèle de processeur programmable, offrant nativement du parallélisme. L'idée sous-jacente est d'offrir un modèle de programmation abstrait permettant d'exécuter un même code sur des architectures aux dimensions diverses, permettant un certain passage à l'échelle.

Ce modèle fournit 3 principaux types d'abstractions :

– Des noeuds de calculs, pouvant être de deux types : automates simples ou Turing-complets ; seuls ces derniers sont autorisés à allouer de la mémoire dynamiquement ou à créer d'autres

noeuds de calcul.
- Des segments de mémoire allouables dynamiquement. Ces segments ont une notion de propriétaire, un seul noeud peut y accéder à la fois. Lorsqu'un noeud essaye d'accéder à un segment ne lui appartenant pas, il est bloqué jusqu'à ce que la propriété du segment lui soit concédée.
- Des « streams » configurables entre une source et une destination uniques. Ces flux sont capable de gérer une notion de pression inverse afin d'éviter des blocages inutiles : si le noeud de destination n'est plus capable d'accepter des données, l'information est transmise au noeud source, qui à son tour peut si nécessaire le faire savoir à ses propres sources de données.

On note également que la concurrence des noeuds est contrôlée, puisque les seules interactions possibles se font selon les « streams ». La figure 2.6 représente ces différents éléments pour le cas d'un encodeur vidéo simple.

C'est l'architecture ou le système qui décident comment mapper tout cela en pratique sur un nombre de coeurs de calculs et de bancs mémoires limités ; cela peut par exemple être fait en utilisant des notions de time-sharing pour exécuter plusieurs étapes de traitement successivement au lieu de parallèlement. Ainsi, un unique programme peut s'exécuter aussi bien sur une petite plateforme embarquée minimale que sur une puce offrant un très large parallélisme.

Le dataflow

Les architectures Dataflow [11][60], populaires lors des années 70-80, avaient pour but d'offrir un support matériel à l'exécution dataflow. Le principe de base est d'exécuter une instruction lorsque toutes ses opérandes sont disponibles ; l'exécution se base donc sur les dépendances, il n'y a plus de pointeur d'instruction courante. Le processeur doit donc avoir une vue globale du programme. L'exécution exploite alors naturellement tout le parallélisme disponible : plusieurs instructions peuvent être exécutées simultanément puisqu'il n'y a pas d'état réellement partagé, seulement des opérandes qui circulent de la sortie d'une instruction à l'entrée d'une ou plusieurs autres.

Il est donc nécessaire d'être capable de gérer dynamiquement la création, l'adressage et la mort de ces opérandes. La solution généralement retenue est la « CAM », pour « content adressable memory ». Le principe est que chaque donnée a un tag ; chaque instruction a une liste de tags nécessaires afin de résoudre ses dépendances. La CAM se comporte donc comme une table de hachage, capable d'adresser la mémoire par tag.

Les architectures dataflow se heurtent cependant à leur généralité. Il est intéressant de profiter du parallélisme offert par le dataflow ; mais dans le cadre d'une machine massivement parallèle, il devient difficile de gérer efficacement l'adressage global des tags [24]. Autrement dit, un programme réel contient suffisamment d'instructions et de dépendances pour rendre la gestion de celui-ci problématique.

WaveScalar [73] propose une architecture à base dataflow mais capable d'exécuter du code provenant de langages impératifs, en fournissant une mémoire ordonnée. L'architecture est organisée en un « WaveCache », un substrat mélangeant éléments de calcul et cache de données. Cela permet d'avoir un système passant à l'échelle, puisqu'il suffit d'étendre le substrat en ajoutant des éléments de calcul et de mémoire pour profiter de l'espace disponible. Le principe de fonctionnement de WaveScalar est de découper la représentation Dataflow du programme en vagues (« waves »), correspondant plus ou moins à la notion de macro-bloc en architecture von-Neumann. Ces vagues contiennent un ensemble d'instructions ne bouclant pas et pouvant donc être exécutées chacune au maximum une fois. Un identifiant est assigné à chaque vague

à exécuter, fournissant un moyen d'ordonner des vagues successives. De plus, à l'intérieur de chaque vague, chaque accès mémoire est annoté afin de pouvoir garder une trace de l'ordre requis. Les identifiants de vague et les annotations d'accès permettent de fournir un modèle mémoire ordonné.

Il est à noter que l'exécution dans le désordre sur des processeurs superscalaires est souvent vue également comme une forme limitée de dataflow. En effet, l'exécution dans le désordre maintient une fenêtre d'instructions dans laquelle les dépendances sont calculées afin de pouvoir exécuter les instructions dès que possible. La différence est donc que le choix du nombre d'instructions possible est très faible (de l'ordre de la taille de la fenêtre d'instructions), alors que dans le cas d'un dataflow générique, n'importe quelle instruction du programme peut être choisie à partir du moment où les dépendances sont résolues.

2.1.4 Aides matérielles au multi-tâche et parallélisme

Les processeurs modernes proposent également des systèmes afin d'aider le système d'exploitation à gérer le parallélisme. Les quelques mécanismes présentés ici sont les mécanismes les plus classiques, mais de nombreux autres existent qui visent souvent des types de parallélisme non génériques. Par exemple, les transferts DMA (pour « Direct Memory Access ») fournissent une forme de parallélisme permettant à un périphérique d'accéder directement à la mémoire, sans interrompre le processeur principal.

Multi-tâche préemptif

La plupart des processeurs modernes offrent des primitives afin d'aider le système d'exploitation à gérer un parallélisme préemptif. Il ne s'agit pas dans ce cas de parallélisme pur dans le sens où plusieurs fils d'exécution s'exécutent simultanément mais de parallélisme type « time-sharing », où un quota de temps d'exécution est alloué à chaque thread, donnant à l'utilisateur l'illusion d'un parallélisme.

Le principe de base est généralement d'avoir deux niveaux d'exécution [1]. Un des niveaux est appelé habituellement mode utilisateur : c'est là que s'exécutent les applications normales, sans privilège particulier. Un deuxième mode, le mode noyau, est réservé au système d'exploitation ; depuis ce mode, le système est capable de gérer l'exécution des programme en mode utilisateur et notamment les interrompre arbitrairement au bout d'un certain temps. Ainsi, grâce à ces fonctionnalités, le système est capable de gérer du multi-tâche préemptif, au lieu de devoir attendre que chaque application rende la main.

Mémoire virtuelle

Pour permettre à plusieurs processus de s'exécuter simultanément sans risque d'interférence, de nombreuses architectures mettent également à disposition la notion de mémoire virtuelle. Cela permet au mode noyau de gérer arbitrairement la manière dont le mode utilisateur voit la mémoire. Ainsi, il est exemple possible de partager certaines zones mémoires entre plusieurs processus et, à l'inverse, de faire que d'autres zones soient spécifiques au processus.

De plus, cela permet de dupliquer facilement un processus à un instant donné, sans pour autant recopier la mémoire. Lorsqu'une duplication est demandée, le système se contente de mettre en place une correspondance mémoire virtuelle/mémoire physique identique pour les

[1] Au minimum. L'architecture IA32 offre par exemple 4 niveaux, appelés « rings ». Cependant, beaucoup de systèmes portables, comme Linux, n'en utilisent que deux.

deux processus. Dès que le nouveau processus écrit sur une page, celle-ci est recopiée (« copy-on-write ») afin d'éviter d'interférer avec le premier processus. Cela permet d'avoir une gestion du parallélisme légère et réactive, et qui de plus simplifie la gestion des zones en lecture seule [2] qui sont automatiquement partagées sans risque.

Aides pour les locks

Certains processeurs proposent des primitives dédiées à la gestion des locks sur multi-processeurs. Ainsi, sur PowerPC, deux instructions, *lwarx* « Load and Reserve » et *stwarx* « Store Exclusive » [72] permettent de gérer les accès concurrent sur une zone mémoire. Typiquement, elles servent à implémenter des locks, qui seront à leur tour utilisés pour gérer l'accès concurrent aux structures.

L'instruction de « load », *lwarx*, charge une valeur depuis l'adresse mémoire spécifiée ; un flag de réservation est alors activé, en gardant une trace de l'adresse physique réservée. Lorsque ce flag est activé, le processeur met en place du «snooping», afin de détecter toute tentative d'écriture sur la zone mémoire en question. Si cela arrive, alors le flag est remis à zéro. Le fonctionnement du « store » *stwarx* est d'effectuer l'opération uniquement si le flag de réservation est toujours activé ; dans le cas contraire, l'opération échoue.

Ainsi, avec ces deux instructions, il est possible d'accéder de manière atomique à une zone mémoire en détectant tout conflit d'écriture et réessayant l'opération le cas échéant. Contrairement à un système simple de lock, il s'agit d'une approche optimiste, espérant que dans la majorité des cas il n'y a pas de conflit. Cela permet de réduire le coût puisqu'il n'est pas nécessaire de systématiquement faire une synchronisation des différents coeurs, au contraire du lock où cela est nécessaire, même si aucun conflit ne se produit.

2.2 La programmation parallèle

La thématique de la programmation parallèle génère presque autant de solutions que de programmeurs. Ainsi, dans [52], plus de 200 frameworks de programmation parallèle sont recensés au milieu des années 90. Bien sûr, même si chaque framework est différent, les idées sous-jacentes sont souvent similaires, les différents frameworks ne faisant qu'en intégrer certaines au gré des contraintes visées.

Il n'est pas visé ici l'exhaustivité mais plutôt un panorama des principales approches existantes.

Ces frameworks essayent de résoudre deux problématiques, souvent de manière entremêlée :

- Donner des informations sur la sémantique parallèle ;
- Donner des informations sur le comportement à l'exécution.

Prenons l'exemple d'une boucle simple :

```
1 Pour i de 1 à n:
2     Faire(etat_global,i)
```

La connaissance de la sémantique de ce code nous permettrait de déterminer si chaque itération est indépendante ou si, à cause de l'état global, chaque itération dépend de la précédente. Si on suppose alors que chaque itération est indépendante, on peut choisir de l'exécuter via des

[2] Ces zones peuvent être nombreuses puisque le code des bibliothèques partagées s'y trouvera typiquement. Ainsi, une seule instance des bibliothèques existera dans la mémoire physique, même si plusieurs processus les utilisent.

instructions de parallélisme vectoriel s'il s'agit de calculs simples ou alors distribuer les itérations sur un cluster ; ce choix relève non plus de la sémantique mais du choix d'exécution.

De nombreux frameworks mélangent les deux aspects, liant les informations sur la sémantique parallèle aux informations nécessaires à l'exécution, notamment pour des raisons de performance.

Parmi les informations liées à la sémantique du parallélisme, on peut distinguer deux grandes catégories :

- la concurrence, pour indiquer quelles tâches doivent être lancées simultanéments ;
- les synchronisations et communications, pour gérer deux tâches exécutées en parallèle.

Les synchronisations limitant la concurrence, la distinction n'est pas nécessairement toujours claire.

Cette section est organisée en fonction de ces deux catégories. La première partie donne aperçu des moyens de décrire du parallélisme. La deuxième porte sur les moyens de communication en mémoire distribuée puis en mémoire partagée. Enfin, quelques approches plus haut niveau sont décrites, basées sur des paradigmes impliquant un parallélisme implicite.

2.2.1 Exprimer le parallélisme

Parallélisme vectoriel

Le parallélisme vectoriel offre un parallélisme très fin grain. Il s'agit d'appliquer une même opération simultanément sur plusieurs valeurs ; cet ensemble de valeurs est habituellement appelé vecteur. Cela est particulièrement utile dans le cas d'algorithmes réguliers, devant traiter un grand nombre de données. Typiquement, le traitement d'images (comme les calculs de distances en estimation de mouvement) permet l'utilisation de parallélisme vectoriel.

L'écriture explicite de code tirant parti de parallélisme vectoriel n'a guère de difficultés particulières, puisqu'il suffit d'utiliser les types ou opérations fournies par le framework, comme avec les *intrinsics* de gcc [69], qui vont permettre de manipuler les vecteurs comme des variables normales.

L'obtention automatisée de code vectoriel est parfois possible dans des cas simples, mais se heurte à des problèmes de détection des cas intéressants [55].

Parallélisme de pipeline

Le parallélisme de pipeline se base sur le fait qu'une même série de traitements doit être appliquée à des données arrivant à la chaîne. Le principe est identique à la notion de pipeline pour l'architecture matérielle (voir section 2.1.1).Le code est découpé en plusieurs tâches, chacune des ces tâches prenant en entrée la sortie de la précédente. Les données arrivent en continu par la première tâche et le résultat du traitement total est obtenu en sortie de la dernière tâche. Chaque tâche peut être exécutée en parallèle : pendant que la tâche A traite la donnée n, la tâche B à sa suite traite la donnée n-1, C traite n-2 et ainsi de suite. Ce type de parallélisme peut notamment être obtenu via les langages de gestion de flux (« stream ») tel que StreamIt [75].

Cependant, les types de flux permettant un parallélisme simple sont limités par la nécessité d'éviter des boucles dans le système. Ainsi, bien que les chaînes de transformations de vidéos se prêtent généralement bien au parallélisme de flux, un algorithme tel que H.264 [2] limite le parallélisme lié au flux à cause de la présence d'une boucle de rétroaction.

Threads

L'approche Unix du parallélisme a une dichotomie similaire à la notion de mémoire partagée/mémoire distribuée. D'un côté, il y a les processus, capables d'exécuter un flux d'instructions simple. Ils ont chacun leur mémoire et doivent mettre en place explicitement des canaux de communication entre eux, en utilisant par exemple des tubes nommés ou des sockets. D'un autre côté, il y a également la notion de *thread* : plusieurs fils d'exécution se partageant un même espace mémoire.

On notera que comme dans le cas de mémoire partagée/mémoire distribuée, la distinction n'est pas exacte. Il est en effet possible de partager des zones mémoires entre plusieurs processus avec par exemple les primitives POSIX *shmem* [71] ; et à l'inverse, il existe un mécanisme appelé « Thread Local Storage » qui permet à des threads d'avoir des zones d'adressage privées [29].

En pratique, deux types de threads sont souvent différenciés : les threads en mode noyau et ceux en mode utilisateur.

Les premiers sont gérés par le système d'exploitation et sont préemptifs : un thread ne peut pas bloquer l'exécution d'autres threads puisque le noyau peut à tout moment et sans coopération du code utilisateur choisir d'exécuter un autre thread. Le noyau peut répartir les threads entre différentes unités physiques. Il peut également choisir d'ordonnancer plusieurs threads sur une même unité d'exécution ; le thread n'a pas de moyen de savoir quand est-ce qu'il sera désordonnancé.

Les threads en mode utilisateur, parfois appelés *fibers*, fonctionnent sans coopération du noyau. Puisqu'ils sont vus comme un seul fil d'exécution par le noyau, ils ne peuvent pas fournir réellement du parallélisme puisqu'il ne leur est pas attribué plusieurs unités d'exécution simultanément. Le changement de thread ordonnancé est ordonné par le thread lui-même, soit explicitement (en demandant qu'un autre thread soit exécuté à sa place), soit lorsque l'utilisateur utilise une des primitives de la bibliothèque de threads qui profite de son appel pour ordonnancer un autre thread. Un thread buggué risque donc de garder la main et de bloquer l'exécution de tous les autres threads. Cependant, puisque le changement de thread ne nécessite pas de passage par le noyau, l'ordonnancement est plus léger que dans le cas de threads noyau. Par exemple, la bibliothèque *protothread* [30] ne nécessite que l'équivalent de quelques sauts pour un changement de contexte et ajoute seulement un surcoût mémoire de 2 octets par thread existant.

Une notation est souvent utilisée pour différencier les cas de threading :

- $M : 1$: Les M threads utilisateur sont exécutés sur un seul thread noyau, comme par exemple dans la bibliothèque GNU Pth [32] ;
- $1 : 1$: Chaque thread utilisateur est exécuté sur un thread noyau, comme par exemple dans la bibliothèque pthread [56] ;
- $M : N$: Les M threads utilisateurs utilisateurs sont exécutés sur un ensemble de N threads noyau, comme par exemple les KSE de FreeBSD [33].

Bien que le dernier cas puisse sembler être le plus intéressant par sa généralité, il est loin d'être systématiquement mis en oeuvre. Le mélange de threads en mode utilisateur et en mode noyau impose d'avoir deux niveaux d'ordonnancement, ce qui peut être pénalisant pour certains types d'applications sensibles à la latence. De plus, bien que les threads puissent sembler similaires, leur sémantique diffère légèrement du fait que le mode utilisateur doit commuter explicitement, contrairement au mode noyau qui est capable de commuter quand il le désire.

La bibliothèque pthread [56] permet sous Linux/Unix de manipuler de manière portable des threads noyau. Il s'agit d'une bibliothèque bas niveau, fournissant des primitives de création / arrêt de threads mais également de synchronisation tels que des mutex ou des conditions. C'est au programme de s'occuper entièrement de la validité de son code : il est nécessaire d'expliciter

chaque création de thread, de mettre en place des mutex sur chaque variable partagée et de les obtenir à chaque accès.

Fonctions asynchrones

La notion de *fonctions asynchrones*, renvoyant des*futures*, offre une approche pour manipuler des threads simples dans un programme. Le principe est d'appeler une fonction mais de ne pas attendre la fin de son exécution. Le code appelant continue directement son exécution pendant que la fonction appelée est exécutée dans un thread concurrent ; ce principe est notamment utilisé dans [19].

Le code appelant, dès que la fonction est lancée, reçoit en valeur de retour une variable dite *future* : elle contient le résultat potentiel de la fonction. Si le code appelant fait référence au contenu de cette variable, alors il bloque en attendant que la fonction ait réellement fini son travail.

Cette gestion automatisée de la création de thread et de la synchronisation facilite l'utilisation de threads pour des code irréguliers, puisqu'il n'est pas nécessaire de mettre en place des structures de contrôle complexes, difficiles à généraliser.

Cependant, le parallélisme obtenu est limité, puisque pour être bénéfique, les fonctions doivent pouvoir s'exécuter longtemps (à comparer au surcoût de la création du contexte concurrent) avant de nécessiter une synchronisation. De plus, bien que l'expression de la création et terminaison d'un thread soient simplifiées, les problèmes de concurrence d'accès ne sont pas résolus pour autant : si la fonction exécutée en parallèle accède à des structures partagées avec l'appelante, des mécanismes de protection classiques (comme des locks) doivent être mis en place.

Cilk [56] ajoute des mot-clés au langage C permettant l'utilisation de fonctions asynchrones :

- Le mot clé *cilk* doit être ajouté sur les fonctions faisant usage de fonctions asynchrones.
- Le mot clé *spawn* permet d'exécuter une fonction en parallèle, l'exécution de l'appelant continuant en parallèle.
- Le mot clé *sync* est une barrière locale : l'exécution est bloquée jusqu'à ce que les fonctions appelées avec *spawn* dans le scope local soient terminées. Cette barrière est implicite avant chaque *return*.

De plus, pour gérer l'accès concurrent aux variables, Cilk fournit des locks ; la documentation conseille cependant de les éviter en profitant des propriétés du modèle.

Afin de faciliter l'usage des fonctions asynchrones, Cilk fournit également en complément ce qu'on appele des *inlets*. Il s'agit d'un mécanisme permettant d'exécuter un morceau de code sur le résultat d'une fonction asynchrone, comme par exemple pour effectuer une réduction. Elles garantissent leur atomicité entre elles, ce qui évite de devoir recourir à des locks.

Découpage en tâches élémentaires

Une approche communément utilisée par l'implémentation de frameworks parallèles est le découpage du travail à effectuer en tâches élémentaires. Chacune de ces tâches peut être ensuite exécutée librement sur un des coeurs disponibles. Cela permet d'avoir un équilibrage de charge naturel. En effet, même si l'assignation des tâches à exécuter sur les fils d'exécution disponibles peut se faire dès leur création, il est généralement possible d'avoir un mécanisme de « work stealing ». En pratique, cela signifie que, lorsqu'un coeur a fini d'exécuter les tâches qui lui étaient imparties, il peut accaparer les tâches d'autres coeurs.

Ce mécanisme de découpage en tâches est rarement offert tel quel, mais plutôt utilisé pour implémenter un paradigme de plus haut niveau. Par exemple, le modèle d'exécution de Cilk est basé sur cette notion de tâche et de « work stealing » : lorsqu'un processeur Cilk n'a plus

rien à faire, il est capable de « voler » une tâche à un autre processeur choisi aléatoirement. De plus, Cilk optimise le cas où une tâche est exécutée localement sur le même processeur que son créateur via l'utilisation d'une version du code adaptée pour réduire le surcoût. Cependant, il reste nécessaire avec Cilk de créer explicitement et statiquement chacune des tâches, ce qui induit un surcoût dans les zones rendues séquentielles par manque de ressources.

La bibliothèque « Threading Building Blocks » (TBB) d'Intel [6] fournit un ensemble de templates C++ permettant d'exploiter efficacement le parallélisme offert par les puces multi-coeurs. L'approche choisie est basée, comme Cilk, sur du découpage en tâches élémentaires. TBB fournit un ensemble de primitives bas-niveau permettant de créer aisément des tâches à partir d'opérations courantes. Par exemple, la fonction template *parallel_do* permet de créer une boucle aux itérations parallèles, dont l'ordonnancement effectif est laissé à charge de la bibliothèque TBB. Des structures de données communes supportant des opérations parallèles sont fournies, comme des files d'attente, des vecteurs ou des tables de hachage. Des algorithmes sont également mis à disposition, générant eux-mêmes des tâches, tels que *parallel_sort* pour du tri ou *pipeline* pour implémenter aisément un parallélisme type pipeline. Des fonctions classiques de gestion du parallélisme, type mutex ou opérations atomiques sont incluses dans la bibliothèque, ce qui permet donc de l'utiliser comme une couche d'abstraction matérielle complète. Cela correspond à l'objectif de TBB qui est de présenter une abstraction relativement bas niveau, se calquant sur la machine afin d'offrir au programmeur les outils lui permettant de tirer un maximum de performance d'un processeur multi-coeurs.

Cependant, aussi bien dans Cilk que dans TBB, le problème de la granularité de découpage du code reste à la charge du programmeur. En effet, il est nécessaire de pré-découper le travail à effectuer afin que l'ordonnanceur puisse adapter la charge efficacement entre les processeurs. S'il n'y a pas assez de tâches disponibles, des ressources de calcul sont gaspillées ; à l'inverse, si trop de tâches sont créées, le surcoût de création pose problème alors qu'elles sont de toute façon séquentialisées. Le découpage nécessaire afin d'optimiser au mieux l'exécution dépend de l'algorithme ainsi que de l'architecture, imposant donc un choix éventuellement difficile pour le programmeur. Dans TBB, il est par exemple demandé au programmeur de choisir la « grain size », taille de grain, notamment pour les boucles parallèles. Cela va déterminer le nombre d'itérations de la boucle à considérer comme une unique tâche. Dans les versions récentes de la bibliothèque, il est également possible d'utiliser une fonction de découpage automatique, à la place de la taille de grain statique, qui essaye d'éviter le surcoût du découpage en tâches via l'utilisation d'heuristiques [3].

2.2.2 Communications en mémoire distribuée

La notion de mémoire distribuée peut ici aussi bien viser des systèmes qui physiquement ont de la mémoire distribuée entre les noeuds de calculs, tel un cluster, que des modèles de programmation où l'on travaille avec des processus indépendants.

Le passage de message

Le passage de message permet d'écrire des codes parallèles fonctionnant sur des systèmes à mémoire distribuée, comme par exemple des ensembles de machines. Le principe est de séparer complètement chaque flot d'exécution séquentiel. Chacun de ces flots doivent expliciter les communications entre eux à l'aide de messages pour partager des informations et se synchroniser. La complexité induite par le parallélisme est entièrement prise en charge par l'utilisateur. Cependant, puisqu'il n'y pas de zone mémoire partagée, il n'est pas nécessaire d'associer des mécanismes de réservation de ressources tels que des mutex sur les structures de données. Les

interactions avec d'autres fils d'exécution sont encadrés à des points précis et ne peuvent intervenir aléatoirement.

Il existe deux optimisations classiques pour le passage de message :

- laisser le programme continuer à s'exécuter une fois que l'envoi du message a été initié ;
- éviter la recopie des messages en mémoire lors de l'exécution sur un système à mémoire partagée, en donnant directement le « pointeur » vers les données au processus destinataire du message.

Si on laisse le processus expéditeur modifier des éléments contenus dans le message alors que le message n'est pas fini d'envoyer ou que le processus de destination le modifie dans le cas de la mémoire partagée, les structures risquent d'être corrompues du point de vue de l'expéditeur. Il devient donc nécessaire d'avoir des mécanismes de protection, par exemple par analyse statique de code ou par des flags dynamiques. Cela devient en partie similaire aux problématiques de locking pour de la mémoire partagée. Cependant, les zones concernées par ces types de problèmes peuvent être identifiées explicitement, ce qui n'est pas le cas pour le parallélisme à mémoire complètement partagée.

MPI [58], pour "Message Passing Interface", définit une API standard pour le passage de messages. De nombreuses implémentations existent, telles que MPich [85], lam-MPI [16], OpenMPI [35]. MPI définit notamment des variantes bloquantes et non bloquantes de chaque fonction de communication ; elle fournit également des fonctions de communication de groupe. Tout doit être fait explicitement, laissant le soin au programmeur de gérer chaque synchronisation et partage de données.

Le modèle acteur

Côté langage, la notion de passage de message existe sous la notion de *CSP* (pour «Communicating Sequential Processes») [43] ou d'*Actor Model* [42].

Dans ces deux approches, le programme est découpé sous forme d'acteurs / processus. Chacun de ces acteurs est un programme séquentiel indépendant. Selon les implémentations, il est possible de concevoir un acteur sous la forme de plusieurs sous-acteurs, rendant l'acteur englobant parallèle, mais fournissant toujours un modèle séquentiel vu de l'extérieur. Les acteurs sont capables de réagir à des messages, d'envoyer des messages ainsi que de créer d'autres acteurs. Les communications se font à travers des canaux de communication explicitement mis en place : un acteur ne peut communiquer qu'avec un acteur dont il connaît l'adresse.

Notons que dans la version originale de Hoare [43] des CSP, la topologie des processus était déterminée statiquement au contraire des acteurs qui peuvent être créés à l'exécution. De plus, une notion de buffer de messages existe sur les liens de communication, ce qui n'est plus le cas dans les évolutions récentes [14] telles que Occam [81].

De nombreuses algèbres de processus modélisent ces concepts ; cependant, le travail de cette thèse ne visant pas les approches et modélisations théoriques, cet aspect ne sera pas développé. On peut néanmoins mentionner le PI-Calcul [64], qui est un formalisme mathématique minimal au même titre que le lambda calcul [12]. Il la particularité de fusionner au sein d'un même concept les notions de variable et de canal de communication. JoCaml [21] implémente une variante du PI-calcul.

Occam [81] implémente le principe des CSP. Ce langage a été à l'origine conçu pour le Transputer [83] afin d'exprimer un maximum de parallélisme. Il s'agit d'un langage impératif, doté d'opérateurs spécifiques pour gérer l'envoi et la réception de données sur les canaux. On notera également l'existence de deux types de blocs de code : « SEQ », exécutant chacune des instructions du bloc séquentiellement et « PAR » permettant d'exécuter chaque instruction en

parallèle.

Toujours dans Occam, un troisième type de bloc existe, « ALT », permettant d'exécuter une seule alternative de code parmi plusieurs. Bien que n'offrant pas en soit du parallélisme, ce mécanisme est classique dans les systèmes concurrents. Cela permet à un acteur de réagir à plusieurs types de messages, sans connaître a priori le type du message suivant. Par exemple, si un acteur représente une file d'attente, il devra réagir à des messages de type « enqueue » et « dequeue ». Les alternatives peuvent être associées à une condition, appelée « garde », qui doit être vérifiée afin que l'alternative puisse être choisie.

Un type de mécanisme similaire existe avec la fonction UNIX « select », qui permet d'attendre l'activité sur un descripteur de fichier quelconque. Cette fonction est souvent utilisée pour écrire un code sans threads mais traitant néanmoins plusieurs requêtes simultanément, comme un serveur web.

Ada [63] implémente également ce mécanisme. Les types de messages acceptés sont déclarés explicitement à l'aide du mot clé « entry » ; la clause « accept » permet de spécifier que faire en cas de réception d'un message du type précisé et peut être rendue conditionnelle à l'aide de gardes également.

Erlang [10] implémente le modèle Acteur, en se basant sur un langage fonctionnel pour les zones séquentielles de code. Chaque processus a sa propre file d'attente de messages ; la sélection d'un message à recevoir se fait via du matching dans une clause dédiée. Par exemple, dans le cas de la figure 2.1, un message est envoyé à un processus nouvellement créé, puis l'envoyeur se met lui même en attente d'une réponse. S'il reçoit alors un objet de type quelconque, il exécute un traitement donné ; cependant, si le message reçu est de type texte, il effectue un traitement adapté, en l'occurrence en afficher le contenu.

Listing 2.1 – Syntaxe Erlang

```
1 Pid = spawn(Mod, Func, Args)    %Cree un nouveau processus
2
3 Pid ! message      % Envoie un "message" au processus
4
5 receive       % Attend un message envoye à ce processus
6 message -> do_something;
7 {hello, Text} -> io:format("Got␣hello␣message:␣~s", [Text]);
8 end.
```

2.2.3 Synchronisations en mémoire partagée

Les threads partageant le même espace mémoire, il est nécessaire d'avoir des mécanismes de synchronisation. On peut en distinguer deux grands types :

- ceux permettant d'éviter un accès concurrent à des zones mémoires et ainsi garantir l'atomicité apparente du code les modifiant ;
- ceux permettant d'endormir et de réveiller les threads, afin d'ordonnancer ces derniers.

Réservations de ressources

Les entiers atomiques sont les versions les plus simples de variables garantissant la cohérence d'une zone mémoire partagée entre plusieurs threads Des opérations sont fournies pour manipuler ces variables sur le principe du « test and set » : il est possible de tester la valeur et éventuellement

de la modifier tout en garantissant qu'aucun autre thread ne modifiera la variable entre temps. Selon les implémentations, les opérations disponibles varient ; on peut mentionner notamment le « dec and test », qui décrémente la variable et indique si le résultat est nul de manière atomique. GCC fournit ces opérations nativement [69], comme par exemple *__sync_fetch_and_add*.

Les *mutexes* (pour "MUTual EXclusion"), souvent appelés *locks*, sont associés à une zone mémoire à protéger des accès concurrents. Pour qu'un thread puisse modifier la dite zone, il doit explicitement obtenir le lock, qu'il relâchera lorsqu'il aura fini ses modifications. Si le lock est déjà pris par un autre thread, le thread doit attendre jusqu'à ce qu'il soit libéré.

Les *spinlocks* désignent des locks dont l'attente, lorsqu'un lock est demandé mais déjà pris, est dite active. Une attente normale demande au système de threading de ne plus ordonnancer le thread demandeur, afin de libérer l'unité d'exécution. Dans le cas de l'attente active, le thread continue à être exécuté mais se contente de vérifier en permanence une propriété donnée, indiquant si le lock est disponible. Le spinlock permet généralement d'être plus réactif puisqu'il n'y a pas d'effets lié à l'ordonnanceur du système, mais au prix d'une occupation processeur plus importante.

Les sémaphores [28] fonctionnent de manière similaire aux mutex mais permettent de protéger un nombre quelconque de ressources identiques. Ils sont implémentés sous forme d'une variable entière, initialisée au nombre de ressources disponibles. Une opération $P()$ demande une ressource ; si l'entier est strictement supérieur à zéro, alors la ressource est réservée et l'entier décrémenté d'un ; dans le cas contraire, l'opération bloque jusqu'à ce qu'une ressource soit disponible. L'opération $V()$ permet de rendre une ressource et incrémente donc de manière non-bloquante l'entier.

Les locks en lecture/écriture [23] (« read/write locks » ou « rwlocks ») permettent à plusieurs threads accédant en lecture à la zone partagée de s'exécuter simultanément. A l'inverse, un seul thread en écriture est autorisé. Un risque de famine existe : lorsque plusieurs threads ont accès en lecture à la zone mémoire, qu'un thread en écriture est en attente, et qu'un autre en lecture arrive, il y a deux principaux choix :

- l'autoriser à accéder en lecture : risque de famine pour le thread en écriture qui risque de ne jamais être ordonnancé ;
- ordonnancer dès que possible le thread en écriture : risque de famine pour le thread en lecture si des threads en écriture arrivent en permanence.

L'approche *RCU* [53] pour «Read Copy Update» est une alternative aux locks en lecture/écriture et permet d'éviter les problèmes de famine. Le principe sous-jacent est, lors d'une modification d'un élément devant être protégé des accès concurrents, de faire une copie de celui-ci sur lequel le thread en écriture travaille. Lorsque la mise à jour est terminée, les pointeurs vers l'élément en question sont eux-même mis à jour. Une fois que tous les threads qui accédaient en lecture à l'ancienne version de l'élément sont terminés, l'ancien élément peut être libéré ; il n'y a pas de risque de famine, puisque les nouveaux threads accèdent directement au nouvel élément à partir du moment où la mise à jour est terminée.

Par exemple, le noyau Linux fournit l'API suivante pour manipuler les RCU :

- *rcu_read_lock*/*rcu_read_unlock* pour délimiter les accès en lecture ;
- *rcu_assign_pointer* qui permet de mettre à jour l'ancien élément vers le nouveau de manière atomique ;
- *call_rcu* qui permet d'indiquer une fonction de callback à appeler lorsqu'un élément n'est plus utilisé ; typiquement cela permet d'appeler une fonction de désallocation de l'ancien élément.

Méthodes d'attente

Les barrières permettent de faire en sorte que tous les threads d'un ensemble donné aient fini leur travail avant de continuer. Une barrière est généralement initialisée par le nombre de threads à attendre. Ensuite, chaque thread, à la fin de son propre travail, signale à la barrière qu'il a terminé : s'il n'est pas le dernier, il est endormi ; sinon, tous les autres threads sont réveillés et lui même continue normalement son exécution.

La barrière désigne aussi un mécanisme permettant de contraindre la séquentialité mémoire dans un morceau de code. Ainsi, la présence d'un appel à une telle barrière garantit que toutes les modifications mémoire précédent l'appel (précédence causale) sont effectuées, évitant ainsi que le compilateur ordonnance des accès plus tard sous couvert d'optimisation.

Les *conditions* permettent d'endormir un ou plusieurs threads en l'attente d'un événement généré par un autre thread de l'application. Plusieurs primitives sont disponibles pour les manipuler :

- *wait* : attendre sur la condition en attendant d'être réveillé ;
- *broadcast/notifyall* : réveiller *tous* les threads en attente sur la condition ;
- *signal/notify* : réveiller un seul thread en attente, les autres restant endormis.

Les listings 2.2 et 2.3 donnent des exemples de code d'attente et de signalement en utilisant la bibliothèque pthread. Le principe du code de l'exemple est d'endormir un thread et de le réveiller quand la condition *needwakeup* devient vraie [3].

Listing 2.2 – "Les conditions : exemple de boucle d'attente"

```
1 pthread_mutex_lock(&mutex);
2 while (!needwakeup)
3     pthread_cond_wait(&cond, &mutex);
4 pthread_mutex_unlock(&mutex);
```

Listing 2.3 – "Les conditions : exemple de code de réveil"

```
1 pthread_mutex_lock(&mutex);
2 needwakeup = true;
3 pthread_cond_broadcast(&cond);
4 pthread_mutex_unlock(&mutex);
```

On remarque dans ces exemples la présence d'un mutex lors de l'appel au broadcast en ligne 3 du listing 2.2. L'utilisation de ce mutex est obligatoire afin d'éviter des race-conditions. L'appel à la fonction d'attente met en attente l'appelant *et* relâcher le mutex de manière atomique. Cela permet donc d'éviter de rater un signal de réveil. Par exemple, en l'absence de cette atomicité et des différentes prise du locks dans les listings 2.2 et 2.3, le scénario suivant pourrait se produire :

- Exécution de la ligne 2 dans le listing 2.2.
- Exécution des lignes 2 et 3 dans le listing 2.3. Le thread n'étant pas encore en attente, le signal de réveil envoyé sera perdu.
- Exécution de la ligne 3 dans le listing 2.2. Le signal de réveil ayant déjà été envoyé, le thread se met en attente et n'est jamais réveillé, il y a un deadlock.

L'existence de ce lock lorsque plusieurs threads sont attente permet également d'avoir un ordonnancement sur l'accès à une ressources partagée entre ces threads via la prise de lock au moment du réveil.

[3] Ce morceau de code est adapté directement du code d'attente des threads non utilisés de Capsule.

Objets protégés

La notion d'événement lié au passage de message peut être transposé dans l'approche objet par la notion de « protected object », objet protégé (appelé également *moniteur*). Les appels de méthode sur une instance sont séquentialisés, une seule méthode peut être active à un moment donné. Pour cela, un lock est implicitement utilisé par le framework fournissant les objets protégés. Chaque appel de méthode essaie d'acquérir le lock ; si le lock est déjà pris, il attend sur la file d'attente associée. Ainsi, on garantit l'atomicité des méthodes vis à vis de l'état représenté par l'instance.

Dans ce cas, chaque méthode fournie par l'objet est en fait équivalente à un type de message qu'on peut lui envoyer. La sémantique diffère cependant un peu puisque les appels de méthodes sont synchrones alors que les envois de message n'attendent pas, de manière générale, de réponse.

Le principe des objets protégés étant souvent vu comme intuitif, de nombreux langages objet offrent ce type de protection sur les objets.

En Java, chaque objet a un lock associé. Le mot clé « synchronized », pouvant être associé à une méthode ou à un bloc de code, indique que pour pouvoir s'exécuter, il est nécessaire d'acquérir le lock. Le comportement est donc similaire aux objets protégés, mais cela doit être déclaré explicitement.

En Ada, il est possible de déclarer les types comme « protected ». Pour accéder aux données d'un type déclaré ainsi, il devient obligatoire de passer par les méthodes associées. Le langage garantit alors explicitement l'exclusion mutuelle, sans devoir nécessiter d'annotations spécifiques à chaque méthode comme en Java. De plus, l'exclusion mutuelle est de type lecture/écriture : plusieurs appels peuvent être effectués simultanément s'ils travaillent exclusivement en lecture ; les appels en écriture sont exclusifs.

