N° d'ordre : 3988 ANNÉE 2009

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention Informatique
Ecole doctorale : Matisse
présentée par

Van-Hoa NGUYEN

préparée à l'unité de recherche : UMR 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Composante universitaire : IFSIC

Intitulé de la thèse :

Traitement Parallèle des Comparaisons Intensives de Séquences Génomiques

Thèse soutenue à IRISA Rennes
le 12 novembre
devant le jury composé de :

Denis TRYSTRAM
Pr. Institut Polytechnique de Grenoble / rapporteur
Bernard POTTIER
Pr. Université de Bretagne Occidentale / rapporteur
François BODIN
Directeur technique CAPS entreprise / examinateur
Jacques NICOLAS
Directeur de recherche INRIA / examinateur
Christophe BLANCHET
IR CNRS IBCP / examinateur
Dominique LAVENIER
Pr. ENS Cachan (IRISA) / directeur de thèse

Remerciements

Je remercie vivement les professeurs Denis Trystram, de l'Institut Polytechnique de Grenoble, et Bernard Pottier, de l'Université de Bretagne Occidentale, d'avoir accepté de rapporter ces travaux, ainsi que François Bodin, directeur technique CAPS entreprise, et Jacques Nicolas, directeur de recherche INRIA et Christophe Blanchet, IR CNRS - IBCP, pour leur participation au jury.

Un merci particulier à Dominique Lavenier pour m'avoir encadré durant ces trois années, pour sa gentillesse, ses conseils, et son dynamisme.

Bien entendu, je remercie tous les rélecteurs de ce manuscrit pour leurs remarques et leurs corrections. Je remercie également chaleureusement toute l'équipe Symbiose pour la bonne ambiance qui règne autour du patio.

Je remercie ma famille et tous mes amis qui m'ont supporté dans tous les sens du terme pendant la préparation de cette thèse.

Table des matières

Chapitre 1

Résumé Étendu

Dans ce chapitre, nous présentons globalement le travail de cette thèse, ainsi que la conception du logiciel PLAST (*Parallel Local Aignment Search Tool*). PLAST a été imaginé pour comparer des banques de séquences en tenant compte des technologies actuelles relatives au calcul parallèle.

Nous débutons en présentant les objectifs du logiciel PLAST ainsi que les motivations de notre travail. La section suivante concerne la démarche de conception de PLAST. Par la suite, nous décrivons l'algorithme générique de PLAST et les accélérations par les jeux d'instructions SSE, les GPU et les composants FPGA. Enfin, nous montrons les principaux résultats de cette thèse.

1.1 Motivations et objectifs de PLAST

Les nouvelles technologies de séquençage conduisent à une croissance des données génomiques et de leurs données associées (annotations et méta-données). La comparaison des banques est un calcul commun. Elle est une étape de base de nombreux traitements bioinformatiques. La croissance exponentielle des données génomiques fait exploser les temps de calcul.

Étant donnée la difficulté d'augmenter la fréquence d'horloge, les processeurs multi-cœurs sont apparus depuis 4 ou 5 ans pour augmenter la puissance de calcul. Ces architectures se composent de plusieurs cœurs physiques gravés au sein de la même puce. Ces nouvelles architectures seront efficaces pour le calcul haute performance si les algorithmes sont conçus pour ces architectures.

De plus, durant la dernière décennie, les GPU ont fortement amélioré leur puissance de calcul, y compris par un grand nombre de processeurs spécialisés. Ils ont de nombreux avantages par rapport à l'architecture CPU pour un certain nombre de calculs intensifs parallèles. Parmi ces avantages, ils utilisent des centaines d'unités parallèles de calcul. Les GPU peuvent désormais être une alternative pour déporter les calculs coûteux en bioinformatiques.

