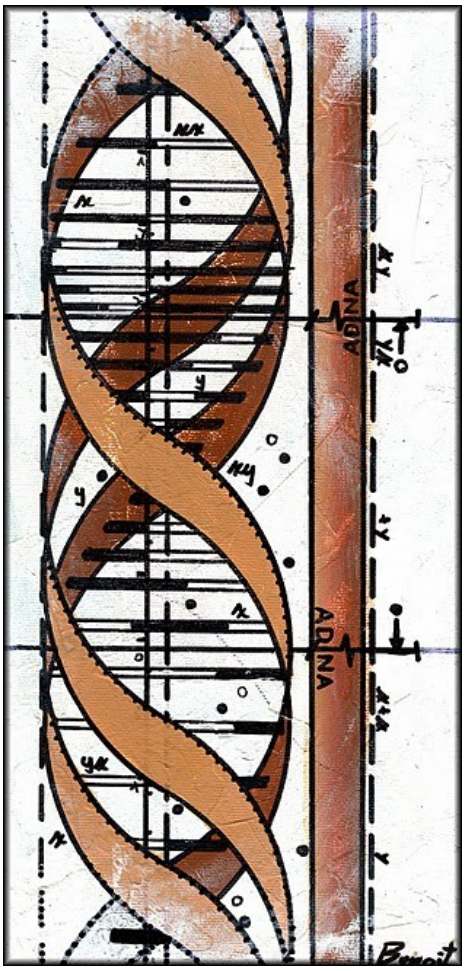
UMONS
Université de Mons

Faculté Polytechnique



Conception et mise en oeuvre d'algorithmes de sélection de ressources dans un environnement informatique hétérogène multi-processeur.

Application à un logiciel de bio-informatique.

Travail de fin d'études
Présenté en vue de l'obtention du grade de Master ingénieur civil en Informatique et Gestion

Sébastien FRÉMAL

POLYTECH MONS

Sous la direction de
Monsieur le Professeur Pierre MANNEBACK
Monsieur Michel BAGEIN

Juin 2010

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Table des matières

Remerciements

En préambule à ce rapport, je souhaite adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce travail.

Je tiens à remercier mes promoteurs, M. Manneback et M. Bagein, qui se sont toujours montrés disponibles et à l'écoute, pour la richesse des enseignements et des échanges tout au long de la réalisation de ce travail, ainsi que pour le temps qu'ils m'ont consacré pour m'aider à finaliser le présent rapport malgré leur charge académique et professionnelle, et les horaires parfois un peu décalés.

Mes remerciements vont également à mes rapporteurs qui me font l'honneur d'accepter de lire et juger ce rapport.

Je remercie également les personnes du service informatique, M. Noël, M. Guttadauria, M. Mahmoudi, et M. Lecron qui m'ont tous accueilli chaleureusement dans le service et qui m'ont aidé à de nombreuses occasions.

Mes remerciements s'adressent également à M. Augonnet, pour les longues heures qu'il a passées à m'enseigner l'utilisation de StarPU et des GPU, ainsi que le temps qu'il a passé pour m'aider à rédiger le chapitre 3 de ce rapport, et ce malgré sa charge professionnelle et l'heure parfois tardive.

Je voudrais également remercier M. Mouton et M. Cassano, tous deux membres du Centre d'Excellence en Technologies de l'Information et de la Communication, pour leur accueil au sein du centre et pour l'aide qu'ils ont pu m'apporter.

J'exprime ma gratitude envers M. Szöke, représentant de la société BioAware pour l'algorithme DNARun, pour m'avoir accordé son temps pour m'aider à rédiger le chapitre 2.

Je n'oublie bien sûr pas ma famillle et mes amis proches pour leur contribution, leur soutien et leur patience.

Introduction

La programmation parallèle permet d'exécuter de nombreux calculs simultanément sur différents nœuds de calcul [1]. Elle s'appuie sur le principe que les grands problèmes peuvent souvent être divisés en de plus petits qui sont alors traités simultanément, en parallèle. L'intérêt pour cette technologie a crû à partir de l'année 2004. Jusque là, les améliorations des performances étaient surtout dues à l'augmentation de la fréquence des processeurs, cette dernière déterminant le nombre d'instructions, d'opérations, qui peuvent être traitées à la seconde. Le problème est que la consommation de puissance, et donc la génération de chaleur, est proportionnelle à cette fréquence et a ainsi atteint un point critique en 2004. Plutôt que de continuer à tenter d'augmenter les fréquences, les constructeurs se sont tournés vers la programmation parallèle qui leur permet de porter leurs calculs sur plusieurs coeurs de calcul. Il n'est d'ailleurs plus rare que les ordinateurs personnels possèdent un processeur comportant deux coeurs de calcul.

Des outils ont été développés afin de distribuer aisément et efficacement les tâches sur les différents coeurs de l'unité centrale de traitement, puis d'autres ont vu le jour afin de distribuer ces mêmes tâches sur un ensemble de machines (cluster de serveurs). Actuellement, il est possible de distribuer un calcul simultanément sur un cluster se trouvant à Paris et sur un autre se trouvant à Bruxelles. Cette méthode permet ainsi de pouvoir utiliser un grand nombre de coeurs de calcul en employant les processeurs de nombreuses machines.

Ces dernières années, une nouvelle manière d'optimiser le temps d'exécution a vu le jour : la programmation parallèle sur accélérateur graphique. Ces derniers sont des processeurs à plusieurs multi-coeurs pouvant posséder plusieurs centaines de cœurs de calcul. Cette technologie permet donc de traiter rapidement une très grande quantité de données.

Leur seule utilisation permet déjà une amélioration des performances, et nous désirons à présent les utiliser conjointement avec les coeurs des unités centrales de traitement. Peu d'outils permettant la parallélisation d'un code sur les différents types de coeurs ont été à ce jour développés. L'un de ces rares outils, la librairie StarPU [3], est développé au sein de l'INRIA[1] [2].

Le but de ce TFE est d'étudier, au sein de la librairie StarPU, l'efficacité des algorithmes sélectionnant les ressources, les processeurs hétérogènes, pour leur assigner les différentes parties d'un même calcul. Pour que cela ne reste pas cantonné à un travail purement académique, il a été décidé que StarPU soit intégré à l'application BioloMICS [4], développée par la société BioAware [5], afin de l'optimiser. Cette application appartient au domaine de la bio-informatique.

Le premier chapitre de ce rapport présente les processeurs graphiques, ainsi que les autres processeurs hautement multi-coeurs. Les performances des processeurs graphiques y sont évaluées à travers l'exemple du tri d'un vecteur. Cet exemple a été choisi car les tris sont les opérations qui prennent le plus de temps lors de l'exécution de l'algorithme présent au sein de BioloMICS. Le

[1] Institut National de Recherche en Informatique et Automatique

second chapitre présente le séquençage de l'ADN, contexte dans lequel l'application BioloMICS a été développée, ainsi que les actions qui ont été menées au niveau de l'algorithme. Enfin, le dernier chapitre présente le fonctionnement du support unifié StarPU ainsi que son intégration à l'algorithme de tri du premier chapitre.

Les annexes contiennent une étude de certains outils libres permettant la distribution de tâches de calcul sur des clusters, un dictionnaire comprenant les définitions de certains termes techniques abordés dans ce rapport et enfin une liste d'acronymes associés à leur signification.

Chapitre 1

L'Unité de Traitement Graphique

1.1 Généralités

Les processeurs graphiques, également appelés accélérateurs graphiques, ou GPU[1], sont les processeurs de nos cartes graphiques. Ces dernières servent, à l'origine, à décharger le CPU[2] de la charge de travail concernant le traitement graphique 2D et 3D. Le premier accélérateur graphique fut conçu en 1984 par IBM [6]. Comme un ordinateur miniature, [7] il possédait un CPU, de la mémoire vive, de la mémoire morte ... Au fil du temps, l'architecture des processeurs graphiques a fort évolué afin de se spécialiser dans le traitement de la géométrie et du rendu. Les processeurs sont maintenant équipés de plusieurs centaines de coeurs de calcul possédant leur propre mémoire, travaillant en parallèle et permettant ainsi d'atteindre une puissance théorique de plusieurs centaines de Gflops [8]. Le lecteur voulant avoir plus d'informations sur l'architecture et le fonctionnement d'un GPU est invité à consulter le dossier [9].

1.2 Vers le calcul générique sur les processeurs graphiques

Le premier paragraphe de cette section se base sur les informations issues de [10].

Le GPU sert à la base à traiter les pixels à afficher à l'écran qui peuvent atteindre un taux de rafraichissement de 75 à 100 Hz pour des résolutions de 2560x1600 pixels, ce qui demande beaucoup de calculs dans le cas des applications pour lesquelles les images changent rapidement. Les processeurs graphiques ont donc été conçus pour que leurs coeurs exécutent toujours les mêmes opérations sur un très grand ensemble de données et suivent donc une architecture SIMD[3]. Cette architecture se porte naturellement candidate de premier choix pour les applications nécessitant d'être parallélisées : il n'est plus question d'effectuer les opérations en même temps sur deux ou quatre coeurs, comme dans le cas des CPU, mais sur plusieurs centaines de coeurs. Par exemple, dans le cas du GPU Tesla T10 du fabricant NVIDIA, processeur présent en 2010 sur le cluster de la Faculté Polytechnique de l'Université de Mons, le nombre de coeurs s'élève à 240, comme vous pouvez le constater sur la figure 1.1. Un nouveau type de programmation est donc apparu au cours des années 2000 : le GPGPU[4] qui tend à utiliser le GPU non plus expressement pour des calculs de rendu 3D, mais pour paralléliser les applications à calculs intensifs et parallélisables massivement comme les heuristiques génétiques, les algorithmes génétiques de comparaison de chaines d'ADN, la cryptographie, les simulations Monte-Carlo, le traitement d'images ...

[1] De l'anglais Graphics Processing Units, signifiant *Unité de Traitement Graphique*
[2] De l'anglais Central Processing Unit, signifiant Unité Centrale de Traitement
[3] De l'anglais Single Instruction, Multiple Data, signifiant *Instruction Unique, Données Multiples*
[4] De l'anglais General-Purpose Computing GPU, signifiant le Calcul Générique sur les Processeurs Graphiques

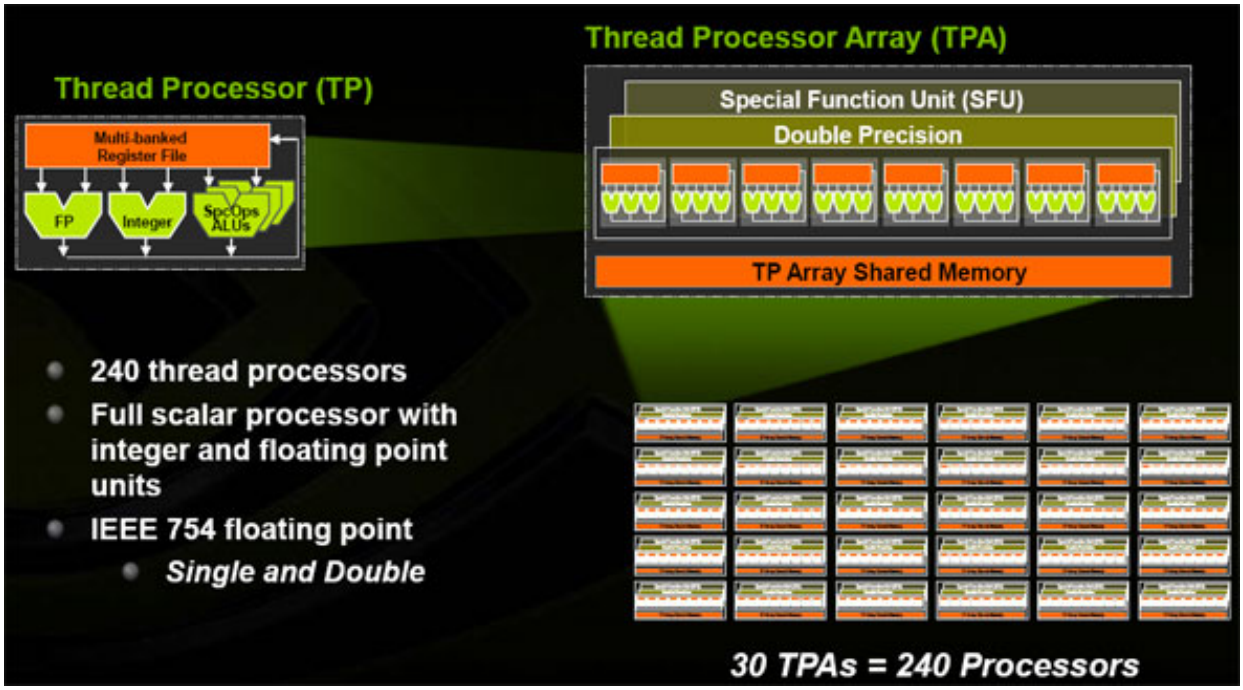


FIG. 1.1 – Spécifications du GPU Tesla T10 de NVIDIA présent dans le cluster de la FPMs (provenance [11])

Le GPU ne permet pas d'exécuter du code écrit en Java, en Python ou en C++. Pour pratiquer le GPGPU, l'utilisateur doit se familiariser avec les langages de programmation prévus pour les GPU. Vous pouvez trouver dans le travail de M. Leclercq [10] une comparaison des différentes manières d'exploiter un GPU. Les langages les plus exploités à l'heure actuelle sont le CUDA et OpenCL qui sont tous deux des standards de programmation. Ils comportent à l'heure actuelle deux grandes différences :

1. OpenCL est un standard ouvert développé par le groupe Khronos [13] alors que CUDA, développé par NVIDIA [12], ne l'est pas.
2. Alors qu'OpenCL peut fonctionner sur n'importe quel processeur (GPU, CPU, CELL 1.5 ...) tant que ce dernier possède le driver correspondant [14], CUDA ne fonctionne que sur les GPUs NVIDIA [15].

OpenCL semble donc plus avantageux que CUDA, j'ai cependant choisi d'utiliser CUDA dans le cadre de ce TFE pour deux raisons :

1. Pour le moment, il n'est pas aisé de trouver un document qui explique comment programmer avec OpenCL. Les ressources mises à disposition sont principalement des exemples et quelques courts tutoriaux. Le but du TFE étant d'étudier l'utilisation de clusters multi-coeurs hétérogènes, j'ai donc préféré consacrer mon temps à cette étude plutôt qu'à celle d'OpenCL.
2. Le TFE m'a amené à manier l'outil StarPU, dont la présentation est faite dans le chapitre 3, car il permet d'employer simultanément les CPUs et les GPUs de manière optimisée. Ce dernier n'étant pas encore entièrement compatible avec OpenCL, il était préférable d'utiliser CUDA.

Comme certaines parties de ce TFE concernent l'implémentation de fonctions en CUDA, je vais en présenter brièvement les principes dans la section suivante.

1.3 Introduction à CUDA

Cette section se base sur les informations issues de [16].

Le CUDA se base sur le langage C et lui ajoute quelques spécifications afin de pouvoir tourner sur un GPU. Le programme tournant sur l'hôte (le CPU) et écrit en C, ou en C++ fait des appels à des kernels[5], les fonctions s'exécutant sur le périphérique (le GPU) et écrites en CUDA. Dès qu'un kernel a terminé son exécution, il rend la main au CPU qui continue à travailler.

Comme vous pouvez le voir sur la figure 1.2, CUDA représente le GPU sous la forme d'une grille divisée en un ensemble de blocs. Dans chaque bloc se trouvent les threads qui effectuent tous le même travail, mais sur des données différentes. C'est le développeur qui spécifie le nombre de blocs qui doivent être lancés ainsi que le nombre de threads au sein de chaque bloc. Lors de l'implémentation d'une fonction en CUDA, le dosage du nombre de blocs et du nombre de threads par bloc est sans doute l'étape la plus importante et la plus difficile : il ne faut ni en mettre trop, sinon le GPU sature et fonctionne plus lentement, ni trop peu, sinon le GPU n'est pas entièrement utilisé. Le nombre de blocs est limité par design à 65536, et le nombre de threads par bloc ne peut pas dépasser 512.

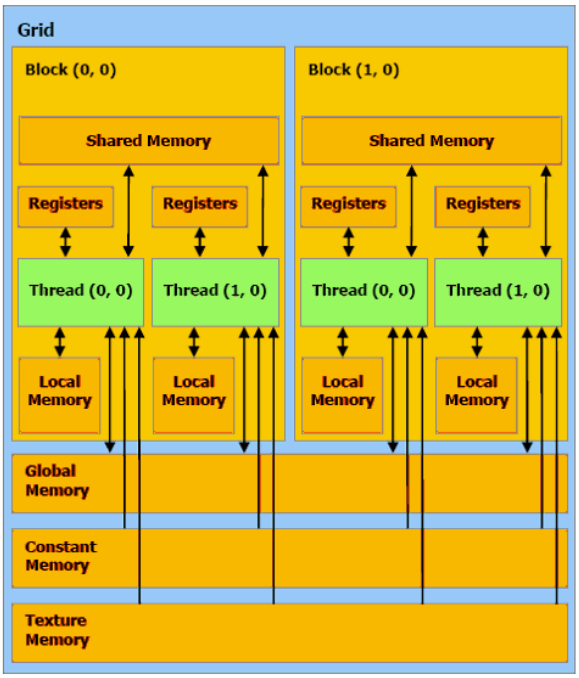


FIG. 1.2 – Architecture de la grille représentant la structure du GPU (provenance [17])

Avec CUDA, la gestion de la mémoire est laissée au développeur. La mémoire d'un GPU est divisée en différentes zones, comme indiqué sur la figure 1.2. Le développeur essaie d'utiliser au maximum les registres mémoire (*Registers*), la mémoire partagée (*Shared Memory*) et la mémoire globale (*Global Memory*) :

- Mémoire globale : avant de lancer un kernel, le développeur déplace les données depuis la mémoire CPU vers la mémoire GPU où les données sont stockées dans la mémoire globale, mémoire dont la latence est élevée : il faut attendre de 400 à 600 cycles d'horloge avant de pouvoir y accéder.
- Registres : les registres sont propres aux threads et ont donc la durée de vie du thread. Ce sont les zones mémoires utilisées par défaut par les threads afin de stocker les données. Cette mémoire est très rapide, son accès ne prenant pas un cycle supplémentaire par instruction,

[5] De l'anglais, signifiant *noyau*

et est limitée à 8192 registres par multiprocesseur. Un GPU est constitué de plusieurs multiprocesseurs, dont le nombre dépend du modèle du GPU.

– Mémoire partagée : la mémoire partagée est une mémoire commune à tous les threads d'un même bloc et possède donc la durée de vie du bloc. L'accès à cette mémoire est aussi rapide que celui des registres tant qu'il n'y a pas de conflit entre les threads. S'il y a conflit, les opérations mémoires sont sérialisées et tous les threads participant au conflit doivent attendre la fin des requêtes mémoires. Cette situation est donc à proscrire. La mémoire partagée permet aux threads de communiquer ou de collaborer entre eux. Elle est limitée à 16 ko par bloc.

Les autres mémoires étant assez lentes, le programmeur évite de les utiliser. Souvent, dans le kernel, le développeur transfère les données de la mémoire globale vers les registres des threads ou vers la mémoire partagée, afin de diminuer le temps d'accès aux données fort utilisées.

Listing 1.1 – Addition de vecteurs sur GPU (provenance [16])

```
1  __global__ void vecAdd(float * A, float * B, float * C)
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      C[i] = A[i] + B[i];
5  }
6
7  int main()
8  {
9      // utilisation du kernel
10     const int nThreadsPerBlocks  = 4;
11     const int nBlocks = (arraySize / nThreadsPerBlocks) + _
12                         ( (arraySize \% nThreadsPerBlocks) == 0 ? 0 : 1);
13     vecAdd<<<nBlocks, nThreadsPerBlocks>>>(A, B, C);
14 }
```

Un exemple de code CUDA est présenté au listing 1.1. Ce code permet d'additionner deux vecteurs sur le GPU, le résultat se trouvant dans un troisième vecteur. L'exécution commence à la ligne 7, avec l'appel à la fonction main. A la ligne 10, le développeur décide qu'il y aura 4 threads par bloc. La ligne suivante calcule le nombre de blocs nécessaires pour traiter l'ensemble des vecteurs, sachant qu'il y a 4 threads par bloc et que chaque thread s'occupe de remplir une case du vecteur résultat. La ligne 13 est un appel au kernel. Un appel à un kernel se différencie d'un appel à une fonction grâce à la partie se trouvant entre le nom du kernel et les arguments. Cette partie sert à indiquer le nombre de blocs et le nombre de threads par bloc qui devront être lancés. Le kernel commence à la ligne 1 avec un mot-clé de CUDA __global__. Ceci indique que le kernel est destiné à tourner sur le périphérique, le GPU, et à être appelé depuis l'hôte, le CPU. Il existe d'autres mots-clés du même type, comme __device__ pour les kernels tournant sur le périphérique et étant appelés depuis le périphérique par un autre kernel. La ligne 3 identifie la position du thread et donc quelles cases des tableaux le thread doit additionner. Pour ce faire, le programme utilise des variables globales de CUDA qui permettent d'identifier l'id du bloc et du thread appelant. Comme chaque thread s'occupe d'une case du tableau, il suffit ici de connaître la position du thread dans la grille du GPU (qui ne possède ici qu'une dimension, mais qui peut en posséder jusque trois) pour connaître la case dont il doit s'occuper.

Le seul élément manquant pour avoir un exemple complet est le transfert des données. Celui-ci est assez simple : il consiste en l'appel de la fonction *cudaMemcpy* qui ressemble fort à la fonction *memcpy*[6] mais qui prend un argument supplémentaire afin d'indiquer si le transfert s'effectue du CPU vers le GPU ou l'inverse.

[6]Cette fonction permet de transférer des données d'un endroit à un autre sur un CPU

Remarquons enfin que, bien que le nombre de blocs soit calculé de manière à pouvoir traiter les vecteurs dont la taille n'est pas un multiple de quatre, il y aura toutefois des problèmes si tel est le cas. En effet, un vecteur de taille 17 entraînera la création de cinq blocs, mais le cinquième comprendra des threads qui essaieront d'accéder aux élements 18, 19, et 20 des vecteurs. Il y aura donc une erreur de segmentation. Ceci peut être empêché en mettant un test dans le kernel qui s'assure que le thread courant n'essaie pas d'accéder à une case inexistante.

1.4 GPU et CPU, des outils complémentaires

Cette section se base sur les informations issue de [10].

Du GPU ou du CPU, aucun des deux n'est totalement supérieur à l'autre. Ces outils sont complémentaires et possèdent chacun leurs points forts. Je vais ici développer la différence de fonctionnement de ces deux architectures afin de montrer dans quel cas il est plus souhaitable d'utiliser l'une ou l'autre solution.

Le CPU est un processeur particulièrement sériel, pouvant pratiquer des instructions fortement dépendantes entre elles et permettant ainsi d'effectuer des opérations de branchement conditionnel complexes, des boucles dont le nombre d'itérations varie à l'exécution ...

Pour les threads s'exécutant sur les coeurs du GPU, celui-ci est perçu sous la forme d'une grille en trois dimensions, constituée de blocs au sein desquels les threads peuvent s'exécuter. Un bloc est un élément de calcul dissociable des autres blocs, abritant plusieurs threads, et conçu pour avoir une exécution indépendante de celle des autres blocs [20]. Cela implique que les threads d'un même bloc peuvent communiquer entre eux, mais il y a des limitations pour deux threads appartenant à des blocs différents : la latence pour accéder à une donnée partagée passe de 1 cycle d'horloge à 400 voire 600 cycles d'horloge. Cet accès mémoire ralentissant les performances du GPU est à proscrire.

Le GPU est utilisé dans le cas où un même ensemble d'instructions, appelé le kernel, doit être exécuté sur un grand ensemble de données, appelé le flux, sans pour autant qu'il y ait trop d'interactions entre les différentes itérations. Il n'y a donc pas ou peu de branchements. L'exemple type est une boucle *for* dont les itérations ne dépendent pas les unes des autres. Le CPU s'occupera plutôt des calculs séquentiels ou plus restreints.

Un autre aspect pouvant influencer le choix concerne le transfert des données : la mémoire du CPU n'étant pas la même que celle du GPU, il faut transférer sur ce dernier les données à traiter avant d'effectuer tout calcul. Une fois les données traitées, leur rapatriement sur le CPU s'avère également nécessaire. Si la quantité de données est importante et que le traitement de celles-ci est fort court, il se peut qu'il soit plus avantageux d'utiliser le CPU plutôt que le GPU. Cet inconvénient tend cependant à disparaître grâce aux nouveaux processeurs présentés ci-dessous.

Les personnes désirant en savoir plus sur la programmation avec CUDA peuvent trouver des informations dans [16], [18], [19].

1.5 Les nouveaux processeurs

Le Cell

Cette section se base principalement sur les informations issues de [22] et [23].

Au début des années 90, Toshiba, Sony et IBM se sont alliés pour détrôner Intel en tant que leader sur le marché des microprocesseurs équipant les ordinateurs personnels. Pour ce faire, ils ont cherché comment optimiser l'application d'un kernel sur un flux de données, ce pour quoi les processeurs traditionnels manquent de performances. Le résultat de cette recherche donna un nouveau type de processeur : le Cell (voir figure 1.3).

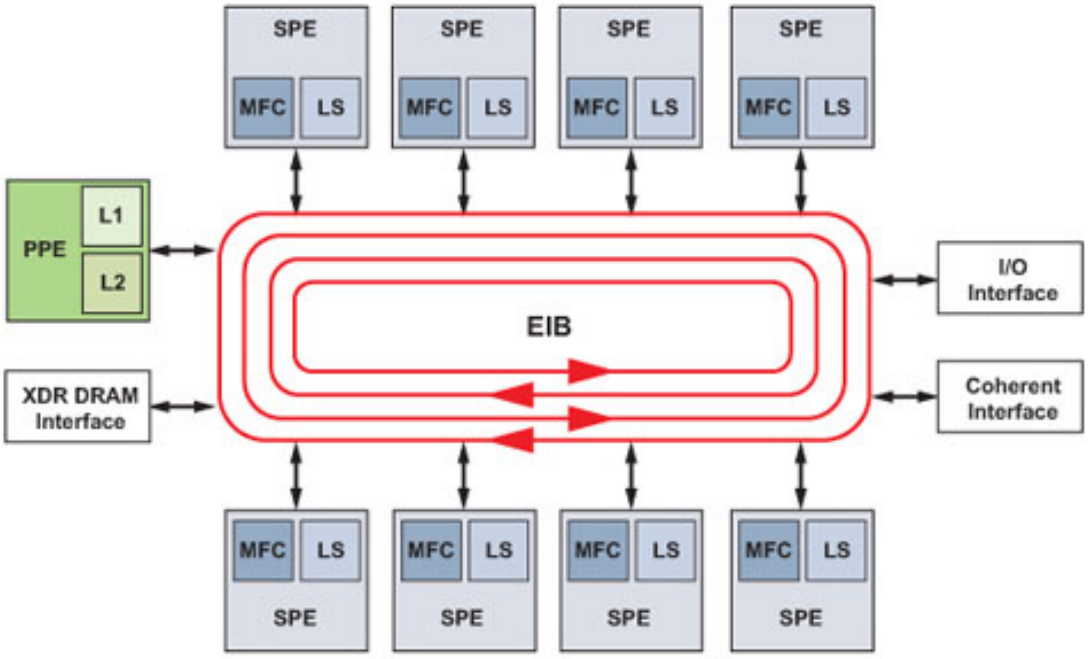


FIG. 1.3 – Architecture générale d'un Cell (provenance [24])

Afin d'éviter la loi des rendements décroisants, l'objectif pour le Cell n'a pas été d'augmenter le parallélisme d'instructions, mais plutôt d'augmenter massivement le parallélisme de threads. A cette fin, les fabricants ont opté pour la même architecture que celle des machines à mémoire distribuée : 8 coeurs SPE[7] possédant chacun leur propre espace mémoire reliés par l'EIB[8], un bus en anneau. Le tout est orchestré par un microprocesseur de type PowerPC appelé PPE [9].

Les SPE ont des processeurs SIMD qui leur permettent donc de travailler comme les processeurs des GPU. Pour les applications non-scientifiques, le processeur d'un SPE peut atteindre une performance de crête de 32 GFlops pour un processeur à 4 GHz. Cette performance tombe à 2,3 GFlops pour les applications scientifiques. Cette chute de puissance est due à l'ajout de contraintes[10]

Les SPE possèdent une mémoire locale restreinte, mais extrêmement rapide (Local Store en anglais, LS sur le schéma), ainsi qu'un contrôleur qui gère cette mémoire (le Memory Flow Controller, ou MFC sur le schéma). Le MFC est responsable du transit des données entre les SPE et le PPE, seul élément ayant accès à la mémoire centrale. Les données transitent via le bus EIB qui possède 4 anneaux. Comme indiqué sur le schéma, les anneaux font circuler les données dans des sens différents afin de diminuer le temps de latence pour accéder à une donnée.