Mémoire transactionnelle

La mémoire transactionnelle présente une alternative aux locks et ce de manière aussi bien matérielle [41] que logicielle [66], voire en utilisant un mélange des deux [26].

Dans un tel système, l'accès aux zones mémoire partagées se fait à l'intérieur de sections *atomiques* : vu de l'utilisateur, le code au sein d'une section est exécuté d'une traite, sans que d'autres threads puissent voir un état intermédiaire incohérent.

Pour cela, au sein d'une section atomique, le système garde trace de l'accès à chaque variable partagée. Pour les accès en lecture, il s'agit de conserver le numéro de version des variables lues ; pour les accès en écriture, il est nécessaire de garder trace de l'ancienne valeur. Si le système détecte un conflit d'accès par rapport à un autre thread, la transaction échoue ; grâce au journal, il est possible de revenir dans l'état initial.

L'intérêt par rapport à un système de lock est qu'il s'agit d'une approche spéculative optimiste : en partant de l'hypothèse que dans la majorité des cas il n'y a pas de conflit d'accès, la mémoire transactionnelle évite le surcoût lié à la prise de lock, qui nécessite une synchronisation. De plus, puisqu'il n'y a pas d'attente d'obtention d'une ressource, le risque de deadlock disparaît. Enfin, il est difficile avec les locks d'avoir une granularité très fine ; par exemple, un seul lock sera associé à un élément de structure de donnée. Si deux threads travaillent sur des variables différentes de ce même élément, le lock les empêchera de travailler simultanément ; dans le cas de la mémoire transactionnelle ; chaque thread peut travailler et, s'ils s'avèrent réellement indépendants, alors aucun blocage artificiel n'est introduit.

Côté programmeur, la mémoire transactionnelle permet de transformer le problème de la gestion du partage d'état d'un problème global en un problème local. Les locks souffrent en effet de deux problèmes rendant leur manipulation complexe. D'une part, plusieurs sections de code

doivent partager un même lock ; ainsi, il est nécessaire d'avoir une compréhension globale du code afin de manipuler les locks correctement. D'autre part, il est difficile de réutiliser et notamment d'encapsuler des constructions dont la concurrence est gérée par des locks. Notamment, cela risque d'entraîner des deadlocks, et il est nécessaire d'avoir une compréhension complète des morceaux à faire coopérer pour pouvoir résoudre cela, ce qui invalide le principe de réutilisation.

A l'inverse, la mémoire transactionnelle peut être en grande partie gérée localement. L'interaction d'une zone transactionnelle avec le reste du système sera prise en charge automatiquement et non pas à la charge du programmeur. De son côté, le programmeur devra indiquer les zones de code atomiques, ce qui est une notion intrinsèquement locale et généralement intuitive pour le programmeur.

Cependant, le journal de transactions, permettant d'annuler une transaction partielle, est complexe à manipuler. Comme il doit garder des traces des opérations des zones transactionnelles, il est nécessaire de limiter la taille de ces dernières afin de pouvoir manipuler le journal sans grever les performances. Cela a un impact sur la réutilisation et sur la composition : en effet, si la taille d'une zone transactionnelle est limitée, cela veut dire qu'une composition de plusieurs sous-systèmes utilisant de la mémoire transactionnelle risque de ne pas être possible. Enfin, il est impossible lors d'une transaction d'effectuer des entrées/sorties, puisque, par définition, celles-ci ne sont pas réversibles.

OpenMP

OpenMP [25] propose des primitives pour gérer du parallélisme adaptatif en mémoire partagée pour le Fortran et pour le C. Des annotations spécifiques sont fournies, permettant d'exécuter en parallèle des portions de code, telles que des itérations de boucles.

Le principe de base est l'annotation *omp parallel*. Cette annotation porte sur le bloc de code suivant. Elle indique au compilateur que ce bloc doit être exécuté en parallèle, pendant que la suite du programme se déroule normalement. Cette annotation est déclinée en plusieurs variations, dont la directive *omp parallel for* permettant d'indiquer à OpenMP que les itérations de la boucle suivante peuvent être lancées en parallèle. La figure 2.4 représente un exemple d'utilisation de cette annotation sur une addition simple de vecteurs.

Listing 2.4 – Exemple de boucle parallèle simple OpenMP

```
1 #pragma omp parallel for
2 for (i=0; i<n; i++) {
3     a[i] = b[i] + c[i];
4 }
```

Bien que cela puisse être manipulé explicitement, par défaut OpenMP est capable de choisir le nombre de threads à utiliser pour une exécution efficace.

OpenMP fournit également des annotations pour la prise en charge du parallélisme. Il est notamment possible de spécifier la sémantique des variables utilisées dans une section parallèle :

- *private*, *shared* : spécifie si une variable doit être dupliquée dans chaque thread ;
- *lastprivate*, *firstprivate*, *threadprivate* : détermine l'initialisation des variables ;
- *reduction* : permet de regrouper la valeur finale de la variable de chaque thread selon une opération spécifiée.

De plus, OpenMP fournit des primitives classiques d'ordonnancement telle que barrières et zone ordonnées.

OpenMP peut à la fois être relativement simple à utiliser et peut s'adapter relativement aisément sur du code existant. De plus, de nombreux compilateurs comprennent les annotations

OpenMP ; on peut notamment citer GCC avec GOMP [1], Intel dans ICC, IBM dans XL C/C++. Cependant, le champ d'application d'OpenMP, venant à l'origine du calcul haute-performance, reste relativement limité. Sa principale application reste l'écriture de boucles parallèles, bien que les nombreuses directives et annotations permettant de spécifier le comportement permettent en pratique de gérer un grand nombre de cas différents. De plus, il n'est pas possible de gérer des sous-groupes de threads, ce qui le rend inadapté pour des algorithmes et programmes travaillant sur des structures de données complexes.

2.2.4 Paradigmes parallèles

Langages fonctionnels

L'approche dataflow [84] représente le programme en un ensemble d'opérations élémentaires. Ces opérations élémentaires ont chacune un ensemble d'entrées et une sortie ; les sorties sont connectées aux entrées d'autres noeuds. Un noeud effectue son calcul lorsque toutes ses entrées sont disponibles. Si chaque noeud représente effectivement une opération élémentaire, alors cette représentation permet d'obtenir le maximum de parallélisme disponible. On remarque qu'il n'y a plus de notion d'instruction courante.

Les langages fonctionnels, basés sur la notion de dataflow [84] offrent par conception des propriétés pour le parallélisme. L'assignation unique permet en pratique que tous les accès à des variables soient en lecture seule, ce qui élimine les problème de concurrence d'accès sur des variables partagées. L'absence d'effet de bord permet de considérer n'importe quelle fonction comme asynchrone si nécessaire, laissant toute liberté au compilateur et au runtime quant à la manière de séparer l'exécution en plusieurs threads.

Le langage fonctionnel Haskell [45] possède plusieurs extensions [67] pour supporter le parallélisme à l'exécution. La primitive *forkIO* permet la création d'un thread, de manière similaire à un langage impératif ; un type appelé *MVar* a été introduit pour permettre la communication entre threads. Il existe également une implémentation de mémoire transactionnelle pour Haskell, en remplaçant le type *MVar* par un type *TVars* ; la création de threads repose également sur la primitive *forkIO*. GPH [76] propose des opérateurs « par » et « seq » [4] permettant d'influencer sur la manière d'exécuter le code Haskell, respectivement en forçant l'exécution en parallèle ou en conservant l'ordre syntaxique.

Il est à noter qu'Haskell en tant que langage fonctionnel respecte l'assignation unique. Cette propriété est utile pour la gestion du parallélisme puisqu'il n'y a pas de conflit d'accès : à partir du moment où une variable est disponible, sa valeur est toujours la même ; il n'y a pas besoin de lock. Cet aspect dataflow, base de l'exécution des langages fonctionnels, fournit donc du parallélisme : il suffit d'instancier les différentes instructions du programme simultanément, puis on peut les exécuter dans un ordre quelconque et de manière concurrente dès que les opérandes sont disponibles. Il est à noter d'ailleurs qu'en Haskell, l'ordre de définition du code n'a pas d'importance.

Cependant, bien qu'Haskell puisse permettre en théorie d'obtenir du parallélisme automatiquement, les extensions précédemment citées restent nécessaires. En effet, le parallélisme obtenu automatiquement a un grain trop fin pour pouvoir être exploité au niveau logiciel ; de plus, il est difficile de prévoir les endroits où ce parallélisme serait intéressant [18].

[4] Haskell étant un langage fonctionnel, il n'y a pas d'ordre d'exécution implicite défini par la syntaxe

STAPL

STAPL [9] permet d'exprimer des traitements parallèles en se basant sur les structures de données à manipuler. STAPL reprend la logique de la bibliothèque STL[70] en l'adaptant afin d'offrir du parallélisme.

- Les *pContainer*s sont les équivalents de la notion de conteneur de la STL, qui regroupe notamment les *vector*, *list* et *map*. Tout en maintenant la compatibilité avec la STL, les *pContainer*s fournissent également, en plus d'itérateurs, des instances de *pRange*s, permettant de faire des traitements concurrents sur la structure de données. Cela implique notamment de permettre un accès partiellement aléatoire aux éléments, indispensable au parallélisme.
- Les *pAlgorithm*s sont les implémentations parallèles des algorithmes de la STL. Ils prennent notamment en paramètre des instances de *pRange*s (alors que la STL prend des itérateurs simples) permettant à l'algorithme de s'exécuter parallèlement tout en respectant les contraintes de concurrence.
- Les *pRange*s offrent une dimension supplémentaire par rapport aux itérateurs conventionnels. En plus de permettre d'itérer sur les éléments, ils peuvent fournir des sous-partitions des données, et ce de manière récursive si nécessaire. Ainsi, ils font la liaison entre les *pAlgorithm*s et les *pContainer*s, permettant aux premiers de manipuler les seconds parallèlement.

En plus des ces éléments reprenant les concepts de la STL, STAPL fournit des concepts additionnels permettant de gérer en pratique l'exécution concurrente. Notamment, le couple *scheduler/distributor* va permettre de déterminer quand partitionner les *pRange*s, quand exécuter le code sur ces partitions et sur quels processeurs. Le but et la difficulté du couple *scheduler/distributor* est donc de faire des choix judicieux afin d'améliorer la performance globale tout en respectant les contraintes de concurrence de l'algorithme.

STAPL permet un modèle de programmation parallèle favorisant la réutilisation, puisqu'il n'est pas nécessaire de connaître ou même de comprendre les problèmes de concurrence sous-jacents pour pouvoir utiliser les conteneurs et algorithmes parallèles. De plus, comme STAPL se base sur des concepts relativement génériques, il est relativement d'adapter de nouvelles constructions ou nouveaux algorithmes. STAPL est également capable de s'adapter efficacement aussi bien sur des systèmes à mémoire distribuée que sur des systèmes à mémoire partagée, ce qui est relativement rare pour un framework parallèle.

Cependant, si STAPL est très bien adapté et simple d'utilisation pour du parallélisme étroitement guidé par les structures de données, il parait moins adapté pour du parallélisme de traitement, où il faudra introduire des structures de données artificielles afin d'aider au parallélisme.

MapReduce

L'approche MapReduce [27] permet de gérer facilement le parallélisme de problèmes décomposables en deux étapes précises, l'étape de « map » et celle de « reduce ». Les dénominations de ces deux étapes viennent des opérations éponymes des langages fonctionnels. L'étape de *map* permet au programmeur d'analyser et de transformer chacune des données d'entrée de manière indépendante ; chaque opération peut donc être menée en parallèle. Ensuite, l'étape de *reduce* permet d'agréger les résultats intermédiaires afin d'obtenir le résultat final.

L'exemple canonique pour cette approche est un programme calculant le nombre d'occurrences d'un terme dans un large corpus de document courts. La phase de « map » consiste à scanner un document d'entrée ; la phase de « reduce » consiste à faire la somme des résultats

obtenus pour chaque document. MapReduce instancie donc un « map » pour chacun des documents, les exécute séquentiellement ou parallélement, puis, dès que des résultats intermédiaires sont disponibles, exécute des étapes de « reduce » afin de sommer les résultats obtenus.

Le programmeur n'a à se soucier que de l'écriture de ces deux phases qui, en soi, ne sont que du code séquentiel normal. MapReduce repose sur le fait que tout l'état du programme ainsi que ses dépendances sont connus par le système, ce qui d'une part lui laisse la latitude de les manipuler à sa guise, mais décharge également le programmeur de cette difficulté. Cependant, à l'inverse des langages fonctionnels qui permettent également au système d'avoir une connaissance complète de l'état et des dépendances du programme, MapReduce obtient cela à une granularité adaptée au parallélisme de threads.

Le système de gestion du MapReduce peut donc choisir quand et comment exécuter les opérations de « map » et de « reduce », ce qui permet d'adapter automatiquement l'exécution aux contraintes de l'environnement. Par exemple, si plusieurs machines sont disponibles, au lieu de devoir calculer séquentiellement plusieurs « map », il est possible de les lancer en parallèle. Si les données d'entrée sont réparties sur plusieurs machines, le système peut ordonnancer les phases de « map » sur les différentes machines en tenant compte de la répartition afin d'éviter des transferts de données inutiles. La fiabilité du système sous-jacent peut être également gérée : comme MapReduce connaît l'état précis du système, il peut facilement relancer une tâche ayant échoué parce que la machine a un problème réseau ou matériel. MapReduce permet donc d'obtenir un vrai passage à l'échelle à partir du moment où le problème s'exprime selon son formalisme.

De plus, en fournissant un formalisme générique, il est aisé d'avoir des outils de suivi, diagnostic et débuggage génériques, ce qui aide également à gérer la complexité liée au parallélisme, la diversité des plateformes matérielles ainsi que la réutilisation.

Cependant, tous les avantages cités, bien qu'importants, découlent du fait que MapReduce est un framework parallèle imposant de découper et d'exprimer son propre programme d'une manière extrêmement précise. Cela limite donc MapReduce à des catégories de problèmes relativement limitées, dont typiquement l'aggrégation de données. De plus, MapReduce semble pour l'instant surtout utilisé pour des traitement de type batch, son utilisation dans le cadre d'algorithmes temps réel semble plus restreint.

MGS

MGS [37] permet de décrire spatialement des structures de données complexes ainsi que des transformations les modifiant. Les données sont décrites sous forme de collections topologiques ; le programmeur décrit les relations entre éléments. Par exemple, un élément de liste pourra être défini en précisant qu'il a un voisin au nord et un à l'est. La figure 2.5 contient quelques exemples de définition de structures régulières (dites «GBF», pour «Group-based data field») mais il est également possible de définir des structures plus complexes et irrégulières.

Listing 2.5 – Exemples de structures régulières MGS : grille tore et plateau hexagonal

```
1 gbf Grid2 =< north, east >
2 gbf Torus2 =< north, east; 12*east = 0, 40*north = 0 >
3 gbf Hexa2 =< east, north, northeast; east + north = northeast >
```

Il est également possible de décrire des transformations en décrivant d'une part une fonction qui permet d'identifier les zones de collections topologiques transformables et d'autre part la transformation à y appliquer.

Il s'agit d'un framework spécifique à un domaine, imposant d'écrire son algorithme sous la forme demandée par le moteur. À partir du moment où cela est possible, le moteur de MGS

s'occupe alors à chaque étape de partitionner et de trouver les sous-ensembles à transformer, permettant potentiellement des transformations parallèles sans difficulté directe pour l'utilisateur. De plus, les transformations et conditions de partitionnement sont séquentielles, ce qui reste donc relativement simple à manipuler pour l'utilisateur.

L'inconvénient étant qu'il s'agit plus d'un système de description de problème que d'un langage de programmation généraliste. Ainsi, s'il est n'est pas possible ou difficile de décrire son programme en phases de partitionnements/transformation sur l'ensemble des données, MGS ne sera pas exploitable.

Blob Computing

L'approche Blob Computing [39] fournit à la fois une architecture abstraite ainsi qu'un modèle de programmation. Les programmes sont composés d'automates cellulaires, capable de se dupliquer, de disparaître ainsi que de créer et détruire des liens entre eux. L'évolution du programme se fait spatialement, sur un substrat d'éléments de calcul élémentaires. Il y a ainsi un parallélisme implicite, fonctionnant de proche en proche et ne nécessitant donc pas de connaissance globale du système.

L'intérêt de cette approche est d'offrir d'emblée un parallélisme massif. Mais les automates cellulaires permettent également d'exploiter naturellement un grand espace de calcul, tout en intégrant la notion de communication qui est sinon souvent un facteur limitant. Il est ainsi possible d'exploiter des machines aux topologies et dimensions variées.

Cependant, la notion d'automate cellulaire reste très bas niveau. Bien qu'adaptée à certains types d'algorithmes, il est difficile de s'en servir pour des programmes plus complets. De plus, il est souvent nécessaire d'effectuer des entrées/sorties, ce qui va restreindre le passage à l'échelle du modèle Blob.

Chapitre 3

Parallélisation adaptative

On peut opposer deux grandes approches pour un framework parallèle : ad-hoc ou générique. Un framework ad-hoc cherche à fournir une exécution parallèle pour un type de problème précis. Par exemple, MGS, mentionné dans le chapitre précédent, offre ainsi un moyen d'exprimer aisément des problèmes relatifs aux collections topologiques. En imposant d'exprimer les données du problème d'une manière précise, le framework est capable d'aisément l'exécuter efficacement. Dans le cas de MGS, les collections et transformations sont décrites d'une manière qui permet de déterminer des ensembles indépendants automatiquement et ainsi obtenir du parallélisme. L'inconvénient de ce type de framework est d'être spécialisé ; il est donc difficile de l'utiliser pour décrire l'ensemble du parallélisme d'un programme complexe.

A l'opposé, les frameworks génériques tentent de fournir des outils de base permettant de construire une exécution parallèle. Un cas classique est la bibliothèque pthread qui fournit les primitives classiques du parallélisme en mémoire partagée : locks, conditions, etc. L'interêt de ce type d'approche est bien sûr de pouvoir utiliser le même framework pour une grande variété de problèmes. Ainsi, la librairie pthread est utilisée pour implémenter la majorité des autres frameworks parallèles existants sous linux. Mais bien sûr, cela signifie également que, pour pouvoir l'utiliser, il faut la plupart du temps reconstruire une couche d'abstraction par-dessus afin de l'adapter à son propre problème.

Bien sûr, souvent un framework sera un mélange des deux approches. Par exemple, l'approche MapReduce est très efficace pour les problèmes d'aggrégation de données ; mais elle est également utilisable pour de nombreux autres problèmes, au prix d'adaptations plus ou moins importantes.

Capsule se rapproche plutôt de la deuxième approche. Il s'agit d'un système générique, permettant l'écriture de code et d'algorithmes irréguliers parallèles mais pouvant également servir de base pour des systèmes plus haut-niveau spécialisés. Pour ce faire, Capsule propose deux mécanismes complémentaires, la *division conditionnelle*, permettant d'adapter dynamiquement le parallélisme d'un programme et la *cellule* aidant à la manipulation des données.

3.1 La division conditionnelle

3.1.1 Principe

Afin de pouvoir conserver une exécution efficace quelque soit le système sous-jacent, un programme utilisant Capsule décrit tous les endroits où il est capable d'exploiter le parallélisme. Cela inclut à la fois les différentes zones du code où le parallélisme est possible mais également - et surtout - tous les instants où il est possible à *l'exécution* de profiter de parallélisme supplémentaire.

Il s'agit d'une description du parallélisme *potentiel* et non pas du détail de l'exécution ; c'est Capsule qui, à l'exécution, détermine s'il est intéressant de profiter ou non de plus de parallélisme.

Pour ce faire, Capsule met à disposition une opération de *division conditionnelle*. La sémantique de cette opération est de demander au système, à l'exécution, s'il est intéressant d'augmenter le parallélisme disponible. De là, le programmeur peut lancer une division effective de son programme, en décrivant la manière de séparer par exemple les structures de données entre le thread existant et le thread nouvellement alloué.

3.1.2 Exemple

Prenons l'exemple d'une addition simple de vecteurs :

```
1  Pour i de 0 à n:
2      c[i] = a[i] + b[i]
```

Les itérations se font l'une après l'autre, comme représenté sur la figure 3.1.

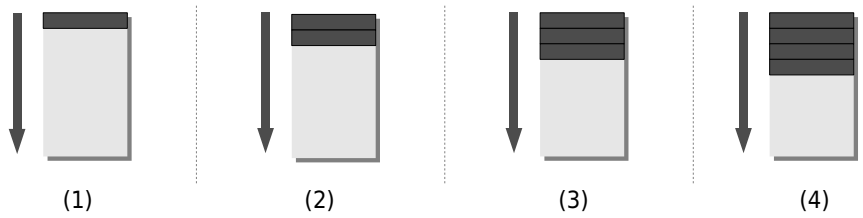


Figure. 3.1 – Évolution des premières itérations sur la version séquentielle de la boucle.

Pour pouvoir adapter le parallélisme dynamiquement via la division conditionnelle, la boucle doit, à chaque itération, essayer de diviser en 2 ensembles les itérations restantes. Si la division échoue, la boucle continue normalement.

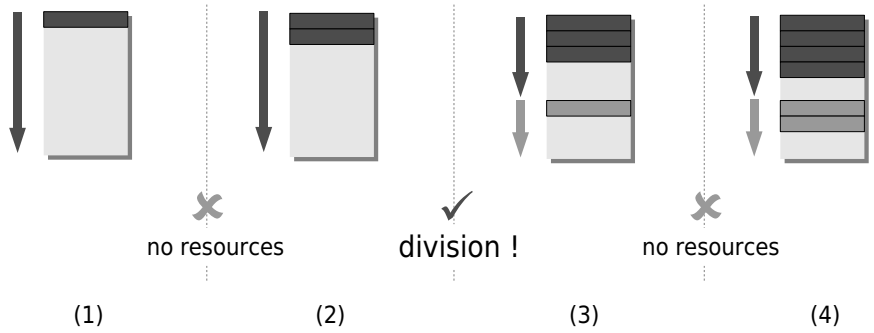


Figure. 3.2 – Exemple d'évolution temporelle d'une boucle Capsule. Les zones foncées représentent les itérations déjà calculées. Sur les 3 demandes de division représentées, seule la deuxième est honorée, conduisant à la création d'un deuxième thread.

Dans l'exemple de la figure 3.2, 4 étapes de calcul de la boucle sont représentées. Au début, la boucle commence normalement, avec un seul thread ; en (1), la première itération est effectuée. A ce moment là, le programme essaye d'effectuer une division puisqu'il est possible d'exploiter plus de parallélisme en séparant les itérations restantes. Capsule refuse par manque de ressources. Le thread continue son travail, exécute une deuxième itération en (2). Ensuite, le programme essaye encore de se diviser ; lors de cet essai, Capsule accepte et un nouveau thread

est alloué. Le programme divise donc en deux les itérations restantes à effectuer : le thread déjà existant continue les itérations de la première moitié de la boucle, le nouveau thread s'occupe de la deuxième moitié des itérations restantes. Ils continuent chacun leur propre exécution en exécutant une itération (étape 3). Puis chacun essaye lui même de se diviser, ce qui est refusé dans l'exemple (étape 4). Ensuite, ils continuent de la même façon, en essayant de se diviser à chaque itération.

Ainsi, dès que des ressources de calcul seront disponibles, quelle qu'en soit la raison, le programme peut s'adapter, quelle que soit la topologie de l'architecture. Tous les threads créés essaieront eux-mêmes de se diviser et de fournir plus de parallélisme, il n'y a pas de centralisation nécessaire de la gestion des divisions.

En pratique, le code doit simplement ajouter la demande de division, appelée « *probe* » à chaque itération (voir listing 3.1). Si la demande est acceptée, alors le programme calcule et adapte les paramètres vis-à-vis du nouveau thread et le lance effectivement à l'aide de « *divide* ». On voit également que lorsque que les divisions (« probe ») sont systématiquement refusées, comme dans le cas d'un mono-processeur, alors le comportement est exactement celui de la boucle simple, à l'exception d'un « probe » à chaque itération.

Listing 3.1 – Pseudo-code d'une implémentation Capsule d'une boucle s'adaptant dynamiquement au parallélisme disponible.

```
1  fonction loop(min, max):
2      Pour i de min à max:
3          c[i] = a[i] + b[i]
4          Si probe(loop):
5              divide((i+max)/2, max)
6              max = (i+max)/2
```

Le partitionnement des données pour les threads nouvellement créés est effectué par l'utilisateur (dans l'appel à *divide*). En effet, partitionner efficacement des données impose de savoir à l'avance lequelles seront utilisées par quel thread, connaissance difficile à construire de manière automatisée et surtout dynamiquement ; l'utilisateur est donc le mieux à même de faire cela puisqu'il a normalement une vision globale du programme. En pratique, dans la majorité des cas, cela ne pose pas de problème particulier.

Mais comme c'est le système Capsule qui décide quand effectuer une division, il a un contrôle implicite sur le partitionnement. Il est par exemple possible au système d'observer différentes métriques d'efficacité (saturation ou non des accès mémoire par exemple) et agir en conséquence sur les demandes de divisions (par exemple, refuser les divisions lorsque la bande passante est saturée).

Ainsi, le programmeur, en décrivant uniquement la manière d'exploiter le parallélisme potentiel, permet au programme de s'adapter à l'architecture sous-jacente et ce, en prenant en compte les propriétés à l'exécution des données utilisées.

3.2 Mémoire structurée

3.2.1 Principe

Le mécanisme de division conditionnelle permet de gérer le parallélisme aussi bien sur des machines à mémoire partagée que sur des machines à mémoire distribuée. Typiquement, dans le cas d'un système à mémoire distribuée, la division conditionnelle se contente de regarder l'état des voisins, évitant ainsi de coûteuses communications avec l'ensemble du système.

Or, avec un modèle mémoire classique, comme représenté sur la figure 3.3, la mémoire est vue comme un seul bloc avec un adressage linéaire simple, sans sémantique spatiale ou temporelle. Vu du système, chaque thread peut accéder à n'importe quelle zone à n'importe quel moment. Un

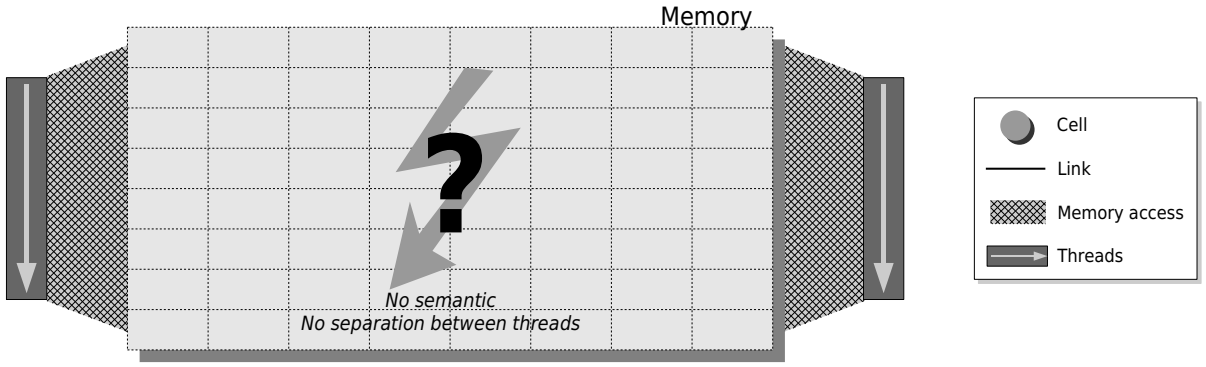


Figure. 3.3 – Vue par le système de la mémoire dans un modèle classique : chaque thread peut accéder à n'importe quel moment à n'importe quelle partie de la mémoire. Il n'est pas possible de construire une frontière sûre entre les zones utilisées par les threads.

moyen d'avoir plus de sémantique avec ce modèle mémoire est d'avoir une approche statistique en se basant sur les accès passés. Cependant, cela ne permet pas de garantir des propriétés précises qui pourraient être nécessaires pour aider le parallélisme, telles que des garanties de non concurrence pour simplifier les locks. Il est également difficile de déplacer des zones mémoire d'un processeur à l'autre afin d'éviter des accès distants

L'analyse statique peut également permettre d'avoir une idée de la localité d'accès. Cependant, cette analyse est souvent limitée lors de la présence de pointeurs et quasiment impossible dans le cas de langages plus dynamiques.

Il est donc nécessaire pour le système d'avoir plus d'informations sur la structure des données et de l'utilisation mémoire afin de pouvoir les manipuler efficacement sur des architectures variées. Cependant, il n'est pas possible de demander à l'utilisateur de gérer tous les aspects mémoire, tels que le placement et le déplacement des données ; cela est complexe, nécessite souvent des informations connues seulement à l'exécution et dépendant beaucoup de l'architecture.

Capsule va donc aider l'utilisateur à gérer ses données. Bien qu'il soit de sa responsabilité d'effectuer la découpe de ses données pour les nouveaux threads afin de répartir le travail, la manipulation en elle même reste toujours de la même forme : assigner des éléments de structure de données à tel ou tel thread.

Capsule propose à l'utilisateur de découper son code et ses données en éléments autonomes, avec des liens explicites (vu du système) ; ces éléments sont nommés *cellules*. Ainsi, tout en laissant l'initiative à l'utilisateur du détail du découpage, le système peut manipuler à sa guise le détail de la répartition de chacun des éléments.

Le programmeur, lorsqu'il écrit son code, précise (implicitement ou explicitement) sur quelles cellules chaque morceau de code travaille. Capsule, en se basant sur cette information, peut alors prendre des décisions, comme par exemple limiter les divisions en cas de conflits trop nombreux ou encore décider de rapatrier certaines données sur la mémoire locale.

Le principe en est représenté sur la figure 3.4. Les cellules sont réparties par le système sur les différents noeuds de calcul ; les threads accèdent aux données via les quelques noeuds qu'ils connaissent : il n'y a pas d'accès aléatoire.

Les cellules sont les porteuses d'information ; les liens représentent uniquement la mise en

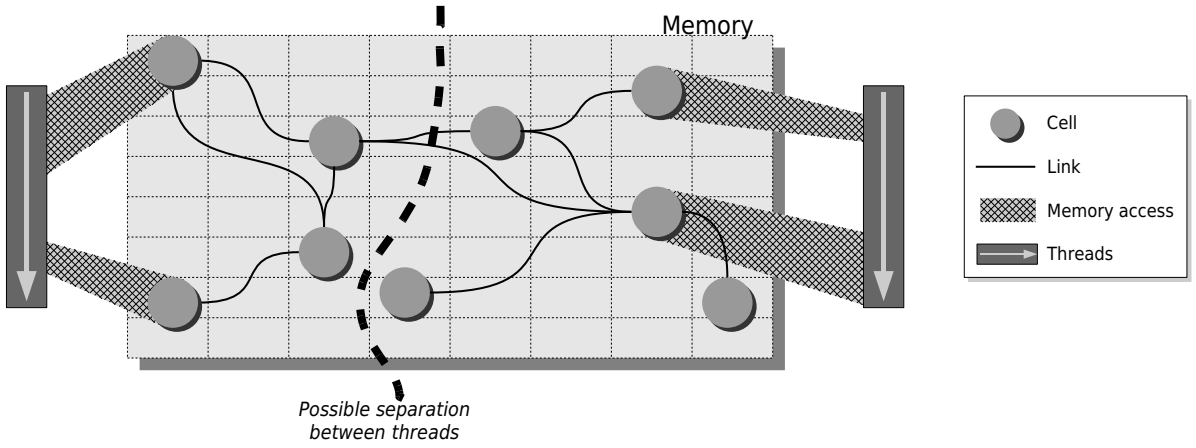


Figure. 3.4 – Connaissance de la mémoire par le système dans le modèle Capsule. Le système a une vision de la structure de la mémoire utilisée par le programme ; de plus, il connaît à chaque instant les zones mémoire accessibles par les threads. Il est donc possible de construire une frontière fiable et d'en tirer parti pour l'optimisation.

relation de plusieurs cellules et ne portent donc pas d'information utilisateur.

3.2.2 Exemple

En pratique, la cellule correspond à un élément de structure de données. Pour comparer à des langages tels que le C ou C++, cela revient à dire qu'une cellule est l'équivalent d'une *struct* ou d'une *class*. La notion de lien est quant à elle similaire à celle de pointeur. La différence qu'apporte Capsule ici est d'imposer la représentation classique en éléments simples liés entre eux (cellules/liens) d'une part mais également de faire, comme dit précédemment, que les liens/pointeurs soient connus du système.

Si on prend l'exemple de la représentation d'un graphe, on a donc, vu du système, quelque chose ressemblant à la figure 3.5. A gauche de la figure, un exemple de graphe simple est représenté ; il s'agit d'un graphe ayant des données aux noeuds (nom du noeud) et sur les arcs (poids de l'arc). Ce que voit le système Capsule est représenté sur la droite ; Capsule a conscience des différents liens existants, ainsi que du découpage mémoire en éléments simples, les cellules.

On note que chaque noeud du graphe est représenté par une cellule mais que chaque arc est également représenté par une cellule. Cela correspond au fait que dans Capsule seules les cellules, à l'opposé des arcs, peuvent stocker de l'information pour l'utilisateur. De plus, cela correspond au code généralement utilisé pour décrire un graphe, comme montré sur le listing 3.2.

Le fait que la représentation d'un arc contienne des pointeurs vers la tête et la queue correspond à ce qui est représenté sur la figure : les cellules encapsulant les arcs ont bien deux liens partant d'eux-mêmes.

Ainsi le système peut tirer parti de l'information fournie par la structure en graphe. Par exemple, lorsque qu'un thread accède à un noeud, le système va pouvoir commencer à charger également les noeuds connexes du graphe, fournissant ainsi une méthode de prefetch simple et efficace.

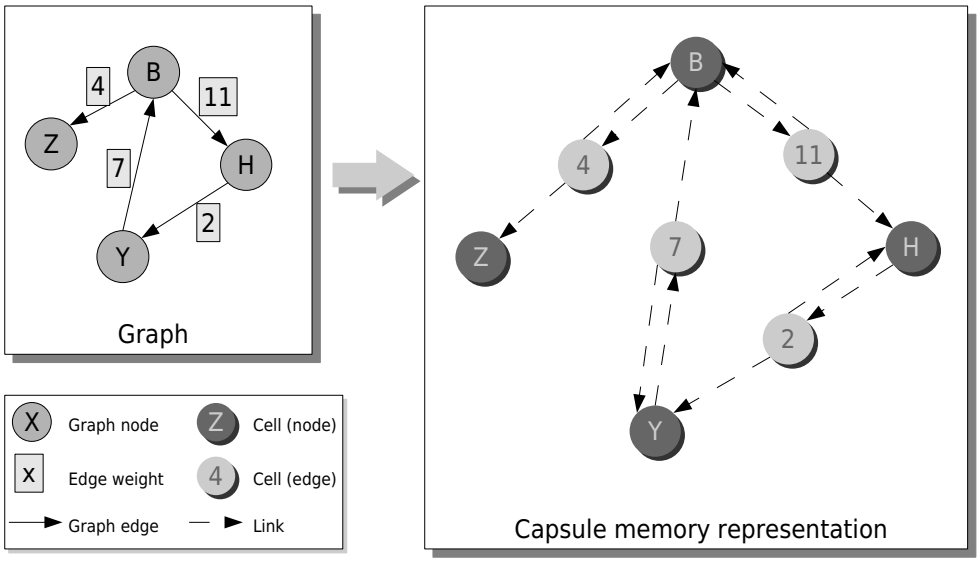


Figure. 3.5 – Exemple de représentation en Capsule d'un graphe simple, ayant des données sur les noeuds et sur les arcs.

Listing 3.2 – Pseudo-code décrivant une approche classique pour représenter un graphe en mémoire.

```
cell Node:
    string                  name
    list<link<Edge>>        edges

cell Edge:
    int                     weight
    link<Node>              tail
    link<Node>              head
```

3.3 Modèle

Capsule, en se basant sur ces notions de division conditionnelle et de cellule propose ainsi un modèle de machine parallèle abstraite, dont un exemple est représenté sur la figure 3.6. La machine est vue comme un ensemble quelconque de fils d'exécutions chacun séquentiel, dont le nombre peut varier au cours de l'exécution.

La mémoire est un modèle spatial : il n'y a pas d'accès aléatoire à n'importe quel élément, mais l'accès se fait via les liens entre chaque cellule, de proche en proche. Chaque fil d'exécution a une liste, contenant généralement assez peu d'éléments, indiquant les cellules sur lesquelles le fil travaille. Pour accéder à d'autres éléments, il doit passer par les liens des cellules déjà connues ; il n'y a pas d'accès aléatoire à la mémoire.

La relation entre code et données peut ainsi se voir de deux manières :

- Le code se déplace sur les données : puisqu'à chaque instant chaque thread a une liste précise de cellules connues (celles sur lesquelles il travaille), le thread va évoluer sur la surface mémoire, en passant d'une cellule à l'autre.
- Les données se déplacent sur la surface de calcul : à chaque fois qu'un thread suit un lien

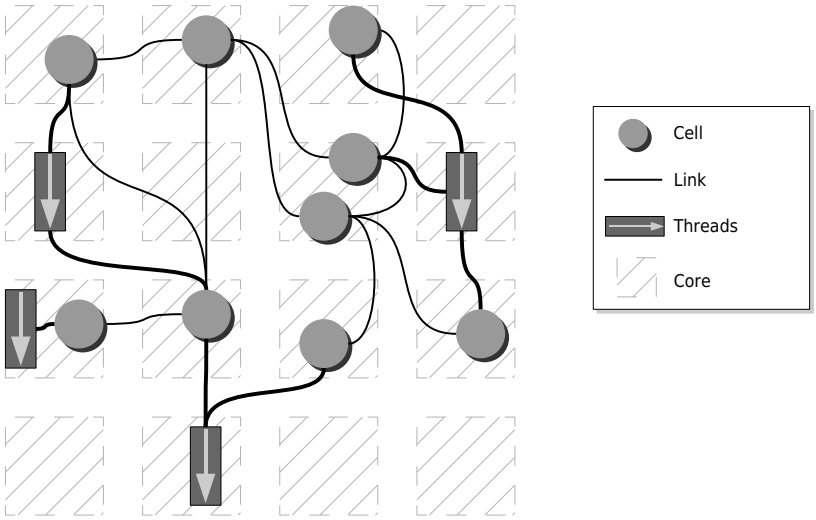


Figure. 3.6 – Modèle abstrait de machine présenté au programmeur. L'assignation des threads et cellules aux différents processeurs et banques mémoires ne lui est pas connu, le système gère cela dynamiquement.

entre cellules, cela revient à tirer la donnée vers le thread.