D'autre part, les FPGA ont été proposés comme une plate-forme reconfigurable de haute performance pour les algorithmes de comparaison de séquences. En effet, les FPGA sont capables de fournir un facteur d'accélération plus grand que des processeurs "general-purpose" et peuvent être une plate-forme idéale pour accélérer des applications bioinformatiques.

BLAST [1, 2] est un des outils de référence dans le domaine de la bioinformatique. Mais BLAST n'est pas optimisé pour comparer des banques génomiques. Il a été conçu pour parcourir systématiquement une large banque. Dans le cadre de la comparaison intensive entre deux banques, par exemple B_0 et B_1, la banque B_1 est scannée n fois (n est le nombre de séquences de la banque B_0), même si l'option de "batch query" peut réduire le nombre de scans de la banque B_1. En outre, l'algorithme de BLAST est fondamentalement séquentiel, même si l'option multithread est disponible. En fait, une recherche multithread coupe simplement la banque de données en morceaux de taille approximative selon le nombre de processeurs disponibles. En conséquence, il est difficile d'obtenir de bonnes performances pour un grand nombre de processeurs, car les processeurs ont besoin d'un dispositif d'E/S intensif.

Le logiciel PLAST a été conçu afin d'accélérer la comparaison intensive des banques de séquences. PLAST se base sur des heuristiques à base de graines comme BLAST, mais il est conçu de façon légèrement différente par rapport à BLAST, ce qui permet de supporter plusieurs technologies de calcul parallèle : les multi-cœurs, les jeux d'instructions SSE, les GPU et les composants FPGA.

Nous avons développé les différentes versions de PLAST en suivant le modèle de la famille de BLAST : PLASTP pour comparer deux banques de protéines ; TPLASTN pour comparer une banque de protéines avec une banque d'ADN traduite en 6 phases ; PLASTX pour comparer une banque d'ADN traduite en 6 phases avec une banque de protéines et TPLASTX pour comparer deux banques d'ADN traduites en 6 phases.

1.2 Démarche de conception

Le processus de comparaison de séquences génomiques peut se diviser en deux principales familles : la recherche dans les banques et la comparaison intensive de séquences. Étant données quelques séquences requêtes, la principale problématique de la recherche dans les banques est d'accéder aux données. De nombreuses solutions (présentées dans le chapitre 2) utilisent la distribution des banques sur des nœuds (cluster de machines) ou des structures d'index efficaces. Les progrès des technologies de séquençage qui conduisent depuis une dizaine d'années à une croissance exponentielle des banques biologiques, permettent de mettre en place des nouvelles applications à grande échelle comme le re-séquençage génomique ou l'analyse métagénomique. Généralement, la tâche de base des applications bioinformatiques à grande échelle utilise la comparaison intensive de séquences. Dans ce cadre, la puissance de calcul est plus importante que l'accès aux données, car les données peuvent se stocker totalement en mémoire principale.

Jusqu'à présent, la plupart des algorithmes de comparaison de séquences ont été conçus et optimisés pour la recherche dans les banques avec quelques séquences requêtes. Cependant, les architectures multi-cœurs existent depuis 4 ou 5 ans et sont, aujourd'hui, de plus en plus populaires. Cette thèse tire son originalité dans le fait d'introduire une méthode pour accélérer la comparaison intensive de séquences. Cette méthode se base sur des heuristiques de graines et la parallélisation du calcul sur les processeurs multi-cœurs, les jeux d'instructions SSE, les GPU et les composants FPGA.

L'approche heuristique peut se paralléliser pour grouper des traitements réguliers et indépendants en utilisant une structure d'index. Pour indexer une banque de séquences, les techniques de graines, telles que les graines de BLASTP [2], les graines espacées [3, 4], et les graines sous-ensembles [5, 6] ont été choisies et examinées. Fondées sur le principe que la structure d'index est non seulement efficace mais aussi capable de supporter la parallélisation, les graines sous-ensembles sont plus efficaces que les autres graines.