Il reste encore deux points importants à indiquer. Le premier est la suppression de l'ordonnancement dynamique des instructions : le PPE et les SPE ne traitent les instructions que dans l'odre du programme. La logique de contrôle est ainsi limitée au maximum, permettant donc de meilleures performances lors de l'exécution. C'est maintenant au développeur et au compilateur d'optimiser l'ordonnancement des instructions. Le deuxième point est que les Cell sont faits pour communiquer entre eux et partager leur charge de travail. Plus il y a de Cell, meilleures seront les performances.

[7] De l'anglais Synergistic Processing Element, signifiant *Element de Traitement Coopératif*
[8] De l'anglais Element Interconnect Bus, signifiant *Bus d'Interconnection d'Eléments*
[9] De l'anglais Power Processing Element, signifiant *Element de Traitement Power*
[10] La chute provient de l'ajout d'une norme définissant un comportement bien précis pour l'arrondi des nombres.

Je termine ici la présentation du fonctionnement général d'un Cell. Le lecteur qui voudrait en savoir plus peut consulter les dossiers [22] et [23] qui présentent en détail le Cell.

Les processeurs Cell ont jusqu'ici été utilisés pour des architectures spécialisées, la plus connue étant la console de jeu PlayStation 3 sortie en 2006 [25]. Son successeur, la PlayStation3 Slim, est sortie en 2009 et bénéficie également de la dernière génération du microprocesseur [26]. En 2005 et en 2006, la société Mercury Computer Systems a conçu des serveurs et un processeur pour station de travail comprenant le Cell. Ce processeur, appelé Mercury Cell Accelerator Board[11] (CAB) peut être branché dans n'importe quel port PCI Express et permet d'atteindre des performances de 180 GFLOPS. Il est principalement conçu pour des applications de rendu, de lancer de rayon, et de traitement d'images, de vidéos et de signaux. Le coût à sa sortie s'élevait à près de 8000 dollars américains [27]. La firme a depuis amélioré son produit et a sorti le CAB 2 [28]. En 2006, IBM a sorti un serveur nommé QS20 qui intègre deux microprocesseurs Cell et qui permet d'atteindre des piques de performance de 410 GFLOPS [29]. Enfin, la société Fixstars sortit en 2008 une carte accélératrice possédant un Cell et permettant d'atteindre des performances de 180 GFLOPS [30].

Le projet de distribution de données de l'Université de Stanford Folding@home [31] a pour but d'utiliser la puissance des ordinateurs des personnes utilisant le programme afin d'étudier le repliement des protéines, les repliements anormaux, l'agrégation des protéines, et les maladies liées. Le site met à disposition les statistiques des clients participant au projet [32]. Ceci permet de montrer, cfr fig. 1.4, que le Cell arrive à un même niveau de performance que les GPUs.

Système d'exploitation	TFlops Natifs	TFlops x86
GPU ATI	934	985
GPU NVIDIA	1283	2702
PlayStation 3 ®	957	2019

FIG. 1.4 – Performances des GPUs et des Cells

1.5.1 Le futur des processeurs

Le processeur CELL est l'un des premiers processeurs multicoeurs hétérogènes, mais est de loin le dernier. En effet, AMD compte sortir le premier APU (de l'anglais Application Processor Unit, signifiant l'Unité Processeur Application), nommé Llano, début 2011. La particularité de ce nouveau type de processeur est qu'il combine des coeurs CPU et des coeurs GPU (deux à quatre coeurs STARS x86 et un GPU compatible DirectX 11 dérivé de l'architecture des Radeon HD 5000 [33]) sur le même *die*, afin de permettre un partage de la mémoire cache et donc une meilleure bande-passante et une latence plus basse. Il n'est plus question d'assemblage, comme c'est le cas pour le Clarkdale d'Intel, sorti début 2010, qui possède un *die* CPU et un *die* GPU [34]. Intel ne se laisse cependant pas faire et compte sortir son APU nommé Sandy Bridge début 2011 également [35] [36]. Intel avait également fait beaucoup de bruit autour du Larabee, un autre processeur qui devait unir les avantages des CPU et des GPU, mais le projet a été abandonné fin 2009. Bien que le monde de l'informatique parle d'un remplaçant, celui-ci n'est toujours pas annoncé [37].

Les fabricants de processeurs semblent donc bien engagés dans la voie du GPGPU et nous pouvons dès lors nous attendre à ce que les futurs APU augmentent significativement les performances des applications issues de ce type de programmation.

[11] Accelerator Board vient de l'anglais et signifie carte accélératrice ou carte d'accélération

1.6 Application : le tri d'un vecteur de données

1.6.1 Introduction : le tri séquentiel

Déjà sur CPU, le tri d'un vecteur de données est un sujet fort populaire, particulièrement dans les domaines de l'optimisation, des bases de données, du data mining[12] ... En effet, les algorithmes ayant besoin de trier des données sont monnaie courante. Dans le cadre de ce travail de fin d'étude, il m'a été demandé d'améliorer l'algorithme DNARun, écrit en C++ et présenté au chapitre 2, en faisant usage du GPU. En analysant les performances de l'algorithme, nous nous sommes rendu compte qu'il perd énormément de temps dans les fonctions de tri. C'est pour cette raison que j'ai cherché une fonction de tri efficace et facile à réutiliser, afin de substituer la version séquentielle à ce tri et ainsi obtenir un gain de temps.

De nombreux algorithmes ont été créés au cours des années précédentes, chacun ayant ses propres spécificités. La bibliothèque STL[13] [38], librairie dont sont issus les classe `vector`, `map` et `list` en C++, utilise le tri Introsort[14]. Ce tri est en réalité un agglomérat de tris [39] [40] :

- Quicksort[15] : algorithme de base d'Introsort, il applique le principe *diviser pour mieux régner (divide and conquer en anglais)* en décomposant récursivement le tableau à trier en sous-tableaux. Le principe est assez simple : un pivot est choisit dans le vecteur, tel qu'il soit à sa position finale, que tous les éléments à sa droite lui soient supérieurs, et que tous les éléments à sa gauche lui soient inférieurs. Le tableau est ensuite décomposé en deux parties : l'une allant du début jusqu'au pivot non inclus, l'autre allant du pivot non inclus à la fin du vecteur. Pour trouver un pivot répondant à ces règles, Introsort procède avec des pointeurs : la position du pivot est choisie arbitrairement, un pointeur parcourt le tableau de la gauche vers la droite et un second pointeur l'explore de la droite vers la gauche. Lorsque le pointeur s'occupant de la partie gauche du vecteur a trouvé un élément plus grand que le pivot et que le pointeur s'occupant de la partie droite a trouvé un élément plus petit que le pivot, ces deux éléments sont permutés. Lorsque les deux pointeurs se rencontrent, le pivot respecte les propriétés énoncées. Le tri se poursuit récursivement de la même manière avec les deux sous-tableaux. La complexité de cette algorithme est de l'ordre de $n * log(n)$, où n est la taille du vecteur. Une illustration de cet algorithme est présentée à la figure 1.5 et une implémentation Java peut être trouvée au listing 1.2. Pour des informations plus détaillées, veuillez vous référer à [40], [41], et [42].
- Heapsort[16] : cet algorithme est très simple : il crée une pile vide, sélectionne à chaque itération l'élément maximum ou minimum du vecteur, selon l'ordre du tri, et le place dans la pile. Une fois que tous les éléments ont été retirés du vecteur, ils se trouvent triés dans la pile. La complexité de cette algorithme est de l'ordre de $n * log(n)$, où n est la taille du vecteur. Pour des informations plus détaillées, veuillez vous référer à [43], et [44].
- Insertion sort [17] : à chaque itération, l'algorithme considère le vecteur comme étant divisé en deux parties : la partie triée à gauche, la partie non triée à droite. A chaque itération, le premier élément du vecteur non trié (a_i) est retiré de celui-ci et comparé à ceux du vecteur trié (les a_j), en commençant par son extrémité droite. Lorsqu'un élément plus petit que l'élément à insérer est trouvé ($a_j < a_i$), ce dernier est placé juste après l'élément plus petit. Au fur et à mesure, la partie triée du vecteur se remplit tandis que la partie non triée se vide. La complexité de cette algorithme est de l'ordre de n^2, où n est la taille du vecteur. Pour des informations plus détaillées, veuillez vous référer à [45].

[12] De l'anglais, signifiant *la fouille de données*
[13] De l'anglais Standard Template Library, signifiant Librairie des Modèles Standards
[14] De l'anglais, agglomération de introspection, et sort (tri). C'est cette fonction qui était la plus appelée par l'algorithme présenté au chapitre 2
[15] De l'anglais, signifiant Tri rapide
[16] De l'anglais, signifiant Tri à pile
[17] De l'anglais, signifiant Tri par insertion

Pour l'algorithme du tri rapide, il est possible de calculer dynamiquement les profondeurs récursives du meilleur et du pire cas. Une moyenne de ces deux profondeurs donne un seuil à partir duquel Introsort préfère utiliser l'algorithme Heapsort, afin d'éviter que Quicksort dégénère vers un comportement quadratique. Insertion sort est utilisé par Introsort lorsque ce dernier rencontre des petits vecteurs, principalement pour finaliser le tri.

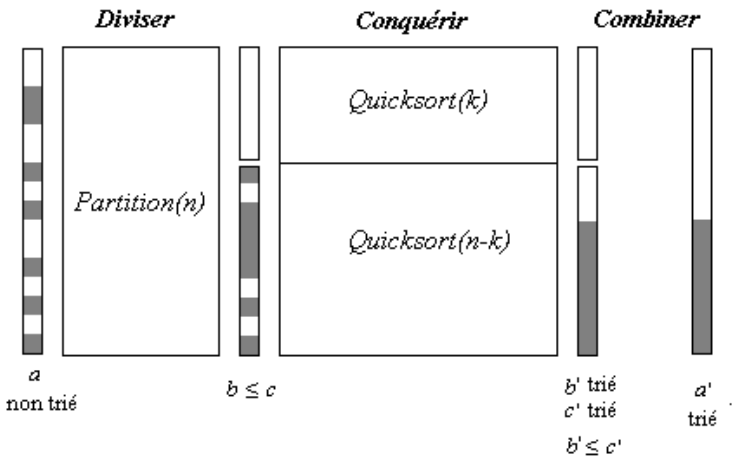


FIG. 1.5 – Algorithme du tri rapide, les zones grises représentent des *1*, les zones blanches des *0* (provenance [42])

A part l'algorithme Quicksort qui permet de par ses sous-tableaux un parallélisme, les autres algorithmes restent fort séquentiels. C'est pourquoi, des recherches ont été menées ces dernières années pour porter des algorithmes de tri sur GPU. Je vais ici présenter l'adaptation de quatre tris disponibles dans diverses librairies : le Quicksort, le Bitonic[18], le Merge Sort[19] et le Radix Sort[20]. Après avoir présenté les tris, je discuterai de leurs réutilisabilités et des modifications apportées par nos soins pour pouvoir utiliser le tri bitonique avec nos contraintes.

Listing 1.2 – Programme Java implémentant l'algorithme Quicksort (provenance [42])

```
1  void quicksort (int[] a, int lo, int hi)
2  {
3  //  lo est l'indice le plus bas, hi est l'indice le plus haut
4  //  de la région du tableau a qui est trié
5      int i=lo, j=hi, h;
6
7      // l'élément x de comparaison
8      int x=a[(lo+hi)/2];
9
10     //  partition
11     do
12     {
13         while (a[i]<x) i++;
14         while (a[j]>x) j--;
15         if (i<=j)
16         {
17             h=a[i]; a[i]=a[j]; a[j]=h;
18             i++; j--;
19         }
20     } while (i<=j);
21
```

[18] De l'anglais, signifiant *bitonique*
[19] De l'anglais, signifiant *tri fusion*
[20] De l'anglais, signifiant *tri par base*

```
22      //  récursion
23      if (lo<j) quicksort(a, lo, j);
24      if (i<hi) quicksort(a, i, hi);
25  }
```

1.6.2 Exemples de tri sur GPU

Adaptation du tri rapide

Le Distributed Computing and Systems Research Group[21] a mis à disposition sa librairie GPU Quicksort [46] dont le but premier est de fournir une implémentation du tri rapide qui soit adaptée au GPU. Comme mentionné dans l'article [47], le problème de cet algorithme est surtout son commencement : à partir d'un certain moment, la taille des sous-tableaux devient suffisamment petite (512 éléments[22]) pour affecter chacun d'entre eux à un bloc du GPU, mais cela nécessite une adaptation préalable du début de l'algorithme : le moment durant lequel plusieurs blocs du GPU doivent collaborer sur une même séquence. Pour ce faire, le tri rapide GPU découpe les tableaux de trop grande taille en sous-tableaux qui sont ensuite chacun assignés à un bloc. Chaque thread s'occupe d'une case d'un sous-tableau. Les threads d'un bloc vont, dans un premier temps, lire et indiquer dans un premier vecteur le nombre d'éléments du sous-tableau associé qui sont supérieurs au pivot ainsi que, dans un second vecteur, le nombre d'éléments qui sont inférieurs au pivot. Une fois que tous les blocs ont fini leur exécution, l'algorithme effectue la somme cumulative des deux vecteurs puis la somme cumulative pour tous les blocs. Pour exécuter cette dernière opération, le CPU doit être utilisé. Ces sommes donnent assez d'informations aux threads pour qu'ils puissent savoir où déplacer la donnée qui leur a été attribuée. Une fois que l'algorithme a effectué plusieurs itérations et que les sous-tableaux possèdent une taille suffisamment petite que pour être triés dans un bloc, l'algorithme réitère les mêmes opérations et la somme cumulative des vecteurs d'un bloc suffit à elle seule pour les trier. Il ne faut donc plus passer par le CPU pour exécuter la somme cumulative sur tous les blocs, un gain de temps est ainsi obtenu.

Le tri bitonique

Cette section se base principalement sur les informations issues de [48] et [49].

Une séquence bitonique est une séquence qui possède au maximum un minimum ou un maximum local. Exemple :

$$1, 2, 3, 4, 5$$

$$3, 4, 8, 15, 20, 17, 10, 2, 1$$

Dans le premier exemple, il y a un minimum local (1) et un maximum local (5) ; le deuxième exemple possède un maximum local (20) et deux minimums locaux (3 et 1).

Pour ce tri, un comparateur B_n, où n est la taille du vecteur à trier, a besoin d'être défini tel que :

$$B_n = [0 : \frac{n}{2}][1 : \frac{n}{2} + 1] \dots [\frac{n}{2} - 1 : n - 1]$$

Les éléments positionnés aux emplacements indiqués par les deux chiffres obtenus dans chaque paire de crochets sont comparés et échangés de manière à ce que le plus petit se retrouve dans la

[21] De l'anglais, signifiant *Groupe de Recherche sur les Systèmes et l'Informatique Distribuée*
[22] Correspond au nombre maximal de threads qu'il est possible de lancer dans un même bloc

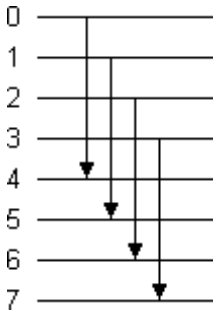


FIG. 1.6 – Comparateur B_8 utilisé pour le tri bitonique (provenance [49])

séquence de gauche. Une illustration de ce comparateur est montré à la figure 1.6. L'application du comparateur B_n au vecteur bitonique a donne :

$$B_n(a) = b_0 \ldots, b_{\frac{n}{2}-1}, c_0 \ldots, c_{\frac{n}{2}-1} \; avec \; b_i \leq c_j \; \forall \; i, j \; \in 0, \ldots, \frac{n}{2} - 1$$

Les séquences b et c sont alors bitoniques. La figure 1.7 démontre ce fait car elle présente l'application de l'opérateur B_n sur tous les types de séquences bitoniques possibles et montre que le résultat est toujours deux séquences bitoniques. Si nous appliquons ensuite l'opérateur $B_{\frac{n}{2}}$ sur b et c, nous obtenons 4 séquences bitoniques qui contiennent d, e, f et g, avec $d \leq e$, $e \leq f$ (car $a \leq b$), et $f \leq g$. Si nous continuons ainsi, nous disposons à la fin de n séquences bitoniques d'un élément. Chacune de ces n séquences étant plus petite que celle qui la suit, nous obtenons un vecteur trié

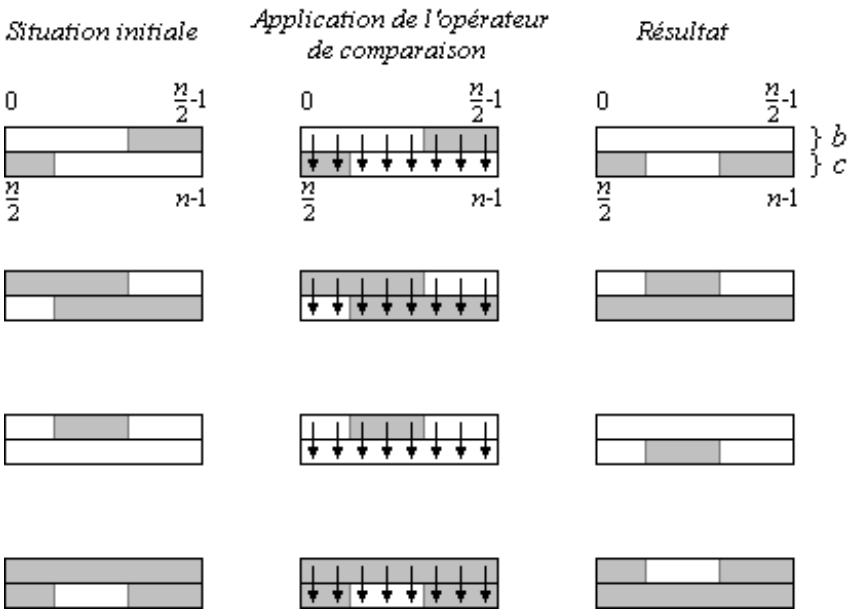


FIG. 1.7 – Mise en évidence du caractère bitonique des séquences b et c après application de l'opérateur B_n. Les zones grises représentent des *1*, les zones blanches des *0* (provenance [49])

Jusqu'ici, le tri a été appliqué sur une séquence bitonique. Si nous voulons l'appliquer sur une séquence quelconque, il faut suivre la procédure indiquée sur les figures 1.8 et 1.9. Une succession de tris sur le vecteur conduisent à obtenir une séquence bitonique qui sera ensuite traitée comme expliqué précédemment. L'implémentation Java de cet algorithme est fourni au listing 1.3. La complexité de cet algorithme est de l'ordre de $n * log^2(n)$.

Listing 1.3 – Programme Java implémentant l'algorithme de tri bitonique (provenance [49])

```
1  public class BitonicSorter implements Sorter
2  {
3      private int[] a;
4      // sorting direction:
5      private final static boolean ASCENDING=true, DESCENDING=false;
6
7      public void sort(int[] a)
8      {
9          this.a=a;
10         bitonicSort(0, a.length, ASCENDING);
11     }
12
13     private void bitonicSort(int lo, int n, boolean dir)
14     {
15         if (n>1)
16         {
17             int m=n/2;
18             bitonicSort(lo, m, ASCENDING);
19             bitonicSort(lo+m, m, DESCENDING);
20             bitonicMerge(lo, n, dir);
21         }
22     }
23
24     private void bitonicMerge(int lo, int n, boolean dir)
25     {
26         if (n>1)
27         {
28             int m=n/2;
29             for (int i=lo; i<lo+m; i++)
30                 compare(i, i+m, dir);
31             bitonicMerge(lo, m, dir);
32             bitonicMerge(lo+m, m, dir);
33         }
34     }
35
36     private void compare(int i, int j, boolean dir)
37     {
38         if (dir==(a[i]>a[j]))
39             exchange(i, j);
40     }
41
42     private void exchange(int i, int j)
43     {
44         int t=a[i];
45         a[i]=a[j];
46         a[j]=t;
47     }
48
49 }
```

Pour porter cet algorithme sur une architecture multi-coeur, il suffit de remarquer qu'il y a $\frac{n}{2}$ comparaisons à chaque étape de l'algorithme et d'attribuer ces comparaisons à $\frac{n}{2}$ threads. Au sein d'un GPU, les threads d'un même bloc peuvent partager des données entre eux grâce à la mémoire partagée. Comme le nombre de threads d'un bloc est limité à 512, la taille maximum d'un tableau pouvant être entièrement trié par ces threads est de 1024. NVIDIA a implémenté un tri bitonique et l'a distribué dans son SDK[23] [50]. Lorsque la taille du vecteur est plus petite ou égale à 1024, un seul bloc est lancé et chaque thread charge deux éléments du vecteur dans

[23] De l'anglais Sample Development Kit, signifiant Kit de Dévelopement Logiciel

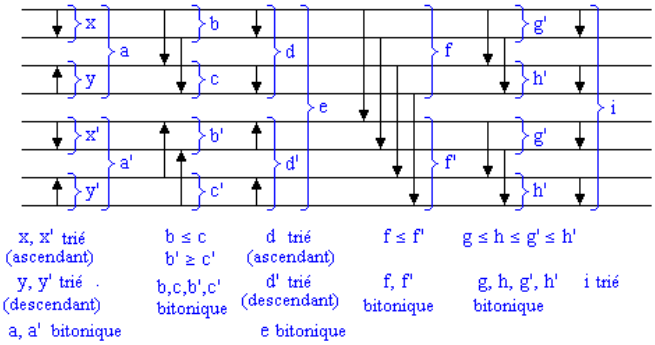


FIG. 1.8 – Tri bitonique complet appliqué sur un vecteur de 8 éléments (provenance [49])

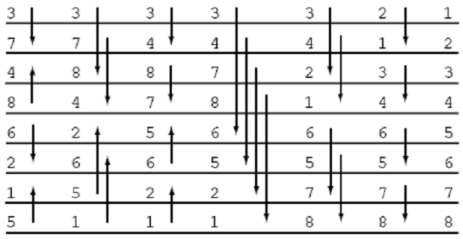


FIG. 1.9 – Tri bitonique complet chiffré appliqué sur un vecteur de 8 éléments (provenance [48])

un tableau placé en mémoire partagée afin de ne pas devoir aller sans cesse les chercher dans la mémoire globale. Les threads du bloc trient le tableau comme expliqué précédemment et chacun replace ensuite deux éléments dans le vecteur se trouvant en mémoire globale. Si la taille du tableau est plus grande que 1024, le tableau est divisé en sous-tableaux de taille égale. Chaque sous-tableau est trié grâce au tri bitonique et ils sont ensuite réunis grâce à un algorithme de fusion.

Le tri fusion

Cette section se base principalement sur les informations issues de [51] et [52].

Comme pour l'algorithme de tri rapide, le tri fusion possède une stratégie *diviser pour mieux régner*. Son principe consiste à diviser le tableau en sous-tableaux qui sont triés indépendamment les uns des autres pour ensuite être réunis grâce à un algorithme de fusion. La figure 1.10 et le listing 1.4 montre la manière séquentielle pour implémenter un tel tri. Pour passer à une implémentation hautement parallélisable, il suffit de diviser dès le départ le tableau en un certain nombre de sous-tableaux. La complexité de cet algorithme est de l'ordre de $n * log(n)$, où n est la taille du vecteur.

Listing 1.4 – Programme Java implémentant l'algorithme de tri fusion (provenance [52])

```
1  void mergesort(int lo, int hi)
2  {
3      if (lo<hi)
4      {
5          int m=(lo+hi)/2;
6          mergesort(lo, m);
7          mergesort(m+1, hi);
8          merge(lo, m, hi);
9      }
10 }
```

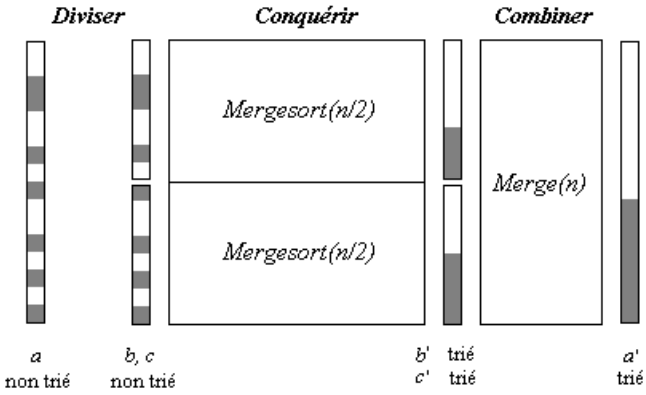


FIG. 1.10 – Tri fusion. Les zones grises représentent des *1*, les zones blanches des *0* (provenance [52])

Cet algorithme a été implémenté par la librairie CUDA Thrust [53].

Le tri par base

Cette section se base principalement sur les informations issues de [51].

Le tri par base est fort différent des tris précédents car les comparaisons qu'il opère s'exercent sur les bits ou les chiffres (selon la version utilisée) des nombres à trier, et non sur les nombres eux-même. Le tri considère en premier les chiffres ou les bits les moins significatifs et termine par les plus significatifs. Si un nombre est constitué de quatre chiffres ou de quatre bits, quatre passes seront nécessaires. Exemple :

Séquence de base :

$$34, 56, 97, 87, 83, 68$$

Première passe (tri selon l'unité) :

$$34, 56, 87, 97, 88, 99$$

Nous pouvons remarquer que le tri est stable : lorsque deux unités sont égales, le tri prend en compte les dizaines pour ordonner les nombres. Dans l'exemple, c'est le cas pour les nombres 87 et 97.

Deuxième passe (tri selon la dizaine) :

$$34, 56, 87, 88, 97, 99$$

La complexité de ce tri est de l'ordre de $n * k$ où n est le nombre d'éléments à trier et k la taille moyenne de ces éléments, la taille étant exprimée en nombre de chiffres ou de bits.

Le SDK de CUDA [50] fournit une implémentation GPU de ce tri. Cette implémentation trie les nombres en considérant les bits. Afin de savoir où un nombre viendra se placer dans le vecteur trié, l'algorithme calcule son rang, celui-ci étant le nombre d'éléments plus petits ou égaux à l'élément considéré. Dans le but d'accélérer l'exécution du programme, l'algorithme charge en mémoire partagée le tableau qu'un bloc de threads triera. La mémoire partagée étant limitée à 16 ko, il n'est pas toujours possible de mettre le vecteur en entier dans le même bloc et les threads ne sont donc pas toujours au courant de la valeur de tous les éléments du vecteur. Afin de pallier à ce problème, chaque bloc de threads crée un histogramme qui contient le nombre d'occurences de chaque valeur rencontrée dans le sous-vecteur. En faisant ensuite la somme des histogrammes, il est possible de savoir où doit aller chaque élément.

Comme la lecture et l'écriture dans la mémoire globale sont des opérations coûteuses, un thread est chargé de trier les éléments sur un nombre b de bits, avec b plus grand que 1.

Chaque passe de l'algorithme se déroule donc en quatre étapes :

1. Chaque bloc charge son sous-vecteur en mémoire partagée et le trie en b passage.
2. Chaque bloc écrit en mémoire globale le sous-vecteur trié ainsi que l'histogramme.
3. Les histogrammes sont sommés et la position de chaque élément est calculée.
4. Chaque bloc copie les éléments à leur bonne position.

1.6.3 Application du tri sur GPU à l'algorithme DNARun

Choix et adaptation d'un tri GPU

L'algorithme DNARun est un algorithme qui effectue de nombreux tris de vecteurs contenant soit une centaine, soit un peu moins d'un millier d'éléments. Vu qu'en GPGPU, il existe une perte de temps due au transfert de données entre le CPU et le GPU, il faut en contrepartie que le traitement sur le GPU soit intensif pour pallier à cette perte de temps. Ce phénomène est illustré sur la figure 1.11 qui présente les performances de la librairie Thrust dans le cas du tri d'un vecteur d'entier. Nous pouvons voir que le tri sur GPU devient plus efficace que le tri sur CPU à partir de 3000 éléments, mais il en faut trois fois plus si les temps de transfert sont pris en compte. Comme les vecteurs de DNARun sont loin d'atteindre de tels nombres, nous les concaténons et les trions en un tri.