Bien sûr, puisqu'un thread peut travailler sur plusieurs cellules et qu'une cellule peut être utilisée par plusieurs threads, le comportement en pratique est un mélange des deux approches.

3.4 Éléments de Capsule

La figure 3.7 représente les différents éléments d'implémentation de Capsule.

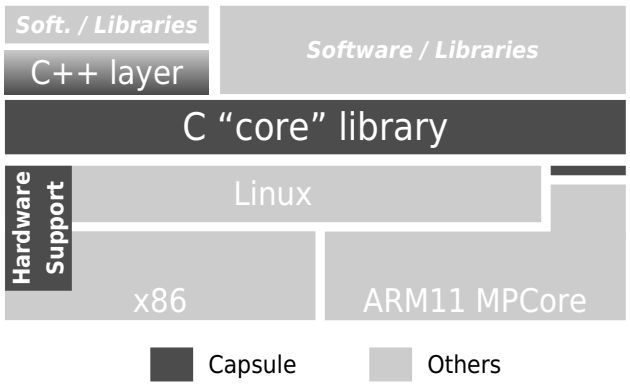


Figure. 3.7 – Éléments de Capsule.

La bibliothèque C fournit une abstraction portable et réutilisable de la division conditionnelle. Cela permet d'écrire des programmes ou des bibliothèques qui peuvent ensuite être utilisés tels quels sur différentes architectures et types de support, la spécialisation se faisant dans la bibliothèque Capsule.

Cette couche C fournissant le modèle d'abstraction matérielle, elle peut également servir de base à des frameworks parallèles fournissant des modèles de plus haut niveau.

Deux implémentations existent actuellement pour la division conditionnelle :

- Une version hardware, fournissant en natif la notion de thread et de division conditionnelle, détaillée dans le chapitre 4 ;
- Une version purement logicielle, fonctionnant sur des processeurs grand public existants, détaillée dans le chapitre 5.

L'implémentation principale de la version software de Capsule se base sur *pthread* et fonctionne sur Linux. Il existe également une version fonctionnant directement sur ARM multi-coeurs, sans utilisation de système d'exploitation ni de la bibliothèque pthread ; l'objectif était de montrer la portabilité et polyvalence de Capsule.

Enfin, la couche C++ de Capsule permet de fournir des concepts plus haut niveau, facilitant la réutilisation. La notion de cellule, avec liens explicites, est implémentée à ce niveau, profitant du pouvoir d'expression du C++ par rapport au C. Cela est détaillé dans le chapitre 6.

Chapitre 4

Implémentation matérielle

4.1 Éléments de conception

La première implémentation matérielle de Capsule est basée une architecture SMT. L'approche SMT pour les processeurs permet d'obtenir des threads légers, à coût de création très faible, ce qui correspond bien à l'approche Capsule. Bien que le cas des CMP ne soit pas traité ici[1] , il s'agirait également d'une plateforme adaptée à Capsule grâce aux communications à faible coût entre les coeurs ; les machines SMP sont également dans ce cas.

Pour que Capsule puisse fonctionner, il est nécessaire d'ajouter quelques fonctionnalités à un SMT normal :

- la division conditionnelle ;
- la création/suppression rapide des threads ;
- des primitives de synchronisation efficaces.

La première est spécifique à Capsule, les autres sont plus classiques pour les SMT. L'architecture est donc capable, via la division conditionnelle, de prendre des décisions quant au parallélisme. Cette architecture Capsule est appelée « SOMT », pour « Self Organized Multi-Threaded processor », processeur multi-threadé auto-organisé.

Le processeur SMT sur lequel l'architecture Capsule est basée est similaire à celui proposé par Tullsen [78]. Son principe est représenté sur la figure 4.1 ; il y a 8 contextes matériels et 32 registres par contexte. A chaque cycle, 16 instructions peuvent être récupérées ; une politique de type Icount 4.4 [77] fait que des instructions de 4 threads (sur les 8) sont récupérées à chacun des cycles. Chaque thread a son propre contexte matériel, incluant entre autres l'état d'exécution du thread, ses registres ainsi que sa position courante (pointeur d'instruction).

4.1.1 Mécanisme de division

L'architecture SMT utilisée comme base permet d'avoir plusieurs threads s'exécutant en parallèle, ayant chacun leur propre contexte. Afin de gérer dynamiquement la création et la durée de vie des threads, plusieurs instructions sont ajoutées.

L'instruction *nthr*, pour « New THRead » permet à un thread de demander la création d'un nouveau thread. Cela correspond à la division conditionnelle de Capsule : cette instruction ne fait que demander s'il est intéressant de créer un thread (et le crée le cas échéant) mais ne l'impose pas. Ainsi, l'architecture est libre d'ignorer cette instruction si elle estime cela nécessaire ou intéressant vis à vis des ressources disponibles.

[1] Des travaux sur le cas des CMP sont en cours actuellement dans le cadre d'une autre thèse.

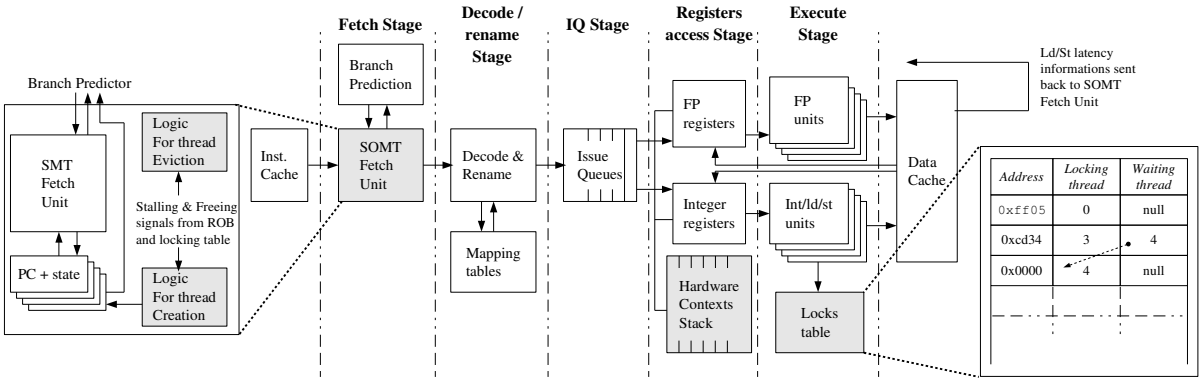


Figure. 4.1 – Multi-threading auto-organisé.

Le traitement de l'instruction *nthr* suit plusieurs étapes. Elle est tout d'abord considérée comme un branchement non-conditionnel. Au moment de son exécution, elle crée un nouveau thread via la réservation d'un contexte matériel. Ces contextes peuvent avoir 3 états :

- *actif* : les instructions du thread continuent à être récupérées ;
- *bloqué* : le contexte est utilisé, mais les instructions ne sont pas actuellement récupérées ;
- *libre* : le contexte n'est pas utilisé actuellement.

Lorsqu'un *nthr* a été décodé, un contexte *libre* est choisi et est mis a l'état *bloqué*.

Quand l'instruction *nthr* se termine, les registres du thread ayant initié la division sont recopiés vers le contexte matériel nouvellement alloué. Le PC (« Program Counter »), indiquant la position actuel du flux d'exécution est également recopié du thread courant vers celui nouvellement créé. Enfin, le nouveau contexte matériel est mis à l'état « actif » et le thread est donc lancé.

Le *nthr* pouvant être spéculatif, les registres sont recopiés tardivement, au moment du commit. Une autre solution possible serait de recopier spéculativement les tables de renommage de registres, mais cela risquerait d'être coûteux. En pratique, le thread d'origine est bloqué pendant un cycle pour la copie, et le nouveau contexte est maintenu bloqué pendant un temps variable, le temps que toutes les copies soient correctement effectuées. Si une instruction *nthr* spéculative était sur le mauvais chemin, le contexte est simplement remis à l'état « libre » une fois l'erreur de spéculation détectée.

Afin d'éviter les problèmes avec la spéculation de division, les instructions des nouveaux contextes démarrent de manière retardée, prenant en compte la longueur du pipeline, assurant ainsi que le nouveau thread existe réellement. En pratique, il s'avère que ce délai d'attente n'est guère perceptible sur la performance grâce à la vaste quantité de parallélisme exploitée ; il ne semble donc pas nécessaire d'ajouter un mécanisme complexe capable d'inverser l'effet d'un thread mal spéculé afin d'optimiser ce cas.

Enfin, l'instruction *kthr* permet à un thread de se terminer et ainsi libérer un contexte matériel. Lorsque l'instruction est décodée, le contexte passe de l'état « actif » à l'état « bloqué », empêchant ainsi la récupération des instructions suivantes. Lorsque *kthr* atteint l'étape de « commit », le contexte est effectivement libéré en passant à l'état « libre ».

4.1.2 Stratégie de division

Comme mentionné précédemment, c'est l'architecture qui décide si une division proposée par *nthr* est réellement effectuée. La politique de base est gloutonne : une instruction *nthr* ne sera

exécutée que si un contexte matériel est disponible à ce moment là ; sinon, elle sera considérée comme un *nop* n'ayant aucun effet.

Cependant, cette politique simple peut poser problème lorsque, par exemple , un algorithme touche à sa fin. Typiquement, cela entraîne beaucoup de parallélisme potentiel, mais chacune des sections est très courte. Par exemple, si on considère un quicksort, à chaque fois que l'on sépare une liste en deux sous listes, une division est tentée afin d'exécuter en parallèle les deux sous tris. Lorsque la liste devient petite (moins d'une dizaine d'éléments) ou que la séparation de la liste n'est pas équilibré, les threads ont très peu d'éléments à trier. Dans ces cas là, il risque d'y avoir une perte de performance, puisque le coût de la division peut alors dépasser le gain obtenu grâce au parallélisme.

Pour réduire ce phénomène, la version matérielle de Capsule suit les terminaisons de threads récentes. Si les threads sont en train de mourir rapidement, alors le processeur bloque les prochains *nthr* même si des ressources sont disponibles. En pratique, cela signifie qu'une instruction *nthr* sera réellement prise en compte à condition qu'il y ait un contexte disponible et que le nombre de threads ayant terminé dans les N derniers cycles ($N = 128$ dans les expériences) soit inférieur à la moitié du nombre de contextes matériels.

4.1.3 Activation et désactivation de threads

Cette version matérielle de Capsule possède également un « swap » pour les threads, permettant ainsi d'avoir plus de threads vivants que de contextes matériels. Cela peut être comparé aux processeurs superscalaires capables d'avoir plus d'instructions en vol que d'unités fonctionnelles réellement présentes.

L'architecture Capsule possède une pile LIFO de contextes matériels au niveau de la banque de registres (voire figure 4.1). Dans [80], Tune et al présentent un concept similaire de contextes matériels virtuels utilisant une pile nommée « buffer de registres inactifs ». Ils estiment dans leur implémentation que la latence de swap de contexte n'excède pas 15 cycles en moyenne, via l'utilisation d'instructions micro-codées.

Comme dans la version Capsule il n'est pas fait usage de masque de registre, que l'implémentation du swap n'est pas particulièrement optimisée et qu'il y a 16 contextes virtuels au lieu de 4 (et 8 contextes physiques au lieu de 2), on estime le coût du swap à 200 cycles pour les 62 registres. En pratique, nous avons estimé qu'une pile de 16 éléments était suffisante pour une architecture à 8 contextes physiques. Comme il y a 62 registres (31 flottants, 31 entiers) ainsi qu'un « Program Counter », la pile fait donc environ 4ko.

Lors de nos expériences, aucun dépassement de pile n'a été detecté. Cependant, dans le cas d'une implémentation réelle, il faudrait prendre en compte cela. Il serait par exemple possible de générer une trappe système afin de pouvoir transférer les contextes bloqués les plus anciens de la pile matérielle à la mémoire.

Pendant qu'un contexte est recopié sur la pile, son état passe à « bloqué » afin d'éviter qu'il continue de récupérer des instructions. Une fois toutes les instructions du thread terminées, le contexte matériel peut être enfin mis à l'état « libre ».

4.1.4 Stratégie d'ordonnancement et de swapping

La stratégie choisie pour l'ordonnancement des différents threads lors du fetch est similaire à un ICount.4.4 [77]. Cette politique privilégie les threads les plus performants qui sont les plus à mêmes d'exploiter efficacement les unités fonctionnelles.

De plus, la stratégie de swap choisie pour l'architecture Capsule permet de mettre de côté les threads induisant de longues latences, essentiellement créées lors du chargement de données. Ce

mécanisme est similaire à celui qu'on peut retrouver dans des grosses machines multi-threadées [8].

Cela veut donc dire que les choix d'ordonnancement et de gestion des threads sont basés uniquement sur des information locales, comme le comportement des caches des threads, et non sur des informations globales.

En pratique, chaque temps de chargement de données est comparé à la moyenne des 1000 derniers chargements. Si la latence est supérieure, un compteur spécifique au thread est augmenté ; sinon, il est décrémenté. Quand ce compteur atteint un certain seuil (256, le compteur ayant une valeur initiale à zéro), le thread est mis sur la pile de swap si plus aucun contexte matériel n'est libre.

4.1.5 Systèmes de synchronisation pour les threads

Comme proposé dans [79], l'exclusion mutuelle pour l'accès aux zones mémoire partagées est basée sur une table de locks, comme indiqué dans le figure 4.1. Cette table, en maintenant la liste des locks, permet d'une part de vérifier lors d'un accès mémoire si l'adresse concernée est actuellement bloquée et permet d'autre part de gérer les listes d'attente de threads.

Pour gérer ces locks, deux instructions, *mlock* («Memory Lock») et *munlock* («Memory Unlock»), ont été ajoutées. Lorsque l'instruction *mlock* est appelée, un lock est pris sur l'adresse précisée en paramètre. Si une autre instruction *mlock* est ensuite appelée sur cette même adresse, le thread demandeur passe à l'état «bloqué» et son identifiant est stocké dans la table de lock, indiquant ainsi qu'il est en attente pour la libération de l'adresse mémoire indiquée.

Chacune des entrées de la table comporte 3 informations : l'adresse lockée, l'identifiant du thread ayant présentement le lock et un identifiant du thread attendant depuis le plus longtemps sur cette adresse. Ainsi, lorsqu'une instruction *munlock* est exécutée, c'est le plus vieux thread en attente qui sera réveillé et deviendra le nouveau propriétaire du lock.

4.1.6 Compilation vers la version matérielle

La version matérielle de Capsule a été développée avant d'avoir le le modèle de programmation proposé dans les sections 5 et 6. Les codes ont donc été écrits dans un style relativement bas-niveau, suivant de près la forme des instructions.

Le listing 4.1 montre un exemple de code bas niveau écrit pour la version matérielle du dijkstra.

Listing 4.1 – Source brut d'un extrait de la version Capsule de dijkstra

```
1  void dijkstra (node_t *node, int distance, node_t *from)
2  {
3      mlock(node);
4      if (distance >= node->distance) {
5          munlock(node);
6          kthr();
7      }
8
9      node->distance = distance;
10     node->parent = from;
11     munlock(node);
12
13     for (edge_t e = node->edges; e != NULL; e = e->next) {
```

```
14          switch(nthr()) {
15              case -1:
16                  //replication denied (sequential)
17                  dijkstra(e->head, node->distance + e->length, node);
18                  break;
19              case 0:
20                  //replication allowed, current worker
21                  break;
22              case 1:
23                  //replication allowed, new worker
24                  dijkstra(e->head, node->distance + e->length, node);
25                  kthr(); //this worker dies upon completion
26          }
27      }
28  }
```

Les appels de fonction *mlock*, *kthr* et *nthr* correspondent quasiment directement à leur équivalent assembleur présenté auparavant. Le switch autour du *nthr* permet de déterminer le comportement en fonction du résultat de la division conditionnelle. Ainsi, si la division échoue, *nthr* renvoie -1 et le dijkstra fait une exploration en profondeur séquentielle. Si la division réussi, la fonction renvoie 0 dans le thread ayant fait la demande et 1 dans le nouveau thread ; dans le cas présent, le thread original continue à itérer sur les autres fils, pendant que le nouveau thread travaille sur le fils courant.

Un exemple de code assembleur obtenu après compilation est donné sur le listing 4.2.

Listing 4.2 – Code assembleur d'un extrait de la version Capsule de dijkstra

```
1  nthr r3,div$ok          ;jump if division is allowed«
2  <»call code>           ;Sequential version
3  br div$end              ;continue normal execution
4  div$ok:                 ;on allowed division
5  beq r3,div$0            ;continue execution in parent
6  div$1:                  ;new cell code
7      -> New stack
8  <new cell code>
9      -> Restore stack
10 kthr                    ;new cell dies upon completion
11 div$0:                  ;nothing special to do
12 div$end:                ;execution continuation
```

L'instruction *nthr* au début effectue la requête effective au processeur pour un contexte additionnel ; si la division est autorisée, le premier paramètre, un registre, stocke le numéro du thread courant (0 pour le thread préexistant, 1 pour le nouveau thread), le deuxième paramètre indique où continuer l'exécution dans le cas d'une division réussie. En pratique, lors d'une *nthr* réussi, les registres sont dupliqués dans le nouveau thread ; le registre en paramètre permet donc au code de savoir dans quel thread il est en train de s'exécuter puisque c'est la seule différence visible facilement depuis le code assembleur.

On voit ainsi que, dans le cas où aucune division n'est autorisée, le surcoût revient à un *nthr* et un branchement inconditionnel afin d'éviter la section correspondant au cas de la division réussie. Le cas où la division est autorisée ajoute du code dans le nouveau thread afin d'allouer et

de désallouer la pile, comme expliqué plus loin en 5.3.5. Dans le cadre de cette version matérielle, l'implémentation de la pile est une approche ad-hoc très simple qui ajoute en moyenne 15 cycles par division réussie.

4.2 Méthodologie

4.2.1 Simulateur

Le simulateur de l'architecture Capsule est basé sur SimpleScalar 3. La partie fonctionnelle a été dupliquée afin de pouvoir gérer plusieurs threads ; cela a impliqué des modifications de l'étage de fetch ainsi que, dans une moindre mesure, de celui de dispatch qui a nécessité la duplication des tables de renommage des registres.

Les expériences ont été exécutées sur l'architecture SOMT Capsule, sur une architecture SMT et sur un super-scalaire sur-dimensionné (voir tableau 4.1). Bien que Capsule puisse également servir à de l'équilibrage de charge entre plusieurs processus, cela n'a pas été implémenté pour le moment sur la version matérielle.

4.2.2 Chargement des instructions

À chaque cycle, jusqu'à 32 instructions peuvent être chargées. Ces instructions sont réparties entre 4 threads au maximum. Chaque thread peut donc récupérer jusqu'à 8 instructions, soit une ligne de cache. Un buffer d'instructions double est utilisé ; seules les 16 premières pourront être utilisées dans le cycle. Les 16 emplacements restants permettent de réduire les accès au cache d'instruction et ainsi de limiter la bande passante nécessaire.

Le buffer d'instructions mélange les instructions des différents threads. Si un thread n'est pas capable de fournir suffisamment d'instructions, la répartition de charge entre les threads sera dynamique. Ainsi, si un thread n'est pas capable de récupérer 4 instructions, d'autres threads pourront récupérer jusqu'à 8 instructions afin de compléter le buffer. La table 4.1 donne le détail des paramètres matériels.

Fetch width	16
Issue / Decode / Commit width	8
RUU size (Inst. window)	256
LSQ size	128
FUs	8 IALU, 4 IMULT, 4 FPALU, 4 FPMULT
Branch prediction	Combined, 1K meta-table size 4K entries bimodal, 8K 2nd level entries Gap predictor,
Memory latency	200 cycles
L1 DCache	8kB, 1 cycle
L1 ICache	16kB, 1 cycle
L2 Unified Cache	1MB, 12 cycles

Table. 4.1 – Configuration des architectures SOMT, SMT et superscalaires utilisées.

4.2.3 Benchmarks

SPEC CINT2000	# Lines	# Modified or added functions	Modified or added lines	% total execution time
`181.mcf`	2412	2	174	45%
`175.vpr`	17729	10	624	93%
`256.bzip2`	4649	3	317	20%
`186.crafty`	45000	8	201	100%

Table. 4.2 – Portage vers Capsule des SPEC CINT2000.

4 SPEC CINT2000, indiqués sur le tableau 4.2 ont été utilisés comme benchmarks, ainsi que 4 algorithmes : compression LZW (utilisé dans 164.*gzip*), calcul de plus court chemin type Dijkstra (dont une version modifiée est utilisée dans 175.*vpr*), un réseau neuronal type perceptron et un tri quicksort. Le temps d'exécution de ces différents algorithmes varie selon les jeux de données de quelques milliers de cycles à plusieurs millions.

Comme mentionné précédemment, une implémentation Capsule d'un algorithme donné peut différer sensiblement d'une implémentation normale C/C++. Ainsi, dans le cas des SPEC CINT2000, il a été nécessaire pour comprendre la spécification réelle du programme de faire du reverse-engineering sur le code. Comme cela est très coûteux en temps, seul 4 SPEC CINT2000 ont été transformés.

Pour chacun de ces benchmarks, nous avons identifié un sous ensemble du graphe de control-flow qui correspondait à une partie conséquente du temps d'exécution du programme. En pratique, cela signifie qu'il est possible de modifier à l'intérieur de ce sous-ensemble le format des structures de données sans pour autant provoquer un surcoût trop importqnt lié à la conversion des données. Ensuite, chacun de ces sous-ensemble a été reprogrammé selon l'approche Capsule. Le tableau 4.2 indique pour chacun des benchmarks le nombre de lignes de code et de fonctions modifiées, ainsi que le temps d'exécution passé dans la partie du programme choisie.

4.2.4 Parallélisation statique

Pour mettre en évidence l'intérêt du coté matériel de Capsule, des versions statiques de chacun des algorithmes ont été conçues pour tourner sur le SMT normal. Ainsi, les versions Capsule fonctionnent sur le SOMT, les versions statiques sur le SMT et les versions de référence sur le superscalaire. Tout ces benchmarks ont été compilés pour Alpha 21264 à l'aide du compilateur *cc* de DEC, avec l'option $-O3$ pour l'optimisation.

Bien qu'il y ait une littérature abondante sur le parallélisme niveau thread et qu'il existe des versions parallèles de plusieurs des algorithmes (multiplication de matrices, quicksort, perceptron, matrice-vecteur) pouvant apporter une meilleure performance que notre parallélisation grain-fin, il n'a pas été possible de trouver des versions parallèles de certains des algorithmes (LZW [2], Dijkstra [3] , MCF).

Pour avoir une comparaison homogène des versions parallélisées statiquement de nos algorithmes, il a été choisi d'adapter la version Capsule de chacun de ces benchmarks en limitant les divisions possible afin d'imiter un découpage statique des données. Ainsi, pour ces versions

[2] Des algorithmes proches de LZW existent en version parallélisées, mais pas LZW lui-même.

[3] Les versions parallèles existantes de Dijkstra ciblent du parallélisme très gros grain et ne sont que peu efficaces sur des petits jeux de données.

statiques, la version Capsule est lancée ; une fois que tous les threads du processeur sont ou ont été utilisés, les divisions suivantes sont systématiquement bloquées. Le découpage dépend donc presque uniquement de la forme de l'algorithme et non pas du détail de la topologie des données, comme dans une approche statique du parallélisme. Il est donc systématiquement possible d'avoir une version avec parallélisme statique, ce qui ne serait pas nécessairement le cas avec un compilateur auto-parallélisant, particulièrement dans le cas de codes utilisant intensivement des pointeurs. De plus, ce compilateur ne serait pas nécessairement capable d'extraire suffisamment de parallélisme pour occuper toutes les ressources matérielles en threads du processeur.

4.3 Évaluation de performance

4.3.1 Algorithmes

Comme mentionné dans la section 4.2, sont comparées dans la plupart des expériences une version super-scalaire (séquentielle puisqu'il n'y pas a de parallélisme de thread), une version parallélisée statiquement sur SMT et la version Capsule parallélisée dynamiquement sur l'architecture SOMT.

4.3.2 Structure de données irrégulières et parallélisme

Un des principaux intérêts de l'approche Capsule est de permettre un équilibrage de charge dynamique au sein de la puce. La stratégie de base pour les divisions, gloutonne, autorise la division tant que des ressources matérielles sont disponibles. Pour un algorithme qui contient des structures de données complexes et irrégulières, la charge de travail de chacun des threads peut être très variable. Avec une parallélisation statique, obtenue à la compilation sans connaissance de la topologie des données, un thread se retrouvant avec une charge de travail faible se retrouve en attente, gaspillant ainsi des ressources de calcul. Avec Capsule, chaque thread essaye de diviser son propre travail en permanence à l'aide de l'instruction *nthr* afin de créer plus de parallélisme. Ainsi, dès qu'un thread termine, un autre voit une de ses requêtes de division accordée, permettant de réutiliser le contexte matériel qui vient de se libérer, maximisant donc l'utilisation des ressources.

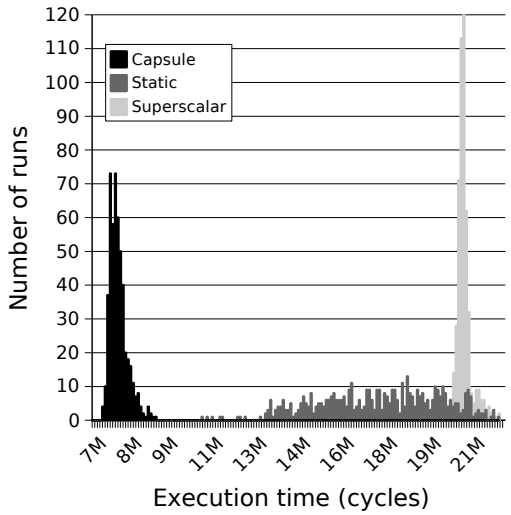


Figure. 4.2 – Répartition du temps d'exécution pour quicksort.

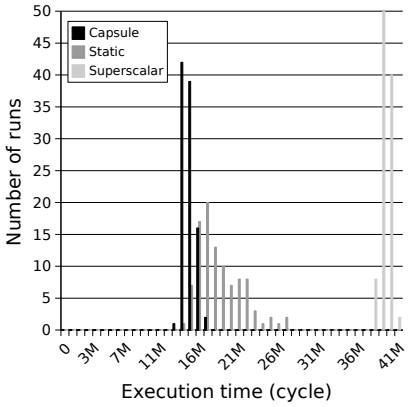


Figure. 4.3 – Répartition du temps d'exécution pour dijkstra.

Ce phénomène est visible dans les figures 4.2 et 4.3 pour les algorithmes quicksort et dijkstra. Quicksort a été testé sur 500 listes de même taille générées aléatoirement et dijkstra sur 100 graphes de 1000 noeuds également construits aléatoirement. En abscisse se trouve le temps d'exécution et selon les ordonnées se trouve le nombre de jeux de données avec approximativement le même temps d'exécution. On voit sur ces figures aussi bien le gain de performance que le gain en variabilité de chaque approche.

Dijkstra découpe le graphe en plusieurs sous-graphes de taille variable selon le nombre de noeuds fils lors de chaque noeud étudié ; quicksort divise à chaque itération la liste en deux sous-listes de taille variable, fonction du choix du pivot lors de l'exécution de l'algorithme. Avec cette approche de parallélisme Capsule, un speedup de 2.51 est obtenu pour quicksort et de 1.23 pour dijkstra par rapport aux versions statiques, et des speedups de respectivement 2.93 et 2.51 par rapport aux versions superscalaires. La performance obtenue est également nettement plus stable en version Capsule qu'en version avec parallélisme statique ; cela est dû au load-balancing dynamique obtenu grâce à la division conditionnelle fréquente.

4.3.3 Zones parallèles de courte durée

La politique de division conditionnelle gloutonne permet d'optimiser l'équilibrage de charge dynamique puisqu'elle maximise l'utilisation des ressources. Cependant, comme mentionné dans la section 4.1, si la durée de vie d'un thread est courte, le coût de la division risque d'être supérieur au gain obtenu grâce à la parallélisation de la section de code en question. Ainsi, pour éviter ce phénomène, comme expliqué précédemment, l'architecture suit le nombre moyen de morts de threads par cycle et, en fonction de cela, détermine si une division est réellement nécessaire. Cette limitation de division s'illustre bien sur les algorithmes LZW et perceptron.

La version Capsule du perceptron essaye en permanence de diviser les groupes de neurones en deux sous-groupes contenant la moitié des neurones considérés ; le groupe initial contient dans cette version 10000 neurones. La version Capsule de LZW essaye de diviser systématiquement les séquences de caractères en deux sous-séquences de même taille afin de paralléliser la phase de recherche ; la séquence initiale contient 4096 caractères. Dans les deux cas, il n'y a que peu de travail à faire sur chaque sous-ensemble de données et il y a comparativement beaucoup d'occasions de division ; la figure 4.4 montre que la limitation dynamique des divisions permet d'obtenir un gain non négligeable.

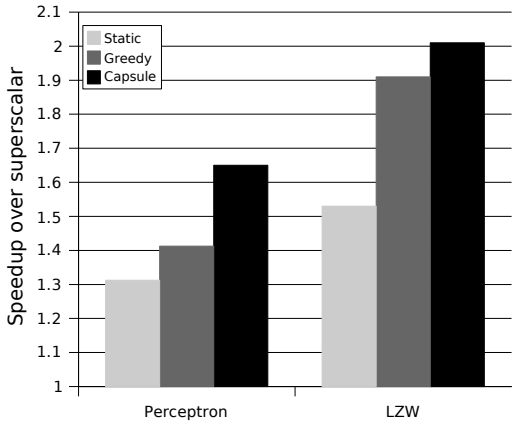


Figure. 4.4 – Limitation des divisions pour les zones parallèles de courte durée

4.3.4 Passage à l'échelle

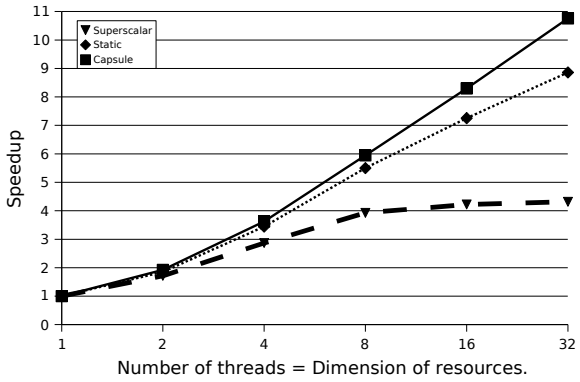


Figure. 4.5 – Passage à l'échelle (moyenne sur les différents algorithmes testés).

La notion de division conditionnelle dont la gestion effective se fait au niveau architectural permet de faciliter l'écriture de programmes supportant un passage à l'échelle correct lorsqu'on augmente le parallélisme matériel disponible. La division conditionnelle permet en effet de profiter dynamiquement d'un nombre quelconque de contextes matériels sans pour autant modifier le programme, aussi bien au niveau source que binaire.

La figure 4.5 illustre cela en augmentant le nombre de ressources matérielles pour les différentes versions de nos benchs. Pour les versions SMT et SOMT, cela a consisté en l'augmentation du nombre T de threads matériels ; pour les version superscalaires cela a consisté en l'augmentation du nombre d'ALU (T ALU entières) et de la fenêtre d'instructions ($T \times 128$ éléments) ; le gain est moyenné sur les différents algorithmes présentés ici. Bien sûr, pour les grandes valeurs de T, les architectures considérées ne sont guères réalistes mais cela permet néanmoins d'avoir un aperçu des possibilités et limitations dues à Capsule pour le passage à l'échelle.

4.3.5 Les SPEC CINT2000 adaptés pour Capsule

La plupart des résultats de performance pour la version matérielle de Capsule ont été obtenus avec des implémentations d'algorithmes dans les sections précédentes. Le but de cette section

est de montrer que Capsule est utilisable sur des applications plus conséquentes que des implémentations d'algorithmes classiques et que même des applications comme les SPEC CINT2000 peuvent avoir suffisamment de parallélisme lorsque l'on s'éloigne du source pour remonter à la définition du problème.

Comme expliqué dans la section 4.2, il a été nécessaire de faire du reverse-engineering des SPEC CINT2000 afin de remonter à la spécification du problème qu'ils cherchent chacun à résoudre ; dans certain cas, pour réduire l'ampleur du problème, ce travail a été restreint à une sous-partie du bench.

Bien que le travail de *modification* d'un programme existant, les SPEC CINT2000 en l'occurrence, soit conséquent, l'écriture directe de code Capsule est plus simple, particulièrement lorsque l'on maîtrise bien la problématique que le programme cherche à résoudre. De plus, utiliser Capsule dès la conception permet également d'avoir de manière générale un code plus efficace puisque suivant au plus près les concepts sous-jacents.

Les résultats de performance présentés ici sont obtenus pour les versions Capsule sur un SOMT à 8 threads et pour les versions séquentielles sur un processeur super-scalaire aux ressources similaires ; sauf indication contraire, la configuration architecturale est la même que celle décrite dans le tableau 4.1.

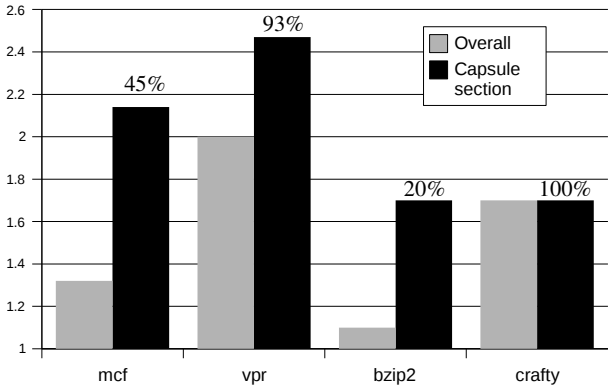


Figure. 4.6 – Gain de performance global et pour les sections modifiées ; les pourcentages au dessus des barres correspondent à la fraction de l'exécution totale passée dans la zone Capsulisée.

Benchmark	# divisions requested	# divisions allowed	% divisions allowed	# insts / division allowed
`mcf`	99598	40532	40%	3.7K
`vpr`	67560	2702	4%	4.5M
`bzip2`	38656	2319	6%	30M

Table. 4.3 – Pourcentage et taux de divisions réussies.

Les gains obtenus pour l'architecture Capsule, sur l'ensemble des programmes et uniquement sur la zone transformée Capsule sont visible sur la figure 4.6. Les pourcentages indiqués sur la figure sont les mêmes que ceux du tableau 4.2, ils représentent la part de code transformé en suivant l'approche Capsule.

Le tableau 4.3 montre les divisions demandées, celles effectivement autorisées ainsi que le taux de divisions réussies obtenues (sous la forme du nombre d'instructions par division réussie).

Ces chiffres permettent d'avoir une idée de la quantité de parallélisme disponible ainsi que de la saturation possible des ressources matérielles disponibles.

Les sous sections suivantes précisent comment chaque SPECT CINT2000 considéré a été « Capsulisé » et donnent quelques détails supplémentaires sur leur comportement.

181.mcf

Dans la version originale de ce benchmark de planification d'itinéraires de bus, il y avait une traversée d'arbre utilisant une implémentation séquentielle ; cette version a été remplacée par une version Capsule parallèle. On remarque que, sur ce benchmark, le taux de division est nettement plus élevé que pour les autres programmes (une division toutes les 3700 instructions en moyenne) ; cela est dû au fait que la version Capsule essaye de diviser à chaque noeud mais que le travail sur chacun de ces noeud est très faible (une comparaison et une opération arithmétique). Il aurait été faisable de choisir une granularité moindre, mais cette version permet de montrer qu'il est réellement possible d'avoir des divisions très fréquentes, quelque soit la granularité mise en jeu.

175.vpr

Ce benchmark résout un problème de routage sur FPGA. La version parallèle Capsule teste les différents routages et placements en explorant simultanément plusieurs graphes de circuits possibles (jusqu'à 8000). Il est à noter que la version parallèle de l'algorithme converge en 9 itérations au lieu de 8 pour la version originale séquentielle, ce qui impacte donc le gain obtenu.

La version parallèle est limitée par la bande passante mémoire. Si la taille de cache et le nombre de ports sont doublés, le speedup d'une itération passe de 2.47 à 3.5, le speedup global passant quant à lui à 3.0.

256.bzip2

Ce benchmark est un programme de compression par blocs, dont la partie la plus coûteuse est en réalité un problème de tri de chaînes. La version Capsule implémente donc une version parallèle de ce tri.

186.crafty

La version Capsule de ce programme d'échecs est en pratique basée sur une implémentation parallèle classique pré-existante de Crafty [4] elle même basée sur la bibliothèque pthread [54] et capable de diviser l'arbre de recherche entre les threads.

L'intérêt de ce SPEC CINT2000 est ici d'illustrer que l'approche Capsule peut être appliquée sur des implémentations de programmes déjà parallélisés et qu'une gestion statique des ressources matérielles risque d'être moins efficace que la gestion dynamique du SOMT. Le programme maintient un pool de threads en attente active et, en pratique, gère les threads de manière logicielle, ce qui empêche quasiment toute division dynamique ; c'est pour cela que ce benchmark n'est pas mentionné sur la figure 4.3. De plus, comme les threads du pool sont en attente active, l'augmentation du nombre de threads utilisés a tendance à dégrader la performance. Le speedup

[4] Crafty est parallélisé depuis la version 19.19, alors que le SPEC CINT2000 correspondant est basé sur la version 14.3. L'algorithme principal reste le même dans les deux versions, seuls quelques heuristiques ont été changées, telles que les heuristiques de notation des solutions.

obtenu sur l'ensemble du programme pour un SOMT à 4 contextes est de 2.3, alors que pour un SOMT à 8 contextes le speedup obtenu n'est plus que de 1.7 à cause de ce problème.