Dans le cadre de la recherche dans une banque, plusieurs méthodes utilisent une seule indexation : soit la séquence requête (comme BLAST [1]), soit la banque (comme BLAT [7]). Cependant, le double index a été proposé dans [8] pour la comparaison intensive de séquences. Cette approche construit une fois l'index de la banque et la stocke dans la mémoire FLASH. Cette thèse propose de construire aussi un double index, mais en mémoire principale.

Les deux principaux points faibles du double index sont le temps d'indexation et l'espace de stockage. Néanmoins, il permet de grouper ensemble efficacement des petits calculs sur les graines pour les effectuer en parallèle. De plus, la parallélisation des comparaisons de séquences avec des technologies telles que les jeux d'instructions SSE [9, 10], les GPU [11] et les composants FPGA [8, 12] est efficace si les calculs de l'extension (sans gap ou avec gap, voir page 19 pour la définition de *gap*) ou de la matrice de programmation dynamique sont formulés et traités de manière régulière. Nous proposons de réarranger aussi les calculs de l'extension sans gap et avec gap sur des sous-séquences de taille bornée (Figure 1.1).

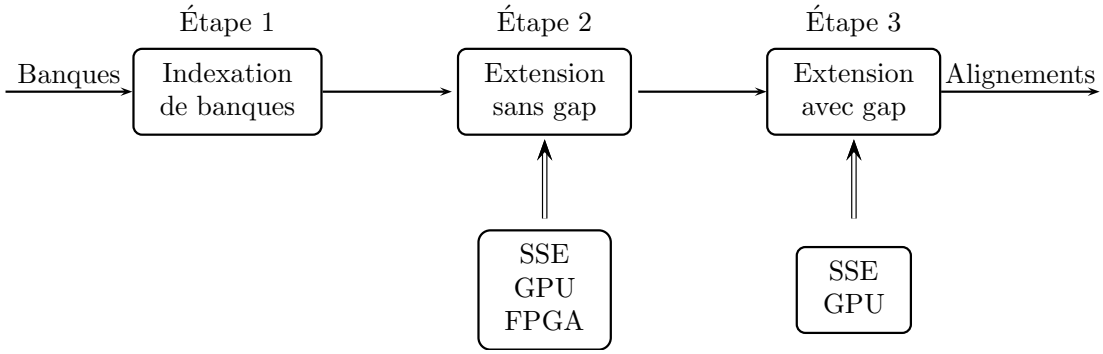


FIGURE 1.1 – Accélération de la comparaison intensive de séquences basée sur l'heuristiques des graines

1.3 Principe et description de l'algorithme de PLAST

Nous présentons dans cette section l'algorithme utilisé dans le logiciel PLAST. PLAST met en œuvre un algorithme heuristique à base de graines. Il fonctionne en trois étapes : (1) indexation, (2) extension sans gap et (3) extension avec gap (Figure 1.1). Nous présentons d'abord l'algorithme de PLAST avant de décrire les schémas de parallélisation mis en œuvre avec les technologies SSE, GPU et FPGA.

1.3.1 Algorithme de PLAST

L'algorithme de PLAST se base sur des heuristiques à base de graines comme BLAST pour détecter des similarités entres deux séquences protéiques. Cette heuristique suppose que deux protéines partagent des similarités suffisantes comprenant au moins un mot identique de W acides aminés. On localise donc, dans un premier temps, les endroits où ces mots apparaissent, appelés points d'ancrage. Puis, à partir des points d'ancrage, des similarités plus grandes peuvent être trouvées en essayant d'étendre, à droite et à gauche.

Pour exhiber des milliers de petits traitements indépendants qui peuvent être exécutés en parallèle, la première étape de l'algorithme indexe les deux banques en utilisant le concept de graine sous-ensemble [5, 6]. Deux tables de T entrées sont construites, T représentant le nombre de graines différentes. Chaque graine est associée à une liste des positions correspondantes à toutes ses occurrences dans la banque.