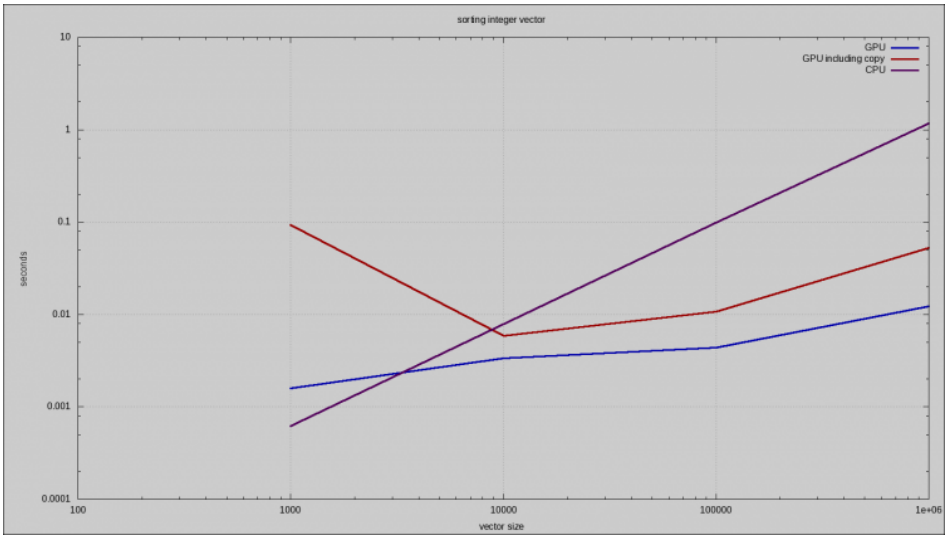


FIG. 1.11 – Performance de Thrust lors du tri d'un vecteur d'entier (provenance [54])

Les données à trier sont contenues dans des structures de 8 octets. Afin de savoir à quel vecteur appartient quelle donnée, nous avons ajouté un identifiant de 4 octets et, pour optimiser l'exécution, le compilateur arrondit la structure à 16 octets. J'ai donc exploré l'existant afin de trouver le tri le plus performant pour des structures de 16 octets :

- Le tri rapide de la bibliothèque GPU Quicksort : ce tri n'a été créé que pour les entiers et les nombres à virgule flottante de 4 octets.
- Le tri bitonique du SDK NVIDIA : ce tri n'a été créé que pour les entiers et les nombres à virgule flottante de 4 octets.
- Le tri fusion de la bibliothèque Thrust : ce tri peut traiter nos données.
- Le tri par base du SDK NVIDIA : ce tri n'a été créé que pour les entiers et les nombres à virgule flottante de 4 octets.

Comme la bibliothèque Thrust demande de passer par ses vecteurs personnalisés, nous avons également voulu adapter un des trois autres tris existants, dans l'espoir d'obtenir de meilleurs résultats. La bibliothèque GPU Quicksort souffre d'un défaut majeur : certains éléments de paramétrisation du tri dépendent du modèle de GPU sur lequel l'algorithme s'exécute et sont encodés en dur dans le code. Comme, d'un côté, nous ne désirons aucune dépendance envers les GPUs sur lesquels le code s'exécute et que, d'autre part, rien n'explique comment calculer ces nombreux éléments ni les éventuelles interactions entre eux, nous nous sommes tournés vers les solutions NVIDIA. Le tri par base possède deux principaux désavantages :

1. C'est le seul tri dont la complexité augmente avec la taille de la donnée. Par rapport au tri implémenté, la complexité quadruple lorsqu'on passe de 4 à 16 octets.
2. C'est celui dont le déroulement varie le plus selon la taille des données. Le code possède plusieurs passages qui optimisent le traitement pour les structures à 4 octets, il aurait donc fallu apporter de profonds remaniements.

Le seul désavantage du tri bitonique est sa complexité. Comme notre but est de créer des tâches qui trieront des vecteurs de taille ni trop grande, ni trop petite, afin de pouvoir utiliser les différents GPUs et CPUs se trouvant sur une machine, ceci n'est pas trop gênant. Nous avons donc décidé d'adapter le tri bitonique.

De base, le tri est optimisé pour trier des vecteurs de clé-valeur[24] dont la valeur peut atteindre une taille de 4 octets. Après avoir retiré cette valeur du code, il pouvait déjà trier des données de 8 octets.

Le dernier obstacle pour passer de 8 à 16 octets fut la limite de la mémoire partagée. NVIDIA ayant optimisé le nombre de threads par blocs par rapport à la taille de la mémoire partagée, il n'était pas possible d'accroître la taille des données sans effectuer un dépassement de la mémoire. Comme l'algorithme nécessite des sous-tableaux dont la taille est une puissance de deux, j'ai divisé par deux le nombre de threads lancés dans chaque bloc et ai multiplié par deux le nombre de blocs.

Une dernière chose à mentionner est que, comme l'algorithme nécessite des vecteurs dont la taille est une puissance deux, le seul moyen de traiter un vecteur de taille $2^y + x$ $(2^y - x)$ avec cet algorithme est de créer un vecteur de taille 2^{y+1} (2^y) et de remplir les $2^{y+1} - 2^y - x$ (x) cases vides par des éléments mis à 2^{y+1} (2^y). Ceci permet de sélectionner notre vecteur d'origine en prenant les $2^y + x$ premiers éléments du vecteur trié.

Résultats

Le tri bitonique de NVIDIA, le tri fusion de Thrust et le tri Introsort de STL ont été appliqués à des vecteurs de taille croissante. Afin de pouvoir comparer l'influence de la taille des données à trier, les tris ont été appliqués sur des données de 8 et de 16 octets. Les résultats sont graphiquement représentés à la figure 1.12. Les courbes correspondant au tri bitonique ne sont pas totalement correctes car ce tri ne peut traiter que des tableaux dont la taille est une puissance de deux et les graphiques devraient donc plutôt ressembler à des fonctions en escalier. Excel ne permettant pas de tracer de tels graphiques et les points des graphiques en nuage de points étant trop gros, je fus contraint d'utiliser des courbes.

La première conclusion à tirer concerne l'importance de la taille des données à trier. Alors que pour les structures de 8 octets, le tri GPU permet, pour les vecteurs dont la taille avoisine les 2000000, un gain de temps de 77% pour le tri bitonique et de 64% pour le tri fusion, ces performances descendent à respectivement 35% et 22% lorsque la structure passe à 16 octets. Nous pouvons retrouver là l'influence des transferts des données entre la mémoire globale et la

[24] La clé est l'élément sur lequel l'algorithme se base pour ordonner le vecteur, la valeur est un élément associé à une clé et qui change de place dans son vecteur en même temps et au même endroit que sa clé

mémoire partagée ainsi que l'influence du nombre de threads à traiter les données à un instant donné. Bien que Thrust soit moins efficace sur les données de petite taille, l'algorithme est plus robuste face au changement de taille des données et est plus performant que le tri bitonique de NVIDIA.

Une seconde réflexion peut être menée sur les temps de transfert des données. Ceux-ci sont loin d'être négligeables et font rapidement tomber les performances des GPU. En effet, pour les structures de 8 octets, l'ajout de ces temps de transfert fait tomber le gain du tri fusion de 64% à 46% et le gain du tri bitonique de 77% à 59%. La nette différence de performance entre les deux tris est due au fait que Thrust gère les transferts de données alors que NVIDIA laisse cette tâche à l'utilisateur. A ce niveau, Thrust semble moins performant. Ceci est dû soit à la présence de mécanismes supplémentaires permettant d'exécuter moins de transferts lorsqu'une même donnée est réutilisée plusieurs fois sur un même GPU pour des calculs différents, soit à une optimisation des transferts de données de 4 octets conduisant à une baisse de performance lorsque la taille de ces dernières augmente. Les performances s'écroulent totalement lorsque la taille des données passe de 8 à 16 octets. Le gain pour Thrust passe de 46% à 5% et le tri bitonique n'arrive plus à faire de gain pour les tailles des vecteurs considérés.

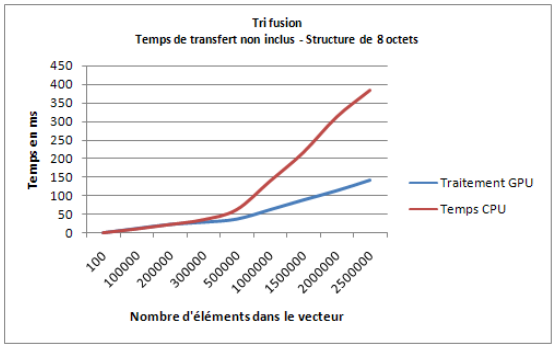
1.6.4 Conclusions

Au vu des résultats obtenus, nous pouvons tirer trois conclusions.

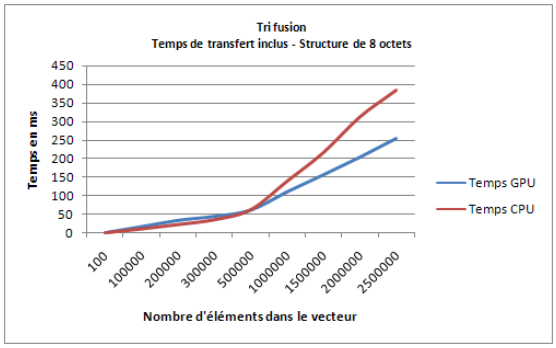
La première est que les tris GPU qui ont été utilisés dans le cadre de ce travail ne sont pas adaptés à des données dont la taille est supérieure à huit octets. En effet, les performances sont fortement dégradées à cause du transfert des données entre mémoire CPU et mémoire GPU ainsi qu'entre les différents types de mémoire du GPU. Ce problème ne se posera sûrement plus lorsque les premiers APU programmables feront leur apparition. En attendant, s'il faut trier des données dont la taille atteint 16 octets, d'autres tris doivent être adaptés, voire même créés.

La seconde conclusion est que les performances des tris appliqués à des structures de 8 octets sont meilleures pour le tri bitonique que pour le tri fusion. S'il est possible de diminuer la taille des données à traiter dans l'algorithme DNARun, c'est ce tri qui doit être retenu. Les résultats numériques du tri NVIDIA appliqué à une structure de 8 octets se trouvent à la figure 1.13. Les colonnes "CPU $\mapsto$ GPU" et "GPU $\mapsto$ CPU" correspondent aux temps de transfert des données entre les mémoires CPU et GPU. Vous pouvez également trouver les résultats du tri CPU ainsi que le gain que procure l'emploi du tri sur GPU par rapport au tri CPU aux figures 1.14 et 1.15.

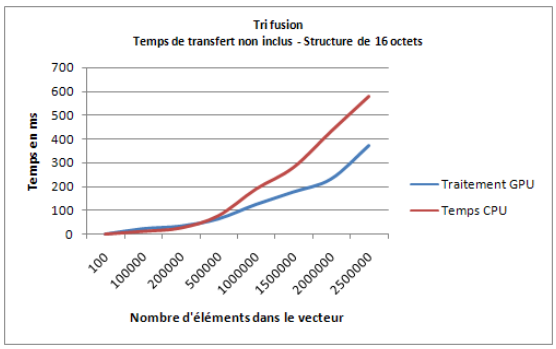
Enfin, une dernière remarque concerne la saturation des performances du tri GPU. En effet, les chiffres présentés à la figure 1.14 montrent qu'entre des vecteurs de 4096 éléments et des vecteurs de 262144 éléments, le gain de temps procuré par l'emploi du GPU passe rapidement de -129% à +48% et qu'ensuite, nous ne gagnons plus que quelques pourcents à chaque fois que la taille du vecteur double. Nous obtenons en réalité une saturation du gain de temps, comme cela peut être mieux observé sur la figure 1.16 qui présente le gain de temps du tri bitonique par rapport au tri Introsort sans prendre en compte les temps de transfert des éléments entre le CPU et le GPU. Ce gain sature aux alentours des 70% dès que la taille du vecteur atteint l'ordre des 65 000 éléments. Nous pouvons donc conclure qu'il y a un certain seuil en-dessous duquel il vaut mieux ne pas se situer si nous voulons tirer parti du maximum des performances du GPU, et au-dessus duquel la taille du vecteur importe peu vu que le gain reste stable.



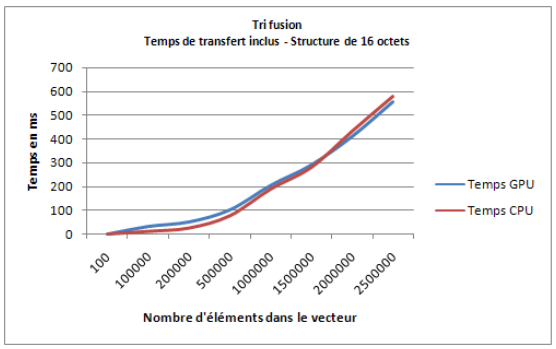
(a) Tri fusion appliqué à des structures de 8 octets, temps de transfert non inclus



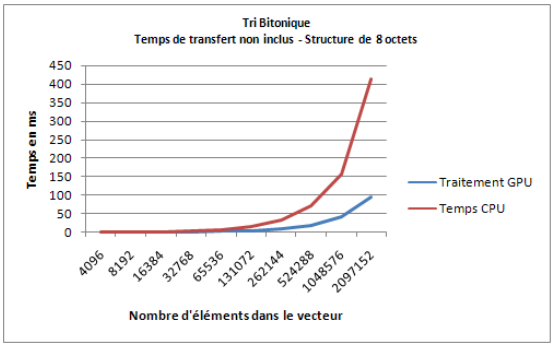
(b) Tri fusion appliqué à des structures de 8 octets, temps de transfert inclus



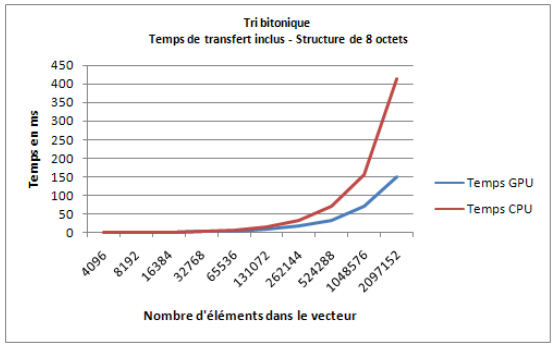
(c) Tri fusion appliqué à des structures de 16 octets, temps de transfert non inclus



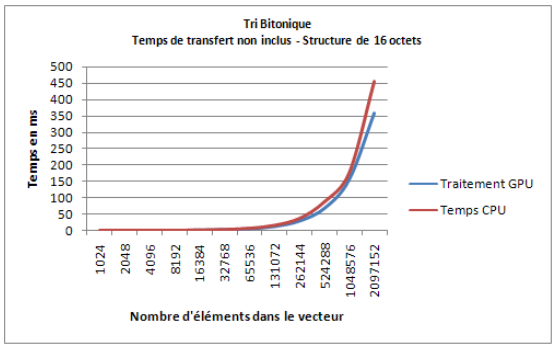
(d) Tri fusion appliqué à des structures de 16 octets, temps de transfert inclus



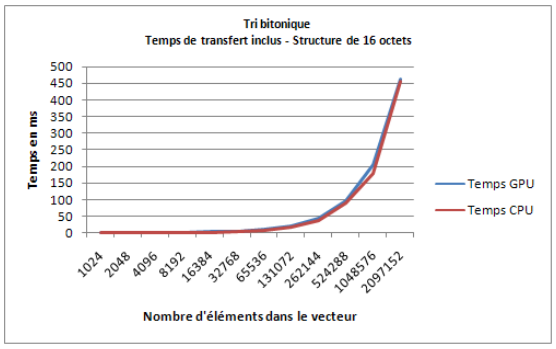
(e) Tri fusion appliqué à des structures de 8 octets, temps de transfert non inclus



(f) Tri fusion appliqué à des structures de 8 octets, temps de transfert inclus



(g) Tri fusion appliqué à des structures de 16 octets, temps de transfert non inclus



(h) Tri fusion appliqué à des structures de 16 octets, temps de transfert inclus

FIG. 1.12 – Performances obtenues pour les tris avec l'algorithme fusion, l'algorithme bitonique et l'algorithme Introsort (CPU)

Nombre d'éléments	CPU ↦ GPU	Traitement GPU	GPU ↦ CPU	Temps GPU
4096	0.040	0.297	0.321	0.658
8192	0.069	0.359	0.418	0.845
16384	0.308	0.472	0.628	1.408
32768	0.576	0.980	0.986	2.542
65536	1.134	1.805	1.817	4.756
131072	2.209	3.704	3.307	9.220
262144	3.682	8.034	5.138	16.854
524288	6.623	18.173	8.531	33.327
1048576	12.449	41.393	15.363	69.205
2097152	24.170	94.307	29.163	147.641

FIG. 1.13 – Performances obtenues pour les tris avec l'algorithme bitonique appliqués à des structures de 8 octets. Les temps sont exprimés en millisecondes.

Nombre d'éléments	Temps CPU	Gain de temps procuré par l'emploi du GPU
4096	0.288	-129%
8192	0.617	-37%
16384	1.379	-2%
32768	2.869	11%
65536	6.332	25%
131072	14.407	36%
262144	32.243	48%
524288	71.570	53%
1048576	155.286	55%
2097152	361.898	59%

FIG. 1.14 – Performances obtenues pour les tris avec l'algorithme bitonique et l'algorithme Introsort (CPU) appliqués à des structures de 8 octets

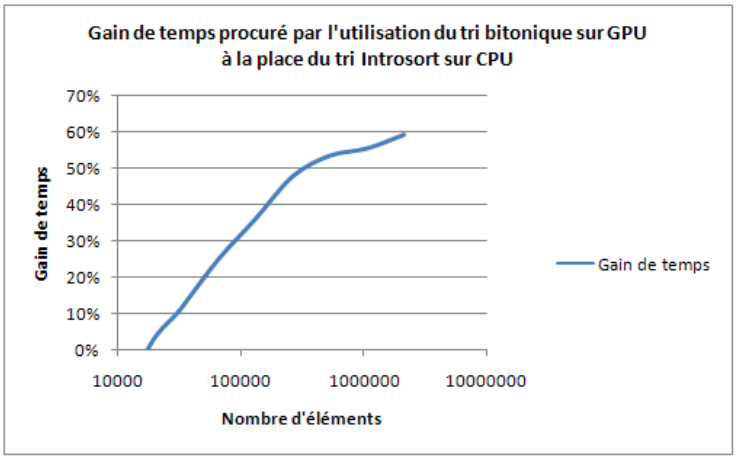


FIG. 1.15 – Gain de temps procuré par l'emploi du tri bitonique sur GPU (temps de transfert compris) à la place du tri Introsort sur CPU dans le cas d'un tri de structures de 8 octets

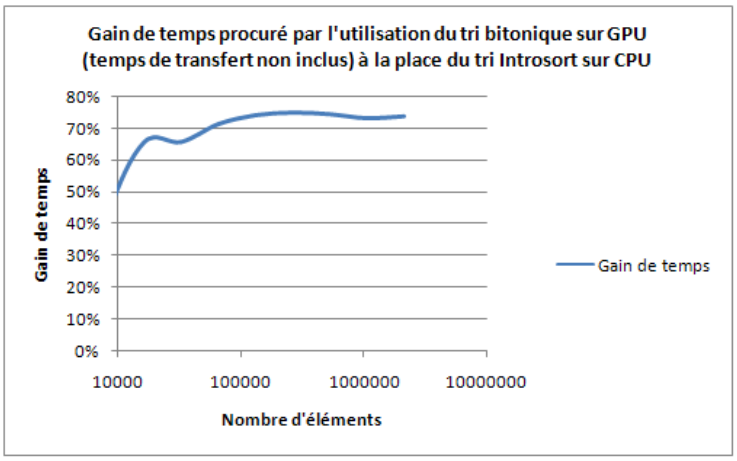


FIG. 1.16 – Gain de temps procuré par l'emploi du tri bitonique sur GPU (temps de transfert non compris) à la place du tri Introsort sur CPU dans le cas d'un tri de structures de 8 octets

Chapitre 2

Application en bio-informatique

2.1 Introduction

Dans le chapitre précédent, il a été présenté les nouvelles technologies de processeur et les possibilités qu'offrent les GPU. Afin de démontrer leurs utilités, nous avons décidé de les utiliser pour améliorer le temps d'exécution de l'application en bio-informatique BioloMICS [4], développé par la société BioAware [5]. L'optimisation de cette application, et plus particulièrement de l'un de ses algorithmes, DNARun[1], nous a été confiée dans le cadre de l'action FEDER-TIC[2]. L'algorithme DNARun est un algorithme s'exécutant dans un thread unique. Notre tâche était de le porter et de le paralléliser sur plusieurs coeurs hétérogènes. Nous avons ainsi travaillé sur deux fronts : M. Bagein fut chargé d'optimiser le code mono-thread et je fus ensuite chargé de le paralléliser sur CPU et GPU.

Avant de présenter DNARun et nos travaux exécutés à son niveau, une introduction du contexte de la création de l'algorithme sera développée, afin de faciliter la compréhension de ses enjeux. Suivra ensuite une introduction sur l'ADN et le séquençage, qui sera complétée par une description de la classification et l'identification des individus grâce à leur ADN. Les explications concernant ces divers sujets ont par endroit été simplifiées afin de permettre au lecteur une meilleure compréhension du sujet.

2.2 Acide désoxyribonucléique (ADN)

Cette section se base sur les informations issues de [56] et [57].

L'ADN, également appelé nucléotide, est une molécule que l'on peut trouver dans l'ensemble des cellules vivantes et qui possède plusieures fonctions :

1. Stocker l'information génétique qui conditionne le développement et le fonctionnement d'un organisme.
2. Transmettre cette information de génération en génération, permettant ainsi l'hérédité.
3. Permettre une diversité des individus de par la modification de certains nucléotides lors de la transmission des séquences ADN.

Un nucléotide est constitué de trois parties :

1. Un groupe phosphate.
2. Un sucre, le désoxyribose.
3. Une base azotée.

[1] Provient de l'anglais, signifie Exécute ADN. L'ADN est présenté en details à la section 2.2.
[2] Cette action porte ce nom car elle est financée par le FEDER [55]

Il existe quatre bases azotées : l'adénine (notée A), la thymine (notée T), la cytosine (notée C) et la guanine (notée G). Puisque les phosphates et les sucres sont toujours les mêmes, la nature du nucléotide est déterminée par la base azotée qu'il contient. Les ADN sont par conséquent appelés selon ces dernières.

Les nucléotides sont complémentaires et, dans le cas des brins d'ADN, liés entre eux par des liaisons hydrogènes. C'est ainsi que les brins d'ADN sont constitués de deux séquences de nucléotides se faisant face et formant une double hélice, comme présenté à la figure 2.1. Le complément de l'adénine est la thymine, et le complément de la cytosine est la guanine. La guanine n'est donc jamais liée avec l'adénine ou la thymine. Une représentation d'un brin d'ADN mis à plat est présenté à la figure 2.2. Vous pouvez y constater la complémentarité des nucléotides ainsi que la manière dont ceux-ci sont liés entre eux.

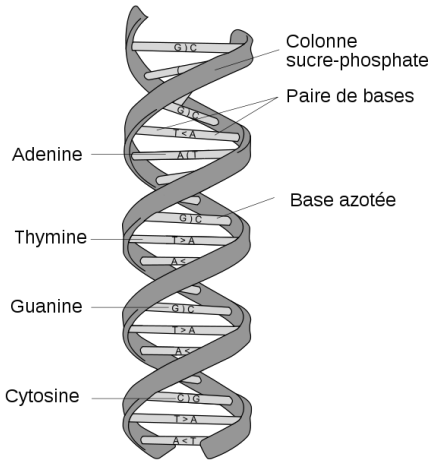


FIG. 2.1 – Brin d'ADN, constitué de deux séquences de nucléotides se faisant face et formant une double hélice (provenance [58])

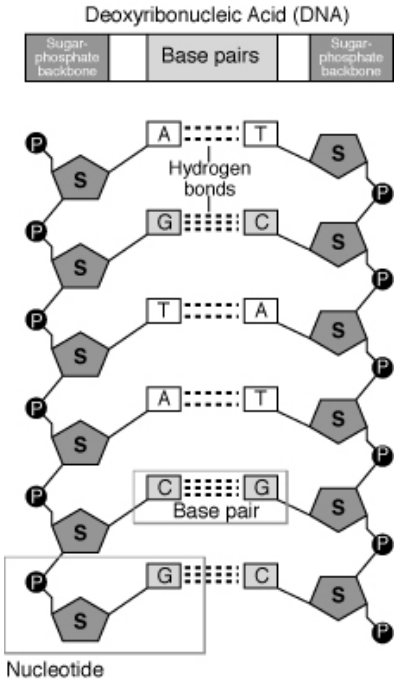


FIG. 2.2 – Représentation d'un brin d'ADN mis à plat et mettant en évidence la structure et la complémentarité des nucléotides (provenance [59])

2.3 Le séquençage de l'ADN

2.3.1 Généralité et utilité

Cette section se base sur les informations issues de [60], [61], [62], [63], [64], et [65].

L'ADN est comme une grande bibliothèque de gènes, ces derniers fournissant l'information qui permet de synthétiser les protéines dont une cellule a besoin pour fonctionner. Le séquençage d'un ADN a pour but de lire les nucléotides d'un brin d'ADN et de mettre en évidence ces gènes. Cette annotation n'est pas simple car, contrairement à un livre, tous les mots d'un génome[3], ne sont pas utiles pour en comprendre l'histoire. Chez les mammifères, les gènes n'occupent que 10% de l'ADN et ceux-ci ne sont pas tous biologiquement significatifs. Au final, seul 3% du génome est vraiment utile, et ces quelques pourcents ne sont pas faciles à délimiter.

En 1990 débuta le Projet génome Humain, entreprise dans laquelle se sont lancés l'Allemagne, la Chine, les Etats-Unis, la France, le Japon et le Royaume-Uni. Le but pour ce consortium mondial public était d'arriver à séquencer le génome entier d'un homme, soit près de trois milliards de paires de bases[4]. Ce projet qui s'est terminé avec succès en 2004 a donc duré 14 ans et a coûté près de 2,7 millards de dollars américains.

Ces investissements n'ont pas été vains et encore moins inutiles, car le génome d'un individu est une mine d'informations pour le domaine médical. Les données récoltées avec le séquençage peuvent être utilisées dans plusieurs buts :

- Identifier les gènes associés à certaines maladies, que celles-ci soient usuelles comme le cancer ou l'arthrose, ou rares, telles l'hémochromatose et l'hyperparathyroïdie. Nous pouvions jusqu'ici prévoir certaines maladies de par leur côté héréditaire, mais il fallait pour cela qu'il y ait eu des antécédents familiaux, ce qui n'est pas le cas avec l'analyse des gènes.
- Mettre en évidence les gènes pouvant avoir de l'influence sur l'efficacité ou les effets secondaires de médicaments. Ceci est fort utile car un grand nombre de médicaments ne sont efficaces que sur 30 à 50 % de la population et peuvent même, dans certains cas extrêmes, aller jusqu'à intoxiquer le patient. Par exemple, la Rézuline, médicament administré contre certains diabètes, est la cause de 60% de décès par empoisonnement hépatique. Un test génétique individuel pourra donc servir à établir des traitements personnalisés comportant des risques bien moindres pour les patients.

Les généticiens reconnaissent qu'il est, pour le moment, difficile d'intégrer ces données en pratique clinique : certains gènes peuvent avoir des effets opposés et être présents dans le même génome, et leur impact peut également dépendre des interactions avec l'environnement et le comportement. Il faut donc maintenant mettre au point des méthodes d'aide à la décision intégrant les données génétiques et cliniques.

Enfin, le sujet est également lié à des questions éthiques : qui devrait avoir son génome séquencé ? Quel conseil devrait être proposé avant et après le séquençage et par qui ? Et qui devrait avoir accès à des informations génétiques individuelles ? Un autre sujet de débat est le brevetage des gènes. En effet, si le brevet garantit des revenus aux entreprises privées qui les analysent, cela ralentit également l'avancement globale des recherches.

A l'heure actuelle, le séquençage d'un individu coûte 10 000 dollars américains et pourrait rapidement tomber à 1000 dollars. Même si le prix du séquençage est donc en baisse, il y a actuellement un faible nombre de conseillers en génétique et de généticiens et ces derniers sont, de surcroît, surchargés de travail. De plus, bien que les techniques de séquençage aillent mille fois plus vite qu'à l'époque du consortium mondial, séquencer un génome humain est une opération encore coûteuse en temps. Les spécialistes espèrent tout de même arriver à exécuter cette opération en

[3] L'ensemble du matériel génétique d'un individu

[4] Ceci représente assez de lettres A, T, C et G pour remplir plus de 200 annuaires téléphoniques

une semaine d'ici une dizaine d'années, mais ce n'est pas encore pour demain que n'importe qui pourra voir son génome séquencé.