4.3.6 Impact du passage au CMP sur la division conditionnelle

La version matérielle de Capsule a été développée sur la base d'un SMT et non pas d'un CMP, bien que cette dernière soit la cible principale de Capsule. Pour extrapoler les résultats du SMT au CMP, les deux paramètres critiques dans le cas de l'approche Capsule sont la latence de division et la latence de probe.

Sur un CMP à mémoire partagée, la latence de probe changera probablement peu, mais la latence de division risque d'augmenter nettement plus à cause du coût de lancement d'un thread sur un coeur distant. Lors de simulations avec des latences de division allant jusqu'à 200 cycles, la variation de performance moyenne observée est de moins de 1% par rapport aux résultats présentés ici. La raison principale de cette relative insensibilité est que le taux de division reste assez limité. Par exemple, dans le cas de 181.mcf, qui a de loin le ratio de divisions réussies le plus haut parmi les SPEC CINT2000 considérés, il n'y a qu'une seule division réussie toutes les 3700 instructions en moyenne.

L'approche Capsule ne nécessite pas nécessairement un grand nombre de divisions pour être efficace, mais plutôt un découpage permettant l'adaptation rapide de charges irrégulières, comme illustré dans le cas de quicksort. Avoir des probes fréquents permet de réduire la latence d'adaptation face à des conditions changeantes. Cependant, même cela considéré, des divisions à plus haute latence risquent d'être dommageables pour des programmes comme 181.mcf qui ont un taux de divisions effectives assez élevé.

Une version matérielle de Capsule pour un système CMP à mémoire distribuée pourrait être également intéressante, ces types de systèmes se retrouvant fréquemment dans l'embarqué. La mémoire distribuée nécessite un support plus complexe pour Capsule, particulièrement pour les probes qui nécessitent des communications efficaces avec les voisins afin d'éviter les conflits de réservation. Le chapitre 6 traite d'un système de gestion de la mémoire qui serait particulièrement adapté au problème de la mémoire distribuée.

Chapitre 5

Implémentation logicielle

5.1 Principes

La version logicielle de la division conditionnelle devait initalement permettre de tester les codes Capsule et de les débugguer aisément sans être contraint par un simulateur. La version logicielle permet donc d'exploiter des multiprocesseurs classiques du commerce. Après optimisations, il se trouve qu'en pratique la version logicielle offre des performances permettant de se passer d'un support matériel spécifique.

Cette version logicielle est vue comme une extension de la bibliothèque pthread. Elle permet d'offrir un parallélisme visant la performance au lieu d'un parallélisme de convenance. En effet, en utilisant pthread normalement, la création de thread réussit systématiquement, quel que soit l'état des ressources. Cela est utile pour avoir un threading permettant l'implémentation de certains types de codes dont le parallélisme est intrinsèque, tels que des serveurs réseau ou par exemple des interfaces graphiques. Le parallélisme dans ces cas simplifie l'écriture du programme, il n'est pas nécessairement introduit pour obtenir plus de performance. Cela se voit également dans de nombreux langages offrant des primitives de parallélisme similaires, mais incapables de profiter de plusieurs threads matériels ; les threads sont systématiquement sérialisés. C'est notamment le cas pour Python, Perl ou les anciennes version d'Erlang.

Bien sûr, il est possible de constuire par dessus pthread des programmes profitant des threads dans un but de performance. Mais cela implique de concevoir une surcouche gérant l'adaptation aux ressources, comme par exemple des mécanismes de work-stealing ; cela peut s'avérer particulièrement lourd quand les zones à paralléliser sont complexes.

Capsule propose une approche permettant d'offrir des threads s'adaptant à la performance et ce, de manière réutilisable aisément. Conceptuellement, il s'agit d'avoir une indication supplémentaire à la création de thread, indiquant que le thread est nécessaire uniquement si les ressources le permettent, autorisant ainsi au système tout latitude de choix.

Plusieurs implémentations successives ont existé :

- Une version en Python, qui a servi de prototype initial, permettant de valider certains concepts d'encapsulation, ainsi que la faisabilité d'avoir une division conditionnelle au fil du code.
- Une version entièrement en C++ qui devait permettre de montrer la faisabilité de l'implémentation des concepts haut-niveau de Capsule dans un langage compilé, tout en profitant de vrais threads système contrairement à la version Python.
- La version actuelle, avec un coeur écrit en C et une surcouche C++ pour les concepts plus haut-niveau que la division ; cela correspond aux éléments de la figure 3.7. Les concepts ayant été validés, l'objectif de cette version était de fournir les primitives de base, réutili-

sables pour d'autres approches, en permettant le maximum de performances possibles.

5.1.1 Opérations de base

La version logicielle de Capsule sépare la division conditionnelle en deux temps :
- une demande de division, qui spécifie le code qui serait à exécuter en parallèle ;
- une activation, qui amorce effectivement le traitement parallèle lorsque la demande a été acceptée au préalable.

Cette dichotomie est présente pour obtenir plus de performance tout en conservant une certaine souplesse. En effet, puisque, une fois alloué, le thread n'est pas exécuté, il est possible de retarder le calcul des sous-ensembles de données à fournir à chaque thread. Ainsi, pour les probes échouant, ce calcul, pouvant être plus ou moins coûteux, n'est pas effectué.

Il est donc effectivement possible d'utiliser autant de probes que l'on désire, la seule contrainte étant que celui-ci soit suffisamment rapide dans la majorité des cas. En pratique, le cas de loin le plus courant est que toutes les ressources sont occupées et que par conséquent le probe échoue. La version logicielle de Capsule optimise essentiellement ce cas.

De plus, puisque le code à exécuter est précisé au moment de la recherche des ressources, il est possible de réserver des ressources spécialisées, adaptées au type de code à exécuter en parallèle. Par exemple, si le code est capable d'utiliser une unité vectorielle pour accélérer un calcul, le probe pourra vérifier qu'une telle unité est disponible.

Du point de vue de l'utilisateur, la demande de division renvoie un *contexte*. Un *contexte* représente en terminologie Capsule un travail à effectuer en parallèle. En pratique, l'utilisateur fournit une fonction qui sera exécutée dans le cadre de ce contexte dans un thread séparé. Quand la fonction se termine, le contexte lui même se termine et est détruit.

Bien que, vu de l'utilisateur, un contexte puisse sembler similaire à un thread de la bibliothèque pthread [1], le terme *thread* ici désigne un concept différent. Dans la version logicielle de Capsule, un *thread* désigne généralement un thread matériel (correspondant par exemple à un thread d'un processeur SMT ou un coeur d'un processeur CMP) ou le thread logiciel correspondant, puisque Capsule crée au début du programme un thread logiciel par thread matériel (voir la section 5.2.1 pour plus de détails). Ainsi, un thread peut exécuter plusieurs contextes successivement, alors qu'un contexte est détruit dès qu'il a fini sa part de travail.

En pratique, une division Capsule suit généralement le schéma donné en figure 5.1.

Lorsque l'application est capable d'exploiter plus de parallélisme et donc de se diviser, elle effectue un *probe*. Si Capsule décide que ce n'est pas intéressant de diviser, alors le programme continue normalement son exécution et doit réessayer régulièrement de diviser. Si la division est autorisée, alors le programme doit séparer en deux les structures de données qu'il doit traiter, lance effectivement l'exécution du nouveau contexte sur la deuxième moitié des données, puis continue lui même son travail. Le nouveau contexte suit un schéma similaire et essaye donc lui-même de diviser régulièrement son travail.

5.1.2 Exemple d'utilisation

La séparation probe/divide se reflète directement dans le code du listing 5.1.

Ce code représente une implémentation possible d'une boucle dont chaque itération peut être exécutée en parallèle. L'approche choisie est d'essayer de diviser après chaque itération le travail restant en deux parties égales ; cela correspond à l'exemple de la figure 3.2 de la section 3.1.

[1] la bibliothèque pthread, lors de la création d'un thread, prend également une fonction en paramètre qui déterminera la durée de vie du thread.

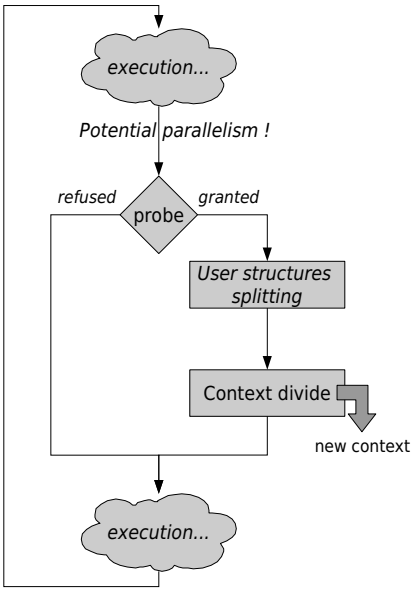


Figure. 5.1 – Squelette d'utilisation classique du couple probe/divide. Le nouveau contexte créé suit également un schéma similaire.

En ligne 9, la fonction *capsys_probe* est appelée avec en argument la fonction qui doit être exécutée en parallèle. Cette fonction représente la demande de division. Capsule regarde si des ressources sont disponibles et si il est intéressant de diviser ; le cas échéant, la fonction renvoie un nouveau *contexte*. Dans le cas contraire, la fonction renvoie un pointeur nul.

La ligne 10 teste le retour de la fonction de probe. Ainsi, si la division échoue systématiquement (cas d'un mono-processeur), la condition est systématiquement fausse et la boucle est strictement équivalente à sa version séquentielle.

Les lignes 11 à 14 sont exécutées si une division a été réussie. Au début de ce bloc, jusqu'à la ligne 13, on calcule les nouvelles limites de boucle. Le travail restant est séparé en deux ; *newmin* et *newmax* représentent les bornes des itérations à exécuter dans le nouveau thread, et $lt \rightarrow max$ (limite de la boucle courante) est adapté de manière similaire.

Enfin, en ligne 14, l'appel à la fonction *capsys_divide* effectue réellement la division. En pratique, cela signifie que le contexte alloué lors du probe commence son exécution. Cette fonction prend un paramètre qui est transmis à la fonction dans le nouveau contexte. Comme il est nécessaire de transmettre les *deux* bornes de boucle, il est fait appel à une fonction *alloc_looparg* ad-hoc à cet exemple qui permet d'encapsuler les différents paramètres dans une seule structure. Plus de détails sur ce sujet seront donnés dans la section 5.3.6.

5.1.3 Notion de groupe

Nécessité de synchronisation complexe

Il est souvent nécessaire d'attendre la fin d'un traitement avant de continuer l'exécution d'un autre morceau de code. Par exemple, dans le cas de la boucle simple parallèle, avant de passer au traitement suivant, il est nécessaire d'attendre que tous les contextes aient fini afin d'être sûr que les accès futurs aux données seront cohérents.

L'approche la plus simple pour cela serait de fournir un système de *join* simple, permettant

Listing 5.1 – Exemple d'utilisation de Capsule : addition de vecteurs

```
void loop(void *arg) {
    context_t *ctx;
    looparg_t* lt = (looparg_t*) arg;
    int i, newmin, newmax;

    for (i = lt->min; i<lt->max; lt ++) {
        lt->c[i] = lt->a[i] + lt->b[i];  //do iteration stuff

        ctx = capsys_probe(loop);
        if (ctx) {
            newmax = lt->max;
            newmin = (i+lt->max) / 2;
            max = newmin - 1;
            capsys_divide(alloc_looparg(lt ,newmin,newmax));
        }
    }
}
```

d'attendre la fin d'un unique contexte depuis un autre contexte. Dans le cas d'une boucle simple, cela revient à faire une synchronisation comme sur la figure 5.2.

Dans ce cas, chaque contexte, lorsqu'il a fini son propre travail, attend la fin des contextes qu'il a créé. À chaque fois qu'un de ceux-ci est terminé, il se réveille donc afin de vérifier s'il en reste encore d'autres ; le cas échéant, il se rendort. Si tous ses contextes fils sont terminés, il peut continuer son propre travail ; dans la majorité des cas, cela veut dire se terminer. Le seul cas où le contexte est toujours utilisé est le cas du contexte principal, où la suite du programme continue à se dérouler (fin de la section parallèle).

Ces séries d'endormissements/réveils afin de se rendormir ou de se terminer ont un coût non négligeable puisque le nombre de synchronisations sera proportionnel au nombre d'itérations. A chaque fois, il faut réveiller le thread, ce qui implique une latence supplémentaire. De plus, conserver des contextes vivants implique de garder leur pile disponible et donc de devoir réallouer une pile si le thread sous-jacent est réutilisé (cf section 5.3.5). Enfin, cela impose à l'utilisateur de conserver une liste des contextes fils pour chaque contexte, ce qui en pratique est relativement lourd à mettre en place.

Un autre surcoût dans l'exemple se pose : même si on est capable de limiter les réveils dûs à l'existence de plusieurs fils pour un contexte, dans l'exemple de la figure 5.2, chaque contexte doit néanmoins attendre ses contextes fils. Or, chaque contexte est indépendant une fois créé. Le seul moment où il est nécessaire de synchroniser, dans le cas de la boucle, est à la fin de tous les contextes. Cela assure que tout soit terminé lorsque l'algorithme de boucle rend la main à la suite du programme. Autrement dit, il est plus intéressant de posséder une sorte de barrière globale pour la synchronisation, comme en figure 5.3.

Cependant, il faut garder à l'esprit que le parallélisme peut être imbriqué : l'algorithme qu'on code peut très bien être inclus dans un autre algorithme lui même parallèle et donc ne représenter qu'une partie des threads existants. L'approche de barrière globale doit prendre en compte cela et permettre de synchroniser en se limitant à un ensemble de threads bien spécifiés.

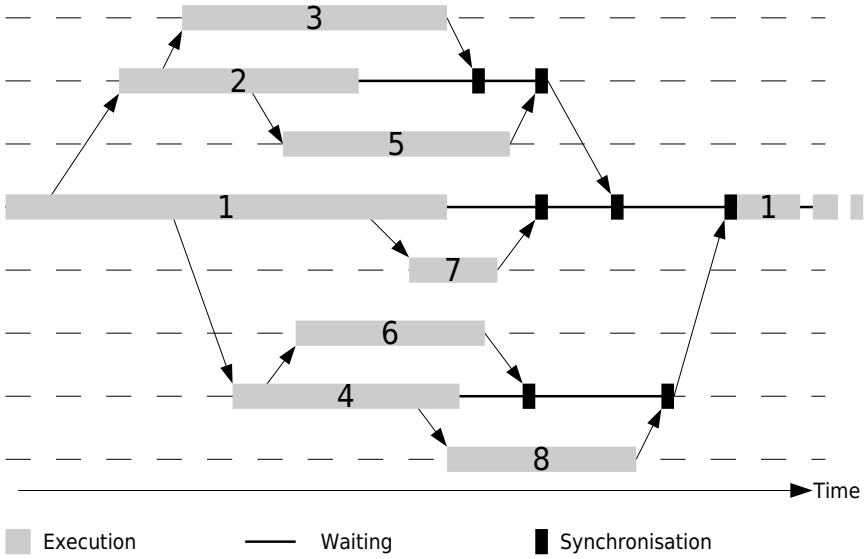


Figure. 5.2 – Représentation d'une exécution avec des synchronisations hiérarchiques. Chaque ligne représente la vie d'un contexte ; il n'a pas été représenté de réutilisation de thread pour éviter d'alourdir la figure.

Principes des groupes Capsule

Capsule propose un mécanisme de groupe afin de simplifier la gestion de la synchronisation. Ces groupes résolvent les deux problèmes :

- synchronisation de plusieurs contextes simultanément, sans nécessité de maintenir une liste de contextes ;
- synchronisation hiérarchique permettant de fusionner les niveaux.

Supposons maintenant, pour l'exemple, qu'il ne s'agit plus d'une boucle simple mais d'un algorithme plus complexe nécessitant, à certains niveaux intermédiaires de calcul, des synchronisations partielles, comme dans la figure 5.4.

Pour manipuler les groupes, Capsule fournit deux fonctions qui ne prennent et ne renvoient *aucun* argument :

- *cap_split()* crée un nouveau groupe ;
- *cap_join()* attend la fin du groupe courant puis le détruit.

Tout le reste est du ressort de Capsule ; la figure 5.5 représente le comportement des groupes lors de l'exécution représentée sur la figure 5.4.

Les règles d'évolution des groupes sont les suivantes :

- Un *groupe* Capsule contient des contextes et d'autres groupes.
- Un groupe initial, englobant récursivement (via les autres groupes qu'il contient) tous les contextes du process existe dès l'initialisation de Capsule (étape 1).
- Un contexte nouvellement crée appartient au groupe de son créateur au moment du *probe* (étapes 2,4,5,7,8,10,11).
- Chaque contexte et chaque groupe appartiennent chacun à un seul et unique groupe, appelé groupe parent, ce qui crée donc une hiérarchie.
- Le groupe courant est le groupe auquel appartient le contexte considéré.

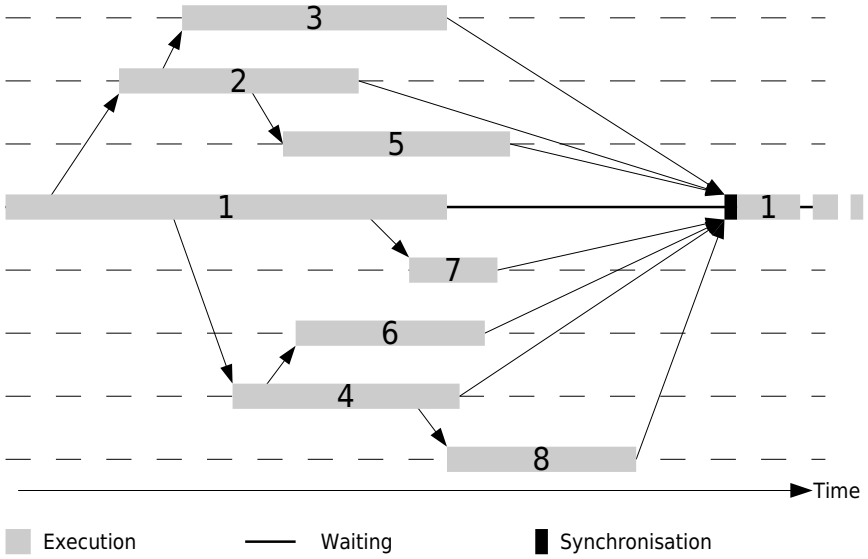


Figure. 5.3 – Représentation d'une exécution avec une synchronisation globale. Chaque ligne représente la vie d'un contexte ; il n'a pas été représenté de réutilisation de thread pour éviter d'alourdir la figure.

- Lorsqu'un groupe est crée, il appartient au groupe courant (étapes 3 et 6).
- Le contexte courant change de groupe pour appartenir au groupe nouvellement crée (étapes 3 et 6).
- Lorsque l'exécution reprend après un *cap_join*, le contexte courant est réassigné au groupe parent du groupe qui vient de terminer (étapes 16 et 19).

Les étapes mentionnées se réfèrent à la figure 5.5.

Bien que les règles puissent sembler nombreuses, l'utilisation reste simple : lorsqu'il est nécessaire d'avoir une synchronisation sur un ensemble de contextes, il suffit d'encapsuler la partie concernée entre un *cap_split()* et un *cap_join()*. Typiquement, un algorithme complet, tel que la boucle parallèle, commence par un *split* et finit par un *join* ; l'algorithme peut être ainsi utilisé sans problème dans un programme lui-même parallèle, sans risque d'interférence.

Enfin, cette gestion de groupe impose une synchronisation en partie hiérarchique : il n'est pas possible d'attendre la fin d'un contexte qui n'est pas un des fils ou petit-fils du contexte demandant la synchronisation. En pratique, aucun des cas rencontré n'imposait une approche différente de la synchronisation. Si cela était nécessaire, il serait de toute façon aisé de rajouter une synchronisation sur un contexte arbitraire unique.

5.2 Implémentation

La version logicielle de Capsule est basée sur la bibliothèque pthread. Elle est conçue comme une extension : l'utilisateur doit utiliser les fonctions pthread pour gérer le parallélisme, telles que les fonctions de gestion des mutex ou des « conditions ». L'objectif est que, à terme, Capsule puisse être intégré de manière transparente au système. Néanmoins, pour des questions de clarté et d'historique de développement, l'API fournie par Capsule ne suit pas les règles de codage

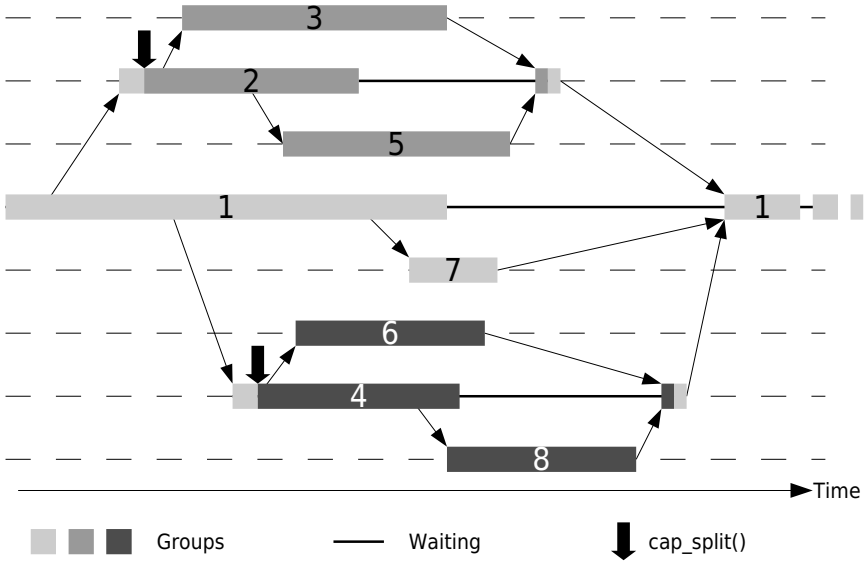


Figure. 5.4 – Représentation d'une exécution avec la gestion Capsule des groupes pour la synchronisation. Chaque ligne représente la vie d'un contexte ; il n'a pas été représenté de réutilisation de thread pour éviter d'alourdir la figure.

actuelles de pthread.

5.2.1 Structures

Capsule est construit autour de quelques structures principales (voir figure 5.6) représentant les différentes portées logiques existantes lors de l'exécution du programme.

Les sous-sections suivantes présentent chacune des structures en précisant leur portée ; pour chaque portée existante, une seule instance de la structure idoine existe.

Processus : *capsys_t*

Elle contient l'état global de Capsule. Une seule instance existe, créée automatiquement au premier probe ou explicitement via les fonctions adéquates de l'API Capsule. Elle contient deux listes principales : la liste des threads utilisés et la liste des threads disponibles. Ainsi, lorsque qu'une division est demandée, c'est via cette structure que la demande passe. De plus, cette structure contient des statistiques diverses sur le comportement de Capsule.

Cette structure est actuellement centralisée puisqu'elle correspond à l'état global de Capsule pour une application. Cependant, il est possible d'imaginer supprimer la liste de threads disponibles via un état par thread. Dans le cas d'une machine à mémoire partagée, cela risquerait d'induire un surcoût de performance puisqu'il faudrait maintenir des informations en plusieurs exemplaires ; éclater les listes de threads disponibles serait donc surtout intéressant pour le cas d'une machine à mémoire distribuée.

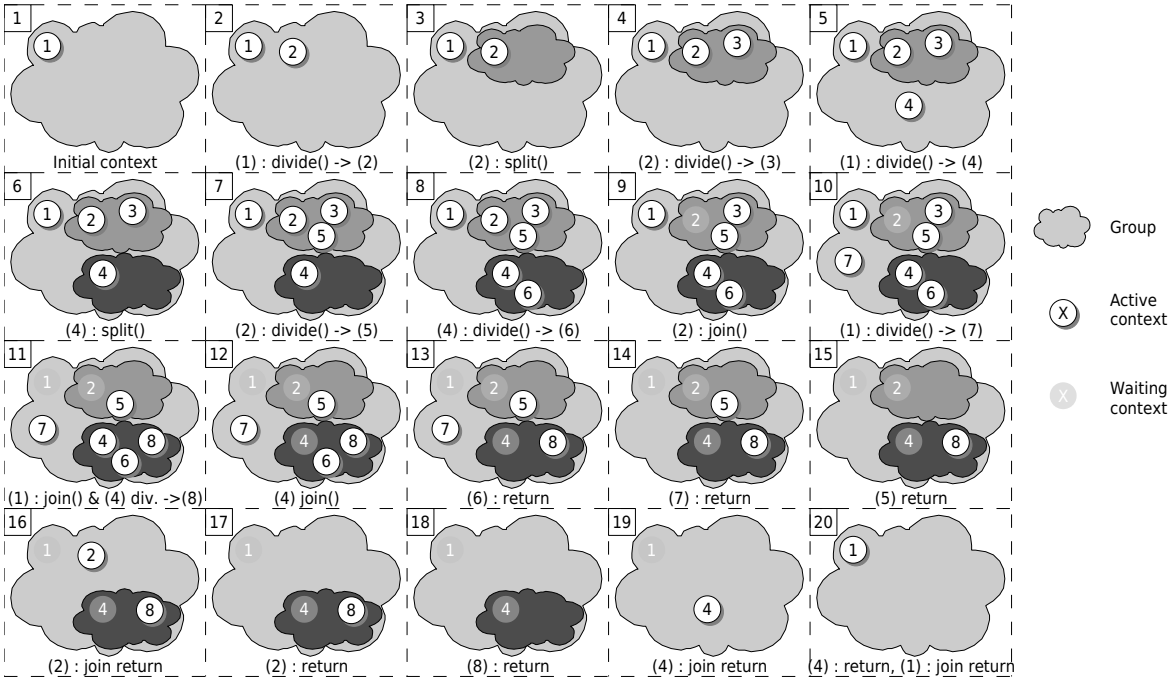


Figure. 5.5 – Évolution des contextes et des groupes correspondant à la figure 5.4. Le libellé de chaque étape précise l'opération que viennent de faire les contextes dont le numéro est précisé entre parenthèses.

Thread hardware : *capthread_t*

Afin d'être capable d'adapter le parallélisme en minimisant la latence, Capsule maintient un pool constant de threads. Le nombre de threads, par défaut, correspond au nombre de threads matériels disponibles sur la machine, chaque thread logiciel étant associé à un thread matériel ; ce nombre peut être modifié au démarrage d'un programme utilisant Capsule.

Chacun de ces threads est décrit via une instance de la structure *capthread_t*. Cette structure conserve les informations pour gérer l'activation et désactivation du thread, dont notamment l'assignation d'un contexte donné. Chacune des instances est créée à l'instanciation de *capsys_t* avec son pthread associé et n'est détruite qu'à la fin du programme. Un *capthread_t* exécute donc, au cours de la durée de vie du programme, plusieurs contextes successifs.

Dans la version logicielle de Capsule, lorsqu'un *thread* est mentionné, il s'agit donc d'une entité non visible par l'utilisateur et représentant un thread hardware. Il ne faut pas confondre cela avec la notion de contexte, visible à l'utilisateur, décrite dans la partie suivante.

Divisions : *context_t*

Cette structure encapsule une fonction à exécuter ; il s'agit de la seule structure Capsule visible à l'utilisateur. Un contexte est créé lors d'une tentative réussie de division via *capsys_probe* et est assigné à un *capthread_t* libre. Cette assignation à un thread peut néanmoins changer durant l'exécution, notamment à l'occasion d'une synchronisation (voir section 5.3.3).

Chaque contexte crée obtient un numéro unique ainsi que le numéro de son parent. Les contextes sont détruits automatiquement par Capsule lorsqu'ils ont fini leur propre exécution. Ils contiennent principalement des informations pour permettre et encadrer l'exécution du code

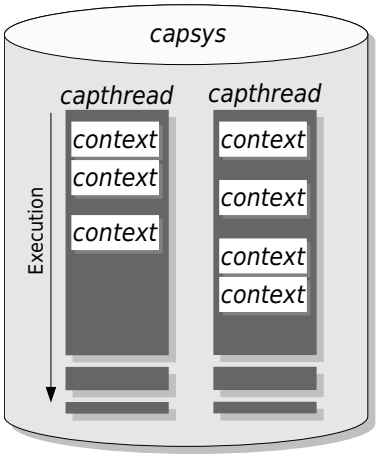


Figure. 5.6 – Structures principales de Capsule. Les groupes, contenant plusieurs contextes, ne sont pas représentés.

de l'utilisateur.

Synchronisation : *cap_group_t*

Les contextes peuvent être regroupés afin de faciliter la synchronisation. Cela permet d'attendre la fin d'un ensemble de plusieurs contextes. Cette structure permet de maintenir les informations nécessaires à la synchronisation d'un groupe, voir partie 5.1.3.

5.2.2 API

Initialisation et terminaison

Capsule maintient un pool de threads, en faisant correspondre un thread logiciel à un thread physique. Il est donc nécessaire, avant de pouvoir faire des divisions, d'initialiser ces threads ainsi que les structures de base de Capsule. Par défaut, lors d'un probe, Capsule s'initialise tout seul s'il détecte que ce n'est pas encore fait. De même, à l'inverse, lorsque Capsule a été initialisé automatiquement, un callback est mis en place afin de détruire proprement Capsule à la fin du programme.

Cependant, il peut parfois être intéressant de contrôler précisément la création et la destruction de Capsule à l'aide des fonctions suivantes :

- *capsys_init* : Lance l'initialisation normale de Capsule, qui crée la structure centrale et initialise un pthread par thread matériel. Lorsque Capsule est initialisé ainsi, il n'y a pas de callback de destruction mis en place, c'est donc à l'utilisateur de s'en charger lui-même.
- *capsys_warmup* : Cette initialisation est principalement destinée aux benchmarks. Elle lance l'initialisation de Capsule, met en place le callback de destruction mais attend également que les threads soient réellement prêts. En effet, après l'initialisation de Capsule, les threads doivent être ordonnancés au moins une fois par le système afin qu'ils fassent leur propre initialisation et se mettent en attente. Cela initialise leur pile et évite donc des effets souvent constatés lors de la première division, comme par exemple la pile allouée de manière paresseuse par le système et donc générant une faute de page au premier appel. De plus, si Capsule est lié dynamiquement à l'exécutable, les fonctions ne sont pas chargées

en mémoire au premier appel, ce qui induit également un surcoût initial.
Dans un programme réel, cette initialisation ne se produit qu'une seule fois à la première division, ce qui ne pose en pratique aucun problème ou risque pour la performance.
- *capsys_destroy* : Détruit et nettoie Capsule. Cela demande aux threads alloués à l'initialisation de s'arrêter. Une fois cela fait, les structures restantes sont désallouées. Une nouvelle tentative de division après cela relance le processus d'initialisation complet de Capsule.

Division conditionnelle

Les fonctions de gestion de la division sont décrites en détail dans la section 5.1.1.
- *capsys_probe* : Demande à créer un nouveau contexte, qui exécute la fonction donnée en paramètre. Si la demande est accordée, renvoie un pointeur vers le contexte, sinon, si Capsule estime que ce n'est pas intéressant, un pointeur *NULL* est renvoyé.
- *capsys_divide* : Lance l'exécution effective d'un contexte alloué précédemment via *capsys_probe*. En plus du contexte spécifié en premier argument, le deuxième argument sera transmis à la fonction spécifiée lors de la demande de création du contexte.

Synchronisation

Le code exécuté dans un contexte correspondant souvent à une portion précise de travail à effectuer, il est généralement intéressant d'attendre que certains contextes aient fini leur travail. Ces deux fonctions de synchronisation, permettant de gérer les groupes, sont décrites en détail dans la section 5.1.3.
- *cap_split* : Création d'un nouveau groupe, enfant du groupe courant. Le contexte courant est réassigné à ce nouveau groupe ; toutes les divisions suivantes assigneront leur contexte à ce groupe.
- *cap_join* : Attend la fin du groupe courant. Une fois que tous les enfants associés ont terminé, le groupe est détruit et le contexte courant réassigné au groupe parent.

Divers

De nombreuses fonctions existent afin d'aider à la gestion de Capsule, notamment pour récupérer des statistiques ou pour forcer certains comportements précis.
- *capsys_get* : renvoie le pointeur vers la structure *capsys_t* principale de Capsule. Si Capsule n'est pas initialisé à ce moment-là, *capsys_init* est automatiquement appelée, et un callback de destruction est mis en place. Du point de vue de l'utilisateur, cette structure est utile pour récupérer des statistiques telles que le nombre de divisions effectives.
- *capsys_current* : renvoie un pointeur vers le contexte courant. Cela permet principalement de récupérer l'ID unique du contexte ainsi que son parent.
- *capsys_block* : permet d'empêcher toute future division. Utile notamment pour les benchmarks, pour étudier le comportement mono-processeur.
- *capsys_unblock* : débloque les divisions après un appel à *capsys_block*.

Variables d'environnement

Lors de son initialisation, Capsule vérifie quelques variables d'environnement, ce qui permet d'influencer sur le comportement d'un programme déjà compilé.
- CAP_VERBOSE : Active des affichages de la part de Caspule, tels que le nombre de threads matériels détectés. N'implique aucun surcoût en fonctionnement.

- CAP_MAX_THREADS : Permet d'influencer le nombre de threads matériels manipulés. Si le nombre N spécifié est inférieur au nombre réel de coeurs, seul les N premiers seront utilisés. Le cas où N est supérieur au nombre de coeurs disponibles n'est utile que pour des tests. En effet, dans ce cas, plusieurs threads Capsule sont alloués par coeur. Cependant, lors d'une division conditionnelle, Capsule ne cherche pas à éviter d'utiliser un coeur, même si d'autre coeurs sont disponibles. Par exemple, si on demande 4 threads sur un bi-core, et qu'il n'y a que 2 contextes en cours d'exécution, il est possible que Capsule ordonnance les deux threads utilisés par les contextes sur le même coeur.
- CAP_MAX_DIVS : Limite le nombre de divisions possibles. Une fois ce nombre atteint, toutes les divisions suivantes sont refusées. Cela permet de tester un partitionnement pseudo-statique des données, en se basant sur les premières divisions.
- CAP_BUSYWAIT : Dans le cas où l'*attente intelligente* (voir section 5.3.4) est utilisée, cela permet de régler le nombre de cycles d'attente active, avant que les threads s'endorment par désordonnancement.
- CAP_PIN_THREADS : Par défaut, Capsule assigne, via un appel système, chaque thread qu'il crée à un coeur du système, empêchant le système d'exploitation de les déplacer s'il estime cela nécessaire. Cette variable permet de désactiver ce comportement et de laisser ainsi le système libre pour le mapping des threads.

5.3 Performance

5.3.1 Coûts d'une division

Une division en Capsule se sépare en plusieurs opérations. La figure 5.7 représente les principales métriques de coût notables.

Notons qu'il s'agit du cas d'une division réussie. Le cas d'une division échouée est beaucoup simple : il suffit de compter combien a coûté l'appel à probe.

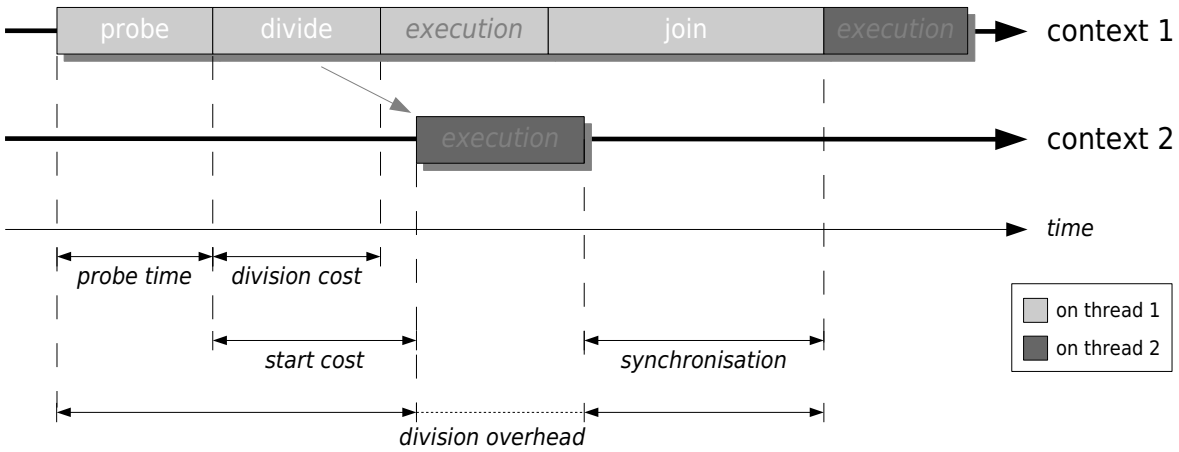


Figure. 5.7 – Les différentes métriques de coût lors d'une division réussie. L'échelle relative entre les coûts n'est pas respectée.

Le scénario est un contexte A qui crée un autre contexte B à l'aide d'un probe/divide. Ensuite, A attend que B ait fini son exécution. On peut distinguer les principales métriques suivantes :

« probe time » Correspond au temps passé lors d'un appel à *capsys_probe* qui renvoie effectivement un contexte. En pratique, ce temps correspond au temps passé à :
- vérifier le prédicteur (voir section 5.3.2) ;
- choisir le premier thread de la liste de ceux disponibles ;
- allouer la mémoire d'une nouvelle structure de contexte.

« division cost » Temps passé dans la fonction *capsys_divide* qui se contente d'activer le thread réservé lors de l'appel à *capsys_probe*.

« start cost » Temps d'activation d'un contexte une fois réservé. En pratique, cela correspond au temps de réveil d'un thread. Ce temps est principalement tributaire de l'ordonnanceur ainsi que de la méthode utilisée pour s'endormir (voir 5.3.4).

« synchronisation » Temps mis pour faire reprendre l'exécution du contexte A à partir du moment où le contexte B a terminé. En pratique, cela correspond à la réactivation du contexte A sur le deuxième thread via les primitives UNIX.