La deuxième étape calcule toutes les extensions : pour chaque graine, deux listes issues des deux tables sont prises en compte et chaque élément d'une liste est comparé à tous les éléments de l'autre liste. Les comparaisons consistent à calculer les scores de petits alignements sans gap en étendant à droite et à gauche des points d'ancrage.

La troisième étape cherche encore à étendre l'alignement en incluant des erreurs de gap. Cette étape n'est lancée que si l'étape précédente a détecté une similarité significative.

Algorithme 1 PLAST séquentiel

```
 1 : IT_0 ← Index (banque-0)
 2 : IT_1 ← Index (banque-1)
 3 : pour toutes les graines k possibles
 4 :   IL_0 ← IT_0[k]
 5 :   IL_1 ← IT_1[k]
 6 :   pour toutes les positions i dans IL_0
 7 :     pour toutes les positions j dans IL_1
 8 :       si extensionSansGap(IL_0[i], IL_1[j])
 9 :         extensionAvecGap(IL_0[i], IL_1[j])
10 :       fin si
11 :     fin pour
12 :   fin pour
13 : fin pour
```

Ainsi, selon ces 3 étapes, l'algorithme générique de PLAST (de manière séquentielle) est présenté dans **Algorithme 1**.

Le but de l'algorithme de PLAST est de regrouper des extensions régulières et indépendantes dans deux petites sections ayant de bonnes propriétés de parallélisation. La première section convient au modèle de programmation multi-thread, ce qui exploite l'architecture multi-cœur des microprocesseurs. La deuxième section convient au parallélisme à grain fin, ce qui est supporté par les jeux d'instructions SSE, les GPU et les composants FPGA.

Effectivement, l'algorithme de PLAST (**Algorithme 1**) a une grande potentialité de parallélisme parce qu'il contient 3 boucles `pour` imbriquées et indépendantes. Concrètement, pour chaque graine, l'extension peut être réalisée en parallèle. Ainsi, elle est considérée comme la première parallélisation orientée vers l'architecture multi-cœur. Cette parallélisation se base sur le modèle de programmation multi-thread. P threads correspondent à P cœurs physiques disponibles qui effectuent en parallèle des extensions correspondantes à P graines. Ce modèle représente la parallélisation de la boucle externe (**ligne 3**). L'algorithme parallèle de PLAST est présenté dans **Algorithme 2**.

Au début, le thread principal indexe deux banques de séquences en construisant deux structures d'index avant de créer P threads d'extension. Il définie aussi un paramètre partagé K à 0 (ligne 4). Ce paramètre représente une graine pour laquelle il faut effectuer l'extension. Chaque thread d'extension incrémente K et réalise l'extension associée à k. Quand toutes les extensions sont accomplies, le thread principal fusionne les résultats de P threads d'extension.

Nous décrivons succinctement les trois différentes étapes de l'algorithme de PLAST.

Indexation

Chaque banque de séquences est indexée en utilisant la structure illustrée dans la figure 1 du *chapitre 5*. Pour chaque graine, toutes ses positions dans la banque sont

Algorithme 2 PLAST parallèle

Thread principal	**P threads d'extension**
1 : $IT_0 \leftarrow Index(banque - 0)$	1 : **tant** que (K<T)
2 : $IT_1 \leftarrow Index(banque - 1)$	2 : $k = K++$
3 : créer P threads d'extension	3 : $IL_0 \leftarrow IT_0[k]$
4 : $K = 0$	4 : $IL_1 \leftarrow IT_1[k]$
5 : **attendre** jusqu'à $K >= T$	5 : **pour** toutes les positions i dans IL_0
6 : fusionner résultats	6 : **pour** toutes les positions j dans IL_1
7 : **fin attendre**	7 : **si** $extensionSansGap(IL_0[i], IL_1[j])$
	8 : $extensionAvecGap(IL_0[i], IL_1[j])$
	9 : **fin si**
	10 : **fin pour**
	11 : **fin pour**
	12 : **fin tant**