2.3.2 Techniques de séquençage

Cette section se base sur les informations issues de [66], [67], et [68].

De nombreuses techniques existent pour séquencer un brin d'ADN mais, comme le but de ce travail n'est pas d'en faire un compte-rendu mais juste de présenter le contexte dans lequel s'est développé DNARun, je ne présente ici que la plus connue : la méthode de Sanger[5].

Avant de présenter la technique en elle-même, deux types de nucléotides, appelés ADN polymérases, doivent être présentés : les désoxyribonucléotides, notés dNTP[6], et les didésoxyribonucléotides, notés ddNTP[7]. Les dNTP peuvent s'ajouter les uns aux autres et synthétiser ainsi un brin d'ADN simple-brin. Les ddNTP peuvent s'ajouter aux dNTP mais, en faisant cela, ils clôturent le brin : il leur manque un groupement OH grâce auquel les dNTP peuvent se lier entre eux (voir figure 2.3). Ces deux familles de nucléotides connaissent une version pour chaque base azotée vue à la section 2.2 : les A (dATP, ddATP), les C (dCTP, ddCTP), les G (dGTP, ddGTP) et les T (dTTP, ddTTP).

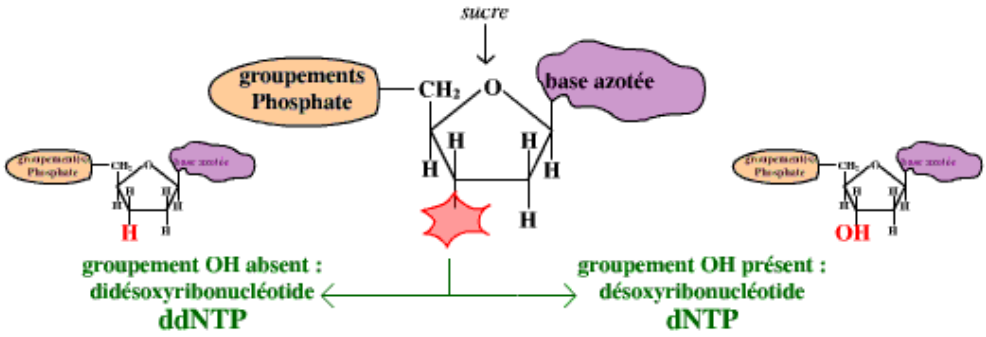


FIG. 2.3 – Représentation d'un ddNTP (à gauche) et d'un dNTP (à droite) (provenance [66])

La technique de Sanger consiste à placer une multitude de clônes du segment ADN simple-brin à séquencer dans quatre milieux différents : chacun des milieux contient les quatre dNTP en grande quantité ainsi qu'une faible concentration d'un des quatre ddNTP. Les dNTP et les ddNTP synthétisent le brin complémentaire aux clônes en se liant à la fois les uns aux autres et en même temps aux nucléotides complémentaires du brin d'ADN séquencé. Cependant, comme les dNTP et les ddNTP doivent se lier à d'autres nucléotides sur leur simple-brin, ils ne peuvent pas entamer le séquençage tant que d'autres ADN n'ont pas commencé à se lier aux clônes. Pour ce faire, une autre enzyme, la primase, est employée pour synthétiser un oligonucléotide[8] complémentaire aux premiers nucléotides du brin ADN à séquencer. Cet oligonucléotide est nommée "amorce" vu qu'elle permet d'amorcer le séquençage.

Une fois l'amorce placée, les dNTP complémentaires aux nucléotides de la séquence analysée viennent se greffer à celle-ci et le brin d'ADN complet se forme petit à petit (voir figure 2.4. Ce phénomène s'arrête lorsqu'un ddNTP vient, par hasard, se placer à la fin de la chaîne en cours de synthèse. Une fois la réaction finie, les quatre environnements contiennent de nombreuses chaînes ADN de longueurs différentes, toutes terminées par l'un des quatre ddNTP en fonction du milieu

[5]Le nom provient de son créateur, Frederick Sanger, qui fut récompensé pour la seconde fois par le prix Nobel de chimie en 1980 pour cette méthode
[6]dntp : désoxyNucléotide TriPhosphate
[7]didésoxyNucléotide TriPhosphate
[8]Courte séquence de nucléotides dont la taille est comprise entre 15 et 25 pour le séquençage

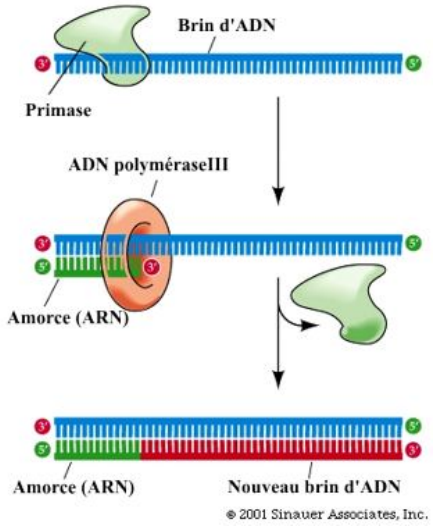


FIG. 2.4 – Présentation de la synthèse d'un brin complémentaire au brin séquencé (provenance [69])

dans lequel elles se trouvent. Celles-ci sont triées par taille et exposées à un laser. Comme les ddNTP sont marqués grâce à l'ajout d'une molécule fluorescente, le laser va pouvoir observer quelle base azotée termine la chaîne, et donc quel nucléotide se trouve à la position déterminée par la longueur de la séquence. Une fois que le laser a identifié les nucléotides se trouvant en fin de chaque chaîne, la séquence de ddNTP est identifiée, et il ne reste plus qu'à prendre les nucléotides complémentaires pour retrouver le simple-brin d'ADN séquencée. Le processus est illustré à la figure 2.5.

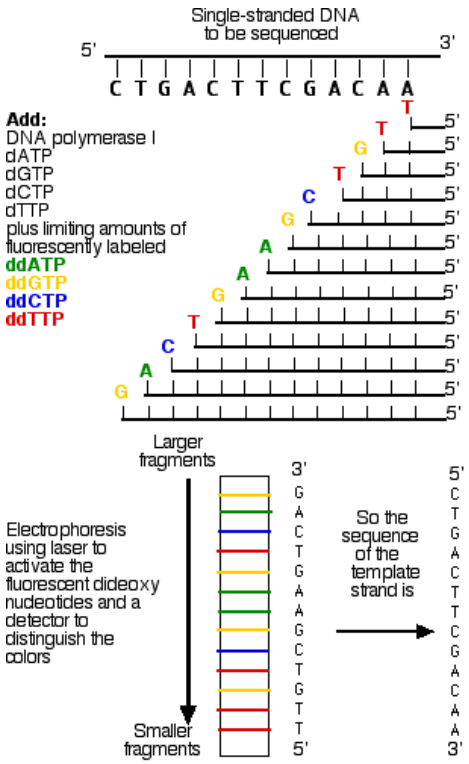


FIG. 2.5 – Illustration du processus de séquençage (provenance [67])

Les techniques de séquençage actuelles ne livrent que des séquences de 1000 nucléotides au

plus. Afin de séquencer le génome humain entier, celui-ci est divisé en petites chaînes ADN qui sont séquencées les unes indépendamment des autres puis réassemblées à la fin du processus. Les techniques qui permettent d'effectuer cette opération ne sont pas présentées ici car elles ne rentrent pas dans le cadre de ce travail. Le lecteur intéressé peut consulter les sources [70], [71] et [72] s'il désire avoir plus d'informations à ce sujet.

2.4 Classification et identification des individus

Cette section se base sur les informations apportées par M. Szöke Szaniszló, représentant de la société Bio-Aware pour l'algorithme DNARun. Ces données ont dans un premier temps été récoltées au cours d'un entretien au CETIC, et elles ont été complétées dans un second temps grâce à un entretien téléphonique.

Les séquences ADN de différents individus[9] comportent des différences et des points communs. En effet, les gènes sont liés à nos fonctions et caractéristiques physiologiques et anatomiques. Ainsi, certains gènes sont responsables de la couleur d'une fleur ou du phénomène de lactation chez les mammifères. Comme ces gènes sont propres à certains individus, certaines espèces, certaines familles ... leur identification et leur recherche dans un génome permettraient d'identifier l'individu à qui ils appartiennent. Cette opération peut néanmoins se révéler assez ardue étant donné que certaines espèces possèdent très peu de différences entre elles. Ainsi, l'ADN du chimpanzé et celui de l'homme sont identiques à 99%, et la différenciation se base donc au mieux sur 1% du génome entier.

Si nous voulons par exemple pouvoir identifier un oiseau, nous allons chercher, dans le génome, le gène qui est à l'origine de la production de plumes. Pour ce faire, nous cherchons le début du gène connu et constant et nous le séquençons de gauche à droite. Le séquençage devenant de moins en moins précis lorsqu'il parcourt le brin ADN, une technique pour augmenter sa fiabilité consiste à séquencer, dans le même temps, le brin de la droite vers la gauche. Il faut donc une paire d'amorces pour effectuer un tel séquençage. Les personnes étudiant les oiseaux vont toutes utiliser les mêmes paires d'amorces et étudier les différences qu'il existe dans les séquences ADN ainsi récoltées entre les oiseaux d'une même espèce et entre les oiseaux d'espèces différentes. Ce processus permet de classifier les individus.

Lorsque nous analysons ainsi la similarité des génomes entre deux individus, nous désirons que celle-ci soit forte si les individus sont fort proches l'un de l'autre et faible si les individus sont éloignés (si nous comparons l'ADN d'un champignon à celui d'un reptile par exemple). Il faut donc trouver dans le génome complet quels sont les meilleurs gènes pour identifier et pour classifier les individus. Pour ce faire, une technique consiste à directement comparer le génome de deux individus. Le problème est que, si les individus sont fort éloignés, nous ne trouvons que peu de gènes en commun, et nous avons donc peu de renseignements sur les éléments à utiliser pour les distinguer.

Une autre technique consiste à ne plus rechercher les gènes mais les amorces communes à certaines espèces. Nous partons d'une série d'amorces et nous regardons combien de fois nous pouvons la trouver dans les différentes espèces d'oiseaux. Nous prenons alors les amorces qui se retrouvent dans la plupart des espèces, mais celles-ci peuvent être nombreuses, de l'ordre de 200 000 unités. Afin de réduire ce nombre, nous recherchons dans ces amorces des paires, espacées de 800 à 1000 nucléotides, présentes chez les différents individus. Une fois ces paires trouvées, nous analysons les similarités et les différences des morceaux d'ADN se trouvant entre les paires et créons un arbre de similarité des espèces à partir de cette analyse. Un arbre de similarité est un arbre dont les branches sont plus ou moins proches si les espèces sont plus ou moins semblables. Si l'arbre de similarité construit à l'aide de la génétique représente bien les différences ou les similarités entre les individus, les paires d'amorces trouvées sont approuvées pour les identifier et les classifier.

[9] L'individu doit ici être considéré comme étant l'unité élémentaire d'une espèce

Le but de tout ceci est de pouvoir identifier un individu à l'aide de quelques paires d'amorces seulement, vingt au maximum. Nous pourrions ainsi, par exemple, découvrir chez un patient victime d'une infection quelle est la bactérie responsable de ses maux grâce à une prise de sang et un rapide séquençage à l'aide de ces amorces. Cela irait beaucoup plus vite que les techniques actuelles (cultures ...).

2.5 DNARun

2.5.1 Description générale

Le but de DNARun est de comparer une série de morceaux d'ADN simple-brin, les sources, à une autre série de morceaux d'ADN simple-brin, les références. Nous pourrions donc charger des amorces dans les sources et les génomes[10] à scruter dans les références. Le nombre de comparaisons potentielles est donc très élévé, ce qui fait de cette application un algorithme de premier choix pour être parallélisé.

Lorsque nous l'avons reçu, l'algorithme entrait directement dans une boucle dont vous pouvez voir le pseudo-code au listing 2.1. L'initialisation des sources se fait très bas dans le code, au niveau de la fonction de comparaison, c'est pour cela qu'elle n'est pas visible à ce niveau.

Listing 2.1 – Pseudo-code de l'algorithme DNARun simplifié

```
Pour SourcesIdentifiants.Début à SourcesIdentifiants.Fin :
    ChargerSourceDepuisBaseDeDonnées()
    Pour RéférencesIdentifiants.Début à RéférencesIdentifiants.Fin :
        ChargerRéférenceDepuisBaseDeDonnées()
        InitialiserRéférence()
        ComparerLesNucléotidesDeLaSourceAvecCeuxDesRéférences()
    Fin Pour
Fin Pour
```

2.5.2 Travail accompli sur DNARun

L'application initiale, écrite en C++, était conçue pour tourner sur un système Windows 32 bits. L'une des premières optimisations fut de le porter sur un système 64 bits. Cette opération a été initiée au CETIC [11] [74] par M. Orlando Casano, notre contact au centre de recherche pour tout ce qui concerne DNARun. La FPMs, disposant d'un cluster de 20 machines bi-processeurs Linux 64 bits, dont une équipée de 4 GPU, a été désignée comme cible test pour la réalisation de la nouvelle version 64 bits de DNARun. Quelques adaptations furent réalisées pour adapter la compilation sur le cluster FPMs (essentiellement du paramétrage des outils de compilation), suivies de nombreuses évolutions du code permettant progressivement de produire un code commun Windows/Linux 32/64 bits optimisé.

Après avoir examiné le code pour comprendre son fonctionnement et optimiser certains passages, nous avons utilisé l'environnement de développement Code : :Blocks [75] et l'outil de profilage GNU Prof [76] afin de déterminer quelles sont les fonctions candidates pour un passage sur GPU. GNU Prof est un outil qui permet de mesurer combien de temps l'exécution passe dans chaque fonction et ainsi quelles sont les fonctions les plus gourmandes en temps de calcul. Ces dernières sont les premières cibles pour une parallélisation étant donné que plus une fonction

[10] Comme un génome est assez grand, il est d'usage de le découper en petits morceaux, un génome occuperait donc de nombreuses références

[11] Signifiant Centre d'Excellence en Technologies de l'Information et de la Communication

prend du temps lors de son exécution, plus un gain à son niveau équivaut à un gain global important. Code : :Blocks est un environnement qui permet de faciliter le développement du code. Il possède entre autres une option qui permet de lancer le profilage du code et un outil qui affiche graphiquement les résultats.

Les résultats obtenus avec ces outils sont présentés à la figure 2.6. Comme vous pouvez le voir, les fonctions qui prennent la majorité du temps de calcul (69,24%) sont *Trace*, *IntroSort*, *StrictSearch*, et *InitSrce* :

- *Trace* : fonction créée par M. Bagein. A chaque appel, elle place une ligne de texte définie par l'utilisateur dans un fichier texte. Elle permet donc d'obtenir des informations sur l'état d'une variable, l'évolution d'une fonction ...
- *IntroSort* : algorithme de tri de la bibliothèque STL [38] présenté à la section 1.6.1 et utilisé au sein de la fonction de tri standard *std : :sort*, bibliothèque qui gère la classe `vector` dans laquelle sont stockés les nucléotides.
- *StrictSearch* : fonction de très bas niveau comparant les nucléotides entre eux.
- *InitSrce* : fonction initialisant les sources, se trouvant dans la fonction de comparaison. Son appel précède celui de la fonction *StrictSearch*.

Vu que la fonction *Trace* sert principalement à avoir des informations sur le déroulement de l'algorithme, elle sera retirée pour la version finale et ne pose donc pas de problème. La fonction *IntroSort* intervient principalement à deux moments : lors de l'initialisation des sources et lors de l'initialisation des références. Elle intervient donc avant et dans la fonction de comparaison. Il est dès lors intéressant de trouver une fonction qui permette de trier sur GPU. Comme la taille des vecteurs qui sont triés pour chaque source et pour chaque référence ne dépasse pas l'ordre des centaines, et que le GPU est optimisé pour le traitement de plus grands ensembles de données, nous avons pensé à sortir le chargement et l'initialisation des sources et des références de la boucle des comparaisons. Ceci nous permettrait de toutes les initialiser en même temps et ainsi de trier un vecteur dans lequel se trouveraient toutes les sources et toutes les références, chacune identifiée par un numéro. Le programme aurait donc l'architecture présentée au listing 2.2. Ceci permettrait également de grandement diminuer le nombre d'appels à la base de données, vu qu'il n'y en a plus que deux grâce auxquels nous chargerions toutes les sources et toutes les références.

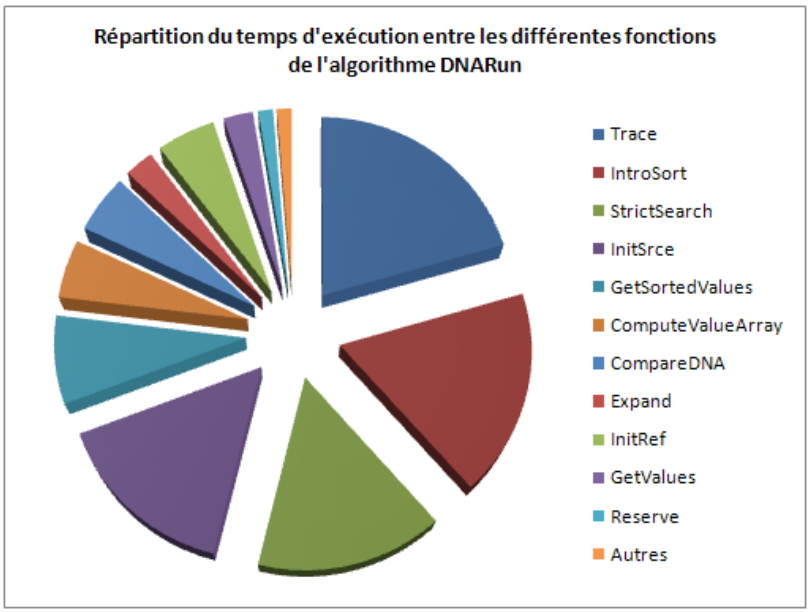


FIG. 2.6 – Répartition du temps d'exécution entre les différentes fonctions de l'algorithme DNA-Run

Listing 2.2 – Pseudo-code de l'algorithme DNARun modifié

```
ChargerSourcesDepuisBaseDeDonnées()
ChargerRéférencesDepuisBaseDeDonnées()
InitialiserSourcesEtRéférences()
Pour SourcesIdentifiants.Début à SourcesIdentifiants.Fin :
    Pour RéférencesIdentifiants.Début à RéférencesIdentifiants.Fin :
        ComparerLesNucléotidesDeLaSourceAvecCeuxDesRéférences()
    Fin Pour
Fin Pour
```

Les données triées dans les fonctions *InitSrce* et *InitRef* sont contenues dans une structure *TTrio* dont le code peut être trouvé au listing 2.3. Les explications concernant les membres de cette structure peuvent être trouvées à l'annexe B. La structure présentée ici tient sur huit octets. Cette version est déjà une version améliorée de la structure. Dans son état initial, la structure occupait plus de huit octets. Nous l'avions réduite afin qu'elle puisse être lue en une seule opération par un système 64 bits et en deux opérations par un système 32 bits. Vu que cette structure est souvent utilisée, cela représente un net avantage.

Comme il faut pouvoir identifier à quelle source ou à quelle référence un *TTrio* du vecteur trié appartient, il est nécessaire d'ajouter un identifiant à la structure `TTrio`. Une fois le tri implémenté sur le GPU, nous avons constaté que trier des données dont la taille est supérieure à huit octets s'avère infaisable avec les choix que nous avons faits (voir section 1.6.3). Il est donc nécessaire de diminuer la taille de la structure *TTrio* si nous désirons y placer l'identifiant. Une analyse de la variable `m_I` nous a indiqué que celle-ci ne dépasse pas la valeur 200. Il y a donc de nombreux bits de cette variable qui restent inutilisés. A contrario, la variable `m_Val` utilise pratiquement tout l'espace disponible et ne laisse que deux à trois bits libres. Nous avons donc choisi de diminuer la taille de `m_I` au profit de l'identifiant.

Listing 2.3 – Structure TTrio

```
struct TTrio
{
    union
    {
        struct
        {
            union
            {
                struct
                {
                    unsigned int m_L : 3 ;
                    unsigned int m_I : 29 ;
                };
                unsigned int m_Pos;
            };
            unsigned int m_Val;
        };
        unsigned __int64 m_Long;
    };
};
```

Afin de tirer profit des performances des coeurs CPU et des coeurs GPU, le long vecteur contenant les sources et les références sera à terme fragmenté et trié sur les différents processeurs. Les *TTrio* étant triés selon leur identifiant et leur valeur, les nucléotides d'un même segment d'ADN restent groupés dans le vecteur. La fragmentation de ce dernier ne pose pas de problème

tant que tous les *TTrio* d'une source ou d'une référence se trouvent dans un même fragment. Je n'entre pas ici plus dans les détails car cette opération est le sujet du chapitre suivant. Etant donné que le gain de temps du tri sur GPU par rapport au tri sur CPU n'augmente plus rapidement une fois les 2^{18} éléments[12] atteints (voir figure 1.16, page 26), il a été décidé que les fragments possèderont une taille maximale de 262 144 éléments. Afin de faire face au pire des cas, nous avons réservé 18 bits pour coder l'identifiant d'un *TTrio*. Ceci permet d'avoir un identifiant par *TTrio* se trouvant dans un même fragment, ce qui est assez grand étant donné que tout vecteur devant être trié possède au moins deux éléments. Il est donc possible d'augmenter la taille des fragments au besoin. Cela laisse également dix bits à la variable `m_I`, soit 2048 valeurs possibles alors qu'elle ne dépasse pas la valeur 200. La modification de taille ne pose donc pas de problème à cette variable. La nouvelle structure *TTrio* est présentée au listing 2.4 et vous pouvez également y voir les opérateurs qui permettent de comparer deux éléments.

La comparaison s'effectue sur deux éléments : d'abord sur l'identifiant et ensuite sur la valeur contenue dans la variable `m_Val`. Comme vous pouvez le constater, les membres de la structure ont été positionnés de manière à ce que la comparaison de deux éléments se fasse en une seule opération. En effet, la comparaison des variables `m_Long` entraine dans le même temps la comparaison de l'`union` contenant l'identifiant, et la structure contenant la valeur. Comme l'identifiant occupe les bits de poids les plus lourds, c'est cette variable qui déterminera la comparaison de l'`union`. Ceci permet une comparaison optimisée tant sur le GPU que sur le CPU.

Listing 2.4 – Structure TTrio modifiée

```
struct TTrio
{
    union
    {
        struct
        {
            union
            {
                struct
                {
                    unsigned int m_L : 3;
                    unsigned int m_I : 11;
                };
                unsigned int m_Pos : 14;
                unsigned int m_Id : 18;
            };
            unsigned int m_Val ;
        };
        unsigned __int64 m_Long ;
    };

    bool operator < ( const TTrioKernel & srce ) const
    {
        return srce.m_Long > m_Long ;
    }

    bool operator > (const TTrioKernel &srce) const
    {
        return m_Long > srce.m_Long ;
    }
}
```

[12] $2^18 = 262144$

Lors de l'adaptation de DNARun pour inclure le tri, nous avons pu remarquer que, quel que soit le changement que nous effectuons, cela altère certains résultats. Les changements incluent :

- L'ajout d'un membre dans une classe
- L'ajout, dans le corps de l'algorithme, de l'appel d'une fonction n'ayant aucune influence sur le déroulement de ce dernier
- L'ajout d'une fonction membre
- ...

Il n'est en réalité pas possible de faire la moindre modification sans que cela produise, pour certaines plages de données, une exécution différente de l'algorithme. Après maintes recherches, nous avons conclu que ces problèmes provenaient de plusieurs sources :

- Existence d'un bogue dans l'algorithme
- Présence de l'opérateur `dynamic_cast` : cet opérateur permet de convertir le type d'une variable. Il est dit dynamique car la conversion se fait lors de l'exécution et non à la compilation. Il est donc moins sécurisé que les autres opérateurs de conversion.
- Comportements erratiques de la classe *stringtpl*

Je ne vais pas ici entrer dans les détails, mais il est possible à toute personne ayant des connaissances en informatique de résoudre les deux derniers problèmes en corrigeant le code source de l'algorithme. Le premier par contre concerne le déroulement de l'algorithme et nécessite donc l'intervention d'un membre de l'équipe initiale de développement. Comme les personnes responsables n'ont pas eu jusque maintenant le temps de s'attarder sur ce problème, il a été décidé de mettre en suspens le passage sur GPU de l'algorithme. En effet, il ne sert à rien de paralléliser une version instable de l'algorithme, cela ne faisant qu'augmenter son instabilité.

Une fois que l'algorithme aura intégré le chargement et l'initialisation en un coup des sources et des références, il serait intéressant de se pencher sur la parallélisation sur GPU des comparaisons elles-mêmes. Etant donné que les sources et les références sont indépendantes les unes des autres, il serait possible de placer sur GPU la comparaison de ces dernières, si la quantité de mémoire locale du GPU suffit. Cette opération risque de prendre un certain temps étant donné que l'application est écrite en C++ et que le GPU n'accepte actuellement que le C. Il faudra donc convertir le code et réécrire certaines fonctions membres de classes auxquelles nous n'avons pas accès (comme les fonctions de la classe *vector*). Le gain serait cependant important étant donné que le corps de l'algorithme se trouverait parallélisé sur de nombreux coeurs.

2.6 Conclusions

La génétique est un domaine très intéressant, particulièrement de par les perspectives qu'offre le séquençage de l'ADN. Les techniques actuelles dans ce domaine prennent cependant encore trop de temps et ne sont pas encore toutes au point. Des algorithmes comme DNARun sont des outils qui permettent d'obtenir rapidement des informations qui permettront d'avancer dans ces recherches. Plus les outils sont rapides, plus les recherches pourront vite avancer. Il est donc important d'optimiser de tels outils.

A cause du bogue dans l'algorithme, nous n'avons pas encore pu lui intégrer le tri GPU, mais ce dernier a déjà été conçu et fonctionne avec les données du problème. Il ne reste donc plus qu'à mettre son appel dans DNARun. De plus, l'architecture de l'algorithme laisse de grands espoirs tant au portage sur les accélérateurs graphiques, qu'une répartition entre plusieurs clusters des comparaisons de chaines d'ADN. Le travail à apporter à son sujet est donc encore important, mais tout autant le seront les gains de performance.

Chapitre 3

La distribution de tâches concourantes sur des architectures multi-coeurs hétérogènes

3.1 Introduction

Jusqu'à ces dernières années, la programmation générale sur GPU n'était pas encore pleinement adoptée par le public, comme le prouve cette discussion [77] datant de 2006 sur le forum français d'informatique Hardware.fr. La raison était le manque d'outils pour programmer et déboguer le code, ce qui engendrait des calculs peu précis. En 2007, CUDA fit son apparition et il est maintenant plus aisé de porter un calcul sur GPU.

La distribution de tâches[1] concourantes permet de diminuer la quantité de travail en divisant un calcul intensif en plusieurs éléments de travail qui seront exécutés en même temps sur différents coeurs de processeur. La majorité des outils actuels permettant de distribuer les tâches ne se sont pas encore tournés vers les solutions GPU. Dernièrement, des chercheurs du MIT[2] ont identifié les faiblesses des anciens outils tels que MPI[3] [78], OpenMP[4] [79], Cilk++ [80] ...et ont mis au point un nouveau système de distribution de tâches : Geonumerics [81]. Bien que ce système s'avère plus efficient que ses prédécesseurs, il ne fait que mieux gérer la distribution des tâches sur les coeurs CPU.

Depuis février 2008, une équipe de scientifiques de l'INRIA a développé StarPU [3], un support exécutif unifié pour les architectures multicoeurs hétérogènes qui permet de distribuer des tâches non seulement sur CPU, mais également sur GPU et Cell. Bien qu'il ne soit pas aussi efficient qu'OpenMP ou MPI pour distribuer des tâches uniquement sur CPU, il s'avère fort efficace pour balancer la charge de travail entre les coeurs hétérogènes. Ceci est réalisé grâce à une rapide estimation du temps d'exécution des tâches sur les différents types de coeurs. Il permet de s'assurer que tous les coeurs aient continuellement du travail, si la présence de tâches le permet, et d'éviter que les dernières tâches soient lancées sur les coeurs les plus lents et retardent ainsi la fin de l'exécution.

Ce chapitre présente dans un premier temps le fonctionnement de StarPU à travers ses atouts et, dans un deuxième temps, l'adaptation de l'algorithme de tri présenté dans les deux derniers chapitres

[1] Une tâche est un ensemble cohérent et contigu d'instructions
[2] De l'anglais Massachusetts Institute of Technology, signifiant Institut de la Technologie du Massachusetts
[3] De l'anglais Message Passing Interface signifiant Interface de Passage de Messages
[4] De l'anglais Open Multi-Processing signifiant Traitement Multiple Ouvert

3.2 Les intérêts de StarPU et leur implémentation

Cette section se base sur les informations issues de [82] et de ma correspondance avec Cédric Augonnet.