« division overhead » Cela correspond approximativement à la somme des durées « probe time », « start cost » et « synchronisation ». L'idée sous-jacente est d'avoir un ordre de grandeur du seuil à partir duquel une division est intéressante. Supposons une durée de travail t. Si on suppose que la division de travail se fait de manière parfaite, chaque contexte aura une durée de calcul de $t/2$. Si le surcoût de division est nul, alors le temps total d'exécution dans le cas où la division est réussie est égal à $t/2$; sinon, il sera de $t/2 + T$, où T est le « division overhead ». Cette mesure est approximative dans le sens où la division n'est pas instantanée, et que donc il est difficile d'estimer le temps passé dans chaque thread. En pratique, le coût dans le contexte A de *capsys_divide* est faible, l'imprécision est donc limitée.

Le tableau 5.1 donne un ordre de grandeur des différents coûts, en cycles, sur un processeur Intel double coeur.

Mesure	Durée en cycles
« probe time »	1150
« division cost »	10
« start cost »	480
« synchronisation »	3450
« division overhead »	5080

Table. 5.1 – Coûts comparés de division. Les mesures ont été effectuées sur un Core 2 Duo ; les chiffres précis sont relativement instables et varient selon les modèles de processeurs.

La plupart de ces coûts ne peuvent être mesurés plusieurs fois successivement au sein d'une boucle, puisque les morceaux de code concernés ne sont pas isolables. La mesure se base donc sur les résultats renvoyés par l'instruction assembleur *rdtsc*, qui renvoie le nombre de cycles écoulés depuis l'initialisation du processeur. Notons que sur processeur Intel, ce compteur est synchronisé entre les coeurs, contrairement à plusieurs générations de processeurs AMD bi-coeurs. Cela permet de mesurer notamment le « start cost », puisqu'il s'agit de faire une différence entre des mesures prises dans deux threads différents.

5.3.2 Prédiction de ressources

Contraintes

Probes	140939
Divisions réussies	3987
Divisions échouées	136952
Taux de réussite	2.8%

Table. 5.2 – Statistiques de division pour des quicksorts sur 50 listes de 1000000 d'éléments, testés sur une machine à 4 coeurs.

Le tableau 5.2 donne des statistiques de division pour un quicksort simple. On constate que la majorité des probes échouent. La politique de base de Capsule est de réserver un thread lors d'un probe s'il y en a un de disponible ; donc, lorsque toutes les ressources sont occupées, le probe est refusé (des politiques plus évoluées rendent cela un peu moins systématique mais la tendance reste la même). Dès lors, avoir beaucoup de divisions refusées est le signe d'une application exploitant efficacement les ressources de calcul disponibles, ce qui est l'objectif.

Ainsi, le surcoût le plus important à prendre en compte est le coût du probe échoué ; les autres coûts laissent plus de marge de manoeuvre à l'implémentation quant aux coûts des opérations effectuées.

Dans le cas d'une boucle simple, dont chaque itération peut être exécutée en parallèle, un probe est effectué à chaque itération. En supposant que la majorité des probes échoue, il est nécessaire que la durée d'un probe échoué soit faible voire négligeable devant la durée d'une itération du corps de la boucle.

Un probe doit vérifier s'il est intéressant de se diviser, vérifier qu'un thread est disponible et, le cas échéant, le réserver. Une vérification explicite de la disponibilité des threads est coûteuse puisqu'il est nécessaire de se synchroniser avec les autres coeurs d'exécution afin d'éviter un conflit d'accès.

Approche prédictive

L'implémentation actuelle de Capsule utilise une liste centralisée des threads disponibles (voir section 5.2.1). Si l'accès à cette liste pour vérifier la disponibilité d'un thread se fait sans précaution, une race-condition peut se produire, comme représenté en figure 5.8 : un premier contexte vérifie qu'un thread est disponible ; mais avant que ce premier contexte ne retire le thread de la liste, un autre contexte vérifie également la présence d'un thread et trouve le même thread. Ensuite, le premier contexte retire le thread de la liste des threads disponibles, et ainsi, le deuxième contexte essaye de réserver un thread déjà occupé.

Pour éviter ce cas classique de race-condition, cette liste centrale de threads disponibles a un lock associé. Cela signifie donc que pour faire une réservation propre de thread, il est nécessaire de prendre ce lock, vérifier l'état et relâcher le lock. En pratique, prendre un lock est coûteux puisque cela nécessite notamment des barrières mémoires, risquant d'impacter la performance des coeurs. Or, un des objectifs de Capsule est de pouvoir utiliser des probes le plus souvent possible.

Pour alléger le coup moyen d'un probe, Capsule utilise un prédicteur logiciel. Cela se base sur les hypothèses suivantes :

- le programme effectue des probes fréquemment ;
- le coût d'un probe échoué est critique ;

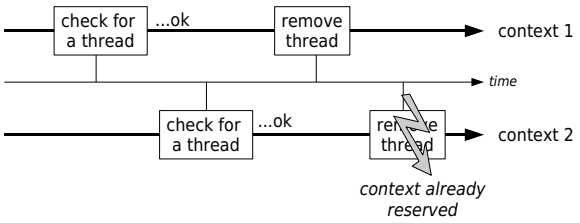


Figure. 5.8 – Race-condition sur la réservation d'un thread. Les deux contextes voient un thread disponible, mais quand le deuxième thread essaye de l'enlever de la liste, il a déjà été pris par le premier contexte.

– le nombre de probes réussis est négligeable devant le nombre de probe échoués.

Le prédicteur logiciel prend la forme d'une simple variable globale, nommée « hint » dont la valeur indique à un moment donné le nombre *approximatif* de threads disponibles. Le prédicteur est mis à jour lors des événements liés aux contextes : quand un contexte est crée et associé à un thread, le prédicteur est décrémenté ; quand un contexte termine et libère un thread, le prédicteur est incrémenté.

Le comportement d'un probe devient celui indiqué en figure 5.9. Lorsque l'utilisateur effectue un probe, Capsule commence par vérifier l'état de ce prédicteur, sans prendre *aucun* lock. Si le prédicteur est négatif ou nul, alors Capsule considère qu'aucun thread n'est disponible et la demande est directement rejetée. Si le prédicteur est strictement positif, alors Capsule vérifie proprement si un thread est *réellement* disponible, en prenant le lock sur la liste de thread et répond en fonction de l'état des threads.

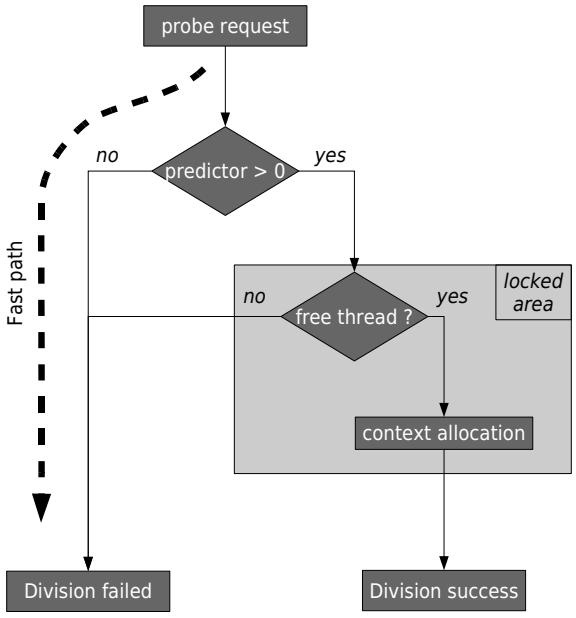


Figure. 5.9 – Implémentation du probe.

Puisque que, lorsque le prédicteur est lu, il n'y a pas de lock pris, sa valeur peut être fausse

et ne pas correspondre à la réalité. En pratique, pour les quantités de code impliquées dans la vérification de thread libre, cette race-condition arrive rarement. La mise à jour du prédicteur est faite le plus tôt possible après la réservation d'un thread (avant même l'allocation du contexte) et le plus tard possible après la fin d'un contexte. Les zones de code Capsule où le prédicteur est incohérent sont donc réduites à quelques dizaines d'instructions assembleur, ce qui est peu comparativement à la durée de vie des contextes.

Bien sûr, les erreurs de prédictions arrivent quand même de temps en temps. Il y a deux cas d'erreur possibles : soit le prédicteur était optimiste, soit il était pessimiste.

Dans le premier cas, cela veut dire que le prédicteur prévoyait un thread ; Capsule passe donc par le chemin avec lock et vérifie à ce moment la disponibilité d'un thread, ce qui échoue. Le seul problème dans ce cas est un surcoût puisque le lock a dû être pris ; cependant, il s'agit bien d'une erreur de prédiction qui doit donc rester rare. Le surcoût généré en pratique sera statistiquement faible.

Dans le deuxième cas, cela implique qu'une division est directement refusée alors qu'un thread est disponible. Comme on suppose qu'un programme Capsule effectue des probes régulièrement, même si on manque un probe alors qu'un thread est disponible, le suivant sera probablement réussi.

Comme dans les deux cas, on peut considérer que le taux d'erreur du prédicteur est faible, alors l'approximation faite par ce prédicteur non-locké est valable.

Au final, cela signifie qu'un probe échoué sans lock - le « fastpath » de la figure 5.9 - qui revient à un simple test de variable, est le cas de loin le plus courant. Le probe moyen en Capsule revient donc à un test de variable simple.

Résultats

Le code assembleur correspondant au cas du probe échoué est montré sur le listing 5.2. La partie correspondant à l'allocation complète quand le prédicteur prévoit la disponibilité d'un thread n'est pas représentée. On voit que le cas où le prédicteur suppose qu'aucun thread n'est disponible revient à 3 instructions assembleur, sans nécessité de synchronisation mémoire. Ces instructions correspondent à la manipulation de la variable globale *capsule_hint* stockant l'état du prédicteur. Elle est d'abord chargée dans un registre, puis sa valeur est testée et ensuite un saut conditionnel est effectué en fonction du résultat du test.

Listing 5.2 – Code assembleur x86 généré pour le probe. Seul le chemin correspondant au probe échoué est montré.

```
1 movl    capsule_hint(%rip), %eax
2 testl   %eax, %eax
3 jle     .L7
4 ...     ;division code
5 .L7:
6 ...     ;normal continuation of code block
```

Le tableau 5.3 donne des résultats de performances entre une version simple de la division conditionnelle qui utilise le lock systématiquement et une version utilisant le prédicteur.

Le test de durée d'un probe est une simple boucle qui essaye d'initier une division à chaque itération ; il est mesuré la différence de temps d'exécution avec la même boucle tournant à vide. Les divisions sont bloquées à l'aide de la fonction *capsys_block*. Cette fonction a pour effet de mettre à zéro le prédicteur, ce qui a donc un effet similaire au cas où les ressources sont toutes occupées. La différence de gain entre le test simple et quicksort donne la tendance : moins on

	Durée d'un probe échoué	QuickSort
Sans prédicteur	121 cycles	194416089 cycles
Avec prédicteur	1 cycle	141740691 cycles
Speedup	121	1.37

Table. 5.3 – Gain du prédicteur logiciel, sur un Core 2 duo (2 coeurs). Le quicksort est effectué sur des listes de 1000000 d'éléments.

fait d'essais de divisions, moins le gain du prédicteur est signifiant (et à l'inverse, plus le fait de rater une division à cause d'une prédiction échouée est important). Cependant, on voit que le coût en pratique d'un probe est suffisamment léger grâce au prédicteur pour pouvoir être inséré très fréquemment.

Notons que le probe n'étant au final qu'un test simple de variable global, son coût réel est très dépendant de l'état du processeur à ce moment-là et donc de son contexte. L'état des prédicteurs, du pipeline, des caches peuvent notamment influencer.

5.3.3 Simplification des groupes

La structure *cap_group_t* maintient les informations liées à un groupe. Elle contient notamment :

- le nombre de groupes/contextes que contient le groupe ;
- un pointeur vers le groupe parent ;
- un pointeur vers un contexte qui attendrait la fin de ce groupe, et, le cas échéant, des informations sur comment continuer l'exécution une fois le groupe fini.

Il n'y a pas de liste explicite des groupes et des contextes parents. En effet, puisque Capsule contrôle la vie des groupes et contextes, et que chacun ne peut avoir qu'un seul parent, un simple compteur de référence suffit pour savoir si un groupe est terminé.

Supposons avoir un groupe G dont le parent est le groupe $P(G)$. À chaque fois qu'un élément E (groupe ou contexte) appartenant à ce groupe G se termine, le compteur de références de G est décrémenté. S'il est non-nul, le groupe G contient toujours des éléments et il n'y a rien à faire. Dans le cas contraire, cela veut dire que le groupe G est terminé. Deux cas existent alors :

- Si aucun contexte n'attendait la fin du groupe G, alors le compteur de $P(G)$ est lui même décrémenté. Le fonctionnement est alors récursif.
- Si un contexte W attendait la fin de G, alors ce contexte, qui par construction appartenait lui-même à G, est réassigné au groupe $P(G)$ et réveillé. Le groupe $P(G)$ troque donc G contre W, son compteur de référence n'est pas modifié.

Notons que dans les deux cas, au maximum un contexte peut être réveillé. Un groupe ne pouvant terminer de sa propre initiative, un contexte vient nécessairement de terminer ; cela implique qu'un thread est disponible. Le contexte réveillé peut donc être assigné immédiatement à un thread et continuer directement son exécution.

Une des conséquences est qu'un contexte peut changer de thread au cours de son exécution. En théorie, cela ne pose pas de problème ; il faut néammoins que l'utilisateur soit attentif à cela, puisque référencer le thread courant peut servir notamment pour le débuggage. D'un point de vue système, Capsule, lors de sa terminaison, force le contexte initial à reprendre son exécution sur le thread initial, afin d'éviter d'introduire des cas non prévus par le code extérieur.

5.3.4 Problèmes d'ordonnancement

Problème du réveil des threads

Les threads étant pré-alloués, ils sont par moment inutilisés et doivent donc attendre d'être sélectionnés via un probe. Le temps que met un contexte à se réveiller, entre le moment où du travail est disponible et le moment où le traitement commence effectivement, est une métrique importante pour la performance de Capsule. En effet, une division est intéressante seulement si le temps de calcul effectif est nettement plus élevé que le temps de réveil.

Lorsque l'on regarde le problème du réveil d'un contexte utilisateur, deux situations se présentent :

- les contextes sont exécutés dans le même thread système ;
- les contextes sont assignés à des threads systèmes différents.

Les systèmes de threading en mode utilisateur comme *GNU Pth* [32] sont généralement dans le premier cas. Lorsqu'un thread doit attendre, que ce soit pour un verrou ou pour autre chose, la bibliothèque de threading utilisateur continue à exécuter un autre thread utilisateur dans le même thread système, réduisant ainsi drastiquement les problèmes de latence, puisqu'il n'y a pas d'aller/retour avec le noyau à effectuer. Cependant, cela n'est possible qu'au sein d'un même thread système, empêchant de facto de faire cela entre processeurs, le parallélisme matériel n'est pas exploité.

Dans Capsule, par construction, on assigne un seul contexte vivant à un thread ; le premier cas n'arrive donc jamais et on devient ainsi tributaire de l'ordonnanceur du système. De plus, cela signifie que le thread auquel le contexte a été assigné ne faisait rien, puisque sinon il n'aurait pu être choisi pour ce contexte. Donc, activer un nouveau contexte nécessite de réveiller un thread système.

Vis-à-vis du système, il y a deux principaux choix pour endormir et réveiller un thread :

- faire de l'attente active : vérifier en permanence l'état d'une variable et, lorsqu'une valeur donnée est atteinte, sortir de la boucle ;
- désordonnancer le thread : le système ne mettra plus le thread dans sa liste de threads vivants, un appel système permettra de le réveiller.

Attente active L'attente active est l'approche la plus simple : le thread vérifie en boucle l'état d'une variable, et lorsque la condition requise est vérifiée, continue son exécution normalement. La latence de réveil est minimisée : elle est uniquement due à la propagation du signal d'un thread à l'autre. Dans le cas de Capsule, où il y a un thread par processeur, la latence est due au passage d'un cache de processeur à l'autre.

Mais à l'inverse, cela implique de vérifier en permanence une condition ; le système d'exploitation doit continuer à ordonnancer le thread. Si le thread est tout seul sur le processeur, cela empêche d'endormir le processeur et donc implique une plus forte consommation électrique et une surchauffe inutile ; ce phénomène est particulièrement néfaste et à éviter sur les systèmes embarqués.

De plus, tout le temps où le thread est ordonnancé à vérifier en boucle la variable, le processeur n'est pas assigné à d'autres processus, provoquant ainsi une perte de performance pour les autres processus ou encore une perte d'équité. D'un point de vue macroscopique, cela a pour effet d'augmenter la latence générale moyenne du système ; on est donc perdant dès que l'on considère autre chose que son propre processus.

Désordonnancement Le désordonnancement laisse au système sous jacent (système d'exploitation ou bibliothèque de threads en mode utilisateur) le soin d'arrêter et de relancer le thread. En pratique, cela signifie que le système a la possibilité d'ordonnancer d'autres threads et processus, ou, le cas échéant, de mettre en veille le processeur et ainsi d'économiser de l'énergie

et de réduire la surchauffe. D'un point de vue global, c'est l'approche la plus juste, puisqu'on laisse la main au système.

Cependant, la latence de réveil risque d'être augmentée : lorsqu'il est décidé de réveiller un thread, le système est notifié et, de là, tout dépend de lui. Pour que le travail du thread à réveiller commence, il doit être ordonnancé ; selon l'implémentation de l'ordonnanceur système, cela peut notamment signifier que le processus courant finisse sa période d'ordonnancement, ce qui provoque ainsi une augmentation du temps de réveil.

Le tableau 5.4 donne les différentes mesures de latence lors d'un probe pour l'endormissement basé sur l'ordonnancement. Les chiffres pour l'attente active sont ceux donnés dans le tableau 5.1 comme référence.

Durée	Attente active (cycles)	Désordonnancement (cycles)
« probe time »	1150	1150
« division cost »	10	3400
« start cost »	480	7300
« synchronisation »	3450	4500
« division overhead »	5080	~12900

Table. 5.4 – Comparaison entre l'attente active et le désordonnancement. Les mesures ont été effectuées sur un Core 2 Duo ; les chiffres ne sont qu'un ordre de grandeur, les résultats étant relativement instables selon les exécutions et processeurs.

On voit que l'introduction du désordonnancement par rapport à de l'attente active multiplie par un peu plus de deux le coût d'une division sur un micro-benchmark. Il faut cependant bien garder en tête que les divisions effectives sont relativement rares et que donc l'impact sera plus limité sur un code complet.

Attente intelligente Pour un cycle d'endormissement/réveil donné, on peut considérer que la latence induite est à peu près constante quel que soit le temps passé endormi [2]. Ainsi, si un thread s'endort pour longtemps, l'impact de la latence due au désordonnancement devient négligeable par rapport au cas de l'attente active.

Bien sûr, il n'est pas aisé de deviner le temps que passera un thread à rien faire. Capsule introduit donc une notion « d'attente intelligente » : quand un thread doit être endormi, il commence par une attente active ; au bout d'un certain temps, s'il n'a pas été réveillé, il s'endort complètement en se faisant désordonnancer.

Dans les zones critiques d'un programme utilisant Capsule, on peut considérer que les probes sont très fréquents, ne laissant les threads endormis que très peu de temps.

De cette manière, l'attente intelligente permet d'obtenir les avantages des deux approches d'endormissement. D'une part, on a une latence de réveil faible quand cela est important et d'autre part, la consommation électrique et l'équité d'ordonnancement sont quand même conservées la plupart du temps.

Une approche possible pour éviter le choix d'une durée arbitraire d'attente active est de s'intégrer à l'ordonnanceur du noyau. Lorsqu'un thread doit s'endormir, il active un indicateur et se met en attente active. Dès que l'ordonnanceur du système le désordonnance au bout de son quantum [3] de temps, alors il est possible de détecter l'indicateur d'endormissement et ainsi

[2] la latence peut en pratique varier si, du fait de la longue période d'endormissement, l'ordonnanceur réduit la priorité du thread ; dans le cas présent, cela ne change guère le problème.

[3] Le quantum de temps correspond au temps maximum pendant lequel le noyau laisse un thread/processus s'exécuter avant de donner la main à un autre processus.

d'éviter de le ré-ordonnancer inutilement.

Comparaison des méthodes d'attente sur système chargé La figure 5.10 compare les 3 types d'attentes sur plusieurs exécutions de quicksort, en faisant varier le « cutoff », qui correspond au temps maximum passé en attente active dans le cas de l'attente intelligente. Le système est chargé avec 8 processus indépendants, répartis équitablement entre les processeurs ; ces processus se contentent d'exécuter une boucle vide en continu, ce qui impacte uniquement l'ordonnanceur mais pas les entrées/sorties ou la mémoire.

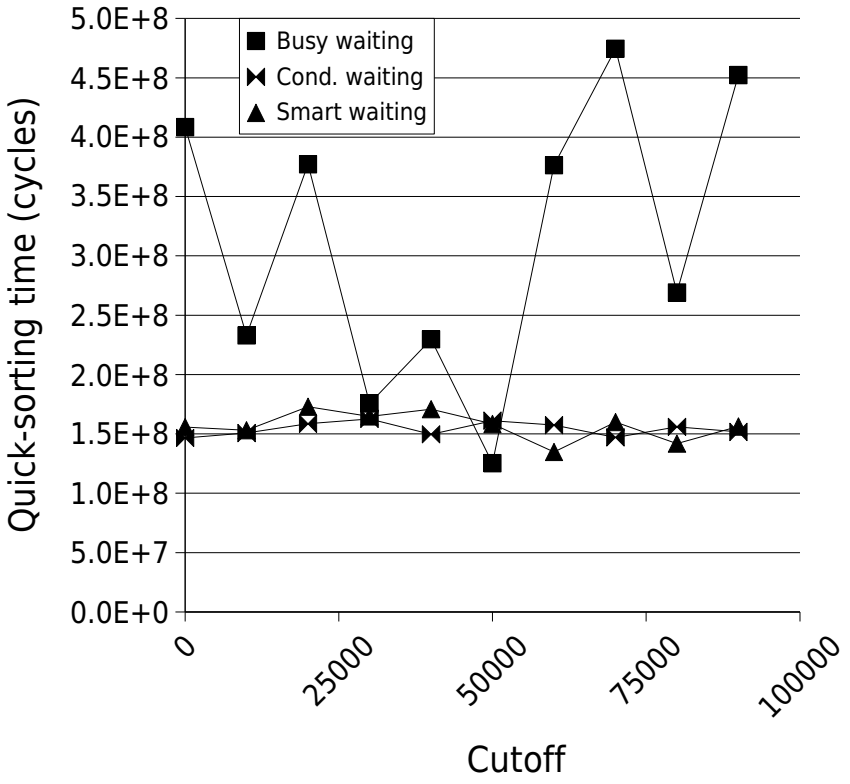


Figure. 5.10 – Comparaison des différentes méthodes d'attente sur un système à 4 coeurs, chacun ayant deux processus tournant à 100%. L'ordonnée représente le temps d'exécution moyen du quicksort sur 16 listes différentes de 1000000 d'éléments. Le « cutoff » correspond au temps en cycles que l'attente intelligente attend en attente active avant de se désordonnancer. Les exécutions pour l'attente active («busy waiting») et le désordonnancement («cond. waiting») ne sont pas dépendantes de cette valeur.

La première chose à remarquer est la grande instabilité des résultats. En effet, seule la version en « attente intelligente » devrait changer de comportement en fonction du « cutoff », les autres versions n'étant pas affectées par ce paramètre. Le système étant lourdement chargé, Capsule est très dépendant de l'ordonnanceur système dans les 3 cas. Comme quicksort est irrégulier dans son équilibrage du travail, cela s'en ressent directement sur la stabilité lorsque que des phénomènes extérieurs entrent en jeu.

On voit que la version « attente active » est nettement plus instable que les deux autres. Cela est dû au fait que l'ordonnanceur, dans ce cas, ne peut pas détecter quand les threads doivent recommencer réellement leur travail. Il introduit donc des périodes de coupure apparaissant aléatoires, le temps d'ordonnancer les autres processus du système. Dans les deux autres cas, lors des longs endormissements qui risquent d'être influencés par l'ordonnanceur, Capsule signale au

système quand il faut réveiller un thread donné. L'ordonnanceur a alors tendance à l'ordonnancer au bout d'un temps relativement stable, puisque, n'ayant pas été ordonnancé depuis une certaine période de temps, sa priorité est à priori suffisante.

Enfin, cette expérience montre qu'en dépit des résultats des micro-benchmarks mesurant les latences des différentes opérations de Capsule, il semble de toute façons préférable de s'en remettre au système (pour l'attente) dans le cas d'un système réel.

Ordonnancement par groupe

Comme vu dans la partie précédente, le temps de réveil d'un thread dépend de la manière dont le système ordonnance les threads du processus. Ainsi, si le système essaye d'ordonnancer simultanément (par construction, Capsule n'a qu'un seul thread par processeur) les threads du processus, alors il est probablement possible de réduire les latences de réveil.

Dans le cas d'une charge du système faible, la différence est imperceptible. Cependant, dans le cas d'un système chargé, le temps entre l'ordonnancement des threads peut rapidement augmenter.

À l'inverse, pour l'exécution du programme à proprement parler, il n'est pas forcément intéressant que les threads soient ordonnancés simultanément. En effet, si la contention inter-threads est forte, alors un ordonnancement désynchronisé entre threads aura tendance à masquer ce phénomène.

5.3.5 Piles et contextes UNIX

Problème de réutilisation des piles

Afin qu'un contexte puisse s'exécuter correctement, il doit avoir à sa disposition une pile avec de l'espace libre. Il s'agit d'une pile classique, permettant de stocker les variables locales des fonctions ainsi que leurs arguments.

Chaque thread de Capsule a nécessairement une pile, puisse qu'il exécute lui-même du code de gestion de contexte, même lorsqu'aucun contexte ne lui est assigné. Par défaut, chacun de ces threads a une pile, créée normalement lors de l'appel de *pthread_create*.

Lorsqu'une synchronisation est effectuée, le contexte appelant est mis en pause, mais doit continuer son exécution lorsque les conditions requises pour son réveil sont satisfaites. La mise en pause / synchronisation est due à un appel de fonction standard (vu de l'utilisateur) à *cap_join*. Lorsque l'exécution reprend, le programme doit donc avoir toujours accès à une pile et notamment aux valeurs qui étaient stockées sur la pile *avant* l'appel à *cap_join*, ainsi qu'à tout l'arbre d'appel. Cela signifie donc que, lorsque l'exécution d'un contexte en synchronisation reprend, il doit avoir à nouveau accès à la même pile ou que les données de son ancienne pile soient présentes dans la nouvelle pile.

Dans le cas où un seul groupe existe, cela ne pose pas de problème puisque cela signifie également qu'une synchronisation nécessite que tous les contextes aient fini leur travail et que donc le thread contenant la pile sur laquelle continuer l'exécution est nécessairement disponible.

Ce cas est illustré sur la figure 5.11. Dans cet exemple, il est représenté l'évolution de quelques contextes avec leur pile sur un bi-processeur, appartenant tous au même groupe. Le contexte initial, $C1$, crée un contexte $C2$, qui est affecté au deuxième processeur. Le contexte $C1$ finit son travail et attend donc la fin des autres contextes. À ce moment, puisqu'un processeur redevient disponible, une division réussit dans $C2$, ce qui crée un autre contexte sur le premier processeur. La pile du premier thread est dans ce cas réutilisée. Mais comme le contexte $C3$ fait partie du groupe, $C1$ ne doit redémarrer qu'après que $C3$ ait fini, libérant ainsi la pile au dessus de $C1$.

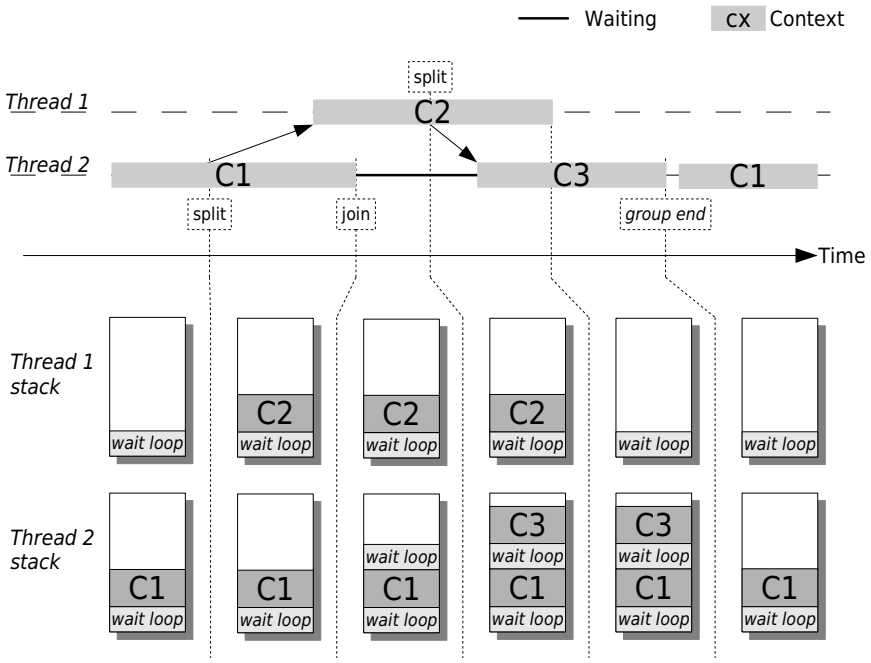


Figure. 5.11 – Évolution des piles dans le cas du groupe unique. Bien que le thread soit réutilisé, le problème de pile ne se pose pas. Les éléments « wait loop » correspondent à la fonction d'attente de contexte, lancée dès qu'un thread est en attente.

Dans le cas où il existe plusieurs groupes, un thread ayant initié une synchronisation peut être réutilisé avant la fin de celui-ci par un autre groupe. Ce cas est représenté sur la figure 5.12. On voit que, lorsque le groupe gris clair finit, ce qui correspond à la fin de $C2$, la pile contenant l'état de $C1$ n'est pas disponible puisque utilisée par le contexte $C3$.

Néanmoins, au moment de réveiller $C1$, un thread est disponible : celui qui exécutait le dernier contexte vivant du groupe. Cette propriété est garantie actuellement par Capsule (voir section 5.3.3). Le problème n'est donc pas lié à la disponibilité d'un thread, mais à la garantie de la disponibilité de la pile au moment de la fin de synchronisation par *cap_join*.

On le voit clairement sur la figure, deux approches sont envisageables :

- Recopier le morceau de pile correspondant à $C1$ de la pile du thread 2 dans la pile du thread 1. On ne peut pas « enlever » l'ancien morceau simplement, puisque cela modifierait l'adressage de la pile de $C3$; on aura donc une perte de mémoire qui risque de grossir. De plus, d'un point de vue strictement technique, il est difficile en C de maîtriser les limites de pile d'une fonction donnée afin de la déplacer, et ce particulièrement de manière portable.
- Se débrouiller pour éviter qu'une pile correspondant à un contexte en attente de synchronisation soit réutilisée. Cela nécessite donc d'allouer des nouvelles piles lorsque nécessaire.

Implémentation des piles dans Capsule

Capsule alloue une nouvelle pile lorsqu'un thread se met en attente de synchronisation.

La figure 5.13 montre l'évolution des piles selon cette approche. Le scénario est le même que celui de la figure 5.12, mais ici on représente selon les stacks et non selon les threads ; les lignes libellées « thread x » représentent quelle pile utilise chaque thread à un moment donné.

Au moment où l'utilisateur effectue un *cap_join*, une nouvelle pile est allouée (stack 3) et la

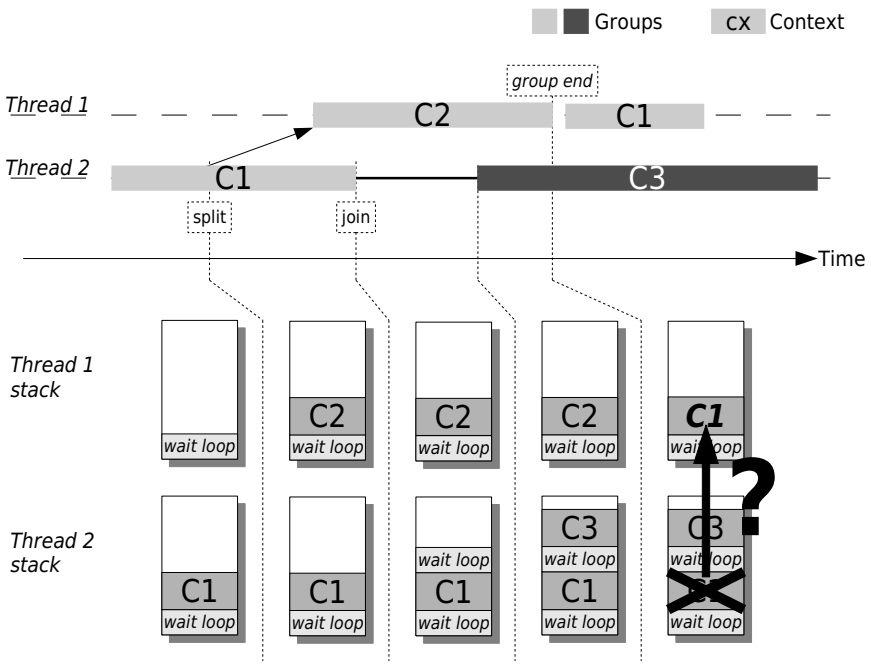


Figure. 5.12 – Évolution des piles avec plusieurs groupes. Lorsque que le groupe gris clair termine, il ne peut réutiliser le thread qui a initié la synchronisation, sa pile n'est plus accessible en écriture à cause du contexte $C3$.

fonction d'attente de thread standard est lancée dessus. Ainsi, au moment où le dernier contexte $C2$ du groupe gris clair a fini son travail, le thread 1 n'a qu'à continuer son exécution sur la pile (stack 2) où le join a été fait. Les piles sont décorellées des threads.

En pratique, cela revient à avoir une pile pour chaque contexte, mais en évitant d'essayer de ré-allouer une pile lorsque qu'un contexte se termine sans faire de *cap_join* (et qui donc ne bloque pas la pile courante).

L'exemple de la figure 5.13 montre une propriété de l'approche actuelle prise par Capsule. Le contexte $C1$, qui a commencé sur le thread 2, continue à la fin son exécution sur le thread 1. En pratique, sur du code propre, cela ne pose pas de problème. Cependant, afin que Capsule rende un environnement propre lorsqu'il termine, il s'assure que l'exécution, après un *capsys_destroy* continue bien sur le thread initial, qui existait avant l'initialisation de Capsule.

Néanmoins, cette migration de contexte d'un thread à l'autre peut poser problème dans le cas d'une mémoire distribuée, puisqu'il faut dans ce cas bouger les données du contexte, dont la pile. Mais il s'agit ici d'un détail d'implémentation propre à la version en mémoire partagée ; une version en mémoire distribuée aurait probablement un schéma de gestion des piles différent.

À cause de ce problème de pile, on se retrouve donc à allouer de temps en temps des nouvelles piles. L'allocation d'une pile est coûteux : il faut d'une part mettre en place l'allocation proprement dite mais il faut également que le système d'exploitation répercute cela sur son propre mapping mémoire, ce qui est fait à la première utilisation.

Capsule implémente un système de pool de piles simple. Chaque pile est conservée dans le pool. Lorsqu'une pile est nécessaire, Capsule essaye d'en prendre une du pool ; si aucune n'est disponible, une nouvelle pile est allouée. Chaque thread a un pool de piles différent, l'allocation

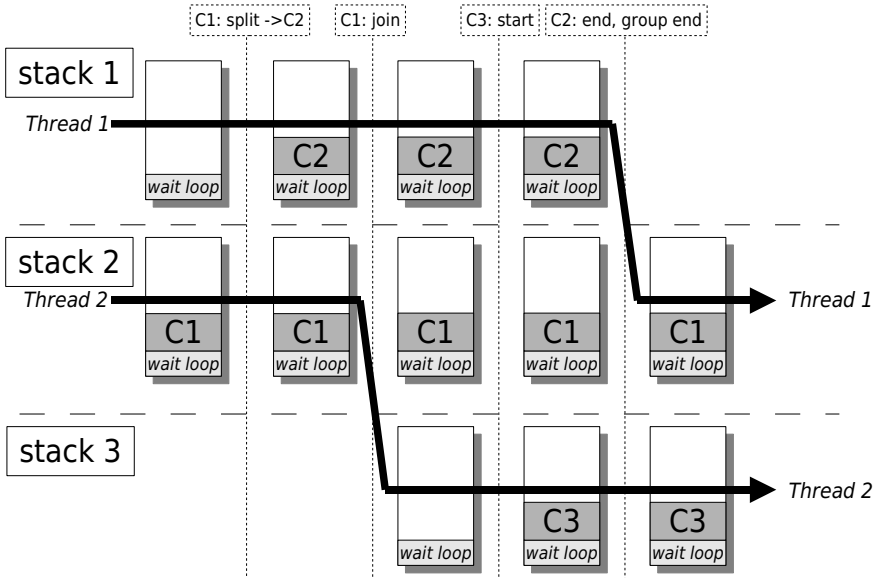


Figure. 5.13 – Évolution des piles dans Capsule. Le scénario est le même que dans la figure 5.12. Les piles sont décorrelées des threads, qui en changent quand nécessaire au moment des split/join.

se fait donc sans lock. Cela permet actuellement d'éviter des problèmes de contention mais à l'inverse risque d'augmenter la consommation mémoire. Bien qu'aucun problème lié à cet effet n'a été rencontré, il serait aisé de mettre en place une approche similaire à *malloc*, à savoir d'avoir des pools privés par threads, mais également un pool partagé (et avec lock).

La réutilisation de pile permet également de diminuer les effets liés au système, puisqu'une pile, une fois utilisée une première fois, a déjà été mappée par le système en mémoire.

Allocation ultra-paresseuse

Comme montré dans les exemples des figures 5.11 et 5.12, une pile peut être réutilisée sans problème pour un contexte à condition que ce contexte appartienne au même groupe que les contextes déjà présents dans la pile considérée. Cela permettrait dans de nombreux cas d'éviter une réallocation inutile de pile, rendant l'allocation encore plus paresseuse.