stockées dans une liste. Pour minimiser la taille de la structure de l'index, une position relative est stockée en calculant la différence entre deux positions successives : $\langle IL : d_1, d_2 - d_1, ..., d_t - d_{t-1} \rangle$, où d_i est un entier qui identifie une position de graine associée à l'appariement i dans la liste IL. En conséquence, la différence peut être stockée sur un entier non signé de deux octets, plutôt que stockée sur un entier de quatre octets (voir la figure 1 du *chapitre 3*). Cependant, pour des graines rares, la différence peut dépasser un entier non signé 16 bits. Pour contourner ce problème, des appariements faux positifs sont ajoutés entre deux positions. Le nombre d'appariements faux positifs augmente la taille de la liste IL d'environ 2%.

PLAST a besoin de 2 octets pour chaque entrée de l'index. Donc il utilise $4 \times 20^W + 2.02 \times n$ octets pour indexer une banque, où le nombre de lettres sur l'alphabet d'acides aminés est de 20 (voir page 18), W est la taille des graines sous-ensembles, n est le nombre d'acides aminés de la banque de séquences. Pour traiter de grandes banques de données, PLAST partitionne automatiquement chaque banque pour s'adapter à la taille de la mémoire disponible.

Extension sans gap

L'extension sans gap de BLAST commence à un point d'ancrage et s'étend à droite et à gauche. L'extension s'arrête lorsque la différence entre le score courant et le score maximal est supérieur à *X-drop*. Cette technique permet de limiter efficacement l'espace de recherche. Cependant, la taille des régions d'extension peut varier d'une séquence à l'autre. Cette technique n'est pas adaptée pour une implémentation parallèle.

Contrairement à BLAST, PLAST effectue l'extension sans gap sur des sous-séquences de taille bornée, à droite et à gauche de L caractères. La figure 1.2 montre un exemple illustrant la différence entre les deux extensions sans gap de BLAST et de PLAST. Plus précisément, pour chaque graine k, si IL_0 a K_0 éléments et IL_1 a K_1 éléments, il y a $K_0 \times K_1$ extensions à calculer. Ainsi, deux listes $BLK0_k$ et $BLK1_k$ de sous-séquences