3.2.1 Un modèle d'exécution unifié

Comme mentionné plus haut, des outils comme Geonumerics ne suffisent pas à tirer profit de toutes les architectures. De tels outils ne sont pas adaptés lorsque la machine sur laquelle s'exécutera le code n'est pas à priori connue. Par contre, le parallélisme de tâches permet de construire des applications portables et efficaces si un gestionnaire de tâches tel StarPU gère leur distribution ainsi que le transfert données.

Afin d'exploiter toutes les architectures existantes, StarPU a implémenté un modèle de tâches multiversionnées en utilisant les notions de codelet et de tâche :

- Codelet : le codelet est un noyau de calcul multi-versionné, indiquant sur quelles architectures il peut s'exécuter et contenant les implémentations associéees.
- Tâche : une tâche est un codelet appliqué à des données.

Pour porter un calcul sur GPU et sur CPU, il suffit de coder les fonctions exécutant ce dernier pour les différentes architectures et de les indiquer au codelet. Lorsque la tâche doit être effectuée, le programmeur passe simplement le codelet et les données sur lesquelles il s'applique à StarPU. Ce dernier, explorant la structure de l'ordinateur, vérifie si un GPU au moins est présent et estime sur quel coeur il est préférable que la tâche s'exécute.

3.2.2 Gestion des données

Lors de l'utilisation de GPUs, il faut transférer les données dans leurs mémoires avant de pouvoir les traiter. Comme nous avons pu le voir au chapitre précédent (voir 1.6.3), cette opération n'est pas sans engendrer une perte de temps. Un moyen de gagner du temps dans une application qui fait plusieurs calculs à l'aide des mêmes données est de laisser les données sur le GPU et les réutiliser ultérieurement. Gérer ceci n'est pas toujours évident pour le programmeur, c'est pourquoi StarPU intègre une bibliothèque mettant en œuvre une DSM[5] logicielle.

Le premier but de la DSM est d'assurer la cohérence des données. Pour ce faire deux types d'approche sont envisageables :

1. Write-Through[6] : les données sont conservées en mémoire centrale et on y accède avant et après chaque calcul
2. Write-Back[7] : les données sont transférées uniquement lorsqu'elles sont nécessaires

La deuxième approche étant moins coûteuse, c'est elle qui est utilisée par le DSM. Chaque donnée est associée à un vecteur qui décrit son état sur chaque noeud mémoire :

- Etat invalide : il n'y a pas de copie valide sur le noeud
- Etat modifié : la donnée est disponible sur le noeud uniquement
- Etat partagé : il existe plusieurs copies valides de la donnée, dont une sur le noeud

StarPU fournit des interfaces mémoires permettant au programmeur d'accéder à la donnée souhaitée. Lorsque ces interfaces sont appelées, le DSM vérifie l'état de la donnée et la transfère depuis le noeud où elle est marquée disponible si cela s'avère nécessaire. Lorsque nous avons pris en main StarPU, celui-ci contenait des interfaces pour gérer des matrices, des vecteurs ... mais n'en avait pas pour gérer des variables uniques comme un entier ou une structure. Afin de pouvoir également gérer ce genre de données, nous avons ajouté une telle interface. Elle figure maintenant avec les sources du projet.

[5] De l'anglais Data Sync Manager, signifiant Gestionnaire de la Synchronisation des Données
[6] De l'anglais, signifiant écrire à travers
[7] De l'anglais, signifiant répondre au moyen de

Une seconde utilité du DSM est de fournir un mécanisme de mémoire virtuelle. Certaines ressources mémoires étant limitées (voir 1.3), le DSM offre un cache logiciel afin qu'une tâche puisse s'exécuter, même lorsque les données nécessaires ne peuvent pas être placées dans la mémoire locale du noeud. Comme cette mémoire cache peut ne pas suffire, StarPU met en œuvre un mécanisme de libération de la mémoire qui supprime les données du cache, tout en assurant leur cohérence et leur disponibilité ultérieure.

Il existe un dernier mécanisme concernant les données et entièrement propre à StarPU : la notion de filtre. Comme le but est ici de répartir les traitements intensifs sur les différents coeurs de calculs d'une machine, une même tâche ne traitera en général pas l'ensemble des données du problème. Plutôt que de transférer tous les éléments de la matrice ou du tableau, le programmeur peut utiliser les filtres afin d'aisément délimiter la plage de données qui sera traitée par une tâche.

3.2.3 Ordonnancement des tâches

Pouvoir soumettre des tâches sur des architectures différentes est un avantage et, comme les temps d'exécution varient selon le type de processeur traitant le problème, il faut également un ordonnanceur qui permettra d'affecter les travaux aux différentes ressources. Ceci est principalement réalisé afin de remplir deux objectifs :

1. Equilibrer la charge de travail afin que toutes les unités de travail soient en tout temps utilisées
2. Eviter que les dernières tâches soient associées aux unités les plus lentes et retardent ainsi la fin du traitement

Comme le but premier de ce travail était de trouver ou de concevoir un tel ordonnanceur, nous avons étudié les différentes politiques d'ordonnancement de StarPU afin de voir comment il procède et ainsi nous assurer que celles-ci soient suffisament efficaces. Lorsqu'une tâche est soumise, elle suit d'abord un même chemin, quelle que soit la politique de plannification, et est ensuite traitée différement selon celle qui est choisie.

Avant de passer aux explications sur le cheminement d'une tâche au sein de l'ordonnanceur de StarPU, la notion de travailleur dans le cadre du support unifié ainsi que son utilisation doivent être expliquées. Un travailleur représente un coeur CPU, un GPU ou un Cell qui est capable d'exécuter des tâches. Avant de pouvoir faire quoi que ce soit avec StarPU, il faut appeler une fonction qui l'initialise. Durant cette étape, la librairie va effectuer de nombreuses actions dont notamment la recherche du nombre de travailleurs de chaque type se trouvant dans le système. Pour ce faire, elle peut procéder de deux manières différentes :

1. Utiliser les informations du système : sont utilisées la fonction Sysconf[8] pour trouver le nombre de coeur CPU et l'API Cuda pour le nombre de GPU.
2. Utiliser la librairie hwloc[9] [84] : cette bibliothèque permet de détecter le matériel présent dans un système, mais de manière plus aboutie que Sysconf [84]. Dans le cadre de StarPU, elle est particulièrement utile dans le cas des processeurs utilisant l'HyperThreading[10], car il serait sinon fort difficile de connaitre le véritable nombre de coeurs sans l'utiliser.

StarPU lance ensuite un thread sur chaque travailleur. Ce dernier sonde en permanence des files de tâches pour voir s'il peut en exécuter une. Les paragraphes qui suivent présentent comment ces files sont remplies afin d'utiliser au mieux les capacités du système.

[8] Sysconf est une fonction permettant d'obtenir de nombreuses informations sur la configuration d'un système, dont le nombre de coeurs CPU [83]

[9] Le nom provient de Hardware Locality, signifiant littéralement *la localité du matériel*

[10] De l'anglais, signifiant HyperFlux en français

Tronc commun

Lorsqu'une tâche est soumise au système, StarPU s'assure qu'il existe au moins un travailleur capable de l'exécuter. Comme il est possible de définir des contraintes d'antériorité entre les tâches, la librairie vérifie que celles de la tâche soumise sont satisfaites. Afin d'éviter des conflits de données, deux tâches manipulant les mêmes données en écriture[11] ne peuvent pas être planifiées en même temps. Si une tâche possède des contraintes d'antériorité ou des contraintes dues aux données, elle est mise de côté en attendant que celles-ci soient toutes levées. Une fois qu'une tâche imposant des astreintes à d'autres se termine, StarPU avertit les tâches asservies.

Une fois toutes les contraintes satisfaites, une file est sélectionnée selon la police d'ordonnancement qui est d'application et la tâche y est ajoutée.

Un mécanisme de priorité a également été implémenté. Une tâche prioritaire est placée dans une file de telle manière qu'elle soit la première à être sélectionnée par un travailleur. Il est aussi possible à l'utilisateur d'attribuer une tâche à un travailleur en particulier. Dans ce dernier cas, la tâche est placée dans une file locale, propre au travailleur, et qui est consultée avant toute autre. Le paragraphe suivant présente les polices d'ordonnancement.

Les polices d'ordonnancement

Ordonnancement rapide A l'initialisation, une file FIFO[12] est créée pour l'ensemble des travailleurs. Lorsqu'une tâche est soumise, elle est simplement placée dans la queue. Lorsqu'un travailleur est apte à exécuter une tâche, il récupère la première tâche de la file.

Ordonnancement rapide et sans priorité Il s'agit du même mécanisme qu'au point précédent, mais il n'y a pas moyen de placer une tâche prioritaire en début de file.

Ordonnancement rapide avec gestion complexe des priorités Il s'agit du même mécanisme que pour l'ordonnancement rapide mais il contient une gestion plus fine des priorités. Le programmeur peut décider dans StarPU d'une certaine quantité de niveaux de priorité. Lors de l'initialisation de StarPU, autant de files que de niveaux de priorité seront créées et les tâches soumises seront placées dans la queue correspondant à leur niveau de priorité. Lorsqu'un travailleur est apte à exécuter une tâche, il sonde la file dont le niveau de priorité est le plus haut. S'il n'y a pas de tâche dans cette file, il examine celle de niveau inférieur et ainsi de suite.

Ordonnancement à modèle de coût A l'initialisation, une file FIFO est associée à chaque travailleur. Lorsqu'une tâche est soumise, une succession de tâches est accomplie pour chaque file :

- StarPU vérifie si le propritétaire de la queue peut exécuter la tâche. Par exemple, si la tâche possède uniquement une version CUDA ou OpenCL, elle ne peut pas s'exécuter sur un CPU.
- La durée de la tâche est estimée en utilisant un modèle de performance. Les modèles de performances sont expliqués plus bas.
- La date de fin d'exécution de la file est estimée au moyen d'un calcul rapide : date de fin d'exécution = date à laquelle la première tâche de la file est exécutée + durée totale des tâches présentes dans la file + durée de la tâche que l'on veut ajouter. La date à laquelle la première tâche de la file est lancée est mise à jour chaque fois qu'on insère une tâche dans la file : StarPU lui additionne le temps d'exécution estimé de la tâche en cours d'exécution sur le travailleur. Cette période est également modifiée lorsqu'une erreur d'approximation

[11] Lorsque des données sont associées à une tâche, le mode d'accès (écriture et/ou lecture) est également défini
[12] De l'anglais First In First Out, signifiant Premier Entré Premier Sorti

a été faite. Ceci arrive lorsque la date réelle est plus proche ou plus lointaine que la période prévue.

La tâche sera placée dans la file pour laquelle la date de fin d'exécution est la plus proche. Lorsqu'un travailleur est apte à exécuter une tâche, il prend la première de sa file.

Ordonnancement à modèle de coût prenant en compte les données Cet ordonnancement débute comme le précédent et exécute quelques calculs supplémentaires. Ainsi, lorsque les files sont passées en revue, nous calculons en plus une pénalité pour les données en fonction de leur taille, ainsi que de la bande passante et de la latence entre les noeuds. Ces deux derniers éléments sont calculés la première fois que StarPU est initialisé sur une machine et sont ensuite stockés dans un fichier. Une fois la pénalité calculée pour chaque file, la date de fin[13] la plus proche est mémorisée dans une variable. Une fois ces opérations effectuées, on effectue à nouveau le tour des files afin de calculer un poids grâce au calcul : poids = date de fin de la file considérée - date de fin de la file la plus proche + pénalité liée aux données. Des pondérations peuvent être apportées à ces termes. La tâche est associée à la file dont le poids est le plus léger. Lorsqu'un travailleur est apte à exécuter une tâche, il prend la première tâche de sa file.

Ordonnancement aléatoire Ici également, une file FIFO est associée à chaque travailleur. Lorqu'une tâche est soumise, elle est placée dans une file sélectionée au hasard. Lorsqu'un travailleur est apte à exécuter une tâche, il prend la première tâche de sa file.

Ordonnancement avec vol de travail Une file LIFO[14] est associée à chaque travailleur. Il n'y a pas ici de gestion de priorités vu que la dernière tâche à être placée dans une file sera la première à être exécutée. Cet ordonnancement prend les files les unes après les autres pour y placer les tâches entrantes. Si une file d'un travailleur est vide, il vole la tâche d'un autre travailleur en sondant les files les unes après les autres et tente de l'exécuter. Si son architecture ne le lui permet pas, il la replace dans la file et passe à la queue suivante.

Les modèles de performances

Les deux derniers modèles de performances de cette rubrique ont besoin d'informations concernant l'exécution des tâches sur les différents types de processeurs. Lorsque StarPU exécute des tâches avec la variable d'environnement STARPU_CALIBRATE non nulle, il place les temps d'exécution des tâches ainsi que la taille des données associées dans des fichiers afin de pouvoir calculer ses estimations. Il faut donc exécuter plusieurs fois une même tâche pour que celle-ci soit bien distribuée sur les différents travailleurs de manière équilibrée.

Les modèles sont chargés la première fois qu'ils sont requis et les mises à jour ne sont replacées dans leur fichier qu'à la terminaison du programme.

Modèle par tâche Avec le modèle par architecture, ce sont les modèles les plus difficiles à employer pour l'utilisateur et ils ne sont donc pas utilisés en pratique. Le programmeur doit fournir au codelet un modèle qui donne, par exemple, le nombre d'opérations que la tâche doit effectuer. Dans le cas d'un produit de matrices carrées qui nécessite $2 * n^3$ calculs, où n est la taille de la matrice, le modèle serait exprimé sous la forme $K * 2 * n^3$. K est une constante, appelée facteur d'accélération, propre au type de coeur sur lequel le calcul s'exécute. Pour la trouver, il suffit de choisir une architecture comme étant la référence et ensuite de regarder le rapport entre le temps d'exécution de la tâche sur la référence et le temps d'exécution sur les autres types d'architecture. Elle pourrait par exemple valoir 1 pour le CPU et $\frac{1}{10}$ pour le GPU. L'ordonnanceur estime le temps d'exécution d'une tâche à partir de ce modèle.

[13] Cette date est calculée comme indiqué au paragraphe précédent

[14] De l'anglais Last In First Out, signifiant Dernier Entré Premier Sorti

Modèle par architecture Pour ce modèle, l'utilisateur doit fournir une fonction semblable à la précédente, mais pour chaque architecture sur laquelle la tâche peut s'exécuter. Le but est ici de supprimer le calcul du facteur d'accélération et également d'obtenir plus de précision. En effet, le facteur d'accélération est calculé pour une certaine quantité de données, mais le temps d'exécution sur le GPU n'évolue généralement pas de la même manière que le temps d'exécution sur le CPU lorsque la quantité de données change. Les modèles permettant de calculer le temps d'exécution peuvent donc fortement varier d'une architecture à une autre.

Modèle basé sur un historique Ce modèle est le plus précis pour les tâches régulières, c'est-à-dire dont la quantité de données ne change pas trop d'une exécution à une autre.

StarPU utilise un historique qui contient entre autres les moyennes et l'écart-type des temps d'exécution précédents en fonction des données associées à la tâche. Le modèle commence donc par calculer un hash des données afin de pouvoir, pour chaque type d'unité de calcul, retrouver le temps d'exécution de la tâche. Si, pour un type de processeur, il n'y a pas d'entrée correspondante dans l'historique, la tâche est attribuée à n'importe quel travailleur de ce type. Lorsque la tâche est effectuée sur le travailleur, le temps d'exécution est mesuré si STARPU_CALIBRATE est différent de zéro. Une fois celle-ci achevée, l'historique est mis à jour : soit une nouvelle entrée est créée, soit la moyenne, l'écart-type ... de l'entrée concernée sont mis à jour.

Modèle basé sur la régression Ici également, un historique est utilisé. Après plusieurs exécutions, un nuage de points est obtenu pour chaque architecture de processeur à partir des temps de calcul mesurés en fonction de la taille des données traitées. Pour calculer le temps d'exécution de la tâche, une régression est appliquée sur le nuage de points afin d'avoir un modèle du type : temps$= \alpha*(\text{taille des données})^{\beta} + \gamma$. Si $\beta = 1$ et $\gamma = 0$, la régression est linéaire, sinon elle est non linéaire.

Dans le cas d'une régression linéaire, seules les composantes α et γ mises à jour à la fin du programme sont stockées dans un fichier. Pour la régression non linéaire, un historique doit être utilisé et toutes les mesures sont stockées.

Ce modèle n'a pas été beaucoup utilisé jusqu'ici, mais il pourrait s'avérer fort utile dans le cas d'une tâche irrégulière pour laquelle la quantité de données varie constamment.

Conclusion

Il est intéressant de constater que les méthodes d'ordonnancement et les modèles de performance sont tels des services que nous branchons sur le corps de l'application. Un programmeur pourrait donc facilement ajouter ses propres méthodes mais les ordonnanceurs présents sont déjà très efficaces. Le seul point négatif est que la distinction des travailleurs se fait actuellement uniquement sur l'architecture et non le modèle du coeur. Cette distinction n'est pas nécessaire au moment où l'utilisateur doit spécifier sur quel type de coeur son codelet peut tourner, mais bien au moment où les modèles de performance mesurent les temps d'exécution des tâches. En effet, un ordinateur pourrait comporter des CPU, des GPU et/ou des Cell de modèles différents et fournissant donc des perfomances différentes. Il serait dès lors intéressant de pouvoir distinguer plus finement les différents coeurs lors de la planification des tâches.

3.3 Intégration de StarPU au tri

Afin d'améliorer les performances du tri des structures `TTrio`, nous avons incorporé la bibliothèque StarPU à celui-ci. Cette section présente d'abord le code qui distribue les tâches sur les différents cœurs et ensuite les résultats de celui-ci.

3.3.1 Le code

Note de l'auteur : je présente ici le code de StarPU alors qu'une documentation est disponible en ligne pour deux raisons. La première est que les développeurs de StarPU n'ont pas encore eu le temps d'établir une documentation complète de la librairie, ce chapitre apporte donc un complément d'information par rapport à celle que vous pouvez trouver sur le site du projet. La seconde est liée à la première par le fait que si la documentation en ligne n'est pas encore complète, c'est dû au fait que la librairie est encore en cours de développement et que la manière de l'utiliser change de temps à autre. Etant donné que nous avons utilisé la version de développement de StarPU pour bénéficier des dernières fonctionnalités de la librairie, le code présenté ici utilise les fonctions telles qu'elles sont en ce mois de juin 2010. Cette présentation sert donc également de mise à jour par rapport à la version disponible sur le site.

Notre algorithme trie un ensemble de sous-vecteurs de `TTrio`, des structures de 8 octets. Il a été montré dans les conclusions de la section 1.6.3 qu'une taille convenable pour les vecteurs est de 262 144 éléments : jusque là, les gains de temps occasionés par le passage du tri sur le GPU croissent rapidement, mais ce n'est plus le cas lorsque la taille continue à augmenter. En plus de cette considération, les tâches seront mieux réparties entre les différentes unités de travail s'il y en a plusieurs : le tri sur GPU s'exécutant deux fois plus vite que le tri sur CPU, il est préférable d'avoir au moins plus de tâches que de coeurs si nous désirons que ceux-ci travaillent tout le temps.

Nous avons vu que le tri connait deux contraintes :

1. Il faut que tous les éléments d'un sous-vecteur (d'une référence ou d'une source) se situent dans un même vecteur (voir section 2.5.2).
2. L'algorithme bitonique ne trie que les tableaux dont la taille est une puissance de deux (ce tri est présenté dans la section 1.6.2). Sur le GPU, il est donc parfois nécessaire de réserver des tableaux plus grands que la taille des vecteurs.

La contrainte la plus ennuyante pour StarPU est la seconde. En effet, la librairie s'occupe de la réservation de la mémoire en fonction de la taille des données à traiter. Nous ne pouvons donc pas lui indiquer de réserver plus de mémoire que celle nécessaire pour les données. Afin de remédier à ce problème, nous créons une matrice de `TTrio` contenant 262 144 colonnes et autant de lignes que de tâches à envoyer sur StarPU. La première contrainte ne garantit pas que les lignes soient toutes pleines, mais nous indiquons tout de même à StarPU que les vecteurs à trier ont tous une taille de 2^{18} éléments. Il se peut qu'il transfère donc plus de données que nécessaire.

Dans DNARun, il n'est pas possible de connaitre à l'avance le nombre de `TTrio` à placer dans la matrice vu que ce nombre varie d'un brin d'ADN à un autre. Pour simuler ce fait, nous remplissons la matrice élément par élément, avec des `TTrio` dont la valeur est aléatoire et dont l'identifiant vaut au départ 0 et est incrémenté d'une unité tous les 74[15] `TTrio`. Au départ, la matrice ne connait qu'une seule ligne. Lorsque celle-ci est pleine, une seconde ligne est allouée. Si un sous-vecteur n'a pas pu entrer entièrement dans la ligne précédente, les éléments de celui-ci déjà placés dans la matrice sont copiés dans la ligne nouvellement créée. Les sous-vecteurs restent ainsi entièrement dans une même tâche.

Nous créons ainsi deux matrices identiques : l'une sera triée avec StarPU et l'autre par un CPU dans un thread unique. Cela permet de vérifier l'exactitude des résultats obtenus avec StarPU. En plus des matrices intitulées `inputTTrio` pour StarPU et `inputCPU` pour le CPU, nous avons la variable `numLines` indiquant le nombre de lignes des matrices, la variable numColumns indiquant le nombre de colonnes des matrices[16] et le tableau `itemPerLine`. La taille de ce tableau équivaut au nombre de lignes des matrices et chacune de ses cases indique le nombre d'éléments à trier se trouvant dans chaque ligne de ces dernières. Tout ceci est initialisé à la ligne 5 du listing 3.1, listing qui présente notre utilisation StarPU. Le code effectuant l'initialisation n'est pas montré car il est assez long et n'est pas utile pour comprendre comment fonctionne le support exécutif.

[15] Cette valeur est choisie de manière arbitraire mais n'influence pas beaucoup le tri

[16] Cette variable a été créée dans un souci de réutilisabilité et d'adaptabilité aux nouvelles technologies, elle est pour le moment toujours initialisée à 262 144

A la ligne 9 de ce listing, nous initialisons StarPU. C'est à ce moment que les travailleurs sont identifiés et que les files sont créées en fonction de la politique d'ordonnancement choisie. Bien qu'à terme des simplifications seront apportées, il est pour l'instant nécessaire d'utiliser certaines variables d'environnement pour indiquer la politique souhaitée. Voici la liste de ces variables ainsi que les valeurs qu'il faut leur donner si nous désirons utiliser l'ordonnancement à modèle de coût prenant en compte les données[17] :

- STARPU_SCHED = "dmda"
- STARPU_CALIBRATE = 1
- STARPU_PREFETCH = 1

Si ces variables d'environnement ne sont pas utilisées, c'est l'ordonnancement rapide qui est utilisé par défaut.

Les lignes 11 à 13 permettent d'indiquer quel modèle de performance nous souhaitons utiliser. La ligne 12 indique que nous désirons utiliser le modèle de performance basé sur l'historique et la ligne 13 permet à l'utilisateur d'indiquer le nom du fichier dans lequel les informations concernant l'exécution des tâches seront stockées. Si ce fichier n'existe pas encore, il sera créé par StarPU lors de sa première exécution.

Les lignes 15 à 20 déclarent un premier codelet. Ses membres indiquent respectivement qu'il peut s'exécuter sur CPU et sur GPU, que la fonction exécutant le tri sur CPU s'appelle `cpu_codelet` et celle exécutant le tri sur GPU `cuda_codelet`, que le modèle de performance à employer est celui indiqué plus haut et qu'une seule donnée sera attribuée au codelet.

Les lignes de 22 à 36 déclarent le codelet qui peut uniquement être exécuté sur CPU. Le codelet effectuant la même fonction que le précédent, sa déclaration lui est fort semblable. En temps normal, StarPU n'a besoin que d'un seul codelet pour ditribuer ses tâches mais, dans notre cas, nous lui indiquerons que chaque tâche trie le même nombre de données, soit 2^{18} éléments. L'un des éléments de décision sera donc tronqué. Comme la dernière ligne risque d'être parfois loin d'être remplie, nous avons décidé que, lorsque le nombre d'éléments descend en-dessous d'un certain seuil, la tâche associée doit s'exécuter sur le CPU.

Les lignes de 28 à 37 déclarent les gestionnaires de données. Ces éléments, à qui nous associons à chacun une ligne de la matrice `inputTTrio`, sont les structures contenant l'état des données sur chaque noeud ainsi que l'interface qui décrit ces dernières (voir section 3.2.2). StarPU possède une interface pour gérer les matrices se trouvant dans un vecteur unique[18], mais comme nos matrices sont représentées par des tableaux de tableaux, nous utilisons celles permettant de gérer les vecteurs. La fonction de la ligne 31 associe l'interface décrivant un vecteur et l'une des lignes de la matrice à un gestionnaire. Les arguments de cette fonction sont respectivement l'interface gérant le vecteur, le noeud mémoire sur lequel les données se trouvent (la valeur 0 indique la RAM du processeur appelant la fonction), un pointeur vers le vecteur[19], le nombre d'éléments à trier, et la taille de chacun de ces éléments.

La ligne 39 marque le début de l'exécution des tris. Les lignes 41 à 47 effectuent les tris sur le CPU à l'aide de la fonction `sort` de la librairie STL. Les lignes 49 à 83 utilisent les tâches StarPU pour effectuer ce tri. Pour chaque ligne de la matrice `inputTTrio`, la boucle commençant à la ligne 49 débute en créant une tâche et en assignant un codelet à celle-ci. Si nous initialisons la tâche s'occupant de la dernière ligne de la matrice et que le nombre d'éléments de celle-ci se situe en-dessous d'un certain seuil, nous la plaçons sur un CPU, sinon nous laissons à StarPU le choix de l'architecture.

[17] Initialement appelé Deque Modeling Policy Data Aware, DMDA

[18] Un vecteur peut aisément être considéré comme une matrice. Pour ce faire, il suffit de désigner un nombre dont la taille du vecteur est multiple. Ce nombre délimite le nombre de colonnes de la matrice et chacune de ses lignes commence donc à un multiple du nombre.

[19] Les pointeurs vers nos vecteurs se trouvent dans la première dimension de la matrice `inputTTrio`

Une fois la fonction de tri achevée, StarPU permet l'appel d'une autre fonction. Le nom de cette fonction doit être placé dans la variable présentée à la ligne 68. Comme nous n'avons pas besoin de cette fonctionnalité, nous plaçons la variable à NULL. Un codelet pouvant porter un argument, nous en profitons pour transmettre à notre fonction le nombre d'éléments qu'elle doit réellement trier. Les lignes 72 et 73 indiquent au codelet quel est le vecteur qu'il doit trier ainsi que le type d'accès aux données qui est employé par la fonction. Dans notre cas, nos tris doivent pouvoir lire et écrire dans le vecteur. Enfin la ligne 75 soumet la tâche à StarPU et le `if` de la ligne suivante vérifie qu'il y a bien un travailleur qui est capable de l'exécuter. En effet, dans le cas d'une architecture ne possédant pas de GPU, une erreur serait soulevée si une tâche ne pouvait s'exécuter que sur un accélérateur graphique.

Avant de récupérer les données dans la boucle commençant à la ligne 85, la fonction de la ligne 84 bloque l'exécution du thread qui a lancé les tâches tant que celles-ci ne sont pas achevées.

Les dernières lignes de ce listing vérifient que le tri s'est bien effectué, libère les matrices allouées dynamiquement ainsi que les gestionnaires de données et "éteint"[20] StarPU.