En pratique, cela nécessite de profondes modifications de la structure du code actuel de Capsule et notamment un changement de la logique de gestion des piles. Cela n'est donc pas implémenté actuellement.

L'idée sous-jacente serait de changer le moment où les nouvelles piles sont mises en place. Actuellement, cela se fait au moment d'un *capsys_join*, afin que le coût lié au changement soit masqué par le temps d'attente. Bien que cela permette de réduire les latences sur les micro-benchmarks de division, il paraît peu probable que cela soit critique en pratique, surtout si l'allocation ultra-paresseuse est utilisée. Ainsi, il serait possible de déporter l'allocation des piles au moment de la division ; le fait d'utiliser un pool évite de toutes façons que le surcoût soit trop important. Dès lors, il devient simple d'avoir l'allocation ultra-paresseuse : il suffit de vérifier si la pile courante du thread est utilisée par le groupe courant ou un de ses parents. Cela peut se faire en conservant une variable simple indiquant le groupe propriétaire de la pile.

5.3.6 Passage d'arguments

Encapsulation des arguments

La version C de Capsule utilise une fonction fournie par l'utilisateur afin de savoir quoi exécuter dans les nouveaux contextes. Capsule doit manipuler lui-même les arguments fournis à cette fonction, puisqu'ils doivent être passés d'un thread à l'autre : fournis à la fonction *capsys_divide*, ils sont transmis à la fonction spécifiée lors du *capsys_probe* et exécutés en parallèle, appelée en pratique par Capsule. Le langage C ne permet guère une manipulation efficace et propre d'un nombre variable d'arguments. Seul les *varargs* permettent cela (comme pour la fonction *printf* par exemple) mais ils nécessitent d'utiliser des fonctions coûteuses en temps pour les manipuler de manière portable.

Il a été fait le choix de n'avoir qu'un seul paramètre à la fonction principale d'un contexte, un *void**, cela de manière similaire à la bibliothèque pthread [4]. Comme il s'agit d'un pointeur non typé, il est possible de passer n'importe quelle structure et ce sans risque de problème d'aliasing.

En pratique, il est souvent nécessaire de passer plusieurs arguments. Par exemple, rien que dans le cas d'une boucle parallèle travaillant sur un tableau, il sera nécessaire de passer au moins un pointeur vers le tableau, ainsi que les bornes de travail demandées à la fonction.

Pour ce faire, il va donc falloir systématiquement passer par une structure encapsulant les arguments à fournir à la fonction, structure qui sera spécifique à chaque type de division du programme.

L'exemple de la section 5.1.2 utilise cette technique. L'exemple de code associé (listing 5.1) nécessite le code du listing 5.3 pour être complet. Ce morceau de code correspond à la partie spécifique (et ad-hoc) nécessaire à la gestion des paramètres.

Listing 5.3 – Gestion des paramètres pour l'exemple de la boucle.

```
1   typedef struct {
2       int *a, *b, *c;   //Source and target vectors
3       int min, max;  //loop boundaries
4   } looparg_t;
5
6   looparg_t* alloc_looparg( looparg_t* ref, int min, int max) {
7       looparg_t *lt;
8
9       lt = (looparg_t*) malloc(sizeof(*lt));
10      lt->a = ref->a;
11      lt->b = ref->b;
12      lt->c = ref->c;
13
14      lt->min = min;
15      lt->max = max;
16  }
```

La structure à la ligne 1 contient l'ensemble des paramètres nécessaires à la fonction de sommation de vecteur : les vecteurs d'entrée, le vecteur de sortie, les éléments sur lesquels la fonction doit travailler. On remarque que cela correspond à la notion de division Capsule, puisque les paramètres correspondent à la structure de données sur laquelle travailler, associés aux données d'une partition précise.

[4] La fonction spécifiée lors d'un appel à *pthread_create*, qui représente le code à exécuter en paralléle, doit prendre en argument un *void**.

La fonction à la ligne 6 est juste une aide, afin d'alléger et factoriser le code de préparation des arguments. Elle se contente d'allouer la structure puis de remplir les différents champs à l'aide des valeurs spécifiées. L'intérêt est de permettre de spécifier les arguments presque normalement sur une seule ligne et éviter une surcharge syntaxique trop lourde.

Problème de l'allocation mémoire

En dehors de la surcharge syntaxique, cette approche a un surcoût à l'exécution par rapport à un passage d'arguments pour un appel de fonction classique [5]. Il est nécessaire d'*allouer* de la mémoire, ici à l'aide de *malloc* à la ligne 9.

Un appel de fonction normal n'a pas ce problème puisque les paramètres sont mis directement sur la pile, qui est déjà allouée. Dans le cas de Capsule cela n'est pas possible directement, puisque ces paramètres sont destinés à une fonction qui sera exécutée dans un autre thread et donc manipulée par Capsule directement ; la pile de l'appelant (en réalité, de celui qui fait le *capsys_divide*) n'est pas la même.

Cependant, il s'avère que cela n'est pas très gênant en pratique. D'une part, cette allocation doit se faire uniquement lorsqu'un nouveau contexte est effectivement créé et exécuté. Or, cela arrive relativement rarement par rapport au nombre effectif de probe échoués, ce qui laisse déjà plus de marge de manoeuvre.

D'autre part, il s'agit généralement d'allocations de petites taille, ce qui est particulièrement bien optimisé par les algorithmes d'allocation de mémoire standard. Les algorithmes d'allocation, tels que celui de la glibc [38] ou tcmalloc [36], utilisent une liste des blocs pré-alloués par thread, sans nécessité de locks. Comme les allocations pour les paramètres sont généralement de petite taille, les blocs pré-alloués suffisent la majorité du temps, permettant ainsi une allocation légère et sans impact sur les autres threads.

Allocation directe par la pile

Une solution envisageable [6] est d'allouer la structure encapsulant les arguments de la fonction sur la pile du *nouveau* contexte. Bien que la surcharge syntaxique reste identique, cela éliminerait l'appel à *malloc* supplémentaire, puisque l'allocation de la pile pour le nouveau contexte est déjà gérée par ailleurs [7].

Cette solution impose de modifier l'API de Capsule, puisque, pour l'instant, la gestion des paramètres est entièrement laissée à l'initiative du programmeur. Dans le cas où l'allocation de la structure passerait par la pile du nouveau contexte, il faudrait que l'utilisateur puisse spécifier la quantité de mémoire dont il a besoin pour allouer sa structure d'une part et d'autre part récupérer le pointeur.

En pratique il y a deux choix pour permettre cela sans modifier le langage : ajouter une fonction ou modifier les appels *capsys_probe* / *capsys_divide*.

Dans le premier cas, décrit dans le listing 5.4, la fonction *cap_argalloc* en ligne 3 permet d'allouer la mémoire sur la pile associée au contexte donné en premier argument.

L'intérêt de cette approche est que cela conserve l'ancienne API et donc préserve la compatibilité. Par rapport au système existant, la surcharge syntaxique est négligeable, puisqu'en pratique cela impose juste de rajouter un argument (le contexte) à la fonction d'allocation.

[5] On suppose que la fonction d'allocation présentée ici est correctement inlinée par le compilateur et que donc la recopie des arguments dans la structure est à peu près équivalente à une mise sur la pile.

[6] Non implémentée actuellement.

[7] La plupart du temps, il n'y a pas d'allocation effective mais reprise d'une pile existante, ce qui rend quasi nul le temps d'allocation.

Listing 5.4 – Ajout d'une fonction afin de permettre l'allocation des paramètres sur la pile.

```
ctxt = capsys_probe(func);
if (ctxt) {
    newlt = cap_argalloc(ctxt, sizeof(looparg_t));
    newlt->arg = ...;
    capsys_divide(ctxt, newlt);
}
```

L'autre approche, dans le listing 5.5, ajoute un paramètre à *capsys_probe* qui indique combien de mémoire est demandée pour stocker les paramètres. Le pointeur vers la zone mémoire allouée est accessible via la structure de contexte renvoyée par la fonction.

Listing 5.5 – Modification de l'API de division conditionnelle pour faciliter le passage de paramètres.

```
ctxt = capsys_probe(func, sizeof(looparg_t));
if (ctxt) {
    newlt = (looparg_t*) ctxt->args;
    newlt->arg = ...;
    capsys_divide(ctxt);
}
```

Cette approche fournit à Capsule plus d'informations au moment de prendre une décision pour une division, puisque le probe est assorti de la taille des paramètres à allouer. Cependant, cette taille a toute les chance d'être constante pour une fonction à appeler donnée, donc l'intérêt en pratique est faible.

Il peut être intéressant que la fonction appelée dans le contexte puisse avoir de vrais paramètres au lieu de se contenter d'un pointeur non typé. Cette deuxième approche permettrait en partie de prévoir cela. Il est en effet nécessaire pour ce faire que Capsule gère les paramètres ; avoir les paramètres accessibles via la structure de contexte permet cela (via le *ctxt→args* de l'exemple). Cependant, il reste nécessaire d'avoir une machinerie au moment de la compilation afin de gérer cela proprement.

Notons que, dans les deux cas, cela permet de gérer la désallocation automatique de la structure de paramètres.

5.3.7 Stabilité induite : cas du quicksort

Algorithme

Quicksort est un algorithme de tri de liste en place. Il s'agit d'un algorithme récursif. À chaque itération, il partitionne la liste courante en deux sous-listes, en s'assurant que les éléments de la première sous-liste sont tous inférieurs à un pivot et que les éléments de la seconde sous-liste sont tous supérieurs à ce même pivot.

La version Capsule exploite cela pour en tirer du parallélisme. A chacune de ces séparations en deux sous listes, une division est tentée : si elle réussit, alors chaque sous-liste est traitée en parallèle ; sinon, le travail est séquentialisé.

La figure 5.14 représente une exécution de cette version Capsule de quicksort. Le graphe montre les différentes divisions : un noeud est un contexte, un arc une division. On voit clairement

que les divisions ne sont pas équilibrées : certaines zones de la liste provoquent plus de divisions que d'autres.

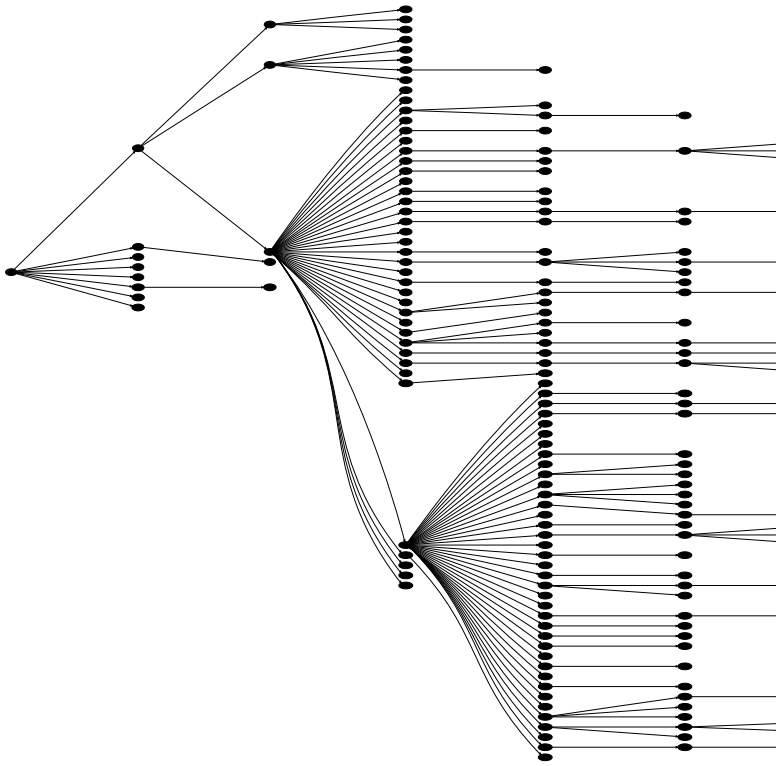


Figure. 5.14 – Répartition des divisions sur une exécution de quicksort en Capsule. Chaque arc représente une division, chaque noeud représente un contexte.

Cela est lié à l'algorithme en lui même. Dans un quicksort, les sous-listes ne sont généralement pas équilibrées, et peuvent même parfois conduire à des sous-listes presque vides. En pratique, cela signifie que le travail à effectuer n'est pas partagé équitablement lors des divisions dans la version Capsule. Cela se reflète donc par des temps d'exécutions plus longs et donc plus d'occasions de se diviser.

Comparaison à une approche statique

La figure 5.15 représente les performances à l'exécution de 3 versions différentes d'un algorithme de tri type quicksort. Chacune des versions a été soumise à 100 listes différentes de 1000000 d'éléments ; les temps d'exécution ont ensuite été regroupés en 50 groupes différents.

La version séquentielle est une implémentation classique de quicksort, utilisant un seul thread. La version statique se base sur la version Capsule mais limite les divisions dès que tous les threads sont occupés : cela serait à peu près équivalent à un découpage statique des données du quicksort, sans connaissance à priori de la topologie de la liste.

La version Capsule nécessite environ 110 millions de cycles pour trier une liste, contre 280 millions pour la version séquentielle. La version dite statique obtient des résultats très variables : le découpage des données dans un quicksort ne peut se faire de manière équilibré sans connaissance préalable de l'organisation de la liste. Ainsi, la performance de la version statique dépend de la liste, ce qui lui permet parfois d'atteindre la performance de la version Capsule mais aussi

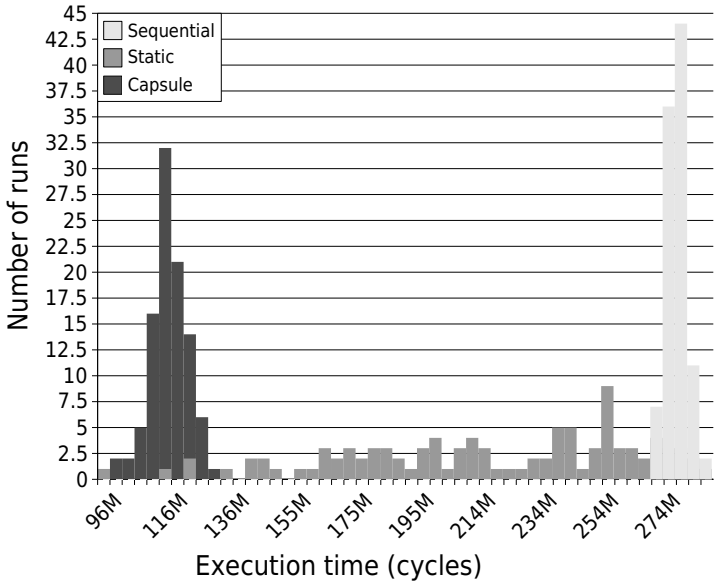


Figure. 5.15 – Comparaison de la stabilité en performance de trois versions de quicksort, sur 100 listes différentes de 1000000 d'éléments. Effectuée sur un AMD bi-processeurs bi-coeurs, soit 4 threads matériels.

parfois fois de se contenter d'une performance similaire à la version séquentielle.

Cette expérience est similaire à celle effectuée sur la version matérielle de Capsule en section 4.3.2. Bien que les paramètres ne soient pas tout à fait les mêmes, on constate que le profil général est exactement le même dans les deux versions.

On remarque que la version Capsule obtient des performances relativement stables, même par rapport à la version séquentielle, et ce malgré l'aspect dynamique du parallélisme Capsule.

Cela est dû à l'approche gloutonne suivie par Capsule : à partir du moment où un programme est capable de se diviser suffisamment souvent (et Capsule aide à cela puisqu'il permet de décrire le parallélisme à n'importe quel niveau), Capsule est capable de réutiliser le parallélisme à n'importe quel moment, permettant ainsi de réajuster le comportement lorsque celui-ci s'éloigne d'une occupation maximum des ressources. Le critère déterminant devient ainsi la quantité de travail à effectuer ; à partir du moment où celui-ci est constant, le parallélisme dynamique permet d'assurer une performance stable. Dans le cas présent, les listes sont tirées aléatoirement, donc la quantité de travail reste sensiblement constante, ce qui est montré par la version séquentielle.

5.3.8 Utilisation en tant que pool de threads : cas de x264

Un des gains obtenu par l'utilisation de Capsule vient du fait qu'il fournit une implémentation générique de pool de threads. Il s'agit d'une approche relativement classique pour la gestion du parallélisme [50], [46], [13].

Dans la majorité des systèmes fournissant des threads, créer un nouveau thread est plus coûteux que maintenir un thread inactif. L'intérêt du pool considéré ici est d'éviter de devoir allouer un thread quand l'intérêt s'en fait sentir et au contraire de les réutiliser, ou même éventuellement d'en allouer préventivement. Ce dernier cas de figure ne s'applique pas à Capsule puisque le nombre de threads vivant simultanément est limité de manière stricte.

L'encodeur vidéo x264 profite de cet aspect. x264 est une implémentation en C d'un encodeur

vidéo H.264 [2] , encore appelé MPEG4 AVC. Le code utilise déjà, de base, un système de parallélisation basé sur pthread.

Le principe en est décrit sur la figure 5.16. En temps normal, lors de l'encodage d'une frame, l'encodeur se sert notamment de l'auto-corrélation existante entre les différentes parties de l'image afin de pouvoir la compresser. En pratique, cela veut dire que, dans le flux binaire de sortie, l'image sera découpée en blocs et que chaque bloc est dépendant du précédent pour pouvoir être décodé.

Afin de pouvoir paralléliser, x264 utilise une possibilité fournie par la norme H.264. La norme intègre la notion de « slices », qui permettent de découper une image en plusieurs parties indépendantes, supprimant la corrélation le long des zones de jonctions. Il devient donc possible d'encoder chaque « slice » en parallèle aisément.

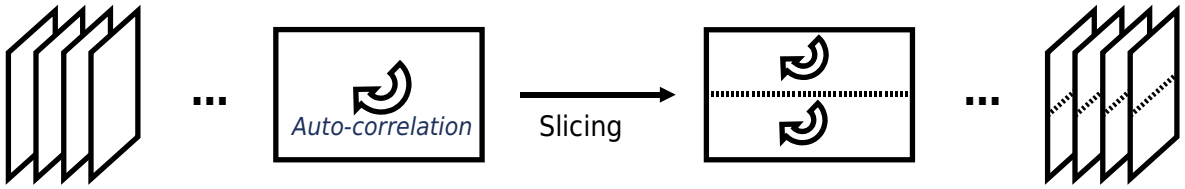


Figure. 5.16 – Principe de parallélisation simple de l'encodeur x264. Chaque image est divisée arbitrairement en parties indépendantes, dont une partie du traitement pour être fait en parallèle.

La zone parallèle de x264 est relativement simple, son principe est montré sur le haut de la figure 5.17. Avant le début du traitement d'une frame, il regarde combien de threads sont disponibles ; à partir de cela, il détermine les différents « slices » et crée les structures afférentes. Ensuite, il crée les threads de traitement, puis attend qu'ils aient tous fini. Enfin, il génère (séquentiellement) le flux H.264 binaire.

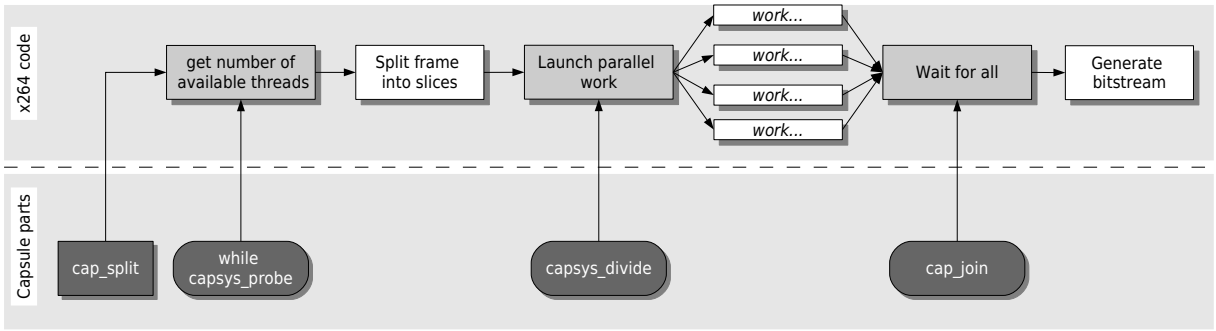


Figure. 5.17 – Coeur parallèle de x264. Les modifications pour utiliser Capsule sont mineures et ne modifient pas le flot du programme.

La version Capsule nécessite peu de modifications. Afin de pouvoir attendre que tous les threads aient fini, il est nécessaire d'insérer un *cap_split* avant le début du traitement de frame et de substituer la synchronisation existante par un *cap_join*.

La détection du nombre de threads disponible diffère quelque peu de la version originale. Dans la version originale de x264, le nombre de threads disponibles, et donc le nombre de « slices » utilisés, était déterminé statiquement, soit via un paramètre en ligne de commande, soit en se basant sur le nombre de processeurs que présente le système. La version Capsule, afin

de pouvoir s'adapter dynamiquement, détermine le nombre de threads disponibles au début de chaque frame.

En pratique, cela passe par le code dans le listing 5.6.

Listing 5.6 – Allocation des contextes en début de frame dans x264

```
1 ctxts[0] = capsys_current();
2 i = 0;
3 while ((i < X264_SLICE_MAX-1) && ctxts[i]) {
4     i++;
5     ctxts[i] = capsys_probe((void*)x264_slice_write);
6 }
```

Ce code essaye d'allouer au maximum $X264\_SLICE\_MAX$ contextes, et stocke chacun dans le tableau *ctxts*. Le contexte d'index zéro correspond au contexte courant, puisqu'il fait lui même une partie du travail ; les contextes sont conservés dans un tableau afin de pouvoir exécuter les *capsys_divide* correspondants. Dès que qu'un probe échoue, le programme considère qu'il a alloué tous les threads disponible et arrête donc d'essayer d'en allouer.

Bien que cette approche gloutonne risque de rater potentiellement des threads qui se libéreraient pendant l'encodage d'une frame, x264 est construit de manière à rendre un découpage dynamique difficile à ce niveau. Cependant, il s'agit de toute façon d'un schéma relativement classique ; il n'est pas toujours possible de découper le travail restant après le début du traitement. De plus, dans le cas de x264, la durée d'encodage d'une frame est relativement faible, ce qui laisse tout de même une latitude dans l'adaptation dynamique.

La version normale de x264 et la version Capsule ont été testées sur deux clips vidéo dont les caractéristiques sont données dans la table 5.5.

Nom	Taille (octets)	Images	Largeur	Hauteur
Clip480	233481651	1201	480	270
Clip1024	1327113047	1500	1024	576

Table. 5.5 – Caractéristiques des clips utilisés pour les tests. La différence de taille des images fait que la quantité de traitement par image augmente pour « Clip1024 ».

Le principal intérêt de ces deux clips est que l'un a une résolution nettement supérieure à l'autre. Cela implique donc que le temps d'encodage d'une frame augmente, ce qui permet de mettre en évidence ce qui est dépendant de la résolution de ce qui ne l'est pas directement.

Le tableau 5.6 donne les tailles des fichiers obtenus pour les différents niveaux de découpe. En effet, puisque pour paralléliser il est introduit des « slices » qui suppriment la corrélation entre certaines zones de l'image, la compression perd nécessairement en efficacité.

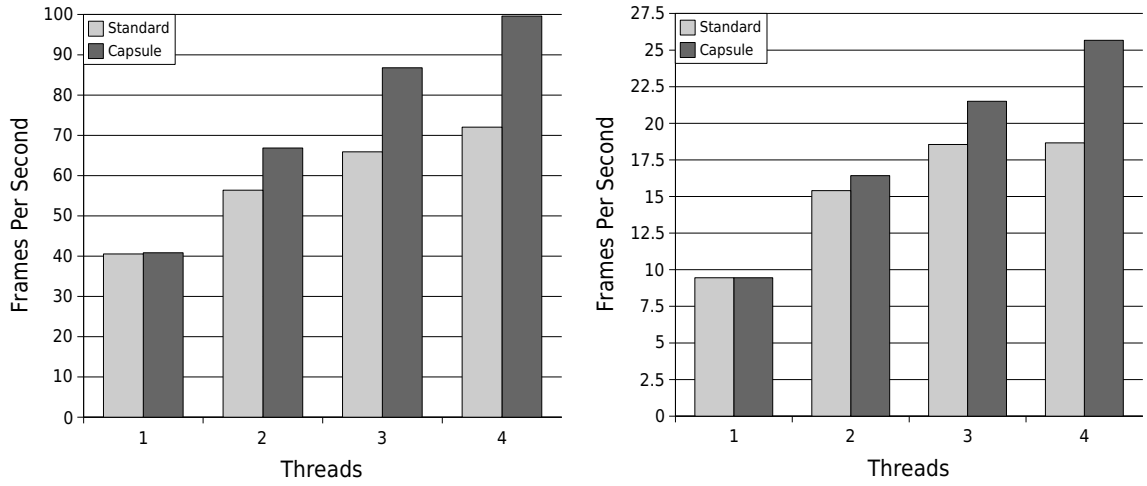
Cependant, on constate que la taille de fichier n'évolue guère pour un nombre faible de processeurs. Même en découpant en 128 morceaux, pour avoir un parallélisme simple de 128, l'accroissement en taille n'est que de l'ordre de 10%.

La figure 5.18 compare les vitesses d'encodage obtenues pour les deux exemples utilisés. On constate que la version Capsule obtient de meilleures performances en parallèle. De plus, l'écart se creuse avec l'augmentation du nombre de threads.

Comme l'approche de parallélisation est la même pour les deux versions, le gain vient d'une réduction du coût de création des threads. La version normale de x264 recrée les threads à chaque image via *pthread_create* ; or, comparativement à la durée d'encodage d'une image, cet appel est coûteux. Capsule, en jouant ici le rôle de pool de threads évite ce surcoût.

Clip	1 slice	2 slices	3 slices	4 slices	128 slices
Clip480	4377046 100%	4407807 100.7%	4437554 101.4%	4467323 102.1%	4801603 109.7%
Clip1024	15829709 100%	15911794 100.5%	15986238 101.0%	16063917 101.5%	17828249 112.6%

Table. 5.6 – Taille des vidéos en octets une fois compressées selon le nombre de « slices ». Les poucentages correspondent à l'augmentation de taille par rapport à un encodage sans « slices ».



(a) Clip480 *(b)* Clip1024

Figure. 5.18 – Comparaison des vitesses d'encodage des deux clips utilisés entre la version normale de x264 et la version Capsule. Plus le nombre d'images par seconde est élevé, plus la performance est bonne.

5.4 Limitations de la version logicielle

5.4.1 Empêcher les divisions trop fréquentes

Dans la version de base de quicksort en Capsule, à chaque séparation en sous-listes, une demande de division est faite. Cela permet d'obtenir de la performance par rapport à une version séquentielle (voir figure 5.15), mais il reste une perte de performance claire liée au choix de diviser ou non. Bien qu'une demande de division qui échoue soit très légère, de l'ordre de quelques cycles, une division réussie prend plus de temps, de l'ordre de quelques milliers de cycles. Dans le cas de listes de quelques éléments seulement, le coût de la division devient nettement supérieur au gain obtenu par la parallélisation.

Afin d'avoir une idée de ce qui est obtenable en ayant une politique de division capable d'éviter ce type de problème, un *seuil* de division a été introduit dans le quicksort Capsule : si une des listes comporte, lors d'une division, moins d'éléments que ce seuil, alors aucune division n'est demandée et le travail est fait séquentiellement.

La figure 5.19 représente le speedup obtenu en faisant varier la valeur de ce seuil, exprimé en nombre d'éléments minimum à considérer dans une sous-liste. On constate que le speedup évolue rapidement vers une valeur proche de son maximum ; il est tout de même nécessaire d'éviter les

listes de moins d'une cinquantaine d'éléments.

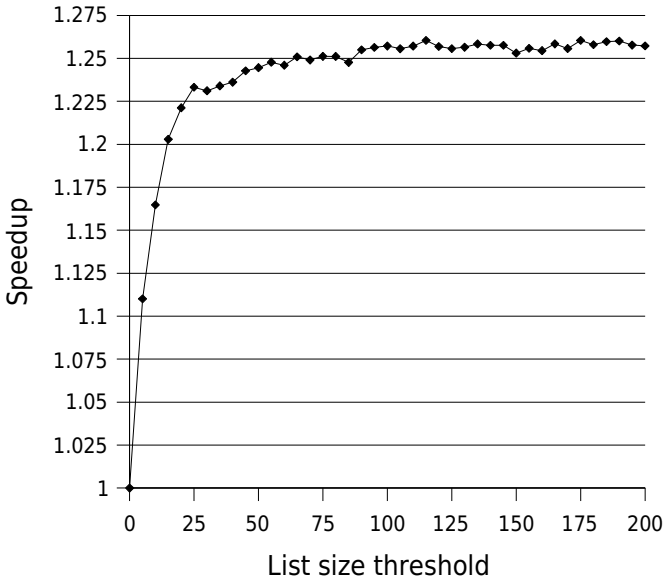


Figure. 5.19 – Effet d'un seuil empêchant la division pour des listes contenant moins de n éléments dans quicksort, moyenné sur 32 listes différentes de 1000000 d'éléments. Effectué sur un AMD bi-processeurs bi-coeurs, soit 4 threads matériels.

5.4.2 Problèmes liés à la taille des jeux de données

La figure 5.20 montre l'évolution du speedup de la version Capsule de quicksort en fonction de la taille de la liste, pour plusieurs nombres de processeurs.

Avant d'atteindre la zone stable, la version 2 processeurs est plus rapide que les versions 3 et 4 processeurs. À cause de la présence de plus de ressources, plus de divisions sont réellement effectuées. Le surcoût lié à l'opération de division elle même, mais aussi celle liée à la division en listes trop petites, provoquent donc une perte de performance relative lorsque l'on augmente le nombre de processeurs.

Pour les petites listes, la version Capsule perd de la performance par rapport à la version mono-processeur. Il faut, pour que la version Capsule soit intéressante, des listes d'environ au moins 10000 éléments. Cela s'explique par le problème du seuil de division. Au lancement du tri tous les threads sont disponibles, les premières divisions sont acceptées, répartissant la liste rapidement entre les threads. Comme les listes sont petites, le surcoût de la division n'est compensé ni par la quantité d'éléments dans chaque liste, ni par effet statistique sur les répartitions en sous-listes.

En pratique, un tri de listes courtes utilisant Capsule peut ne pas être gênant s'il est invoqué rarement : l'impact du surcoût sera négligeable. Le cas problématique se pose lorsque le programme fait appel très souvent à des tris de petites listes.

Cependant, il sera possible de faire appel dans ce cas à des prédicteurs se basant sur quelques exécutions initiales du tri considéré. Cela n'est pas spécifique au quicksort : un prédicteur se basant sur la durée d'exécution moyenne des divisions récentes afin d'inhiber certaines futures divisions ne dépend pas du code sous-jacent.

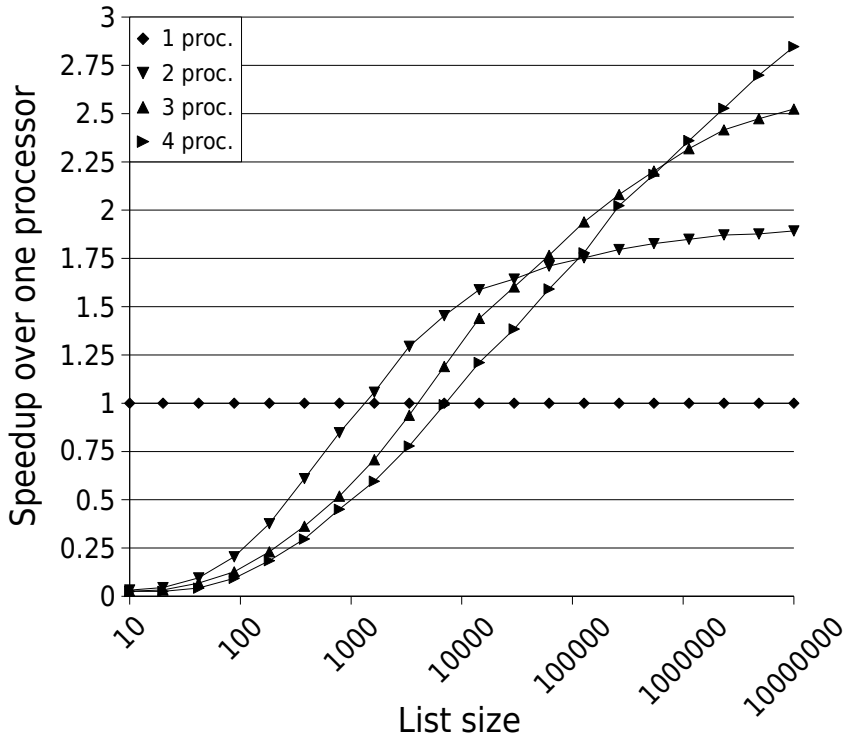


Figure. 5.20 – Évolution du gain de performance par rapport à une version mono-processeur selon la taille de la liste. Moyenné sur 32 listes différentes.

Sur la figure 5.21, on voit que l'introduction d'un seuil permet d'éviter les pertes de performances par rapport à la version séquentielle sur de petites listes. Ce qui est intéressant est que le seuil de division est fixé à 100 éléments, alors que l'on voit sur la figure 5.20 que la perte de performance se faisait jusqu'à des listes de 10000 éléments environ. La perte de performance n'est donc pas dû à un problème intrinsèque de Capsule ralentissant l'exécution mais simplement du nombre trop grand de divisions effectuées.

La figure 5.22 montre l'effet du seuil sur le speedup global. Elle représente le speedup de la version 4 processeurs pour différents seuils, pour de grandes listes.

Un phénomène apparaît : l'introduction d'un seuil permet d'améliorer le speedup jusqu'à un certain point. Les seuil améliorent la performance dans cette expérience entre 0 et 1000, mais entre 1000 et 100000, on voit que la performance chute à nouveau. En fait, le seuil joue sur deux phénomènes à l'exécution :

- La division en listes trop petites pour être intéressantes à paralléliser, c'est ce qui est montré précédemment, notamment sur la figure 5.19.
- L'absence de division pour des listes assez grandes finit par limiter le parallélisme alors même qu'il serait intéressant.

5.4.3 Contraintes sur les métriques pour la décision de division

Bien que le seuil soit efficace, c'est une solution d'une part spécifique au quicksort mais qui de plus doit être réglée pour chaque type d'implémentation et de système.

Il est donc nécessaire d'améliorer le processus de décision de division afin d'éviter les divisions

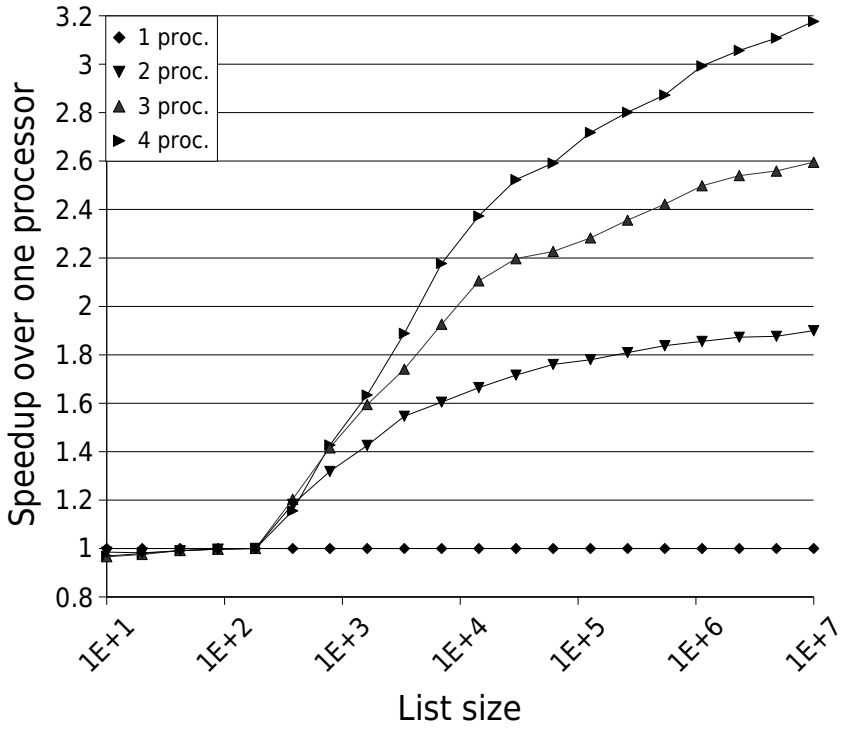


Figure. 5.21 – Évolution du gain de performance par rapport à une version mono-processeur selon la taille de la liste. Un seuil de division a été fixé à 100 éléments, empêchant la parallélisation pour de trop petites listes. Moyenné sur 32 listes différentes.

inutiles, voire coûteuses.

La version actuelle logicielle de Capsule a une politique de division conditionnelle très simple : tant que des threads matériels sont disponibles, la bibliothèque autorise les divisions, et les refuse sinon.

Cependant, comme vu en section 4.1.2, il peut être intéressant d'avoir des politiques allant au-delà de la politique actuelle. Pour ce faire, il faut manipuler des métriques d'efficacité de l'exécution plus complexes que la simple occupation des ressources. Bien que l'utilisation d'un prédicteur logiciel permette d'implémenter logiciellement un probe simple efficace, cela laisse peu de marge de manoeuvre pour ce type de métriques.

Tant que le test d'un probe peut se réduire à un test de variable, la version logicielle est facilement envisageable. Ainsi, si on veut ajouter une métrique sans influencer sur la performance du probe échoué, il est nécessaire que cela puisse se contenter d'influencer ce prédicteur logiciel, sans ajouter de tests lourds dans le probe.