Listing 3.1 – Code source présentant l'utilisation de StarPU dans le cas d'un tri sur CPU et sur GPU

```
1  int main(int argc, char** argv){
2
3      // INITIALISATION DE LA MATRICE DE TTRIOS //
4
5      ...
6
7      // UTILISATION DE STARPU //
8
9      starpu_init(NULL);
10
11     starpu_perfmodel_t cl_model;
12     cl_model.type = STARPU_HISTORY_BASED;
13     cl_model.symbol = "sort-model";
14
15     starpu_codelet clCPGPU;
16     clCPGPU.where = STARPU_CPU|STARPU_CUDA;
17     clCPGPU.cpu_func = cpu_codelet;
18     clCPGPU.cuda_func = cuda_codelet;
19     clCPGPU.model = &cl_model;
20     clCPGPU.nbuffers = 1;
21
22     starpu_codelet clCPU;
23     clCPU.where = STARPU_CPU;
24     clCPU.cpu_func = cpu_codelet;
25     clCPU.model = &cl_model;
26     clCPU.nbuffers = 1;
27
28     starpu_data_handle tab_handle[numLines];
29     for(uint i = 0; i<numColumns; i++)
30     {
31         starpu_vector_data_register(
32             &tab_handle[i],
33             0 ,
34             (uintptr_t)inputTTrio[i],
35             numColumns,
36             sizeof(TTrio));
37     }
```

[20] Shutdown provient de l'anglais et signifie éteindre

```
38
39      // EXECUTION DES TRIS //
40
41      for(uint i=0; i<numLines; i++)
42      {
43          std::sort(
44              inputTTrioCPU[i],
45              inputTTrioCPU[i] + itemPerLine[i] ,
46              compare());
47      }
48
49      for (uint i = 0; i< numLines;i++)
50      {
51          struct starpu_task *task = starpu_task_create();
52
53          if(i == numLines - 1){
54              if(itemPerLine[i]<0.6*limit){
55                  task->cl = &clCPU;
56              }
57
58              else{
59                  task->cl = &clCPGPU;
60              }
61          }
62
63          else
64          {
65              task->cl = &clCPGPU;
66          }
67
68          task->callback_func = NULL;
69          task->cl_arg = (void*)&itemPerLine[i];
70          task->cl_arg_size = sizeof(uint);
71
72          task->buffers[0].handle = tab_handle[i];
73          task->buffers[0].mode = STARPU_RW;
74
75          int ret = starpu_task_submit(task);
76          if (STARPU_UNLIKELY(ret == -ENODEV))
77          {
78              fprintf(stderr, "No worker may execute this task\n");
79              exit(0);
80          }
81      }
82
83      starpu_task_wait_for_all();
84
85      for(uint i = 0; i<numLines;i++){
86          starpu_data_sync_with_mem(
87              tab_handle[i],
88              STARPU_R);
89      }
90
91      // FINALISATION //
92
93      printf(
94          verify(
95              inputTTrio,
96              inputTTrioCPU,
97              numLines,
98              itemPerLine) ? "TEST FAILED\n" : "TEST PASSED\n");
99
100     for(uint i = 0; i < numLines; i++){
```

```
101         free(inputTTrio[i]);
102         free(inputTTrioCPU[i]);
103         starpu_data_unregister(tab_handle[i]);
104     }
105     free(inputTTrio);
106     free(inputTTrioCPU);
107     free(itemPerLine);
108
109     starpu_shutdown();
110 }
```

Les listings 3.2 et 3.3 présentent les fonctions appelées par les codelets et triant respectivement sur le CPU et sur le GPU. La première fonction commence par récupérer le pointeur vers le vecteur à trier puis la valeur indiquant le nombre d'éléments à trier. Il termine par appeler la fonction `std : :sort` de la librairie STL. La seconde fonction récupère une variable supplémentaire à la ligne 4 : la taille du tableau alloué sur le GPU. Cette donnée est tenue par l'interface décrivant le vecteur. La fonction n'appelle plus la fonction `std : :sort` mais celle présentée à la section 1.6.3. Nous lui passons deux fois le pointeur vers le vecteur à trier car la fonction place le résultat du tri dans un vecteur différent de celui qui contient les données à trier. Dans le cas présent, nous ne désirons pas utiliser cette fonctionnalité et indiquons donc que le vecteur contenant les données est également celui qui accueillera le résultat.

Listing 3.2 – Code source présentant le codelet effectuant le tri d'un vecteur sur un CPU

```
1 void cpu_codelet(void *descr[], void *args)
2 {
3     TTrio *v = (TTrio *)STARPU_GET_VECTOR_PTR(descr[0]);
4     uint *size = (uint *)args;
5     std::sort(
6         v,
7         v+*size ,
8         compare());
9 }
```

Listing 3.3 – Code source présentant le codelet effectuant le tri d'un vecteur sur un GPU

```
1 void cuda_codelet(void *descr[], void *args)
2 {
3     TTrio *v = (TTrio *)STARPU_GET_VECTOR_PTR(descr[0]);
4     unsigned nx = STARPU_GET_VECTOR_NX(descr[0]);
5     uint *size = (uint *)args;
6     bitonicSort(
7         v,
8         v,
9         *size,
10        nx);
11 }
```

3.3.2 Les résultats

Les tris de données de 8 octets utilisant StarPU, un GPU et un CPU ont été appliqués sur un vecteur dont la taille passe de 2^{20} à 2^{22} d'éléments[21]. La manière de mesurer les temps du GPU

[21] $2^{20} = 1048576$ et $2^{22} = 4194304$

et du CPU n'a pas changé par rapport à la section 1.6. Seul le tri pour le plus grand des vecteurs a donc nécessité une nouvelle mesure. Les temps relatifs aux tris utilisant StarPU mesurent toutes les actions effectuées par les fonctions de la librairie, à part l'initialisation et l'"éteignage" de celle-ci. Le support unifié a présenté un bogue lors de son utilisation avec l'ordonnanceur à modèle de coût prenant en compte les données : il n'arrive pas à charger le fichier contenant l'historique des exécutions. L'ordonnanceur rapide a donc été utilisé pour effectuer ces mesures. Les résultats sont présentés aux figures 3.1 et 3.2. Les gains de temps procurés par l'utilisation de StarPU et du GPU à la place du CPU sont également présentés aux figures 3.3 et 3.4.

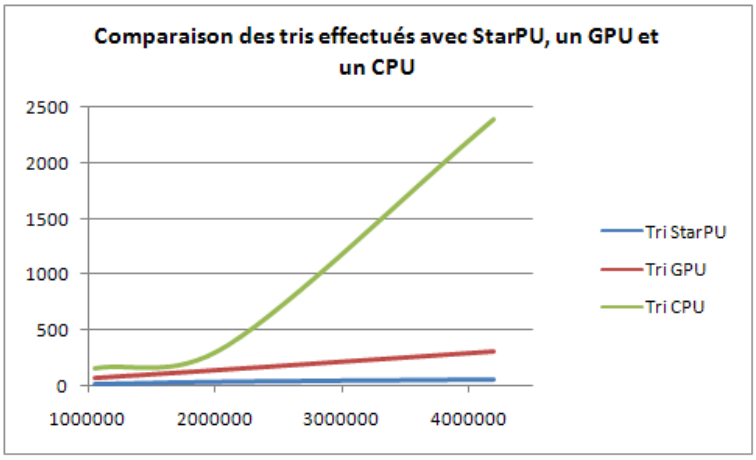


FIG. 3.1 – Comparaison des performances des tris appliqués à des structures de 8 octets effectués avec StarPU, un GPU et un CPU

Nombre d'éléments	Tri StarPU	Tri GPU	Tri CPU
1048576	21.545	69.205	155.286
2097152	43.224	147.641	361.898
4194304	62.787	311.492	2390.265

FIG. 3.2 – Performances des tris appliqués à des structures de 8 octets effectués avec StarPU, un GPU et un CPU. Les temps sont exprimés en millisecondes.

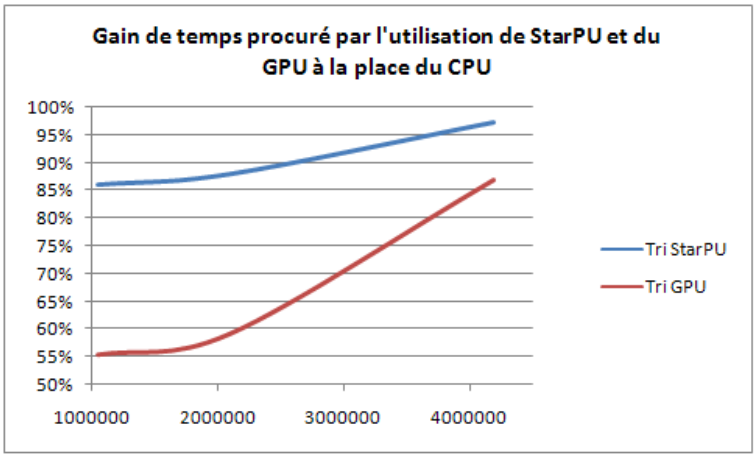


FIG. 3.3 – Gain de temps procuré par l'utilisation de StarPU et du GPU à la place du CPU dans le cas d'un tri de structures de 8 octets

Nombre d'éléments	Tri StarPU	Tri GPU
1048576	86%	55%
2097152	88%	59%
4194304	97%	87%

FIG. 3.4 – Gain de temps procuré par l'emploi de StarPU ou d'un GPU par rapport au temps d'exécution obtenu en utilisant un CPU lors du tri de données de 8 octets

Ces données nous permettent de constater un phénomène qui n'avait pas encore été observé : le tri sur CPU s'emballe lorsque nous arrivons à des vecteurs de taille 2^{22}, ce qui augmente considérablement le gain procuré par l'emploi du GPU ou de StarPU. En effet, ceux-ci passent respectivement de 59% à 87% et de 88% à 97% alors qu'ils semblaient tous deux se stabiliser. Cette constatation donne un argument de plus en faveur de l'utilisation de l'une de ces deux technologies.

Avant de comparer les performances obtenues lors de l'utilisation de StarPU par rapport à celles obtenues avec un GPU, une seconde réflexion doit être menée. Comme déjà mentionné plus haut, le fichier dans lequel sont stockées les données relatives aux précédentes exécutions des tâches n'arrivait pas à s'ouvrir et il n'a donc pas été possible d'employer l'ordonnanceur le plus optimisé. Comme plusieurs mesures sont à chaque fois faites pour le même tri, nous avons pu constater que les résultats avec l'ordonnanceur rapide sont instables (voir figure 3.5). Ainsi, le pire temps d'exécution vaut presque le double du meilleur temps dans le cas du plus petit des vecteurs. La plupart des mesures étant plus proches des pires temps d'exécution, la moyenne de ces derniers est plus proche de ceux-ci que des meilleurs temps[22]. Le plus grand des vecteurs fournit des résultats plus stables car c'est le premier à utiliser pleinement les ressources disponibles. Les essais ayant été effectués sur une machine bi-processeurs Linux 64 bits équipée de 4 GPU, nous disposions de 8 travailleurs (4 CPU et 4 GPU)[23]. Vu que les GPU travaillent deux fois plus vite que les CPU, le système est à plein régime une fois que 12 tâches sont distribuées. Ce chiffre est pour la première fois atteint par le vecteur de 2^{22} éléments qui génère 17 tâches (voir figure 3.6). Nous retrouvons d'ailleurs chez ce vecteur le même écart entre le meilleur et le pire des temps que pour le vecteur le plus petit : les 12 premières tâches se terminent plus ou moins toujours en même temps, et les 5 dernières sont distribuées comme pour le vecteur de 2^{20} éléments. L'écart entre les temps d'exécution ne devrait donc pas augmenter avec la taille du tableau et deviendra ainsi de moins en moins significatif. Nous pouvons toutefois espérer qu'une fois le bogue résolu, nous puissions réussir à toujours obtenir le meilleur temps.

Nombre d'éléments	Pire temps	Meilleur temps	Ecart
1048576	33.132	17.239	15.893
2097152	54.357	31.690	22.667
4194304	74.459	59.253	15.206

FIG. 3.5 – Comparaison des temps d'exécution du tri de structure de 8 octets utilisant StarPU dû à l'utilisation de l'ordonnancement rapide qui ne produit pas un résultat stable. Les temps sont exprimés en millisecondes.

[22] L'écart-type présenté à la figure 3.6 est assez important pour le plus petit vecteur, surtout si nous le rapportons au meilleur temps. Il reste plus ou moins le même par la suite alors que les temps d'exécution grandissent, et son importance diminue donc.

[23] Les kernels exécutés sur les GPU doivent être lancés depuis un CPU. Pour éviter un manque de réactivité de ces derniers, StarPU n'utilise pas un CPU contrôlant un GPU. CUDA ne permet pas qu'un CPU contrôle plus d'un GPU à la fois, mais OpenCL l'autorise. L'utilisation d'OpenCL permettrait donc une augmentation du nombre de travailleurs

Nombre d'éléments	Ecart-Type	Nombre de tâches
1048576	5.935	5
2097152	7.312	9
4194304	5.900	17

FIG. 3.6 – Ecart-type et nombre de tâches utilisées relatifs aux tris de structures de 8 octets utilisant StarPU. L'écart-type est exprimé en millisecondes

Les figures 3.7 et 3.8 présentent les gains de temps procurés par l'emploi de StarPU à la place du GPU. Le gain moyen et le gain dans le meilleur des cas diffèrent fort au départ, mais ceux-ci se rapprochent par la suite pour les raisons énoncées plus haut. Nous atteignons donc des gains de 80% vis-à-vis du GPU, ce qui signifie que l'exécution se passe maintenant 5 fois plus vite. Si nous calculons que, dans notre cas, deux CPU valent un GPU, c'est équivalent à dire que nous possédons 6 GPU. Le meilleur gain que nous aurions pu escompter vaut donc $1 - \frac{1}{6} = 83,33\%$ et nous n'en sommes pas loin. StarPU est donc un système fort efficace, même avec un ordonnanceur assez simple.

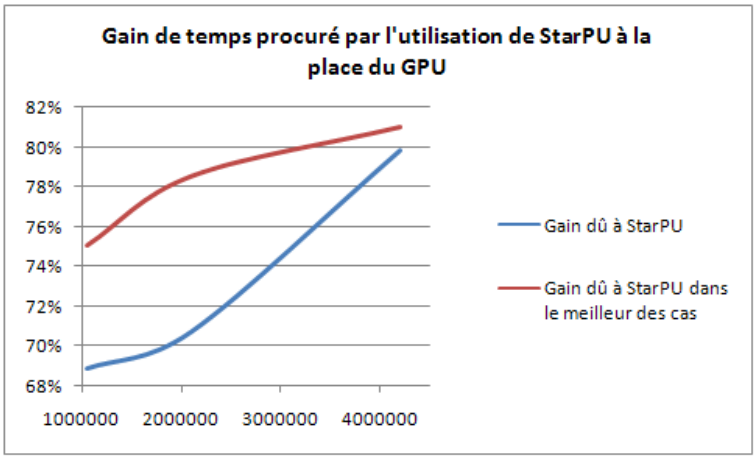


FIG. 3.7 – Gain de temps procuré par l'utilisation de StarPU à la place du GPU dans le cas d'un tri de structures de 8 octets

Nombre éléments	Gain dû à StarPU	Gain dû à StarPU dans le meilleur des cas
1048576	69%	75%
2097152	71%	79%
4194304	80%	81%

FIG. 3.8 – Gain de temps procuré par l'emploi de StarPU par rapport au temps d'exécution obtenu en utilisant un GPU lors du tri de données de 8 octets

Une dernière réflexion est menée sur le gros désavantage de StarPU : il nécessite une initialisation et un éteignage très coûteux. En effet, ceux-ci prennent en moyenne respectivement 5,4 s et 0,45 s, soit près de 6 s alors que les tris n'atteignent pas ici la seconde (à l'exception du tri sur CPU). Ce seul point noir rend donc au final StarPU plus de deux fois moins efficace qu'un simple CPU triant un tableau de 2^{22} éléments. Bien que cela soit donc fort mauvais pour les vecteurs ici envisagés, ces temps restent constants pour toute exécution d'un programme utilisant StarPU. Il ne faut pas non plus perdre de vue que DNARun a été conçu pour comparer une très grande quantité de simples-brins d'ADN, et que ces 6 s ne représenteront plus grand

chose si nous l'alimentons assez pour qu'il puisse tourner, par exemple, toute une journée. S'il était possible pour StarPU d'adopter l'architecture client-serveur, la librairie tournant alors sans cesse sur des machines dédiées, nous pourrions l'utiliser même pour des programmes possédant des temps d'exécution courts.

3.4 Conclusions

La distribution de tâches sur une machine dont nous ne connaissons rien à priori n'est pas chose facile, et pourtant StarPU arrive à fort bien exploiter les ressources disponibles. De plus, CUDA imposant plus de contraintes qu'OpenCL au niveau du contrôle des GPU, la prise en main de ce dernier standard pourrait encore accroître les perfomances obtenues avec le support unifié.

Comme la librairie a besoin d'être initialisée et éteinte à chaque exécution et que ces opérations prennent actuellement 6 secondes sur notre système, son emploi doit rester dans le cadre d'algorithmes possédant des durées d'exécution suffisament grandes que pour masquer cette perte de temps.

Enfin, StarPU peut être utilisé sous Windows, sous Linux et sous Mac OS/X, ce qui fait de lui un support fort portable. L'utilisateur n'a donc pas à se soucier du système sur lequel s'exécutera son code.

Conclusion

Bien que l'exécution séquentielle de l'algorithme de BioloMICS ait été optimisée, nous lui avons découvert un bogue et quelques mauvaises pratiques de programmation qui rendent les résultats instables. Comme la parallélisation du code aurait augmenté cette instabilité, il a été décidé de ne pas modifier davantage l'algorithme. En effet, la parallélisation ne ferait qu'augmenter l'instabilité. Nous avons donc développé les améliorations en dehors de BioloMICS afin qu'elles soient aptes à être intégrées lorsque l'algorithme sera stable. Une analyse du programme a été effectuée afin de découvrir quelles sont les fonctions qui consomment le plus de temps de calcul. Comme il s'est avéré que la première candidate est l'algorithme de tri de la bibliothèque STL [38], et que les suivantes étaient liées de près ou de loin à celle-ci, le tri a été porté sur les accélérateurs graphiques.

Plutôt que de réinventer la roue, nous avons choisi d'utiliser le tri bitonique du SDK de NVIDIA [50]. Comme la plupart des tris pour processeur graphique existants, le tri de NVIDIA n'ordonne que les données dont la taille est de 4 octets. La taille des structures triées dans BioloMICS, appelée `TTrio`, atteint cependant les 8 octets. Il a donc fallu légèrement remanier le tri afin qu'il puisse être utilisé dans BioloMICS. La structure `TTrio` a également été remaniée afin d'inclure un identifiant jusqu'à présent inexistant tout en gardant sa taille et afin de permettre une comparaison selon deux de ses variables en une seule opération. Il n'aurait pas été possible d'avoir des données de taille supérieure car les performances se dégraderaient rapidement à cause du transfert des données entre mémoire CPU et mémoire GPU ainsi qu'à l'intérieur du GPU, entre ses différents types de mémoire. Les performances obtenues sont concluantes : le gain de temps de calcul atteint les 48% à partir du moment où le vecteur à trier possède au moins 218 éléments. Nous observons également que le tri sur CPU dégénère à partir du moment où le vecteur comprend au moins 2^{22} éléments alors que le tri sur processeur graphique reste stable. Nous obtenons à ce moment-là un gain de 87% du temps de calcul.

StarPU est un outil qui permet d'affecter aux ressources des tâches créées par le programmeur. Comme BioloMICS trie un ensemble de vecteurs, nous avons pris comme tâches des sous-ensembles de ce dernier. Diverses politiques d'ordonnancement existent au sein de la librairie. La plus aboutie est assez précise car elle se base sur un historique des exécutions précédentes pour estimer le temps de travail d'une tâche sur les différentes architectures disponibles. Connaissant la charge de travail de chaque ressource, elle affecte la tâche à celle qui terminera le plus tôt ses calculs.

Nous n'avons pas réussi à faire fonctionner cet ordonnanceur sur notre système à cause d'un bogue au sein de StarPU. En attendant que celui-ci soit fixé, nous avons pris une méthode d'ordonnancement qui distribue les tâches aux ressources lorsque celles-ci sont libres de travail. Même avec cet ordonnanceur nous arrivons déjà à de très bons résultats : par rapport aux performances sur processeur graphique, le tri voit son temps d'exécution pratiquement divisé par 5 dans le meilleur des cas et par 3 en moyenne. Nous pouvons espérer qu'une fois résolu

le bogue de l'ordonnanceur, nous atteindrons toujours les meilleurs cas. Comme notre système aurait tout au plus permis un facteur 6 d'accélération, StarPU est fort proche du cas idéal.

Les résultats obtenus sont très bons étant donné que StarPU permet un gain de temps par rapport à la version initiale du tri de 88% avant que le tri sur CPU ne dégénère et de 97% après. Ces performances pourront encore être améliorées grâce au déboguage de l'ordonnanceur basé sur l'historique.

Lorsque le bogue de DNARun sera résolu, notre tri pourra y être intégré et la parallélisation poursuivie. Comme le coeur de l'algorithme consiste en la comparaison de chaines d'ADN, il serait intéressant de placer chaque comparaison dans une tâche StarPU. Le centre de recherche CETIC [74] placera ensuite des ensembles de comparaisons sur différents clusters, permettant ainsi un nouveau gain de temps.

Annexe A

Etat de l'art des workflows et des environnements libres permettant leur exécution sur une Grille Informatique

A.1 Les représentations des workflows

A.1.1 Représentation graphique

Flowchart

Un flowchart permet de représenter aisément un processus. Un exemple, réalisé grâce au Java Workflow Toolkit (JWT [87]), est montré à la figure A.1. C'est en réalité une succession d'activités avec de temps à autre un noeud de décision (représenté par un losange) qui évalue une expression booléenne. Cette représentation est très aisée à comprendre et permet de modéliser un grand nombre de processus.

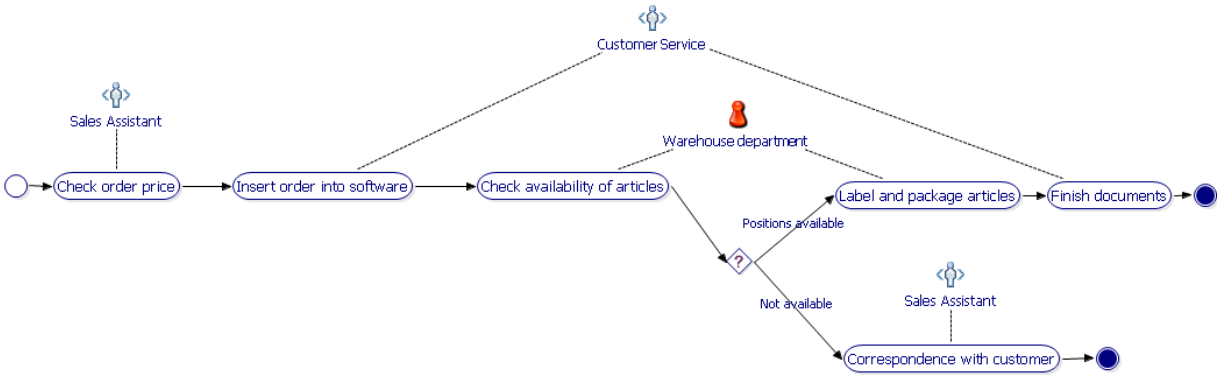


FIG. A.1 – Modélisation d'un processus sous forme d'un flowchart grâce à l'outil JWT (provenance [87])

Le JWT permet donc de modéliser des flowcharts et génère du code grâce au langage XML Process Definition Language (XPDL [88]). Ce langage permet de définir des processus grâce au XML. C'est un standard de la Workflow Management Coalition (WfMC [89]). Un autre avantage est qu'on pourrait éventuellement modifier le XPDL afin d'ajouter les informations nécessaires pour le mapping des activités sur les ressources offertes par les Grilles.

Business Process Modelling Notation (BPMN)

Comme indiqué dans le document [85], BPMN est une notation graphique standardisée, basée sur le modèle flowchart, conçue dans le but d'être compréhensible par tous les acteurs de l'entreprise, de l'analyste qui crée les premières ébauches des processus, jusqu'aux développeurs techniques responsables de l'implémentation des technologies.

Des exemples d'extensions amenées par BPMN sont présentés aux figures A.2 et A.3, les plus intéressantes étant celles de la première figure.

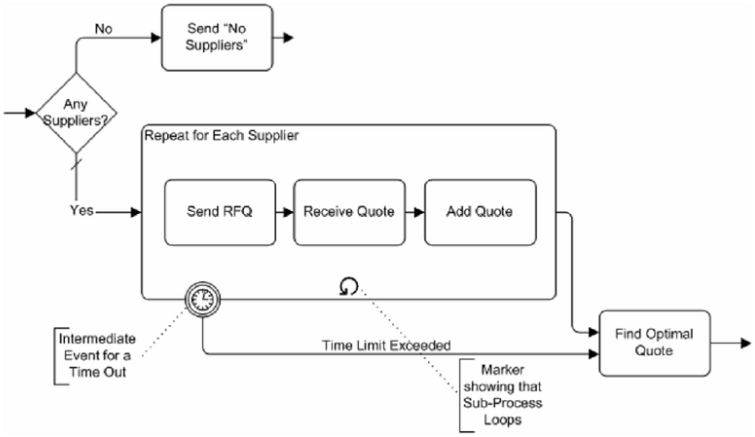


FIG. A.2 – Modélisation d'un processus à l'aide de BPMN, une notation basée sur le modèle flowchart (provenance [85])

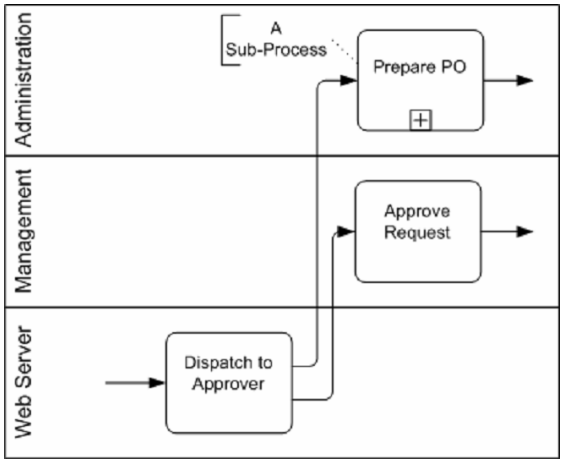


FIG. A.3 – Modélisation d'un processus à l'aide de BPMN, une notation permettant une assignation des activités (provenance [85])

Cette notation peut être modélisée à l'aide des designers des projets open-source Intalio|BPM Community Edition [86] et Java Workflow Toolkit [87]. Intalio génère du code BPEL orienté Web Service, JWT du code XPDL. Il serait donc plus intéressant d'utiliser JWT que Intalio, étant donné qu'il n'oblige pas un langage orienté Web Service.

Graphes Orientés Acycliques (DAG - Directed Acyclic Graph)

Les workflows peuvent également être représentés par des graphes orientés acycliques. Les noeuds du graphe représentent les activités et les arcs les contraintes de précédence de ces dernières. Vu que les graphes sont acycliques, il ne peut y avoir de boucles dans la structure du

workflow. Ainsi, bien que cette représentation permet de modéliser de nombreux workflows, elle ne permettre pas de tout faire.

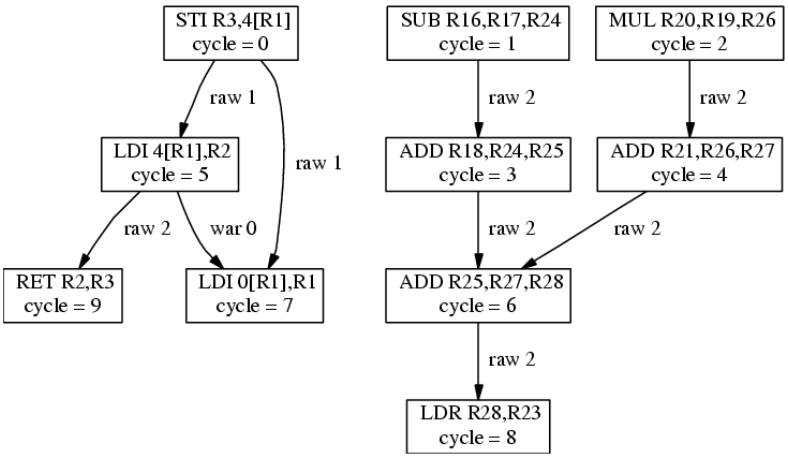


FIG. A.4 – Deux exemples de modélisation DAG (provenance [112])

Les environnements Chimera et Java Cog Kit (voir [95] et paragraphe A.2.2) permettent la conception graphique de tels workflows.

Réseaux de Petri et YAWL - Yet Another Workflow Language

Le réseau de Pétri est un moyen de modéliser le comportement des systèmes dynamiques et à événements discrets ; ainsi que de décrire des relations existant entre des conditions et des événements (voir [91]).

Un réseau est constitué de petits cercles représentant un état et de traits représentant des transitions, les uns étant reliés aux autres à l'aide d'arcs. Dans chaque état, on trouve un nombre entier positif ou nul de jetons. Lorsqu'une transition à laquelle est liée un état possédant au moins un jeton est activée, un jeton est retiré de cet état et un jeton est ajouté dans chaque autre état dont l'arc sort de la transition (voir fig. A.5). L'avancement du jeton représente donc l'évolution du workflow.

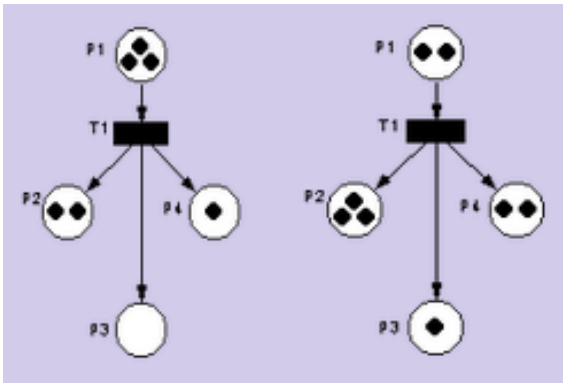


FIG. A.5 – Un exemple de réseau de Pétri avant et après l'activation d'une transition (provenance [91])

Améliorations Afin de pouvoir représenter un plus large ensemble de workflows, ce concept a été complexifié à l'aide de trois nouvelles représentations :

1. Les réseaux de Pétri de haut-niveau [128] : ce terme englobe la famille de représentations se basant directement sur les réseaux de Pétri et se contentant de lui apporter de petits avantages supplémentaires. Un exemple est le réseau de Pétri coloré dans lequel les jetons possèdent une couleur permettant de désigner une valeur.
2. YAWL [125] : YAWL est un langage qui se propose d'améliorer les réseaux de Pétri. Il a été mis en évidence que les réseaux ne sont pas optimaux pour tous les patrons. Ainsi, il est difficile d'exécuter avec eux de l'instantiation multiple, des synchronisations complexes ou des annulations non-locales. YAWL introduit de nouveaux éléments permettant de gérer ces trois cas.
3. *new*YAWL [127] : ce langage est une évolution du précédent, le but étant de fournir un meilleur support pour les flux de contrôle et une meilleure perspective des données et des ressources.

Bien que ces langages soient très poussés et semblent donc en mesure de modéliser de nombreux processus métiers, les logiciels les utilisant pour exécuter les workflows sont encore rares et fort méconnus.

A.1.2 Représentation XML

XPDL - XML Process Definition Language

XPDL [88] est destiné à décrire les processus métier à l'aide du langage XML. Comme présenté dans [119], la description du workflow est basée sur cinq éléments :

- Activity : bloc de base de la définition d'un workflow
- Transition : élément connectant les activités
- Participant : spécifie les participants d'un workflow (les entités effectuant le travail,...)
- DataField et DataType : spécifie les données manipulées par le workflow

Une activité peut avoir une restriction de transition de type Split (diviser) ou Join (joindre). Les transitions possèdent également une restriction de type AND (et logique) ou XOR (ou exclusif) :

- Join AND : joint tous les threads concurrents du processus dont les transitions arrivent en l'activité. La synchronisation est requise et le nombre de threads peut dépendre des précédents Split AND.
- Join XOR : joint les threads alternatifs ; la synchronisation n'est pas requise.
- Split AND : toutes les transitions sortant de l'activité seront exécutées en parallèle.
- Split XOR : une seule des transitions sortant de l'activité sera exécutée : la première qui n'a pas de conditions ou dont la condition est évaluée à Vrai.

Comme introduit dans le dernier point, chaque transition peut avoir une condition qui indique si, oui ou non, nous passons de l'activité A à l'activité B. Ces transitions et restrictions sont les outils qui permettent de faire du contrôle de flux dans XPDL.

Ce langage permet de prendre en compte les contraintes de précédence entre activités, l'exécution parallèle et la synchronisation de threads, les structure de contrôle tel `if`, les boucles tel `while do`,... mais ne supporte pas de nombreux patrons telle la Fusion Multiple (deux branches convergent sans synchronisation vers une activité qu'elles exécutent toutes les deux)(voir [119] pour plus d'exemples sur ce qui peut être fait avec XPDL).

JSDL

Il est aisé de générer et de lire du code XPDL valide, mais les définitions concernant les jointures données par la WfMC peuvent être interprétées de différentes manières. La difficulté est donc de savoir comment l'outil qui va exécuter le workflow va interpréter le code XPDL.

XPDL semble ainsi convenir pour la réalisation de workflow assez simple, mais devient plus difficile à maîtriser lorsque le développeur souhaite créer des structures plus complexes.

JaWE (Java Workflow Editor [120]) et JPEd (JaWE based Process Editor [121]) sont tout deux des éditeurs graphiques de workflows XPDL.

AGWL - Abstract Grid Workflow Langage

Comme présenté dans [123] et [124] AGWL [122] construit ses workflows en agençant des activités. Il y a ainsi un bloc pour les activités de base, les activités composées (activités `sequence`, et `parallel`), les activités de boucle (activités `while`, `dowhile`, `foreach` et `for`), les activités conditionnelles (activités `switch` et `if`), et les activités avancées de contrôle de flux (activités `parallelFor` et `parallelForEach`). Il existe également une activité DAG qui permet de modéliser des graphes acycliques orientés. Les contraintes de précédences sont fixées grâce à l'attribut `predecessor` de chaque noeud.

A ces activités sont associés des attributs qui permettent de définir leurs arguments d'entrée et de sortie. Des contraintes et des popritétés peuvent également être ajoutées pour donner des indications sur le déroulement des activités ou des contraintes que le milieu exécutant le workflow doit remplir (contraintes sur le temps d'exécution, le système d'exploitation, la mémoire,...).

Des documents sur la page d'AGWL [122] apportent plus d'informations sur la modélisation d'un workflow destiné à tourner sur une grille. Ce langage est celui utilisé par ASKALON (voir paragraphe A.2.6) pour modéliser et exécuter les workflows.

WS-BPEL - Web Services Business Process Execution Language

WS-BPEL [134] possède le même principe de base que l'AGWL, ce sont également des blocs emboîtés les uns dans les autres afin de définir un processus. La différence est que ce langage est orienté Web Services. Ceux-ci sont définis au moyen du langage WSDL (Web Services Definition Language) et WS-BPEL va les ordonnancer afin de former un workflow.

Il existe donc de nombreuses balises permettant d'agencer les Services. Parmi ceux-ci, nous retrouvons une balise permettant de définir des variables, une balise permettant de contenir un flux séquentiel, des balises de contrôle de flux tel que `If`, `forEach`, `while`,... La balise `flow` permet de préciser que les différents blocs `séquence` se trouvant à l'intérieur doivent s'exécuter en parallèle. Bien que cette balise existe, [92] met en évidence qu'il n'est cependant pas aisé de paralléliser les itérations d'une boucle lorsque le nombre d'itérations est variable.

Une caractéristique de ce langage est sa capacité de gérer les erreurs. Il est possible de déclarer des erreurs là où il risque d'y avoir des problèmes.

Seules les propriétés permettant de préciser les contraintes que le milieu, la Grille doit remplir pour exécuter un Web Service semblent manquer.

WSCI - Web Service Choreography Interface

WSCI [135] est un équivalent de WS-BPEL : Il permet également d'ordonnancer des Web Services décrits en WSDL et propose les mêmes balises que le WS-BPEL. La différence entre les deux est que WSCI décrit le comportement observable d'un Web Services grâce à ses interfaces, alors que WS-BPEL décrit le comportement d'un processus métier en se basant sur les interactions de ce dernier avec ses partenaires (les autres Web Services étant les partenaires).

WSFL - Web Services Flow Language

Les informations fournies dans cette partie reposent sur la documentation trouvée dans [129], [130], [131], [132], et [133].

Ce langage appartenant à IBM et basé XML est fortement orienté Web Service, comme son nom l'indique. Il consiste à décrire un processus métier comme étant un DAG dont les noeuds sont des Web Services et dont Les arcs représentent l'ordre d'exécution des services.

Afin de pouvoir faire du contrôle de flux, il est permis d'associer une condition de transiton aux arcs. Pour que les services auxquels aboutissent les arcs s'exécutent, il faut que les valeurs de sortie des services qui les précèdent vérifient la condition booléenne.

Comme ce langage est orienté DAG, il suffit de faire sortir deux arcs d'une même activité pour créer des sections pouvant s'exécuter en paralèlle. Une activité vers laquelle pointent plusieurs arcs sert de jointure : elle ne sera jamais commencée tant que les conditions de transition des arcs entrant n'ont pas toutes été évaluées. Une telle activité peut posséder une condition de jointure se reposant sur les conditions de transition. Pour que l'activité commence, il ne faut donc pas nécessairement que toutes les conditions soient vraies mais bien qu'elles aient été évaluées.

Les activités possèdent également une condition de sortie qui est examinée à la fin de leur exécution. Si la condition n'est pas remplie, l'activité se réexécute. Ce mécanisme permet de vérifier que tout s'est bien exécuté. Il permet ainsi de recommencer l'activité si un événement non désiré est survenu et d'implémenter un mécanisme de boucle. En effet, une activité peut être exécutée autant de fois qu'il sera nécessaire pour que la condition de sortie soit satisfaite. Comme il est possible qu'une activité appelle un sous-workflow, il est possible d'effectuer une boucle sur une succession d'activités.

Les activités peuvent se voir attribuer des propriétés désignant des obligations supplémentaires devant être remplies lors de leur exécution, tels une durée maximum d'exécution, un nombre maximum d'essais,... Ces propriétés servent à localiser le meilleur fournisseur de services selon les propriétés imposées.

A.2 Les environnements permettant l'exécution de workflow sur une Grille informatique

A.2.1 Introduction

Sauf précision, tous les paragraphes de cette section sont basés sur l'analyse du livre [92].

Cette section présente une liste non-exhaustive des environnements qui permettent l'affectation des ressources (les serveurs, processeurs voire même coeurs d'une grille informatique) aux tâches du workflow. Certains environnements n'ont d'emblée pas été retenus pour ces recherches :

- P-Grade Grid Portal (un environnement basé web pour le développement, l'exécution et la surveillance des workflows, voir [93]) et MOTEUR (un moteur de workflow, voir [113]) : ces deux éléments n'ont pas comme point fort la répartition des tâches sur des grilles.
- Taverna (voir [94]) : ce logiciel est fort orienté services (Web Services, BioMart databases, BioMoby services,...) et conçu pour conduire des expériences *in silico* en biologie. Le but de ces recherches n'étant par l'utilisation de services existants, cet environnement a été écarté.
- Triana (voir [114]) : comme pour P-Grade, cet environnement permet l'exécution de workflows sur une grille, mais la répartition des tâches sur un ensemble de ressources n'est pas mentionné.
- Cactus (voir [115]) : ce framework est destiné à la conception graphique de workflows, mais pas à leur déploiement ou à leur exécution.
- Sedna : cet environnement basé BPEL est aussi plus destiné à la conception des workflows qu'à leur exécution sur des Grilles.

Ci-dessous sont présentés les outils qui possèdent des aptitudes dans le domaine de l'affectation des ressources aux activités d'un workflow.

A.2.2 Java CoG Kit Workflow

[95] est une source complémentaire d'informations sur l'environnement Java Cog Kit.

Un des buts de Java Cog Kit est de permettre aux différents acteurs d'utiliser facilement des programmes, et d'administrer les grilles à partir d'un framework haut-niveau.

L'environnement utilise beaucoup le concept d'abstraction et de fournisseurs : en utilisant des abstractions, une couche a été construite au-dessus de l'intergiciel Grille. Celle-ci permet un accès aisé aux fonctions de la Grille, tels le transfert de fichiers ou la soummission d'activités. Pour intégrer de nouveaux services, le développeur doit seulement définir un ensemble de fournisseurs répondant aux définitions d'interfaces. Ainsi, le concept de fournisseurs de Grilles permet d'intégrer une grande variété d'intergiciels Grille au sein de Java Cog Kit et les abstractions permettent au développeur de choisir au moment de l'exécution les intergiciels à qui seront soumis les transferts de fichiers et la soumission d'activités.

Le framework Karajan est utilisé comme moteur de workflow pour faire tourner les workflows représentés par des DAGs. C'est ce framework qui contient le plannificateur qui assigne les ressources aux activités, aux composants du workflow. Il est facile d'intégrer des algorithmes de plannification définis par l'utilisateur.

La définition des workflows est sauvegardée sous un format XML ou dans un langage propre au framework Karajan. Le passage d'un mode de représentation à un autre est possible. Il est possible de concevoir graphiquement les workflows (voir A.6).

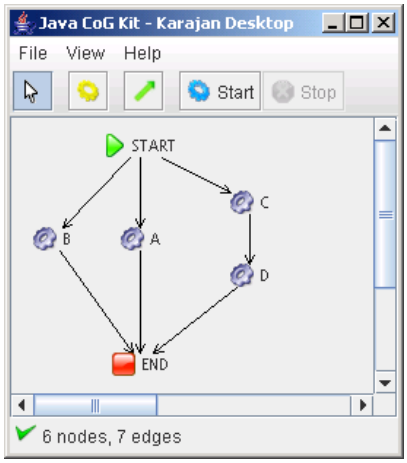


FIG. A.6 – L'éditeur graphique de workflows de Java Cog Kit (provenance [95])

Le plannificateur peut être configuré avec un ensemble de protocoles et de ressources, mais ne peut pas collecter dynamiquement les informations sur les ressources.

Les tâches sont exécutées de manière asynchrône. Un mécanisme de sondage est donc peut-être utilisé pour vérifier si une tâche a terminé son travail. Ceci ne pose pas de problèmes pour les workflows composés de tâches ayant une grande durée d'exécution, mais dans le cadre de ces recherches, les tâches s'enchaînent rapidement et un contrôle plus actif est nécessaire.

A.2.3 Condor, Stork et DAGMan

Condor (voir [96]) est un système batch optimisé pour l'exécution de workflows composés de tâches possédant une longue durée d'exécution, et notamment sur Grilles. Stork (voir [118]) est utilisé pour déployer les données et DAGMan (Directed Acyclic Graph Manager, Gestionnaire de Graphes Orientés Acycliques, voir [97]) permet de soumettre les workflows à Condor. Condor

trouve les machines pour l'exécution des programmes, et DAGMan ordonnance les activités en se basant sur les dépendances qui existent entre celles-ci.

Le principal attrait pour cette technologie est que chaque tâche du workflow peut être précédée de l'exécution d'un pré-script. Ce dernier peut actuellement être utilisé pour désigner sur quelle Grille l'utilisateur désire faire tourner la tâche. Le système pourrait être réutilisé non plus pour définir sur quel site une tâche doit tourner, mais bien sur quelle ressource (processeur,coeur,...).

Cet environnement est principalement conçu pour les workflows possédant des tâches nécessitant une longue durée d'exécution. Ainsi, pour suivre le statut des tâches soumises, il va observer le fichier log du système batch qui effectue les jobs. Lorsque DAGMan détecte qu'une tâche est terminée, il met le statut de la tâche à "achevée" et prépare alors l'exécution des tâches ayant toutes leurs contraintes de précédence satisfaites. Le problème est donc le manque de réactivité du système : il faut que DAGMan vienne lire les fichiers log pour passer d'une tâche à ses successeurs, alors que nous souhaiterions que cette opération s'effectue rapidement.

A.2.4 Pegasus

Pegasus [98] est un outil qui permet de générer un workflow exécutable grâce à l'instance d'un workflow et aux informations sur les ressources. Il permet ainsi aux scientifiques de concevoir des workflows au niveau applicatif, sans se soucier de l'environnement d'exécution. Pour créer un workflow, l'équipe ayant travaillé sur Pegasus utilise trois méthodes :

1. Concevoir les instances de workflows directement en accord avec un schéma prédéfini.
2. Utiliser des environnements comme Chimera ou Triana pour concevoir les instances de workflow.
3. Utiliser un editeur intelligent de workflow tel le CAT (Composition Analysis Tool) qui conseille l'utilisateur sur l'organisation des tâches au sein d'un workflow et complète les instances de workflow.

Pegasus utilise un catalogue de sites contenant des informations sur chaque Grille accessible pour attribuer les tâches du workflow à celles-ci. L'algorithme d'attribution peut également prendre en compte la localisation des données d'entrée des tâches. Pegasus permet à l'utilisateur de personnaliser l'algorithme de sélection des ressources, ce dernier étant un plug-in de l'environnement. De plus, il est mentionné que le catalogue de site peut contenir des informations sur le nombre de processeurs disponibles, la quantité de mémoire disponible et l'espace disponible sur le disque. Nous pouvons donc espérer pouvoir plannifier le workflow à un niveau assez bas au sein de la Grille.

L'exécution des workflows s'effectue grâce à DAGMan et Condor-G. Cette partie n'a donc pas changé depuis le paragraphe précédent, hormis une amélioration fort intéressante : la tâche fictive (placeholder). Une tâche fictive est une unité de travail placée dans la file d'attente d'une Grille et qui peut exécuter plusieurs tâches du workflow lorsqu'elle commence son exécution. Ainsi, les tâches ne durant que quelques minutes voire même quelques secondes peuvent être regroupées ensemble (Pegasus regroupe les tâches se trouvant au même niveau de profondeur dans le workflow), soumises en même temps à la Grille et donc éviter de perdre trop de temps dans les files d'attente. L'exécution d'une tâche fictive est paramétrisable et peut être de type séquentielle ou parallèle (basé sur MPI). Cette technique testée sur un workflow de 1513 tâches a présenté une amélioration du temps d'exécution du workflow de 61% en passant de 104 minutes à 40 minutes.

A.2.5 Imperial College e-Science Networked Infrastructure (ICENI et ICENI II)

Pour ICENI II, la documentation se trouve dans [99].

ICENI (voir [117]) est une collection d'intergiciels Grille utilisée pour fournir et coordonner des services Grilles pour des applications scientifiques (voir [109]). ICENI se concentre principalement sur trois éléments :

- prototyper les services et leurs interfaces nécessaires à la construction des intergiciels Grilles orientés services
- développer une meilleure plateforme pour le support des applications Grilles
- étudier les informations concernant les services et applications nécessaires pour permettre un placement optimal des activités du workflow sur les Grilles

Le cycle de vie du workflow est décomposé en trois étapes : spécification du déroulement du workflow, validation de ce dernier et attribution des ressources aux activités (étape réalisation), et l'exécution du workflow. ICENI semble fort orienté Web Services.

Pour la spécification, une interface graphique a été implémentée et permet de faire des graphes orientés (les cycles sont admis) (voir figure A.7). La représentation graphique est transposée en code grâce au langage XML.

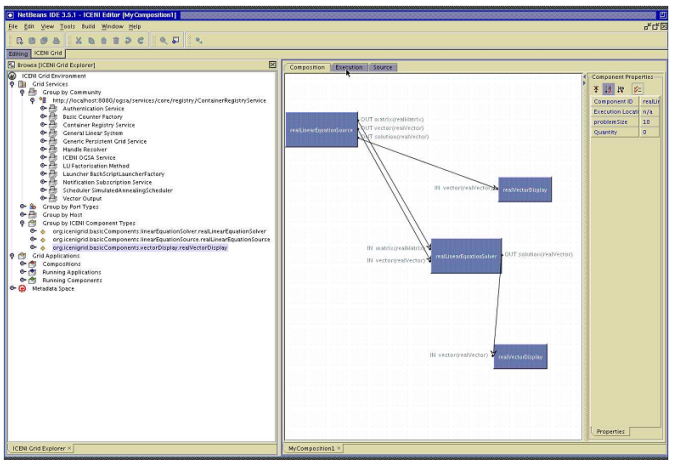


FIG. A.7 – L'éditeur graphique de workflows d'ICENI (provenance [110])

Pour ce qui est de l'affectation des tâches aux ressources, un service permet de découvrir les ressources disponibles et des algorithmes attribuent les activités aux ressources. ICENI fournit un framework pour ajouter de nouveaux algorithmes. En plus de plannifier l'exécution des activités sur les ressources, ICENI permet d'optimiser les workflows (remplacement d'activités par des équivalents plus optimisés, réordonnancement des activités pour rendre le workflow plus efficace,...) et de plannifier les activités juste-à-temps grâce à un service de réservation.

A.2.6 ASKALON

Le but de l'environnement de développement et d'exécution d'applications Grille ASKALON (voir [116]) est que les développeurs n'aient pas à connaître la couche Grille pour concevoir et exécuter des workflows.

Les workflows peuvent ici être conçus soit à l'aide d'une interface graphique, en utilisant les standards UML, notamment les diagrammes d'activité, soit en les codant avec le langage AGWL (voir A.1.2), un langage basé XML. Les workflows sont ensuite décomposés en DAG lors de leur exécution. Il est possible d'associer aux activités des contraintes concernant les ressources. On

peut ainsi spécifier qu'une activité doive s'exécuter sur un certain type de processeur, avec une fréquence d'horloge minimum ou un certain système d'exploitation. La figure A.8 présente sur la gauche la modélisation d'un workflow à l'aide de l'interface graphique d'ASKALON, et sur la droite son équivalent DAG.

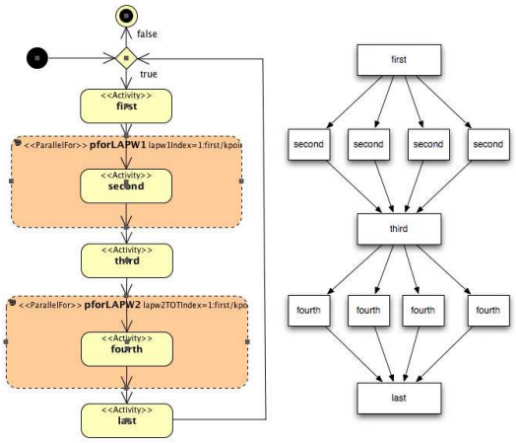


FIG. A.8 – A gauche : workflow conçu à l'aide de l'interface graphique d'ASKALON ; à droite : équivalent DAG du workflow (image issue de [108])

Le gestionnaire de ressources utilisé dans ASKALON est GridARM. Ce dernier permet d'explorer les ressources disponibles, de réserver celles-ci à l'avance, de les négocier sur des critères de temps, de coût et de Qualité de Service, de déployer les workflows sur les Grilles grâce au framework distribué GLARE,... GridARM couvre les ressources physiques tels les processeurs, les dispositifs de stockage (la mémoire), et les interconnexions réseaux, mais aussi les ressources logiques comme les services Web/Grille et les exécutables.

A côté du gestionnaire de ressources, le plannificateur spécifie sur quelles ressources les activités vont s'exécuter. Pour ce faire, il utilise le gestionnaire de ressources pour avoir les informations concernant les Grilles afin de connaître les déploiements possibles. Le pannificateur fonctionne avec trois algorithmes : un génétique, un myopic juste à temps, et un algorithme date de fin au plus tôt hétérogène (HEFT - Heterogeneous Earliest Finish Time). Les algorithmes ayant le même format pour les données d'entrée-sortie, il serait à priori possible de placer un quatrième algorithme sans trop de difficultés. Si un workflow rencontre un problème lors de son exécution (disparition d'une ressource,...) le plannificateur effectue alors un réordonnancement des activités en prenant compte de l'état actuel du workflow.

Le moteur d'exécution coordonne l'exécution du workflow et résoud les dépendances liées aux flux de données (accès aux bases de données, les arguments des activités,...). Le moteur doit sonder les Grilles pour savoir lorsqu'une activité est terminée et donc savoir à quel moment il peut lancer l'exécution de ses successeurs. Afin de rendre le système plus réactif, les tâches peuvent être fusionnées. Il y a ainsi moins de tâches à surveiller, un gain de temps au niveau des files d'attente et du système de sondage, et une plus grande réactivité entre les tâches fusionnées.

A.2.7 UNICORE 6 - UNiform Interface to COmputing Resources 6

UNICORE est un environnement permettant de concevoir et d'exécuter des processus, des workflows (voir [100]). Il possède une architecture trois tiers : le tiers client, le tiers service et le tiers système.

Le tiers client propose un client Riche basé sur Eclipse pour concevoir les workflows sous forme de DAG. UNICORE fournit les activités If et While, permettant ainsi de faire des boucles

malgré la modélisation DAG. Le tiers client fournit également un client en ligne de commande qui permet l'accès aux fonctionnalités offertes par le tiers service. Il permet ainsi d'exécuter les workflows, de surveiller leur statut,...

UNICORE possède une architecture orientée service, et tous ses services se trouvent dans le tiers service. Il y a ainsi un service qui permet d'authentifier les requêtes ; un service qui s'occupe de stocker les ressources, de transférer les fichiers, et de gérer les jobs ;... Deux services concernent particulièrement les workflows :

- Le moteur de workflow : le moteur d'UNICORE est basé sur Shark XPDL, mais il est possible de le remplacer par un autre moteur grâce à l'approche orientée service.
- L'orchestrateur : le but de ce service est d'exécuter les tâches du workflow, manipulant l'exécution des jobs et surveillant les Grilles. Des algorithmes attribuant les ressources aux tâches du workflow sont implémentés, mais il est aisé d'en ajouter d'autres.

Enfin, le tiers système permet de faire le lien entre UNICORE et les ressources.

Annexe B

La structure TTrio

La structure `TTrio` est présentée à la figure B.1. La variable `m_Val` contient huit *TTrios*, chacun codé sur quatre octets. Ces deux variables sont reprises dans la variable `m_Long` grâce à une `union`. La structure tiens donc ainsi sur huit octets.

Listing B.1 – Structure TTrio

```
struct TTrio
{
    union
    {
        struct
        {
            union
            {
                struct
                {
                    unsigned int m_L : 3 ;
                    unsigned int m_I : 29 ;
                };
                unsigned int m_Pos;
            };
            unsigned int m_Val;
        };
        unsigned __int64 m_Long;
    };
};
```

Annexe C

Définition

C.1 API (Application Programming Interface)

Une API a pour objet de faciliter le travail d'un programmeur en lui fournissant les outils de base nécessaires à tout travail à l'aide d'un langage donné. Elle constitue une interface servant de fondement à un travail de programmation plus poussé [103].

C.2 Carte accélératrice - Carte d'accélération

Carte augmentant la puissance du microprocesseur principal d'un ordinateur afin d'en améliorer les performances [139].

C.3 Cluster

Ensemble de plusieurs machines vues comme une seule permettant d'obtenir de grandes puissances de traitement [141].

C.4 Die

Petite portion rectangulaire d'un wafer, sur laquelle ont été gravés les circuits d'une puce [145].

C.5 Datamining

Ensemble de méthodes et de techniques qui permet d'extraire des informations à partir d'une grande masse de données. Son utilisation permet par exemple d'établir des corrélations entre ces données et de définir des comportements-type de clients [144].

C.6 Définition de workflow

La définition d'un workflow est la représentation des étapes du workflow. Elle présente ainsi la structure générale du processus mais n'identifie ni les données qui seront traitées, ni les ressources qui exécuteront le workflow (voir [92], p. 377).

C.7 Deque

Double ended queue : queue qui supporte l'insertion et la suppression aux deux extrémités [151].

C.8 Données cohérentes

Des données, dont le contenu correspond à une information et représente un état du processus à un instant précis, sont dites cohérentes. Afin que des données soient cohérentes, aucune modification ou actualisation ne doit être effectuée durant le traitement ou le transfert [137].

C.9 Fichier log

Fichier regroupant l'ensemble des événements survenus sur un logiciel, une application, un serveur ou tout autre système informatique [106].

C.10 FIFO (First In First Out)

Méthode de gestion de stockage dans laquelle on fait sortir en premier ce qui est rentré le plus tôt [138].

C.11 Flops

FLoating-point Operation per Second, signifiant opération en virgule flottante par seconde. Unité de mesure de la vitesse de calcul d'un ordinateur ou d'un processeur [150].

C.12 Framework

En programmation orientée objet, infrastructure logicielle qui facilite la conception des applications par l'utilisation de bibliothèques de classes ou de générateurs de programmes [107].

C.13 Hash

Algorithme de protection utilisé par Internet pour les signatures numériques, MD5 est l'algorithme hash le plus utilisé [143].

C.14 HyperThreading

La technologie HyperThreading consiste à définir deux processeurs logiques au sein d'un processeur physique. Ainsi, le système reconnaît deux processeurs physiques et se comporte en système multi-tâche en envoyant deux threads simultanés, on parle alors de SMT[1]. Cette "supercherie" permet d'utiliser au mieux les ressources du processeur en garantissant que des données lui sont envoyées en masse [140].

[1] De l'anglais Simultaneous Multi Threading, signifiant Multi Flux Simultané

C.15 Instance de workflow

L'instance d'un workflow est composé de la définition du workflow et des données d'entrée. Les ressources sur lesquelles va s'exécuter le workflow ne sont pas encore précisées (voir [92], p. 377).