Conserver l'approche d'un prédicteur simple implique les contraintes suivantes sur le type de métriques utilisables :

1. Le prédicteur est global, cela signifie que les métriques ne doivent pas porter sur des aspects spécifiques à tel ou tel contexte. Cela empêche notamment de favoriser certaines parties de code du programme utilisant Capsule par rapport à d'autres quand on voit qu'elles ont un comportement qui nous intéressent.
2. Puisqu'il n'est pas possible de vérifier l'état d'une métrique à la division, il faut pouvoir

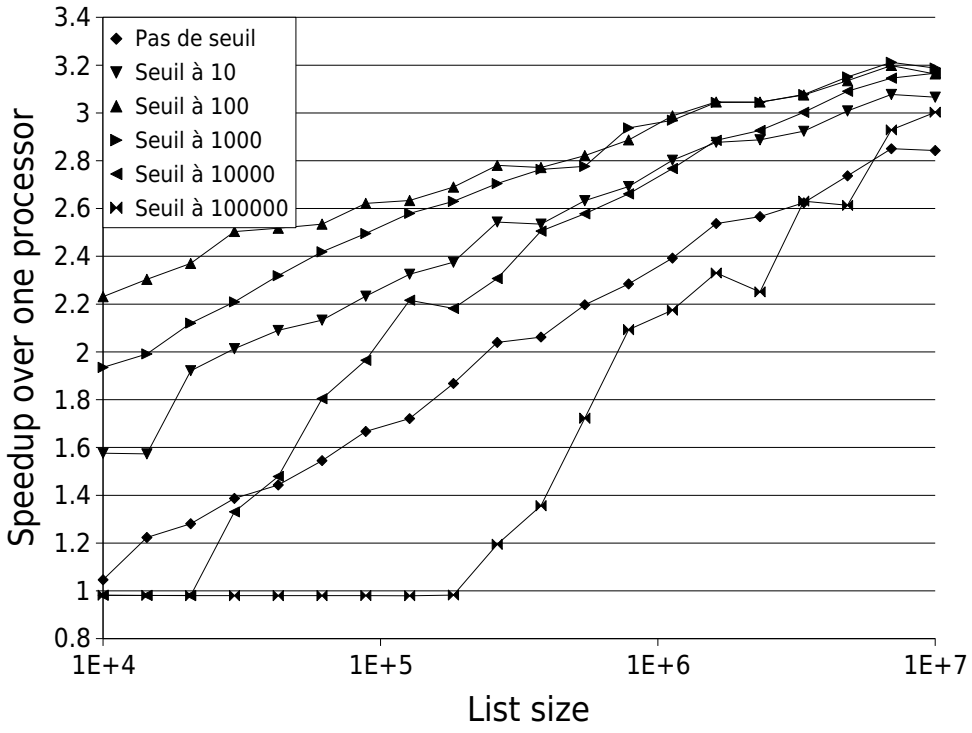


Figure. 5.22 – Évolution du gain de performance sur 4 processeurs par rapport à une version mono-processeur selon la taille de la liste et le seuil choisi. Moyenné sur 16 listes différentes. L'instabilité est grandement amplifiée par l'échelle logarithmique.

influencer le prédicteur logiciel de manière asynchrone. Autrement dit, il faut pouvoir le modifier de temps en temps, en dehors des probes échoués. Bien sûr, ces modifications ne doivent pas elle même induire un surcoût.

3. Enfin, le taux de mauvaise prédiction doit rester très faible afin de ne pas passer à côté de ressources intéressantes. Le fait que le prédicteur doit être mis à jour de manière asynchrone risque d'augmenter le risque d'erreur. Dans le cas de la métrique simple du nombre de threads disponible, il était facile de suivre cela de manière exacte, puisqu'il suffit de suivre les naissances et morts de contextes ; dans le cas d'une métrique dépendant de détails d'exécution, cela peut être plus complexe.

Dans le cadre de la version matérielle, la plupart de ces contraintes ne se posent pas, puisqu'il est possible d'avoir accès directement à des mesures matérielles et avoir des unités dédiées à la gestion des prédicteurs.

Cependant, plusieurs approches sont envisageables afin de conserver une implémentation complètement logicielle, tout en prenant en compte les problèmes précédemment cités.

Unicité du prédicteur

Il est possible d'avoir plusieurs prédicteurs à la place d'un seul. Typiquement, il serait possible d'avoir un prédicteur par groupe, puisque cela correspond généralement à un découpage en tâches similaires, ce qui signifie généralement un comportement similaire. Au lieu d'accéder à une variable globale, il faut récupérer la variable correspondant au groupe, ce qui peut introduire

un surcoût. Cependant, sur la plupart des architectures, le « Thread Local Storage », zone de stockage privée à chaque thread, est implémenté de manière aussi efficace que la zone de données globales, ce qui permet donc d'implémenter des prédicteurs différenciés efficaces.

Avoir plusieurs prédicteurs nécessite de mettre à jour plusieurs variables au lieu d'une seule. Ici, le surcoût dépend essentiellement de la métrique, notamment selon sa fréquence de mise à jour. Le fait d'avoir une mise à jour asynchrone, et donc potentiellement relativement rare peut permettre de réduire l'impact de ces mise à jour.

Mises à jour du prédicteur

Dans un contexte parallèle, la première idée pour résoudre le problème des mises à jour asynchrones serait de dédier un thread à cela qui passerait son temps à observer le comportement des autres threads et adapterait les prédicteurs en fonction. Bien que relativement souple, cette approche risque d'induire également un coût non négligeable. Par exemple, il paraît peu intéressant de dédier un thread physique à cela ; mais sans thread matériel, cela signifie que l'on devient tributaire du système et que l'ordonnancement du thread de mise à jour du prédicteur risque d'être chaotique.

Cette dépendance au système peut cependant être mise à profit. Au final, c'est le système d'exploitation qui a la meilleure connaissance du comportement des différents threads, qui a accès à tous les compteurs matériels et qui est en charge des différentes politiques de répartition des ressources. Implémenter la gestion des prédicteurs au niveau du système permettrait de profiter de ces informations, mais permettrait également d'introduire une mise à jour à chaque changement de contexte système et/ou d'appel système.

Enfin, il est également possible d'avoir une approche probabiliste, en effectuant une mise à jour des prédicteurs uniquement tous les N probes. En jouant sur N, on peut ainsi diminuer le surcoût de mise à jour des prédicteurs. Cela peut se faire relativement simplement : il suffit d'ajouter une variable qui compte le nombre d'appels à probe ; un test sur la valeur de cette variable détermine s'il faut mettre à jour les prédicteurs à ce moment-là. Bien que rajouter ainsi un deuxième test augmente la quantité d'instructions pour un probe échoué, si on compare cela au nombre d'instructions du programme par probe, cela reste probablement tolérable dans la majorité des cas.

Le principe de ce probe légérement modifié est représenté sur la figure 5.23. Le probe commence par incrémenter un compteur d'appel. Si ce compteur dépasse une valeur N, alors le prédicteur est mis à jour ; la suite se déroule normalement.

La figure 5.24 montre le coût d'une mise à jour périodique du prédicteur.

La méthode de mesure utilisée est la même que dans la section 5.3.2. La division est bloquée, le temps d'exécution d'une boucle lançant des *capsys_probe* est mesuré et il est ôté le temps d'exécution de la même boucle, mais vide. En divisant par le nombre d'itérations, on obtient une mesure du coût d'un probe échoué.

On voit que, dans les deux cas représentés, pour deux processeurs différents, le surcoût s'estompe rapidement, se stabilisant vers quelques cycles en moyenne. Il est à noter cependant que le coût d'une mise à jour du prédicteur reste elle-même relativement faible, se contentant de faire quelques comparaisons.

Une variation du coût se produit vers $n = 8$ pour le processeur AMD et vers $n = 60$ pour le processeur Intel. Cela est manifestement causé par la différence d'implémentation des prédicteurs matériels dans chacune de ces architectures.

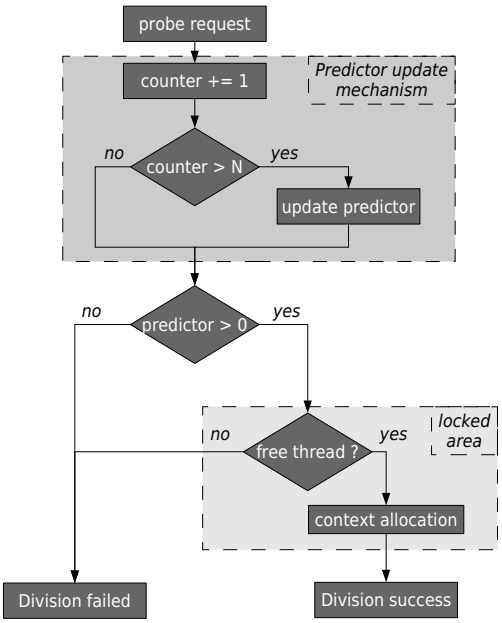
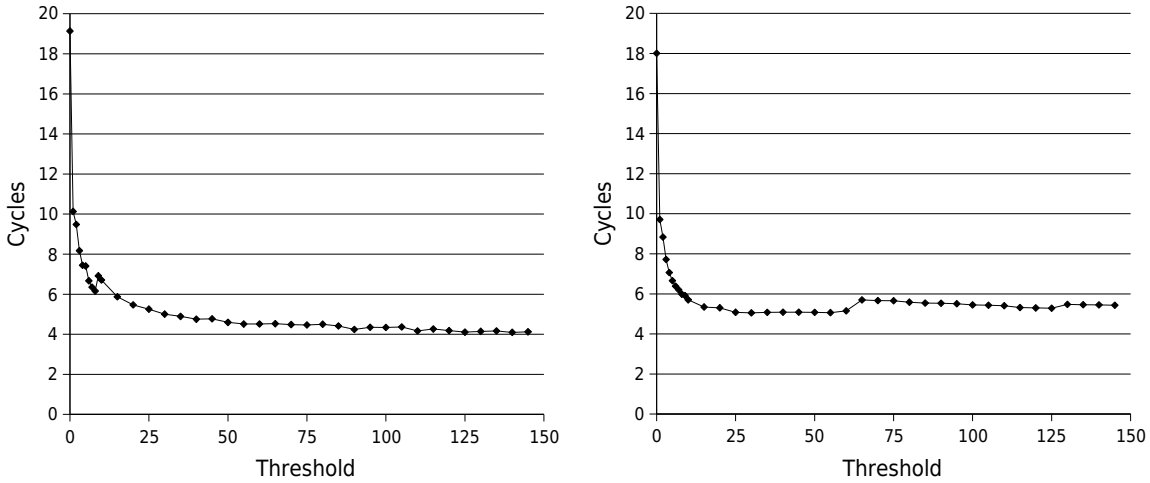


Figure. 5.23 – Workflow d'un probe avec mise à jour périodique du prédicteur.



(a) AMD DualCore Opteron 264 *(b)* Intel Core 2 Duo

Figure. 5.24 – Variation du coût du probe échoué lors du micro benchmark de la section 5.3.2. L'abscisse représente la période de mise à jour du prédicteur.

Fiabilité du prédicteur

La notion de fiabilité du prédicteur reste en fait particulièrement dépendante de la prédiction du nombre de threads matériels. En effet, il est peu probable que d'autres prédicteurs aient un impact aussi important sur la performance que le fait de ne pas utiliser un thread hardware complet. La plupart des autres métriques envisageables étant - à priori - des améliorations de la politique de base, il est tout à fait possible que le taux d'erreur soit plus élevé sans que cela ne soit visible sur la performance.

5.5 Intégration au noyau Linux

La version logicielle de Capsule a dès le départ été conçue pour une intégration la plus fine possible avec le système existant, notamment la bibliothèque pthread. L'étape suivante serait donc d'intégrer cela directement au noyau. Il est discuté ici succinctement l'approche pour l'intégration à Linux ; il est tout à fait possible d'envisager cette intégration sur d'autres noyaux ou au niveau de machines virtuelles de langages tels que Python, Java ou .Net.

5.5.1 Sémantique

Sous Linux, la fonction *pthread_create* qui permet de créer un thread fait appel à l'appel système *clone*, qui permet de manière générique de créer de nouveaux fils d'exécution gérés par le noyau. De nombreux flags existent pour manipuler le comportement de cet appel, contrôlant les ressources qui seront partagées ou au contraire spécialisées entre les threads. Ainsi, cet appel permet aussi bien de créer des threads (mémoire partagée entre les threads d'une même application) que des nouveaux processus, de manière similaire à $fork$.

L'intégration la plus simple de Capsule au niveau du noyau est de rajouter un flag à l'appel système *clone*. La sémantique de ce flag est la même que celle de la division conditionnelle. Si le noyau estime, lors de l'appel système, qu'il n'est pas intéressant de créer un nouveau thread et que ce flag est présent, alors l'appel système échouera en renvoyant un code d'erreur tel que $EAGAIN$ [8].

Côté utilisateur, la bibliothèque pthread doit rajouter également un attribut afin d'offrir la division conditionnelle lors du *pthread_create*. Les attributs pthread permettent de spécifier des propriétés spécifiques lors de la création de threads, telles que des propriétés d'ordonnancement. L'activation des propriétés se fait via des fonctions dédiées ; il serait donc nécessaire de créer une fonction *pthread_attr_setconditional*.

Il est à noter que cette approche diffère légèrement de l'API Capsule actuelle. Avec la bibliothèque pthread, une fois un *pthread_create* exécuté avec succès, le nouveau thread commence son exécution aussitôt, contrairement à Capsule qui actuellement attend un appel à *capsys_divide* pour initier l'exécution effective.

Cela n'est pas nécessairement gênant. Le choix de pthread étant rentré dans les moeurs [9] , il semblerait préférable d'adapter Capsule à cela.

Deux solutions simples sont possibles selon les cas :

- Mettre en place une *condition* qui ferait attendre le thread nouvellement créé. Cela émulerait le comportement actuel de la division conditionnelle dans le Capsule logiciel ;
- Éclater le code habituellement présent entre le probe et le divide ; la partie spécifique au nouveau thread est mise au début de celui-ci. Cela revient à laisser chaque thread calculer son propre ensemble de données sur lesquelles il doit travailler. Le risque est alors d'augmenter les cas possibles de race-condition, puisque les deux threads doivent manipuler des structures communes.

5.5.2 Séparation mode utilisateur / mode noyau

Il n'est ni possible ni intéressant de déporter toute l'implémentation de la division conditionnelle efficace au sein noyau. Le principal obstacle à cela est le coût d'un appel système,

[8] Le code d'erreur $EAGAIN$ signifie actuellement que trop de processus existent au moment de l'appel, ou, autrement dit, que les capacités du système ont été atteintes.

[9] Dans la page de manuel de *pthread_create*, il est expliqué plusieurs raisons à ce choix. Notamment, cela permet de réduire le nombre d'appels système et d'éviter l'introduction d'un nouvel état possible pour un thread.

nécessaire pour activer le noyau : il n'est simplement pas possible d'implémenter le probe directement avec un appel système, puisque le cas échéant, chaque probe, même échoué, conduirait à un aller/retour en mode noyau.

Il est donc nécessaire d'avoir au moins une partie du probe, typiquement le probe échoué, en mode utilisateur. Cela correspond déjà à l'existant : une partie dans le noyau, utilisable via l'appel système *clone* et une partie en mode utilisateur, via la bibliothèque pthread.

Afin que le probe échoué puisse se faire la majorité du temps sans passer par le noyau, la bibliothèque doit être capable de décider par elle-même s'il est intéressant d'essayer de demander une division ou s'il vaut mieux s'en abstenir. Il paraît donc intéressant de conserver la notion de prédicteur ; simplement, dans le cas présent, le prédicteur peut être maintenu par le noyau.

Le noyau devrait donc fournir au processus/thread considéré une variable qui servirait de simple indication pour que le code en mode utilisateur puisse savoir s'il est intéressant de lancer l'appel système clone correspondant. Plusieurs méthodes sont envisageables en pratique pour mettre à disposition cette variable. Bien que ce ne soit pas dans l'habitude du noyau Linux de mettre à disposition des données en continu (par opposition à des buffers remplis lors de la lecture de fichier par exemple), il est possible par exemple de fournir un appel système qui permette d'indiquer au noyau où stocker le prédicteur pour chaque thread. Cette adresse mémoire peut même être fournie directement lors de l'appel à *clone*.

Le fonctionnement serait donc celui décrit sur la figure 5.25.

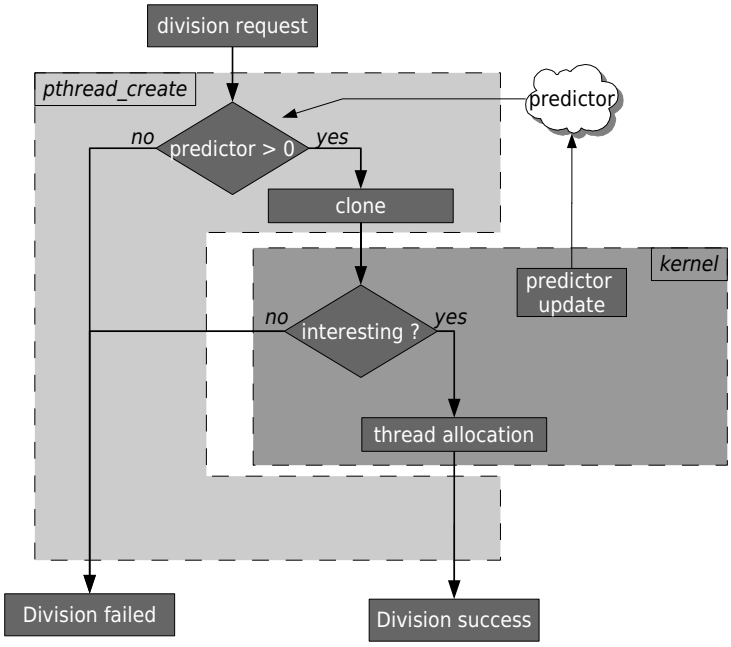


Figure. 5.25 – Workflow d'un probe avec la division conditionnelle au niveau noyau. Le prédicteur est mis à jour lors d'événements divers (dont un clone) par le noyau.

Le prédicteur, maintenu par le noyau, est lu par la fonction *pthread_create* ; s'il s'avère intéressant de continuer, alors un appel système est effectué, passant la main au noyau.

Enfin, avoir ces paramètres et fonctionnalités au niveau noyau permet de facilement avoir des politiques de gestion des threads à l'échelle du système. Avoir les politiques de gestion de threads gérés par une bibliothèque rend en pratique difficile un réglage commun, notamment à cause des différences de version possibles ou encore de la présence de logiciels linkés statiquement.

5.5.3 Gestion du prédicteur

Le prédicteur, utilisé par la bibliothèque en mode utilisateur, est mise à jour principalement par le noyau. Il est donc possible de le mettre à jour au moment de l'ordonnancement. Comme a priori l'ordonnancement se fait relativement régulièrement, cela permet d'avoir une mise à jour asynchrone du prédicteur sans pour autant nécessiter un thread supplémentaire. De plus, le prédicteur peut être mis à jour au moment des appels système.

L'intérêt dans les deux cas est que cela permet de suivre les événements du système qui semblent corrélés avec les besoins du prédicteur. Le noyau a directement accès à des informations permettant de calculer le prédicteur :

- ordonnancement effectif ;
- occupation de chaque processeur ;
- compteurs matériels ;
- profil du processus, comme par exemple la part d'entrées/sorties.

5.5.4 Gestion des threads

La phase de «thread allocation» a en réalité deux aspects : l'allocation de la pile et la création du fil d'exécution proprement dit.

Comme c'est au noyau de décider de la création ou non d'un thread, le plus logique est de lui laisser également la gestion de la création des threads. Cela s'oppose à la solution actuelle choisie par Capsule qui laisse la partie en mode utilisateur gérer un pool de threads. Bien sûr, il est possible d'implémenter également un pool de threads pré-alloués au niveau du noyau. Cependant, cela est probablement inutile, l'allocation en elle même d'un thread au niveau noyau n'étant pas particulièrement coûteuse. Le coût de création d'un thread est habituellement causé par les problèmes d'allocation de pile et d'aller/retour en mode noyau nécessaires.

L'allocation de la pile est traditionnellement laissée au mode utilisateur. En fait, le noyau ne sait pas du tout comment le processus organise sa mémoire et est donc incapable de gérer une allocation propre au sein de celui-ci. L'allocation reste donc l'apanage du mode utilisateur. De même, avec les threads Capsule, il peut être parfois intéressant d'utiliser une pile contenant déjà des données afin de reprendre une exécution. Il n'est donc pas intéressant que le (potentiel) pool de threads que maintiendrait le noyau conserve également les piles.

Ainsi, lors d'une division, la bibliothèque devra fournir une pile lors de l'appel de *clone*, comme cela se fait actuellement. La bibliothèque pourra dès lors implémenter toutes les politiques d'allocation qui l'intéressent, y compris bien sûr réutiliser des piles pour éviter de devoir allouer systématiquement un large espace mémoire, comme ce qui se fait déjà dans Capsule.

Les problèmes d'ordonnancement mentionnés à la section 5.3.4 sont simplifiés lorsque l'on intègre la division conditionnelle au niveau du noyau. En effet, puisque chaque création réussie passe par le noyau, il est envisageable dans ce cas pour l'ordonnanceur d'agir en conséquence, afin d'éviter des latences inutiles si possible.

Chapitre 6

Modèle mémoire

Dans les sections précédentes, il était essentiellement question de l'implémentation de Capsule, de comment avoir des primitives de parallélisme efficaces et portables. Cependant, un des principes de l'approche Capsule est d'essayer de redistribuer la complexité entre les différentes couches de programmation / exécution. Ainsi, la notion de division conditionnelle permet au matériel de prendre des décisions qu'il est le mieux à même d'assumer grâce à une information simple qui lui est fournie. Ce chapitre aborde un aspect plus haut niveau, le modèle mémoire mentionné dans la section 3.2, permettant d'aider l'utilisateur à utiliser les mécanismes de Capsule.

Il s'agit d'un chapitre avant tout prospectif. La plupart des idées présentées ici ont influencé la conception des couches bas niveaux présentées dans les chapitres précédents. Cependant, ces idées n'ont pas été implémentées ou ne l'ont été que superficiellement. Notamment, certains problèmes sous-jacents n'ont pas encore de réponses satisfaisantes ou mériteraient une exploration plus poussée.

La première partie de ce chapitre présente deux modèles mémoire classiques et met en relief la perte de sémantique qu'ils entraînent. La deuxième partie détaille le modèle mémoire proposé pour Capsule avec la notion de cellule en le comparant aux deux modèles précédents ; un exemple illustre cela. Ce modèle mémoire offre plus de sémantique que les modèles mémoire classiques ; la troisième section présente un exemple exploitant cette sémantique pour faciliter le parallélisme. Enfin, la dernière section mentionne succinctement deux problèmes causés par ce modèle mémoire.

6.1 Les modèles mémoires classiques

Deux modèles mémoire s'opposent dans le cadre du parallélisme. La mémoire partagée est le modèle naturel du parallélisme de thread ; à l'opposé, les modèles acteur se basent sur une mémoire non partagée, utilisable par un seul fil d'exécution.

Ces modèles sont présentés dans les deux sous-sections suivantes afin de les positionner par rapport au modèle mémoire proposé pour Capsule. Cependant, il faut garder à l'esprit qu'il ne s'agit que de modèles et qu'il ne sont donc pas nécessairement corrélés avec le type de mémoire que la machine possède. Par exemple, il est possible de présenter un modèle mémoire partagée sur une machine à mémoire distribuée [51] ou, à l'inverse, de travailler en passage de message sur une machine à espace d'adressage linéaire [10].

6.1.1 La mémoire partagée

Avec un modèle de mémoire partagée via un tas («heap»), le système n'a pas d'autre information lui permettant d'adapter la performance en influant sur l'ordre d'exécution : le moindre changement peut potentiellement corrompre le programme, sans possibilité de détecter cela de manière automatisée.

Même si les frameworks parallèles fournissent des primitives de synchronisation plus ou moins évoluées, ils ne donnent que la sémantique minimale à respecter afin de conserver une cohérence des données et des calculs.

Tout ce que le système voit, c'est ce qui est représenté sur la figure 6.1 : des fils d'exécutions travaillant sur un espace de données commun, avec juste quelques données privées stockées dans une pile à la sémantique limitée. Il lui est difficile de savoir sur quelles données un thread travaille à un moment donné. La seule approche possible est statistique, en étudiant les accès mémoire passés et en considérant que le futur sera similaire ; il s'agit d'une approche basée sur la *localité*.

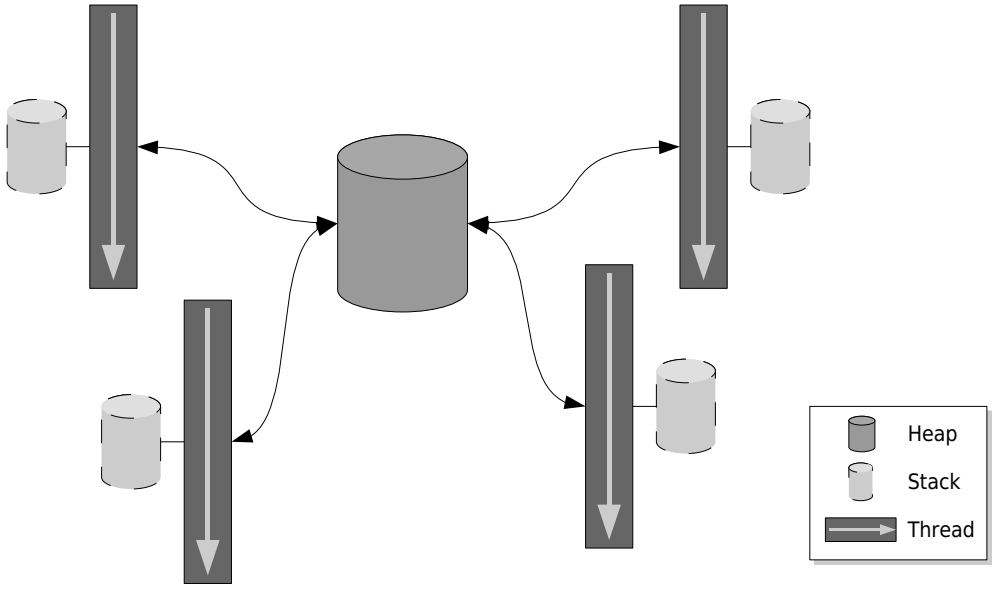


Figure. 6.1 – Vue depuis le système de la répartition thread/mémoire en modèle mémoire partagée.

Cette approche fonctionne relativement bien. Elle est par exemple utilisée pour la gestion des caches ou encore par Linux sur les machines à architecture NUMA, afin d'éventuellement déplacer des pages de données entières vers des zones de mémoires plus rapide vis à vis du processus les utilisant.

Cependant, puisqu'il ne s'agit que d'une approche statistique, il est difficile de se baser dessus afin de prendre des décisions précises. Par exemple, il n'est pas possible de prendre en compte cela afin d'accélérer certains locks en supposant que les autres threads n'accéderont pas à la mémoire en question. De manière similaire, il est difficile de prévoir ce qui se passe en dehors des approches répétitives ; par exemple, un parcours de graphe est complexe à prédire.

6.1.2 La mémoire en modèle acteur

Le modèle Acteur (voir section 2.2.2), à l'opposé de la mémoire partagée, circonscrit de manière précise la mémoire sur laquelle travaille chaque fil d'exécution. Le système sous-jacent

voit ce qui est représenté en figure 6.2 : des fils d'exécution, ayant chacun leur mémoire privée, les échanges d'informations se faisant uniquement via des messages ponctuels selon des canaux de communication préalablement déclarés.

Puisque chaque acteur possède sa propre mémoire, à laquelle aucun autre ne peut accéder, il est facile au système de par exemple migrer des données. Cependant, la conséquence pour ce modèle est qu'il doit nécessairement travailler par passage de message, puisqu'il n'y a pas d'état partagé entre les fils d'exécution.

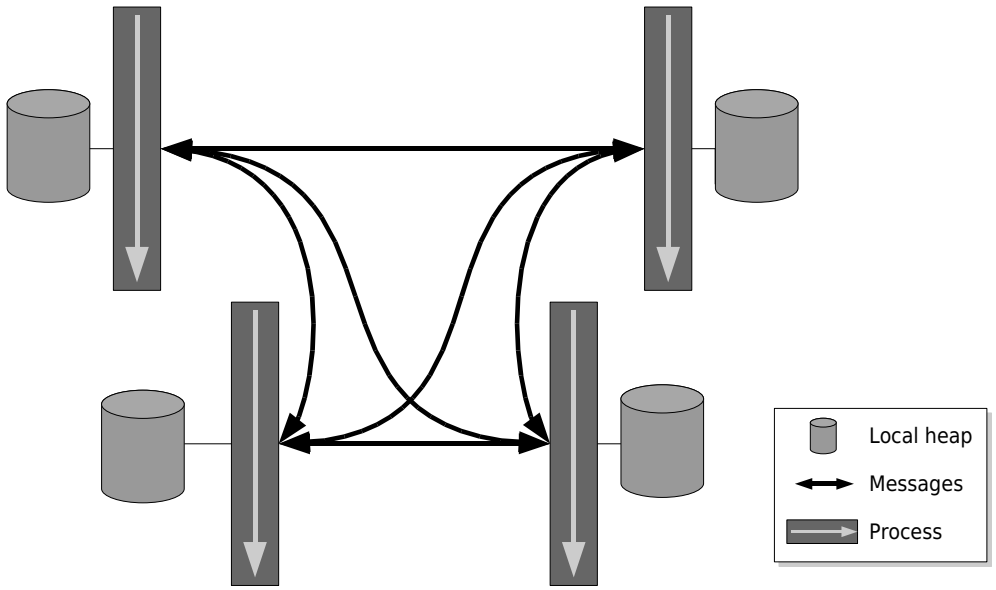


Figure. 6.2 – Vue depuis le système de la distribution de la mémoire entre processus dans le modèle acteur.

Dans ce cas, il n'est donc pas possible d'identifier les données partagées, puisque le passage de message impose de recopier les données lorsque celles-ci doivent être utilisées par plusieurs Acteurs. Cela empêche donc une optimisation simple, évitant des réécritures constantes de données similaires.

6.1.3 Contraintes sur la mémoire imposées par les langages

La mémoire partagée et le modèle acteur s'opposent sur la manière de conserver l'*état* du programme en cours d'exécution entre les fils d'exécution. Cependant, dans les deux cas, le modèle considéré n'est pas idéal. En mémoire partagée, chaque thread a sa propre pile pour conserver un état local. Dans le modèle acteur, les communications entre les acteurs passent souvent par un buffer afin d'essayer de réduire la recopie des données.

Pourtant, beaucoup de langages, aussi bien impératifs que fonctionnels, offrent par défaut une sémantique bien plus forte sur l'accès aux données que celle exploitée dans les modèles précédents, avec très peu d'exceptions. Par exemple, en C++, n'importe quelle zone de code ne peut accéder qu'à un ensemble précis de variables (représentant l'état) :

- les variables locales ;
- les variables d'instance ;
- les variables globales (dont les variables statiques).

De plus, les relations entre les différentes variables existantes sont déterminées directement par les pointeurs présents dans les structures et tableaux. Pour une portée de code donnée, comme par exemple une fonction, il est aisé de voir à l'exécution quelles variables un code est capable d'accéder, à condition d'extraire l'information nécessaire depuis le source.

Notons qu'il est possible en C++ d'accéder à une zone mémoire arbitraire via l'arithmétique de pointeurs. Cependant, cette pratique est habituellement très fortement déconseillée et dans l'immense majorité des cas inutile.

Ainsi, toute la sémantique utilisée pour aider à structurer et concevoir un programme est perdue lors de l'exécution.

6.2 La mémoire par cellules

6.2.1 Principes

Comme présenté dans la section 3.2, l'approche haut-niveau de Capsule propose un modèle intermédiaire entre le modèle de threads à mémoire partagée et le modèle à acteurs. Au lieu d'imposer la relation existante possible entre les éléments d'état (comme les variables) et les fils d'exécution, l'attribution est dynamique.

Chaque fil d'exécution conserve un ensemble de *liens* vers les éléments sur lesquels il travaille à un moment donné ou qu'il utilise. Ces éléments sont appelés *cellules*, ils peuvent représenter soit du code, soit des données, éventuellement les deux. Typiquement une cellule correspond à un élément de structure de données, tel qu'un noeud de graphe. Il peut aussi correspondre à un élément de code tel que l'implémentation d'une fonction ou d'une méthode.

La mémoire existe vis à vis du système comme représenté sur la figure 6.3. Le système connaît explicitement à l'exécution le découpage de la mémoire ainsi que les relations existantes à un moment donné.

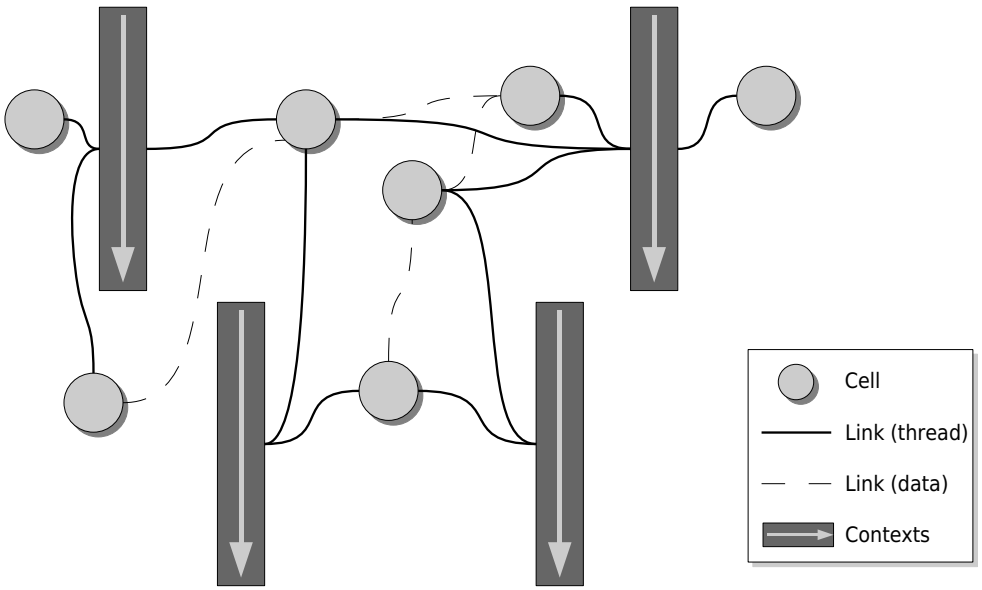


Figure. 6.3 – Vue depuis le système de la répartition des données par rapport aux fils d'exécution dans le modèle haut-niveau de Capsule. La finesse du découpage et d'adressage des données est explicite, au contraire des deux modèles précédents.

Lorsqu'une cellule est utilisée par plusieurs fils d'exécution, cela correspond à un fonctionnement en mémoire partagée ; à l'inverse, une cellule utilisée par un seul fil sera semblable à du modèle acteur.

Un des principes de cette approche est de supprimer, vu du programme, la notion d'adressage global de la mémoire. L'adressage global devient un « détail » d'implémentation, dont le système sous-jacent peut profiter afin d'optimiser l'exécution du code.

Pour le programmeur, l'adressage se fait de proche en proche. Un bloc de code connaît un certain nombre de variables, généralement obtenues via les paramètres d'appel ou par des variables d'instance par exemple. De là, le seul moyen qu'a un bloc de code pour accéder à d'autres données est de suivre les pointeurs depuis les variables auxquelles il a déjà accès. Il n'y a pas de connaissance globale à priori de l'état du système.

6.2.2 Exemple

Cette approche pour structurer la mémoire a l'avantage de correspondre à la manière de programmer dans de nombreux langages. Par exemple, en C ou C++, chaque instance de *struct* ou *class* correspondent à une cellule. Les fonctions et les méthodes correspondent à des cellules de code. Ces fonctions ont une connaissance de la mémoire basée sur les paramètres qu'elles ont reçu explicitement ou implicitement (pointeur *this* vers l'instance en C++). Les liens entre cellules existent également, sous la forme de pointeurs (ou de références en C++).

L'exemple du listing 6.1 représente le code simplifié d'un calcul naïf de plus court chemin entre un noeud source et tous les noeuds d'un graphe.

Listing 6.1 – Exemple d'une recherche simple des plus court chemins d'un graphe.

```
1  struct Edge {
2      Node* head;
3      Node* tail;
4      int length;
5
6      Edge(Node* _tail, Node* _head, int _length);
7  };
8
9  struct Node {
10     std::list<Edge*> edges;
11     int id;
12     int distance;
13     Edge* path;
14
15     Node(int _id) : id(_id), distance(-1), path(this, NULL) {}
16
17     bool update(int newdist, Edge *from) {
18         if ((distance != -1) and (distance < newdist)) return false;
19
20         distance = newdist;
21         path = from;
22         return true;
23     }
24 };
25
```

```
26 void shortest(Node *node, int newdist, Edge *from) {
27     if (not node->update(newdist, from)) return;
28
29     std::list<Edge*>::iterator edge;
30     for ( edge = node->edges.begin();
31           edge != node->edges.end(); edge++ ) {
32         probecount++;
33         shortest((*edge)->head,
34             node->distance + (*edge)->length, *edge);
35     }
36 }
```

Le découpage serait le suivant : chaque instance de *Edge* et de *Node* correspondent, vu de Capsule, à une cellule. Les pointeurs qu'ils contiennent (*head* et *tail* pour *Edge*, *path* et la liste *edges* pour *Node*) représentent les liens existants. En pratique, le code accède déjà aux données de proche en proche ; l'algorithme est initialisé via un appel à la fonction *shortest* prenant en paramètre le noeud de départ.

Afin que le système sous-jacent puisse tirer parti de ce modèle, il est nécessaire que la majorité du code d'un programme le suive. Néanmoins, quelques points apparaissent comme pouvant poser problème si on impose ce modèle dans le code :

- L'arithmétique de pointeurs. En pratique, il est généralement déconseillé d'en faire usage, et une programmation propre n'en a pas besoin. Bien que la machine sous-jacente a probablement toujours une notion d'adresse rendant toujours possible l'arithmétique de pointeur quand nécessaire, dans de nombreux cas elle n'a pas de sens si le système se permet de manipuler librement les cellules.
- La gestion des tableaux. Paradoxalement, cette approche de la mémoire rend plus facile la gestion de structures telles que des arbres ou des graphes qui étaient difficiles à gérer dans un modèle de mémoire linéaire mais rend plus complexe la gestion des tableaux. Cependant, des solutions sont envisageable afin de représenter efficacement les tableaux tout en se basant sur cette notion de cellules. Voir la section 6.4.1 pour plus de détails.
- Chaque variable globale nécessite de systématiquement gérer un lien supplémentaire depuis chaque bloc de code. Une recommandation classique de style de code est d'éviter les variables globales[1] ; en pratique, il est souvent nécessaire d'en avoir quelque-unes [2]. Néamnoins, le compilateur est normalement capable de détecter les zones de code accédant à des variables globales.