C.16 Intergiciel (middleware)

Classe de logiciels permettant de faire la relation entre des couches de bas niveau d'un ordinateur (système d'exploitation) et des couches de haut niveau (applications) [104].

C.17 LIFO (Last In First Out)

Type de file d'attente dans laquelle le dernier arrivé et le premier servi [153].

C.18 Moteur de workflow

Dispositif logiciel fournissant l'environnement d'exécution d'une instance d'un workflow [102].

Le système de gestion de workflow peut être constitué d'un ou plusieurs moteurs de workflow.

C.19 Participant

Une ressource qui effectue le travail d'une instance d'une activité d'un workflow [102].

C.20 PCI Express (PCI-E)

Le bus PCI Express (Peripheral Component Interconnect Express, noté PCI-E ou 3GIO pour "Third Generation I/O"), est un bus d'interconnexion permettant l'ajout de cartes d'extension dans l'ordinateur. Le bus PCI Express a été mis au point en juillet 2002. Contrairement au bus PCI, qui fonctionne en interface parallèle, le bus PCI Express fonctionne en interface série, ce qui lui permet d'obtenir une bande passante beaucoup plus élevée que ce dernier [149].

C.21 Processeur 64 bits

Un processeur 64 bits est un processeur dont la largeur des registres est de 64 bits sur les nombres entiers. En effet, des processeurs dits 32 bits gèrent depuis longtemps les nombres flottants sur 64 voire 128 bits [152].

C.22 Processus

Suite d'opérations ou d'événements [111].

C.23 Service de workflow

Service logiciel pouvant être composé d'un ou plusieurs moteurs de workflows. Son but est de créer, gérer et exécuter des instances de workflows [102].

C.24 SDK (Software Developement Kit)

C'est l'ensemble des outils de développement de logiciels, surtout chez Microsoft [148].

C.25 Standard ouvert

Tout protocole de communication, d'interconnexion ou d'échange et tout format de données interopérable et dont les spécifications techniques sont publiques et sans restriction d'accès ni de mise en œuvre [142].

C.26 Système batch

Système qui exécute et retourne le résultat des tâches qu'on lui soumet, et ce sans l'intervention humaine [105].

C.27 Système de gestion de workflow

Système définissant, créant et gérant l'exécution de workflows en exécutant des programmes ; tournant sur un ou plusieurs moteurs de workflows ; capable d'interpréter la définition du processus, d'interagir avec les participants du workflow et d'utiliser des outils et des applications informatiques lorsque cela est requis [102].

C.28 Thread

Il s'agit d'un processus léger qui va permettre à un programme court d'être exécuté [147].

C.29 Wafer

Le wafer est une galette de silicium très pure utilisée pour la fabrication de circuits intégrés : processeurs ... C'est sur cette galette que seront gravés les millions de transistors qui composent le circuit [146].

C.30 Web Service

Application web pouvant interagir dynamiquement avec d'autres programmes en utilisant des protocoles d'échanges basés sur XML comme SOAP, XML-RPC ou XMLP [136].

C.31 Workflow

Automatisation d'un processus métier, dans sa globalité ou seulement une partie de celui-ci, dans lequel des documents, de l'information ou des tâches sont transmis d'un participant à un autre pour effectuer des actions, en accord avec un ensemble de règles procédurales [102].

C.32 Workflow exécutable

Un workflow exécutable est une instance de workflow à laquelle sont associées les ressources nécessaires à son exécution (transfert des données, exécution des tâches, gestion de l'exécution,...) (voir [92], p. 377).

Annexe D

Glossaire d'acronymes

ADN - Acide désoxyribonucléique
AGWL - Abstract Grid Workflow Langage
BPEL - Business Process Execution Language
BPMN - Business Process Modeling Notation
CAT - Composition Analysis Tool
CPU - Central Processing Unit
DAG - Directed Acyclic Graph
DAGMan - Directed Acyclic Graph Manager
ddNTP - didésoxyNucléotide TriPhosphate
dNTP - désoxyNucléotide TriPhosphate
DSM - Data Sync Manager
EIB - Element Interconnect Bus
GPGPU - General-Purpose Processing on Graphics Processing Units
GPU - Graphics Processing Units
ICENI - Imperial College e-Science Networked Infrastructure
INRIA - Institut National de Recherche en Informatique et Automatique
JaWE - Java Workflow Editor
JPEd - JaWE based Process Editor
JWT - Java Workflow Toolkit
MIT - Massachusetts Institute of Technology
MPI - Message Passing Interface
OpenMP - Open Multi-Processing
PPE - Power Processing Element
SDK - Sample Development Kit
SIMD - Single Instruction, Multiple Data
SMT - Simultaneous Multi Threading
SPE - Synergistic Processing Element
STL - Standard Template Library
UCT - Unité Centrale de Traitement
UNICORE - UNiform Interface to COmputing Resources
W3C - Workflow Wide Web Consortium
WfMC - Workflow Management Coalition
WS-BPEL - Web Services Business Process Execution Language
WSCI - Web Service Choreography Interface
WSDL - Web Service Definition Language
WSFL - Web Services Flow Language
XML - Extensible Markup Language

XPDL - XML Process Definition Language
YAWL - Yet Another Workflow Language

Bibliographie

[1] J.-F. Remacle, `Introduction to Parallel Computing`, Université Catholique de Louvain

[2] `Institut National de Recherche en Informatique et en Automatique`, http ://www.inria.fr/, consulté le 03 juin 2010

[3] `StarPU : A Unified Runtime System for Heterogeneous Multicore Architectures`, http ://runtime.bordeaux.inria.fr/StarPU/, consulté le 23 mai 2010

[4] `BioloMICS`, http ://www.bio-aware.com/BioloMICS.aspx, consulté le 30 mai 2010

[5] `BioAware`, http ://www.bio-aware.com/, consulté le 30 mai 2010

[6] `Chronology of Personal Computers`, http ://pctimeline.info/comp1984jul.htm, consulté le 11 avril 2010

[7] `Professional Graphics Controller : Notes`, http ://www.seasip.info/VintagePC/pgc.html, consulté le 11 avril 2010

[8] Matthieu Ospici, Luigi Genovese, Thierry Deutsch, `Exploitation et partage de GPU dans les grappes de calcul hybrides`, 2009

[9] `Architecture et fonctionnement d'un GPU`, http ://www.pcworld.fr/article/materiel/carte-graphique/architecture-et-fonctionnement-d-un-gpu/108881/, consulté le 11 avril 2010

[10] Antoine Leclercq, 2009, Faculté Polytechnique de Mons, `GPGPU, ou la programmation parallèle générale de processeurs graphiques`

[11] `Hardware Update - NVIDIA Tesla T10 : GPU Computing di seconda generazione`, http ://www.hwupgrade.it/articoli/skvideo/1989/nvidia-tesla-t10-gpu-computing-di-seconda-generazione_2.html, consulté le 12 avril 2010

[12] `NVIDIA CUDA Zone : Qu'est-ce que CUDA ?`, http ://www.nvidia.fr/object/what_is_cuda_new_fr.html, consulté le 21 avril 2010

[13] `Khronos Group : OpenCL Overview`, http ://www.khronos.org/opencl/, consulté le 21 avril 2010

[14] `macgeneration : OpenCL amadoue les trous noirs supermassifs`, http ://www.macgeneration.com/news/voir/139591/opencl-amadoue-les-trous-noirs-supermassifs, consulté le 21 avril 2010

[15] `NVIDIA CUDA`, http ://demo.manifold.net/doc/nvidia_cuda.htm, consulté le 21 avril 2010

[16] Developpez.com : Une introduction à CUDA, http ://tcuvelier.developpez.com/gpgpu/cuda/introduction/ ?page=theorie, consulté le 21 avril 2010

[17] Dr. Dobb's : CUDA, Supercomputing for the Masses : Part 4, http ://www.drdobbs.com/high-performance-computing/208401741 ;jsessionid=NNNULZALOZKDHQE1GHRSKH4ATMY32JVN ?pgno=3, consulté le 21 avril 2010

[18] NVIDIA NVIDIA CUDA - Programming Guide - version 3.0, publié le 20 février 2010

[19] NVIDIA Developer Zone - CUDA Training, http ://developer.nvidia.com/object/cuda_training.html, consulté le 21 mai 2010

[20] Developpez.com - Une introduction à CUDA, http ://tcuvelier.developpez.com/gpgpu/cuda/introduction/ ?page=theorie, consulté le 12 avril 2010

[21] Comment ça marche .net - Processeur, http ://www.commentcamarche.net/contents/pc/processeur.php3, consulté le 13 avril 2010

[22] Tom's Hardware - Le processeur Cell, http ://www.presence-pc.com/tests/Le-processeur-Cell-366/6/, consulté le 13 avril 2010

[23] IBM - IBM BladeCenter QS20 blade with new Cell BE processor offers unique capabilities for graphic-intensive, numeric applications, http ://www.realworldtech.com/page.cfm ?ArticleID=RWT072405191325, consulté le 13 avril 2010

[24] Nasa - Représentation d'un Cell, http ://www.hec.nasa.gov/news/gallery_images/cell.chip_diagram.jpg, consulté le 13 avril 2010

[25] Tom's guide - Le processeur Cell de la Playstation 3, http ://www.bestofmicro.com/actualite/12075-playstation-cell.html, consulté le 26 mai 2010

[26] CrunchGear - 45nm Cell microprocessor confirmed in PS3 Slim, http ://www.crunchgear.com/2009/08/20/45nm-cell-microprocessor-confirmed-in-ps3-slim/, consulté le 26 mai 2010

[27] Entrepreneur - Mercury brings Cell processor to PC workstation architecture, http ://www.entrepreneur.com/tradejournals/article/149501911.html, consulté le 26 mai 2010

[28] Mercury Computer Systems - Cell Accelerator Board 2, http ://www.mc.com/products/boards/accelerator_board2.aspx, consulté le 26 mai 2010

[29] Real World Technologies - Cell Microprocessor III, http ://www-01.ibm.com/common/ssi/rep_ca/7/897/ENUS106-677/index.html, consulté le 26 mai 2010

[30] Fixstars - Fixstars Releases Accelerator Board Featuring the Latest Cell/B.E. Chip, http ://www.fixstars.com/en/company/press/20080403.html, consulté le 26 mai 2010

[31] Folding@home, http ://folding.stanford.edu/, consulté le 13 avril 2010

[32] Folding@home - Client statistics, http ://fah-web.stanford.edu/cgi-bin/main.py ?qtype=osstats, consulté le 13 avril 2010

[33] Tom's Hardware : AMD distribue ses premières puces Llano, http ://www.presence-pc.com/actualite/AMD-Llano-Fuzion-38969/, consulté le 09 mai 2010

[34] PCWorld.fr & matbe : ISSCC 2010 - AMD dévoile Llano, le premier APU (CPU avec GPU intégré), http ://www.pcworld.fr/2010/02/09/materiel/cpu/isscc-2010-amd-llano-apu-32-nm/473451/, consulté le 09 mai 2010

[35] macgeneration : Sandy Bridge arrivera à la fin de l'année, http ://www.macgeneration.com/news/voir/151051/sandy-bridge-arrivera-a-la-fin-de-l-annee, consulté le 09 mai 2010

[36] PCWorld.fr & matbe : Intel montre Sandy Bridge, le CPU 32 nm intégrant CPU et GPU sous un seul die, http ://www.pcworld.fr/2010/04/14/materiel/cpu/intel-parle-sandy-bridge-cpu-integrant-cpu-gpu-seul-die/484871/, consulté le 09 mai 2010

[37] PC Inpact : (MàJ) Intel se remet au GPU Computing : projet Moïse (Poisson d'avril), http ://www.pcinpact.com/actu/news/56197-intel-moise-gpu-computing-larrabee-idf.htm, consulté le 09 mai 2010

[38] sgi : Standard Template Library Programmer's Guide, http ://www.sgi.com/tech/stl/, consulté le 20 mai 2010

[39] Algorithmist : Introsort, http ://www.algorithmist.com/index.php/Introsort, consulté le 20 mai 2010

[40] Ralph Unden .net : A guide to Introsort, http ://ralphunden.net/ ?q=a-guide-to-introsort, consulté le 20 mai 2010

[41] Algorithmist : Quicksort, http ://www.algorithmist.com/index.php/Quicksort, consulté le 20 mai 2010

[42] Sorting algorithms : Quicksort, http ://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm, consulté le 20 mai 2010

[43] Algorithmist : Heap sort, http ://www.algorithmist.com/index.php/Heap_sort, consulté le 20 mai 2010

[44] Sorting algorithms : Heapsort, http ://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/heapen.htm, consulté le 20 mai 2010

[45] Sorting algorithms : Insertion sort, http ://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertionen.htm, consulté le 20 mai 2010

[46] GPU Quicksort Library, http ://www.cs.chalmers.se/d̃cs/gpuqsortdcs.html, consulté le 09 mai 2010

[47] Daniel Cederman, Philippas Tsigas, A Practical Quicksort Algorithm for Graphics Processor, 2008,

[48] Joseph T. Kinder Jr., GPU as a Parallel Machine : Sorting on the GPU, mars 2005,

[49] Sorting networks : Bitonic sort, http ://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, consulté le 21 mai 2010

[50] NVIDIA CUDA C SDK Code Samples, http ://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html, consulté le 21 mai 2010

[51] Nadathur Satish, Mark Harris, and Michael Garland, Designing efficient sorting algorithms for manycore GPUs, septembre 2008,

[52] Sorting algorithms : Mergesort, http ://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm, consulté le 22 mai 2010

[53] Thrust : C++ Template Library for CUDA, http ://code.google.com/p/thrust/, consulté le 22 mai 2010

[54] Linux.com : C++, the GPU, and Thrust : Sorting Numbers on the GPU, http ://www.linux.com/news/software/developer/81090-c-the-gpu-and-thrust-sorting-numbers-on-the-gpu, consulté le 22 mai 2010

[55] FEDER : Fonds Européen de Développement Régional, http ://europa.eu/-legislation_summaries/employment_and_social_policy/job_creation_measures/-l60015_fr.htm, consulté le premier juin 2010

[56] ADN - Acide désoxyribonucléique, http ://www.adn.wikibis.com/acide_desoxyribonucleique.php, consulté le 28 mai 2010

[57] ADN - Nucléotide, http ://www.adn.wikibis.com/nucleotide.php, consulté le 28 mai 2010

[58] Wikipedia - Fichier :DNA structure and bases FR.svg, http ://fr.wikipedia.org/wiki/Fichier :DNA_structure_and_bases_FR.svg, consulté le 28 mai 2010

[59] Wikipedia - Fichier :Nucleotide.gif, http ://fr.wikipedia.org/wiki/Fichier :Nucleotide.gif, consulté le 28 mai 2010

[60] Une nouvelle ère pour l'ADN, http ://adnhumain.blogspot.com/2008/02/le-squenage-de-ladn-un-projet-arriv.html, consulté le 28 mai 2010

[61] Kathryn Brown, La lecture du génome humain, Pour la science, numéro 275, septembre 2000, http ://www.fil.univ-lille1.fr/FORMATIONS/BIOINFO/Bioinfo/275_040_045.pdf, consulté le 28 mai 2010,

[62] L'annotation du génome humain, l'exemple du chromosome 14, http ://www.cns.fr/spip/IMG/pdf/Annotation_genome_humain.pdf, consulté le 28 mai 2010,

[63] Jean-Yves Nau, Séquençage génomique en pratique quotidienne ?, http ://revue.medhyg.ch/infos/article.php3 ?sid=3730, consulté le 28 mai 2010

[64] Génome - Projet génome humain, http ://www.mon-genome.com/projet_genome_humain.php, consulté le 28 mai 2010

[65] Hopital.fr : Le génome entier d'un homme séquencé pour évaluer son risque de maladies, http ://www.hopital.fr/Hopital/Actualites/Actualites-medicales-et-soignantes/Le-genome-entier-d-un-homme-sequence-pour-evaluer-son-risque-de-maladies, consulté le 28 mai 2010

[66] Michel Delarue et Gilles Furelaud, Le séquençage d'un ADN, http ://www.snv.jussieu.fr/vie/dossiers/sequencage/sequence.htm, consulté le 28 mai 2010

[67] DNA Sequencing, http ://users.rcn.com/jkimball.ma.ultranet/BiologyPages/D/DNAsequencing.html, consulté le 28 mai 2010

[68] Dictionnaires et Encyclopédies sur 'Academic' : primase, http ://fr.academic.ru/dic.nsf/frwiki/1368019, consulté le 28 mai 2010

[69] Cégep de Sainte-Foy, http ://www.cegep-ste-foy.qc.ca/profs/gbourbonnais/pascal/nya/genetique/notesadn/-imagesadn/amorce.jpg, consulté le 28 mai 2010

[70] Gilles Furelaud, et Yann Esnault, Le séquençage d'un génome : comment ça marche ?, http ://www.snv.jussieu.fr/vie/dossiers/genomes/methodes_carte.htm, consulté le 28 mai 2010

[71] Bioinformatique.eu : Séquençage, http ://www.bioinformatique.eu/wiki/Séquençage, consulté le 28 mai 2010

[72] Stratégies de séquençage, http ://www.cns.fr/spip/IMG/pdf/Strategies_sequencage.pdf, consulté le 28 mai 2010,

[73] Hominidés : Le séquencage du génome du chimpanzé , http ://www.hominides.com/html/actualites/actu080905-adn-chimpanze.php, consulté le premier juin 2010,

[74] CETIC, http ://www.cetic.be/, consulté le premier juin 2010

[75] Code : :Blocks, http ://www.codeblocks.org/, consulté le premier juin 2010

[76] GNU Prof, http ://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html, consulté le premier juin 2010

[77] Hardware.fr : Un bon bouquin pour faire du GPGPU, http ://forum.hardware.fr/hfr/-Programmation/C-2/bouquin-faire-gpgpu-sujet_97762_1.htm, consulté le 23 mai 2010

[78] The Message Passing Interface (MPI) standard, http ://www.mcs.anl.gov/research/projects/mpi/, consulté le 23 mai 2010

[79] The OpenMP API specification for parallel programming, http ://openmp.org/wp/, consulté le 23 mai 2010

[80] Intel Cilk++ Software Development Kit, http ://software.intel.com/en-us/articles/intel-cilk/, consulté le 23 mai 2010

[81] Geonumerics, http ://geonumerics.mit.edu/, consulté le 23 mai 2010

[82] Cédric Augonnet, StarPU : un support exécutif unifié pour les architectures multicœurs hétérogènes, septembre 2009,

[83] Pages du manuel de la fonction Sysconf, http ://manpagesfr.free.fr/man/man3/sysconf.3.html, consulté le 27 mai 2010

[84] Portable Hardware Locality (hwloc), http ://www.open-mpi.org/projects/hwloc/, consulté le 27 mai 2010

[85] Stephen A. White, Introduction to BPMN, IBM Corporation

[86] Intalio|BPM Community Edition, http ://www.intalio.com/products/bpm/community-edition/, consulté le 07 novembre 2009

[87] Java Workflow Toolkit, http ://www.eclipse.org/jwt/components/transformations/index.php, consulté le 07 novembre 2009

[88] XPDL Support and Resources, http ://www.wfmc.org/xpdl.html, consulté le 07 novembre 2009

[89] Workflow Management Coalition, http ://www.wfmc.org/, consulté le 07 novembre 2009

[90] International Science Grid This Week - Acronym of the Week - DAGMan, http ://www.isgtw.org/ ?pid=1000449, consulté le 11 novembre 2009

[91] Les réseaux de Pétri, notions de base, http ://www.tn.refer.org/hebergement/cours/sys_disc/notions_de_base_RdP.html#def, consulté le 09 novembre 2009

[92] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, Matthew Shields, Workflows for e-Science, Scientific Workflows for Grids, **1ère édition**, Springer, 2007

[93] Gergely Sipos, The P The P- -GRADE Portal : GRADE Portal : An easy to use graphical An easy to use graphical interface for the Grid, **Hungarian Academy of Sciences**, Mars 2006

[94] Taverna, **Taverna project website**, http ://taverna.sourceforge.net/, consulté le 28 octobre 2009

[95] Java CoG KiT, http ://wiki.cogkit.org/index.php/Java_CoG_Kit, consulté le 28 octobre 2009

[96] Condor, **High Throughput Computing**, http ://www.cs.wisc.edu/condor/, consulté le 28 octobre 2009

[97] Dagman, sur le site de Condor, http ://www.cs.wisc.edu/condor/dagman/, consulté le 09 novembre 2009

[98] Pegasus, Panning for Execution in Grids, http ://pegasus.isi.edu/, consulté le 21 novembre 2009

[99] Stephen A. McGough, William Lee, John Darlington, Workflow Deployement in ICENI II, **London e-Science Centre, Department of Computing, Imperial College London, London**, 2006

[100] UNICORE, http ://www.unicore.eu/index.php, consulté le 12 novembre 2009

[101] Enhydra Shark, http ://www.enhydra.org/workflow/shark/index.html, consulté le 12 novembre 2009

[102] Workflow Management Coalition - Terminology & Glossary, The Workflow Management Coalition Specification, version 3.0, février 1999

[103] Le Dico du net - API, http ://www.dicodunet.com/definitions/developpement/api.htm, consulté le 14 novembre 2009

[104] Le Dico du net - Intergiciel, http ://www.dicodunet.com/definitions/developpement/intergiciel.htm, consulté le 14 novembre 2009

[105] The Free Dictionary by Farlex - Batch Processing, http ://encyclopedia2.thefreedictionary.com/batch%20processing, consulté le 15 novembre 2009

[106] Le Dico du net - Fichier log, http ://www.dicodunet.com/definitions/hebergement/fichier-log.htm, consulté le 15 novembre 2009

[107] Le signet informatique - Cadre d'applications, http ://w3.olf.gouv.qc.ca/terminologie/fiches/8872480.htm, consulté le 19 novembre 2009

[108] S. Ostermann, R. Prodan, T. Fahringer, A. Iosup, D. Epema, On the Characteristics of Grid Workflows, CoreGRID TR-0132, 10 avril 2008

[109] The Grid Workflow Forum - Change of ICENI from 6 to 7, http ://www.gridworkflow.org/snips/gridworkflow/exec/-diff ?name=ICENI&oldVersion=6&newVersion=7, consulté le 20 novembre 2009

[110] ICENI Research Group, Imperial College e-Science Networked - Infrastructure, 5 janvier 2005

[111] LINTERNAUTE - Encyclopédie - Processus, http ://www.linternaute.com/dictionnaire/fr/definition/processus/, consulté le 20 novembre 2009

[112] IRISA - Graphe acyclique dirigé, http ://www.irisa.fr/master/COURS/CAPS/-CoursCD/HTML/RepresentationsDesProgrammes/RepresentationDesProgrammes/-DAG.htm, consulté le 20 novembre 2009

[113] MOTEUR's home page, http ://modalis.polytech.unice.fr/softwares/moteur/start, consulté le 20 novembre 2009

[114] Triana, http ://www.trianacode.org/index.html, consulté le 20 novembre 2009

[115] Cactus, http ://cactuscode.org/, consulté le 20 novembre 2009

[116] Askalon, http ://www.dps.uibk.ac.at/projects/teuta/, consulté le 20 novembre 2009

[117] The London e-Science Centre - ICENI, http ://www.lesc.ic.ac.uk/iceni/, consulté le 20 novembre 2009

[118] Stork, http ://www.storkproject.org/, consulté le 20 novembre 2009

[119] Wil M.P. van der Aalst, Patterns and XPDL : A Critical Evaluation of the XML Process Definition Language,

[120] Enhydra JaWE - Open Source Java XPDL editor, http ://www.enhydra.org/workflow/jawe/index.html, consulté le 23 novembre 2009

[121] JaWE based Process Editor, http ://www.jped.org/index.html, consulté le 23 novembre 2009

[122] AGWL : Abstract Grid Workflow Language, http ://www.dps.uibk.ac.at/projects/agwl/, consulté le 25 novembre 2009

[123] Thomas Fahringer, Sabri Pllana, and Alex Villazon, `AGWL : Abstract Grid Workflow Language`

[124] Thomas Fahringer, Jun Qin, Stefan Hainzer, `Specification of Grid Workflow Applications with AGWL : An Abstract Grid Workflow Language`

[125] `YAWL`, http ://www.yawlfoundation.org/, consulté le 12 décembre 2009

[126] W.M.P. van der Aalst, A.H.M. ter Hofstede, `YAWL : Yet Another Workflow Language (Revised version)`

[127] Nick Russell, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, `newYAWL : Specifying a Workflow Reference Language using Coloured Petri Nets`

[128] `High-level Petri Nets - Concepts, Definitions and Graphical Notation`, Final Draft International Standard ISO/IEC 15909, Version 4.7.1, 28 octobre 2000

[129] `Web Services Flow Language 1.0`, Prof. Dr. Frank Leymann, mai 2001

[130] `The Web services insider, Part 4 : Introducing the Web Services Flow Language`, http ://www.ibm.com/developerworks/webservices/library/ws-ref4/, consulté le 13 décembre 2009

[131] `The Web services insider, Part 5 : Getting into the flow`, http ://www.ibm.com/developerworks/webservices/library/ws-ref5/, consulté le 13 décembre 2009

[132] `The Web services insider, Part 6 : Assuming responsibility`, http ://www.ibm.com/developerworks/webservices/library/ws-ref6/, consulté le 13 décembre 2009

[133] `Web services insider, Part 7 : WSFL and recursive composition`, http ://www.ibm.com/developerworks/webservices/library/ws-ref7/, consulté le 13 décembre 2009

[134] `Web Services Business Process Execution Language Version 2.0` , OASIS Standard, 11 avril 2007

[135] `Web Service Choreography Interface (WSCI) 1.0`, 2 août 2002, http ://www.w3.org/TR/wsci/, consulté le 15 décembre 2009

[136] `Le Journal du Net : Web Services`, http ://www.journaldunet.com/encyclopedie/definition/221/51/20/services_web.shtml, consulté le 15 décembre 2009

[137] `Siemens - Support produit`, http ://support.automation.siemens.com/WW/-llisapi.dll ?func=cslib.csinfo&objId=36748706&load=treecontent&lang=fr&siteid=cseus-&aktprim=0&objaction=csview&extranet=standard&viewreg=WW, consulté le 23 mai 2010

[138] `agro Job : Définition Fifo`, http ://www.agrojob.com/dictionnaire/definition-fifo-3382.html, consulté le 25 mai 2010

[139] `Toulinfo - Dictionnaire, page C`, http ://toulinfo.free.fr/dico/pagec.htm, consulté le 26 mai 2010

[140] `Comment ça marche : Processeur`, http ://www.commentcamarche.net/contents/pc/processeur.php3, consulté le 27 mai 2010

[141] Dico info : cluster, http ://dictionnaire.phpmyvisites.net/definition-CLUSTER-4286.htm, consulté le premier juin 2010

[142] Un article de loi définit ce que sont les formats ouverts, http ://formats-ouverts.org/blog/2004/07/01/12-un-article-de-loi-definit-ce-que-sont-les-formats-ouverts, consulté le 03 juin 2010

[143] Dicofr - Hash, http ://www.dicofr.com/cgi-bin/n.pl/dicofr/definition/20010101002434, consulté le 03 juin 2010

[144] Futura-Techno : Datamining, http ://www.futura-sciences.com/fr/definition/t/high-tech-1/d/datamining_3927/, consulté le 03 juin 2010

[145] Futura-Techno : Die, http ://www.futura-sciences.com/fr/definition/t/informatique-3/d/die_2459/, consulté le 03 juin 2010

[146] Futura-Techno : Wafer, http ://www.futura-sciences.com/fr/definition/t/informatique-3/d/wafer_2460/, consulté le 03 juin 2010

[147] Dico info : Thread, http ://dictionnaire.phpmyvisites.net/definition-Thread-13277.htm, consulté le 03 juin 2010

[148] Dico info : SDK, http ://dictionnaire.phpmyvisites.net/definition-SDK-9806.htm, consulté le 03 juin 2010

[149] Comment ça marche : Bus PCI Express (PCI-E), http ://www.commentcamarche.net/contents/pc/pci-express.php3, consulté le 03 juin 2010

[150] Le Jargon Français : Flops, http ://www.linux-france.org/prj/jargonf/F/flops.html, consulté le 03 juin 2010

[151] [dot]Myself : Le Java nouveau est arrivé, http ://www.dotmyself.net/posts/history-y-2006.html, consulté le 03 juin 2010

[152] Techno-Science : Processeur 64 bits, http ://www.techno-science.net/ ?onglet=glossaire&definition=331, consulté le 03 juin 2010

[153] Dico info : LIFO, http ://dictionnaire.phpmyvisites.net/definition-LIFO-4708.htm, consulté le 03 juin 2010