6.2.3 Dualité code/données

Bien que les cellules soient intuitivement plus faciles à représenter en termes d'éléments de structures de données, elles peuvent également servir pour du code.

Prenons l'exemple d'une architecture ayant plusieurs accélérateurs matériels, possédant chacun leur propre mémoire. Ces mémoires privées sont adressables depuis le coeur principal et depuis les accélérateurs. À l'inverse, les accélérateurs ne sont pas capables d'accéder à la mémoire « principale » du système, ni d'interagir avec les entrées/sorties. Le processeur « The Cell » [47] suit ce principe.

[1] A noter que le concept de *singleton*, propre à l'objet et souvent présenté comme solution pour éradiquer les variables globales s'implémente également via une variable globale ; le problème est donc similaire.

[2] Capsule utilise par exemple une variable globale pour stocker le pointeur vers la structure principale *capsys_t* ainsi qu'une autre pour accéder rapidement au prédicteur.

Dans ce cadre, il n'est plus seulement important pour la performance d'avoir les bonnes données au bon endroit au bon moment mais également d'avoir le bon code à exécuter dans les mémoires des accélérateurs. Dès lors, il pourrait être intéressant de déporter un traitement sur une de ces unités spécialisées si le code est déjà présent. Dans le cas contraire, la copie du code étant trop coûteuse, il est préférable de continuer l'exécution séquentiellement sur le coeur générique.

6.3 Exemple d'utilisation des cellules pour faciliter le parallélisme

La connaissance des cellules manipulées par un fil d'exécution n'est pas statique, elle évolue au cours de l'exécution du programme. Des données sont « oubliées » par un fil d'exécution, d'autres « découvertes ». Chaque fil d'exécution a une liste de cellules à maintenir qui représente les cellules dont il a connaissance à un moment donné. Mais cette liste peut aller au-delà de l'information binaire « utilise/n'utilise pas ». Chaque morceau de code, lorsqu'il rapatrie une cellule, sait pourquoi il la récupère et donc quel va en être son usage. Cela permet de classer l'usage des données selon des critères potentiellement intéressant pour le système.

Cette connaissance à la compilation peut être utilisée afin d'influencer le comportement à l'exécution, dont notamment la concurrence. Il est possible de distinguer plusieurs types d'usages d'une cellule dans un bloc de code donné :

- écriture : la cellule est lue et modifiée ;
- lecture : la cellule est lue, mais non modifiée ;
- référencement : la cellule est manipulée mais on n'accède pas à son contenu ;
- suppression : la cellule est supprimée et n'existe plus en sortie de bloc.

À l'aide des ces propriétés, encore appelées « tags », il est possible de déterminer les droits des fils d'exécutions pour l'accès concurrent à une même cellule. Les règles de base que doivent suivre les fils d'exécutions sont décrites dans le tableau 6.1.

	Lecture	Écriture	Référencement	Suppression
Lecture	√		√	
Écriture			√	
Référencement	√	√	√	
Suppression				

Table. 6.1 – Compatibilité entre les différents types d'accès possibles pour une cellule donnée. Deux fils d'exécutions peuvent travailler simultanément sur une même cellule si leurs tags sont compatibles.

Cela décrit ce qu'il est possible pour un fil d'exécution donné de faire vis à vis d'une cellule déjà utilisée par un autre fil. Cependant, il est possible d'améliorer cela, notamment en dupliquant une cellule. Les « RCU » ([53], pour *Read Copy Update*, voir section 2.2.3) permettent par exemple d'avoir simultanément des lectures et une écriture, en dupliquant les données temporairement lors de l'écriture.

Le listing 6.2 met en évidence l'ajout de ces tags (ligne marquées d'un ⇒) en suivant une syntaxe C++. Le fonctionnement se base sur le comportement de l'allocation des variables locales en C++ : à l'entrée du bloc de code, le constructeur est appelé et, dès que l'exécution sort du bloc de code, le destructeur est appelé. Ainsi, une déclaration comme celle de la ligne 17 permet d'avoir une variable de type *capsule* :: *Writer* pendant la durée du bloc de code ;

les constructeurs/destructeurs de ce type permettent de tagguer en conséquence la variable. Les déclarations de type *capsule* :: *Reader* correspondent aux accès en lecteur uniquement, les *capsule* :: *Writer* correspondent aux accès en lecture/écriture et les *capsule* :: *Referrer* correspondent au référencement d'une cellule sans accès au contenu.

Listing 6.2 – Exemple du listing 6.1 annoté avec les types d'utilisation de cellules.

```
1  struct Edge : capsule::Cell {
2  Node* head;
3  Node* tail;
4  int length;
5  Edge(Node* _tail, Node* _head, int _length);
6  };
7
8  struct Node : capsule::Cell {
9  std::list<Edge*> edges;
10 int id;
11 int distance;
12 Edge* path;
13
14 Node(int _id) : id(_id), distance(-1), path(this, NULL) {}
15
16 bool update(int newdist, Edge *from) {
17     ⇒capsule::Writer t1(*this);
18     ⇒capsule::Referrer t2(*from);
19
20     if ((distance != -1) and (distance < newdist)) return false;
21     distance = newdist;
22     path = from;
23     return true;
24 }
25 };
26
27 void shortest(Node *node, int newdist, Edge *from) {
28 ⇒capsule::Reader t1(*node);
29 ⇒capsule::Reader t2(node->edges);
30 ⇒capsule::Referer t3(*from);
31
32 if (not node->update(newdist, from)) return;
33
34 std::list<Edge*>::iterator edge;
35 for ( edge = node->edges.begin();
36       edge != node->edges.end(); edge++ ) {
37     ⇒capsule::Reader r2(**edge);
38     probecount++;
39     shortest((*edge)->head,
40         node->distance + (*edge)->length, *edge);
41 }
42 }
```

Considérons maintenant que la boucle de la ligne 35 est en fait une boucle parallèle, qui peut exécuter chaque itération en parallèle.

Si on considère que chaque cellule a un lock simple associé et que les *capsule* :: *Reader* et *capsule* :: *Writer* se contentent chacun de le prendre le lock en début de bloc et de le relâcher en fin de bloc, alors on obtient rapidement le problème de la figure 6.4. Le travail commence sur la cellule 1, il est donc locké. Ensuite, l'algorithme suit les arcs et continue donc sur la cellule 2, qu'il verrouille, puis la 3 où il fait de même. Ensuite, il essaye de suivre le lien de 3 à 1 ; mais comme 1 est déjà verrouillé, il bloque, créant ainsi un deadlock fatal.

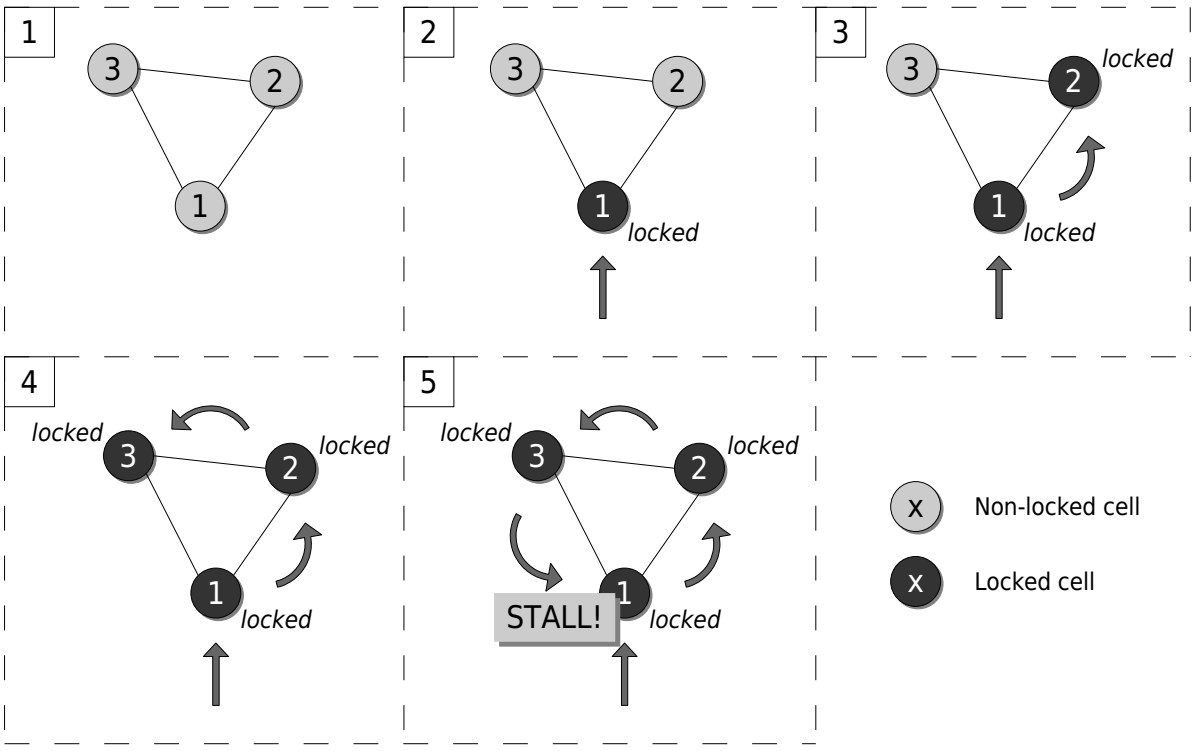


Figure. 6.4 – Utilisation d'un lock simple : problème contre-intuitif simple de réentrance sur un graphe.

Cela est dû au fait que en ligne 28, le lock est pris en début de bloc et n'est relâché qu'à la fin. La récursivité se faisant au sein de ce bloc, le lock n'est pas encore relâché lorsque les autres cellules sont explorées.

Cependant, les tags de la figure 6.2, bien que représentés comme du code « normal », ont été ajouté de manière systématique. Il n'existe pas pour l'instant de pré-processeur ou compilateur adapté, mais ils pourraient être ajoutés automatiquement, sans intervention de l'utilisateur. Le principe est simple : chaque paramètre d'entrée d'une fonction ou d'un bloc de code est tagué en fonction de l'utilisation qui en est faite, chose aisément détectable automatiquement, même si cela peut être parfois plus grossier qu'une annotation manuelle.

Pour le problème de dead-lock, il n'est pas suffisant d'affiner les annotations. Au sein de la boucle, il y a des accès au noeud, notamment pour itérer sur les arcs disponibles. Si le lock est relaché avant l'appel récursif puis repris à son retour, cela règle le problème dans le cas où il n'y pas de parallélisme, mais pas dans le cas contraire.

Pour résoudre cela, au lieu d'associer un simple lock aux cellules, on leur associe un lock

en lecture/écriture, permettant la présence simultanée de plusieurs threads en lecture sur une même cellule, mais un seul en écriture. Les tags servent donc à prendre et à relâcher ce lock en lecture/écriture. En pratique, ces tags apparemment simples permettent de gérer le parallélisme de cet algorithme sans nécessiter de travail supplémentaire de la part du programmeur. Le cas de la figure 6.4 ne se pose plus, puisque la boucle n'a besoin que de locks en lecture et autorisera donc la réentrance.

La zone où il est nécessaire d'avoir un accès en écriture, dans la fonction *Node* :: *update*, est localisée, il n'y a notamment pas de récursion. Cela correspond au style de programmation nécessaire afin que les annotations automatiques en lecture/écriture suffisent pour gérer le parallélisme. Il est nécessaire de conserver les zones en écriture très restreintes, typiquement au sein d'une fonction garantissant la cohérence des données.

Encore une fois, cela a de toute façon l'avantage de correspondre déjà souvent au style de programmation employé, comme par exemple en programmation orientée objet où la cohérence de l'état est garantie par l'appel de méthodes simples. Il est ainsi envisageable d'approcher cela comme un modèle de programmation en soi. Ainsi, l'objectif ne va pas être de rajouter des annotations manuelles supplémentaire afin de gérer les cas difficiles. Il va suffire de se restreindre à ce modèle plus simple, où les modifications de données doivent être localisées. Ne pas le faire reviendrait ainsi à un bug de programmation du programme plutôt qu'une erreur de verrouillage.

6.4 Problèmes d'implémentation

6.4.1 Représentation des tableaux

L'approche de mémoire structurée à base de cellules pose problème pour les tableaux. En effet, le principe d'un tableau, sur un système conventionnel, est que l'accès aux différents éléments se fait en accès aléatoire et qu'il y a donc une connaissance implicite de l'adresse de destination. Le modèle Capsule, par construction, cherche au contraire à éviter cela.

Il y a deux manière d'implémenter directement les tableaux par-dessus les cellules et les liens :

- En allouant une cellule par élément de tableau, chacun ayant un lien vers l'élément suivant. Le surcoût aussi bien en mémoire qu'en temps pour un tableau d'entiers est simplement inacceptable.
- Stocker l'intégralité du tableau dans une seule cellule. On perd ici l'intérêt de la structuration de la mémoire. Par exemple, si chaque élément du tableau est un *struct*, alors ces *struct* n'auront pas de cellule associée et il sera donc impossible de leur assigner des propriétés telles que les locks.

La mémoire non structurée telle qu'existant actuellement favorise la notion de tableau à adressage aléatoire ; ici, l'approche par cellule favorise au contraire les structures à liens plutôt que celles à adressage aléatoire. Il est donc nécessaire d'avoir une approche plus complexe pour gérer efficacement les tableaux, afin d'éviter une perte de performance trop importante lors de la gestion des tableaux.

6.4.2 Gestion de l'adressage

La notion de lien est directement liée à la notion de pointeur. Une implémentation simple et efficace d'un lien au sein d'une cellule est de stocker le pointeur vers l'endroit où est stockée la cellule mémoire visée. Dans un contexte de mémoire partagée, cela est efficace. Cependant, dans le cas d'une mémoire distribuée, cela impose de ne pas déplacer les données puisque sinon cela

invaliderait les pointeurs déjà existants vers la cellule déplacée. Pourtant, il semble intéressant de pouvoir faire bouger les données d'un noeud de calcul à un autre afin d'améliorer la localité des accès mémoire, chose critique dans un cas de mémoire distribuée.

Les problèmes de routage et de placement des données dépendant dynamiquement de l'architecture ainsi que l'exécution réelle. Il est donc bien sûr hors de question de laisser l'utilisateur gérer cela manuellement. Il semble donc nécessaire de présenter un adressage fixe à l'utilisateur. C'est donc le compilateur et le processeur qui devront prendre en charge les problèmes d'adressage.

Cela n'est pas sans rappeler la notion de mémoire virtuelle courante sur les processeurs modernes. La mémoire virtuelle découpe classiquement l'espace mémoire en *pages*, dont la taille est généralement de quelques kilo-octets. Pour gérer cela, le processeur demande au système d'exploitation de maintenir une table des pages, assurant la correspondance entre adresse virtuelle et adresse physique. Ce qui est proposé ici est d'ajouter une couche de virtualisation des adresses, mais à une granularité bien plus fine afin d'en tirer des propriétés intéressantes quant à la gestion du parallélisme. Cependant, les tables de pages actuelle posent déjà souvent un problème dû à l'augmentation de la mémoire disponible sur les machines, entraînant mécaniquement une augmentation des tables de pages. Ainsi, il a été introduit sur architecture x86 la notion de pages de 4Mo afin de relâcher la pression sur la table des pages sur les systèmes avec beaucoup de mémoire. On cherche ici au contraire à diminuer la granularité des éléments de mémoire (cellules vs. pages). Il semble donc impossible de maintenir une équivalence complète et permanente entre adresse de cellule et adresse mémoire afin de pouvoir déplacer les cellules entre les noeuds de mémoire.

Il est à noter que, puisque le système et le processeur ont une connaissance explicite des cellules et des liens, il est possible de réécrire les pointeurs vers les autres cellules lorsqu'une cellule est déplacée. Au sein d'une cellule, le système stockera en entête le nombre de liens contenu dans ladite cellule ; ces liens seront stockés au début de la charge utile. Ainsi, lorsqu'une cellule transitera, il sera aisé de lister les liens et de les réecrire si nécessaire. Néammoins, cela ne résoud pas le problème d'adressage des cellules qui sont déplacées de noeud mémoire puisqu'il n'est pas possible de lister toutes les cellules pointant vers une autre (liens inverses).

Chapitre 7

Conclusion

Tirer parti de l'existant

Capsule, en touchant à plusieurs niveaux de la chaîne de programmation et d'exécution, montre qu'il est possible avec peu de changements d'obtenir beaucoup d'informations utiles à l'optimisation. Même si cela ne résout probablement pas tous les problèmes liés au parallélisme, on peut voir qu'il n'est pas forcément nécessaire de tout remettre en cause mais qu'il peut suffire simplement de trouver les bonnes propriétés à exploiter.

Une comparaison est possible avec le phénomène du passage de l'assembleur aux langages dits structurés, tels que le C. La différence entre assembleur et langage structuré n'est pas nécessairement énorme ; il est d'ailleurs de plus en plus courant d'entendre dire que le C n'est qu'une sur-couche syntaxique légère à l'assembleur. En effet, pour des programmes similaires, on ne code guère différemment entre de l'assembleur et du C. Bien sûr, ce n'est plus vrai aujourd'hui, puisque justement l'assembleur est uniquement utilisé lorsqu'il n'est pas possible d'utiliser proprement du C. Mais lorsque le C est apparu, ce n'était pas le cas. Lors de l'écriture de programmes en assembleur, des règles d'écriture permettant de se faciliter la tâche étaient souvent adoptées. On peut par exemple mentionner la notion de sous programme ne modifiant qu'un ensemble précis de registres. La mise en place d'un cadre de pile lors de l'entrée dans ces sous-programmes permettait d'accéder facilement à variables dites « locales » et autorisait également la récursivité ou la réentrance. Les codes dit « spaghettis » étaient évités en limitant ce qu'il était possible de faire avec les sauts : utilisation de sous-routines avec un point d'entrée unique, interdiction des sauts au milieu d'un traitement. En fait, on se rend compte que tout ce qu'ont fait les langages dit structurés a été de reprendre toutes ces habitudes et lignes de conduites et de les rendre obligatoire par construction. Il devenait alors possible de se baser dessus pour, par exemple, simplifier la syntaxe, améliorer la détection d'erreurs ou permettre des optimisations automatiques. Bien sûr, toute la difficulté était de savoir quelles propriétés rendre obligatoires, ce qui explique pourquoi les langages structurés ne sont pas apparus immédiatement. En rendant trop de propriétés obligatoires, on empêche l'écriture de certains types de codes et on rend le langage spécifique à un domaine. En conservant trop de libertés, il n'est plus intéressant de faire évoluer le langage avec tous les coûts que cela engendre. L'équilibre est subtil.

Un exemple similaire est l'apparition des langages orientés-objet. On peut prendre pour exemple l'API POSIX qui, bien que datant d'avant la notion d'orienté objet, a une notion primitive d'objet. L'état est encapsulé dans une structure ; un ensemble de fonctions permettent de travailler avec. Les langages orientés-objet n'ont fait que reprendre cette habitude (intuitive il pourrait sembler) de programmation.

Ainsi, la bonne approche vis-à-vis de l'évolution des architectures et du parallélisme n'est

pas nécessairement de chercher des paradigmes exotiques, mais peut-être d'essayer de prendre du recul et de voir qu'est-ce que l'on pourrait exploiter dans ce que l'on fait déjà de manière intuitive en écrivant du code parallèle.

Évolutions de Capsule

Deux grands axes peuvent être considérés pour les futurs développements de Capsule. D'une part, Capsule offre une abstraction de l'architecture. Il est possible de profiter de cela afin d'exploiter efficacement des puces avec accélérateurs dédiés ou des puces pour les plateformes embarquées. D'autre part, le principe de division conditionnelle de Capsule est générique. Il est donc intéressant de voir comment cela peut s'intégrer dans d'autres environnements de programmation ou frameworks parallèles.

Prise en charge d'architectures variées

L'augmentation du nombre de coeurs n'implique pas nécessairement de conserver une puce régulière, où chaque coeur est une copie conforme du précédent. Il est également possible de concevoir des multi-coeurs hétérogènes, tels que les systèmes sur puce utilisés en embarqué. Le principe est simple dans ce cas : comme il est souvent aisé d'obtenir du parallélisme pour des types de traitement assez spécifiques, tel que le classique décodage vidéo, la spécialisation des coeurs va permettre d'exploiter les transistors disponibles.

Dans le cas des systèmes sur puce, les coeurs hétérogènes accompagnant le processeur principal sont le plus souvent dédiés à un problème donné : décodage vidéo, communications bluetooth, etc. Cependant, on commence à voir apparaître des processeurs hétérogènes dont les différents coeurs restent généralistes, mais présentant un modèle de programmation différent. Ainsi, le Cell, mentionné à la section 2.1.2, propose deux types de coeurs sur une même puce : d'un côté les PPE, qui sont des coeurs généralistes type POWER et qui présentent un modèle type von-Neumann, et d'un autre côté les SPE, coeurs vectoriels à gestion de cache explicite.

Les coeurs d'un processeur hétérogène vont donc être adaptés chacun à certains types d'algorithmes et de traitements. Cela impose de choisir où exécuter les différentes parties du programme, et, le cas échéant, d'adapter le code afin qu'il puisse fonctionner sur le type de coeur choisi. Cela impose également de connaître le déroulement précis du programme, puisque le nombre de processeurs spécialisés est nécessairement limité. Il va donc falloir éviter d'avoir trop de tâches les nécessitant simultanément. Dans le cas de programmes multi-threads, cette propriété n'est pas nécessairement facile à obtenir.

La division conditionnelle de Capsule pourrait permettre de prendre en compte le type de code à exécuter dans le nouveau thread. Ainsi, si le programme est compilé pour une architecture hétérogène, la division conditionnelle pourra essayer d'allouer un coeur spécialisé pour l'opération demandée. En pratique, ce mécanisme pourrait tirer parti du fait que le code à exécuter dans le nouveau thread est déjà indiqué en paramètre du probe. Il serait donc possible de faire en sorte que le compilateur génère une version vectorielle du code. Ainsi le probe ne demande pas un coeur généraliste mais uniquement un coeur vectoriel. Si ce dernier n'est pas disponible, la création de thread est refusée.

Cependant, le succès de CUDA, surtout vis-à-vis du Cell, pose la question de l'intérêt d'avoir de l'hétérogénéité sur une même puce. CUDA propose un modèle adapté au calcul vectoriel aisément parallélisable (comme typiquement les traitements graphiques) à l'instar du Cell. Néanmoins, CUDA a été conçu dès le départ pour fonctionner sur les GPU, qui sont des puces séparées

du processeur principal [1], cela étant donc opposé au Cell qui intègre tout sur une même puce.

Les deux approches présentent bien évidemment des avantages et inconvénients. Tandis que l'intégration sur une même puce permet d'optimiser les communications entre les types de coeurs, comme par exemple pour la gestion de la concurrence, des puces séparées peuvent profiter d'une mémoire dédiée séparée. À l'inverse, un processeur hétérogène doit être capable de fournir suffisamment de bande passante mémoire pour tous les coeurs tandis que plusieurs processeurs séparés subissent un temps de communication élevé pour les synchronisations avec le coeur principal.

CUDA et le Cell sont conçus pour des types de calculs similaires : énormément de parallélisme, des traitements similaires entre les threads et du travail sur des données locales ou par flot. L'existence actuelle de CUDA semble montrer que les inconvénients de la puce dédiée ne sont pas gênants pour ces types de problèmes. On peut donc se demander quelle est la place des processeurs hétérogènes et si des modèles de calcul permettront d'en exploiter les avantages.

Mais Capsule peut de toute façon s'adapter également à des systèmes multi-processeurs hétérogènes. En effet, comme pour le cas des puces hétérogènes, Capsule est capable de déterminer lors de l'exécution s'il est intéressant d'utiliser une puce distante. Or, les communications pour arbitrer la décision de division sont très légères, ne nécessitant que quelques allers/retours. Il est donc possible d'envisager un canal de communication simple, à bande passante faible, mais à haute priorité par rapport aux autres bus et à faible latence. Ce canal permettrait de déterminer efficacement l'intérêt d'une division ; Capsule de son côté est capable de prendre en compte dans ses choix les autres coûts associés à une division, comme la nécessité de recopier des données.

Intégration de Capsule

Autres environnements

Bien que la version de Capsule décrite dans ce document soit fortement basée sur le C, le principe de division conditionnelle est générique. Il serait intéressant, et utile, d'intégrer cela dans d'autres environnement de programmation. Cette division peut avoir sa place aussi bien dans des langages compilés comme C++ que dans des langages fonctionnant sur des machines virtuelles ou de script.

Une version C++ de Capsule a été commencé au cours de cette thèse. Mais on peut également remarquer que Qt [5], framework populaire fournissant un vaste ensemble de fonctionnalités, dont des éléments d'interface graphiques, propose maintenant un ensemble de classes et fonctions pour le parallélisme. Cette API, nommée « Qt Concurrent » [4], est capable de s'adapter au nombre de coeurs disponibles via des fonctions telles que « filter » ou « map ». Il serait intéressant d'intégrer Capsule dans Qt, aussi bien en tant que base pour l'implémentation des primitives parallèles déjà existantes, qu'en tant que fonctionnalité mise directement à disposition du programmeur.

Les langages de scripts et/ou faisant appel à une machine virtuelle, tels que Python ou le « Common Language Infrastructure » de Microsoft, ont généralement d'autres priorités que la performance. Il est cependant quand même possible de porter Capsule sur ces types de langages. L'implémentation de la division doit être performante, et serait typiquement implémentée au niveau de la machine virtuelle ou de l'interpréteur. Par contre, l'utilisation par le programmeur de la division n'a pas de contrainte intrinsèque de performance. Il est donc possible de manipuler la division depuis ces langages pour paralléliser du code. La division devra juste prendre en compte les surcoûts spécifiques au langage, comme par exemple un coût de création de contexte

[1] CUDA est un modèle de machine virtuelle, on pourrait donc néammoins imaginer faire fonctionner du code CUDA sur le Cell.

élevé.

Chaîne de compilation

L'intégration des principes de Capsule ne se fait pas nécessairement uniquement au niveau d'un langage mais peut également s'imaginer au coeur même de la chaîne de compilation.

On remarque que la plupart des frameworks parallèles fournissent soit une API, soit un langage. Les constructions fournies sont donc figées et limitées à ce qu'a prévu le système considéré. Or, les frameworks spécialisés simples d'utilisation montrent qu'un langage adapté peut permettre de construire un système parallèle simplement. Il semblerait donc être intéressant, pour fournir un système parallèle générique, de travailler au niveau transformation de code / compilation. Autrement dit, au lieu d'avoir un framework parallèle fournissant du code permettant de manipuler des données en parallèle, il serait peut être intéressant de voir cela comme un outil prenant du code en entrée afin de générer du code parallèle.

Prenons l'exemple des objets protégés, où une seule méthode d'une instance peut être en cours d'exécution à un moment donné. Si on veut pouvoir implémenter cela avec un framework parallèle tel que pthread, il est nécessaire d'explicitement prendre et relâcher un lock pour chacune des méthodes, sachant que le moindre oubli est probablement fatal pour l'application, tout en étant difficile à détecter. À l'inverse, s'il est possible de modifier la chaîne de compilation, il suffit alors de créer un type « objet protégé », où les locks sont pris et relâchés automatiquement.

Bien sûr, il existe des langages ayant, de base, des objets protégés. Mais il ne s'agit que d'un exemple. Les constructions permettant de simplifier l'écriture de programmes parallèles sont probablement infinies et peuvent varier subtilement d'un programme à un autre. Il semble donc intéressant de pouvoir modifier la chaîne de transformation/compilation en fonction du programme. Cela permettrait d'exprimer le code de chaque programme d'une manière adaptée à ses algorithmes et structures. Cela voudrait dire qu'un programme aurait deux niveaux : d'un côté le code proprement dit, décrivant comment gérer les données d'entrée du programme, et d'un autre la description de la transformation du code en quelque chose d'exécutable.

Capsule pourrait s'intégrer dans une telle approche. Par exemple, les premières versions de Capsule étaient en fait des générateurs de code, créant différents morceaux de code pour les cas division ratée, division réussie / premier thread et division réussie / deuxième thread. Il serait donc possible d'étendre Capsule afin de mieux contrôler ce processus de génération de code pour, par exemple, écrire automatiquement le cas où la division a échoué. D'autre part, le mécanisme de division conditionnelle de Capsule, ainsi que les accélérations matérielles seraient probablement intéressantes à reprendre pour d'autres modèles de parallélisme.

Nouveaux besoins

La question de l'usage de la puissance de calcul offerte par la multiplication des coeurs se pose également. L'apparition de processeurs plus puissants donne généralement lieu à la création de nouveaux types d'applications et de nouvelles fonctionnalités. Cela est particulièrement vrai pour les logiciels grand public. Aller plus vite sur une fonctionnalité n'est souvent guère intéressant en soi, les logiciels étant taillés pour une certaine puissance disponible. Par exemple, les premiers correcteurs orthographiques nécessitaient une longue passe sur le texte afin de l'analyser et d'identifier les erreurs. Il est maintenant possible de faire cela à la volée, ce qui modifie dès lors son utilisation. Plus récemment, il a été possible d'introduire des effets graphiques plus évolués dans le système de fenêtrage. Bien que les effets en question sont possibles en temps réel depuis

longtemps, la puissance disponible les rend suffisamment peu coûteux pour qu'ils n'accaparent pas tout le processeur.

Quelles fonctionnalités vont être rendues possibles par la puissance offerte par les processeurs multi-coeurs ? Les technologies web côté client, en plein développement, semblent des candidats naturels. Par exemple, de nombreuses fonctionnalités et applications sont désormais réécrites en Javascript, avec une architecture client/serveur, afin de pouvoir les utiliser à travers un navigateur web. Cela permet de les utiliser de n'importe où mais facilite également les échanges et partages entre plusieurs personnes. Dans ce cas comme dans d'autres, il est donc nécessaire d'avoir une machine virtuelle pour l'exécution, afin de notamment faciliter l'isolation. Comment mettre la puissance des multi-coeurs à profit pour ce type de tâches ?

Bibliographie

[1] GOMP - an OpenMP implementation for GCC. http://gcc.gnu.org/projects/gomp/.

[2] H.264 : Advanced video coding for generic audiovisual services. http://www.itu.int/rec/T-REC-H.264.

[3] Intel's threading building blocks reference manual. http://www.threadingbuildingblocks.org/uploads/81/91/LatestOpenSourceDocumentation/ReferenceManual(OpenSource).pdf.

[4] Qt concurrent. http://labs.trolltech.com/page/Projects/Threads/QtConcurrent.

[5] Qt cross-platform application framework. http://trolltech.com/products/qt/.

[6] Threading building blocks. http://www.threadingbuildingblocks.org/.

[7] Donald Alpert and Dror Avnon. Architecture of the pentium microprocessor. *IEEE Micro*, 13(3) :11–21, 1993.

[8] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th international conference on Supercomputing*, pages 1–6. ACM Press, 1990.

[9] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL : A Standard Template Adaptive Parallel C++ Library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, July 2001.

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[11] K. Arvind and Rishiyur S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39(3) :300–318, 1990.

[12] H. P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[13] Andrew D. Birrell. An introduction to programming with c# threads.

[14] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3) :560–599, 1984.

[15] Ian Buck. Gpu computing : Programming a massively parallel processor. In *CGO '07 : Proceedings of the International Symposium on Code Generation and Optimization*, page 17, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Greg Burns, Raja Daoud, and James Vaigl. LAM : An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[17] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *FPL*, pages 605–614, 2000.

[18] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell : a status report. In *DAMP '07 : Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM Press.

[19] Robit Chandra, Anoop Gupta, and John L. Hennessy. Cool : a language for parallel programming. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 126–148, London, UK, UK, 1990. Pitman Publishing.

[20] J. Chaoui, K. Cyr, J.P. Giacalone, S. Gregorio, Y. Masse, Y. Muthusamy, T. Spits, M. Budagavi, and J. Webb. "OMAP : Enabling multimedia applications in third generation (3g) wireless terminals, December 2000.

[21] Sylvain Conchon and Fabrice Le Fessant. Jocaml : Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[22] Intel Corp. Next leap in microprocessor architecture : Intel core duo.

[23] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10) :667–668, 1971.

[24] David E. Culler, Klaus Erik Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. Technical Report UCB/CSD-92-716, EECS Department, University of California, Berkeley, 1992.

[25] Leonardo Dagum and Ramesh Menon. Openmp : An industry- standard api for shared-memory programming. *IEEE COMPUTATIONAL SCIENCE & ENGINEERING*, pages 46–55, 1998.

[26] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : simplified data processing on large clusters. In *OSDI'04 : Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[28] Edsger W. Dijkstra. Over seinpalen. circulated privately, n.d.

[29] U. Drepper and I. Molnar. The Native POSIX Thread Library for Linux. *White Paper, Red Hat, Fevereiro de*, 2003.

[30] A. Dunkels and O. Schmidt. Protothreads – lightweight stackless threads in c.

[31] John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro*, 15(2) :33–43, 1995.

[32] Ralf S. Engelschall. GNU Pth – the GNU portable threads.

[33] Jason Evans. Kernel-scheduled entities for freebsd, nov 2000.

[34] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '83 : Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press.

[35] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open

MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[36] S. Ghemawat and P. Menage. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[37] J-L. Giavitto and O. Michel. MGS : a rule-based programming language for complex objects and collections. *Electronic Notes in Theoretical Computer Science*, 59(4), 2001.

[38] Wolfram Gloger and Doug Lea. PTmalloc, a fast, memory-efficient implementation of malloc for Unix systems. http://www.malloc.de/en/.

[39] F. Gruau and P. Malbos. The Blob : A basic topological concept for hardware-free distributed computation. In *Unconventional Models of Computation (UMC'02), Kobe, Japan*, pages 151–163, 2002.

[40] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[41] Maurice Herlihy and J. Eliot B. Moss. Transactional memory : Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[42] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[43] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[44] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps : caches as a shared resource. In *PACT '06 : Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22, New York, NY, USA, 2006. ACM Press.

[45] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. A history of haskell : being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

[46] Paul Hyde. Java thread programming. http://java.sun.com/developer/Books/javaprogramming/threads/chap13.pdf.

[47] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5) :589–604, 2005.

[48] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara : A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2) :21–29, 2005.

[49] David Koufaty and Deborah T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2) :56–65, 2003.

[50] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, 34(2) :42–55, 2000.

[51] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. Openmosix, openssi and kerrighed : a comparative study. *ccgrid*, 2 :1016–1023, 2005.

[52] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison-Wesley Professional, September 2004.

[53] Paul E. McKenney and John D. Slingwine. Read-copy update : Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[54] Frank Mueller. A library implementation of POSIX threads under Unix. In *Proceedings of theWinter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, 1993.

[55] Dorit Naishlos. Autovectorization in GCC. In *Autovectorization in GCC*, pages 105–117, 2004.

[56] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[57] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1) :80–113, 2007.

[58] Peter S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[59] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *ISCA*, pages 206–218, 1997.

[60] G. M. Papadopoulos and D. E. Culler. Monsoon : an explicit token-store architecture. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 82–91. ACM Press, 1990.

[61] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02 : Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712, New York, NY, USA, 2002. ACM Press.

[62] J. Rabaey, K. Keutzer, M. Sheets, S. Malik, A. Mihal, A. Sangiovanni-Vencentelli, and M. Sgroi. Addressing the system-on-a-chip interconnect woes through communication-based eesign. *dac*, 00 :667–672, 2001.

[63] Patrick Rogers. Book review : Concurrency in ada, by alan burns and andy wellings. *Ada Lett.*, XVII(6) :108, 1997.

[64] Davide Sangiorgi and David Walker. *PI-Calculus : A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[65] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

[66] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[67] Andrew Gordon Simon Peyton Jones and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[68] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA '02 : Proceedings of the 29th annual international symposium on Computer architecture*, pages 25–34, Washington, DC, USA, 2002. IEEE Computer Society.

[69] R. Stallman. Using the gnu compiler collection, 2006.

[70] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Hewlett-Packard Laboratories, 1994.

[71] W.R. Stevens et al. *Advanced programming in the UNIX environment.* Addison-Wesley, 1992.

[72] Janice M. Stone and Robert P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 15(2) :50–58, 1995.

[73] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[74] Michael Taylor, Walter Lee, Jason Miller, David Wentzlaff, Benjamin Greenwald, Volker Strumpen, Nathan Shnidman, Ian Bratt, Henry Hoffmann, Jason Kim, Arvind Saraf James Psota, Jonathan, Matthew Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor : An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st annual international symposium on Computer architecture*. ACM Press, 2004.

[75] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt : A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[76] P.W. Trinder, H-W. Loidl, E. Barry Jr., K. Hammond, U. Klusik, S.L. Peyton Jones, and Á.J. Rebón Portillo. The Multi-Architecture Performance of the Parallel Functional Language GPH. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, LNCS, Munich, Germany, 29.8.-1.9., 2000. Springer-Verlag.

[77] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice : instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual International Symposium on Computer Architecture*, pages 191–202. ACM Press, 1996.

[78] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading : maximizing on-chip parallelism. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 392–403. ACM Press, 1995.

[79] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, page 54. IEEE Computer Society, 1999.

[80] Eric Tune, Rakesh Kumar, Dean Tullsen, and Brad Calder. Balanced Multithreading : Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Dec. 2004.

[81] P. H. Welch. An occam approach to transputer engineering. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 138–147, New York, NY, USA, 1988. ACM Press.

[82] C Whitby-Strevens. The transputer. pages 320–328, 1986.

[83] Colin Whitby-Strevens. Transputers-past, present and future. *IEEE Micro*, 10(6) :16–19, 76–82, 1990.

[84] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4) :38–59, 1994.

[85] G. William and E. Lusk. User's guide for mpich, a portable implementation of mpi.