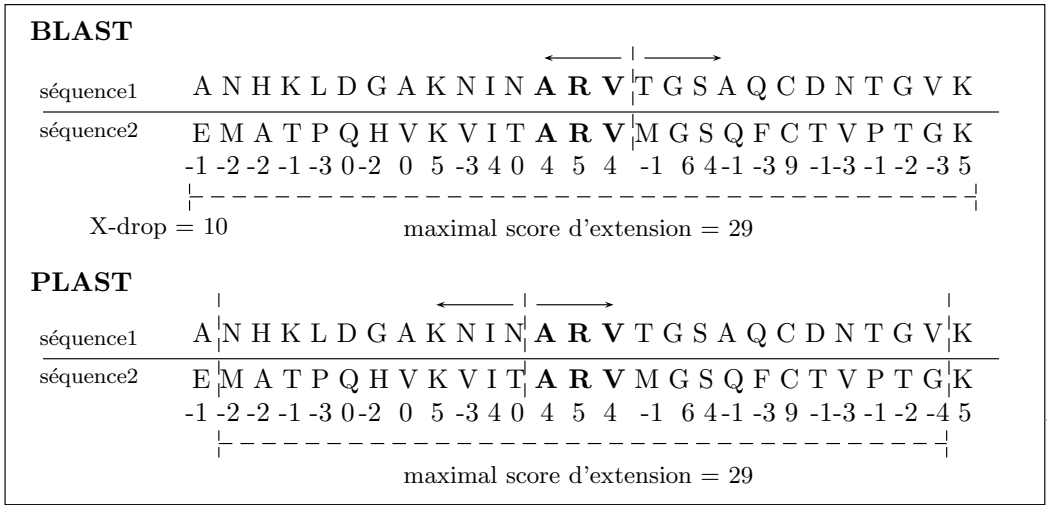


FIGURE 1.2 – Exemples de l'extension sans gap de BLAST et de PLAST. Les paramètres utilisés sont W=3, L=11, X-drop=10 (voir page 25), BLOSUM62 (voir page 19). L'extension sans gap de BLAST commence à la fin du point d'ancrage et s'étend à droite et à gauche. L'extension s'arrête lorsque le score courant devient inférieur au score maximal moins 10. L'extension sans gap de PLAST commence au début du point d'ancrage et s'étend de 14 acides aminés à droite et 11 acides aminés à gauche.

sont construites. Chaque sous-séquence est composée d'une graine de W caractères avec ses extensions à droite et à gauche de L caractères (voir la figure 2 du *chapitre 3*).

Extension avec gap

Les alignements sans gap avec une similarité significative sont passés à cette étape pour étendre d'avantage l'alignement en incluant des erreurs de gap. Le temps de calcul de cette étape, présenté dans l'étude de profil de BLASTP [2], représente jusqu'à 30 % du temps d'exécution [13]. L'accélération de cette étape est également importante pour réduire au maximum le temps d'exécution.

Nous avons divisé cette étape en deux sous-étapes : la première, *petite extension avec gap*, intervenant comme un filtre. La petite extension calcule un score sur une bande par programmation dynamique. La taille de la bande est définie par deux paramètres : la largeur ω et la longueur λ. Si le score dépasse un seuil pré-calculé, alors la procédure de programmation dynamique standard est lancée.

1.3.2 Accélération par jeux d'instructions SSE

Extension sans gap

Nous avons $K_0 \times K_1$ extensions sans gap à effectuer, pour chaque graine, qui peuvent se calculer en parallèle. Par ailleurs, les instructions SSE nous permettent d'utiliser une

parallélisation à grain fin. Plutôt que d'effectuer une opération arithmétique et logique sur 128 bits en un seul cycle machine, les microprocesseurs offrent la possibilité de considérer 128 bits en 16 opérandes de 8 bits et de réaliser simultanément 16 opérations. Ce parallélisme peut avantageusement être exploité pour calculer $\mathcal{N}$ scores en parallèle. Dans ce modèle, la valeur du score est suffisamment faible pour être encapsulée dans un entier d'un octet ou de deux octets. Le registre SIMD de microprocesseur contient simultanément $\mathcal{N}$ scores. En conséquence, nous pouvons calculer en parallèle $\mathcal{N}$ sous-séquences de $BLK0_k$ et une sous-séquence de $BLK1_k$. Nous calculons simultanément 16 scores sur 128 bits. Un score est forcement fixé entre 0 et 255 (8 bits). Le pseudo-code d'extension est présenté dans la figure 3 du *chapitre 3*.

Petite extension avec gap

Les petites extensions avec gap sont également indépendantes. Donc les jeux d'instruction SSE peuvent être utilisés pour réaliser simultanément plusieurs petites extensions avec gap. Les alignements sans gap provenant de l'étape 2 sont stockés dans une liste. Quand cette liste contient au moins $\mathcal{K}$ alignements sans gap, ils sont traités dans un mode SIMD.

Contrairement à l'extension sans gap, les paires de sous-séquences sont assez similaires car une similarité significative a été détectée dans l'étape 2. De plus, la taille des sous-séquences est plus longue (128 acides aminés) que celle de l'étape 2. Donc, la valeur du score ne peut pas être encapsulée dans un entier non signé 8 bits. Ainsi, huit scores sont réalisés en parallèle, chaque score est stocké dans un entier signé 16 bits. Le pseudo code d'extension est présenté dans la figure 4 du *chapitre 3*.

1.3.3 Accélération par GPU

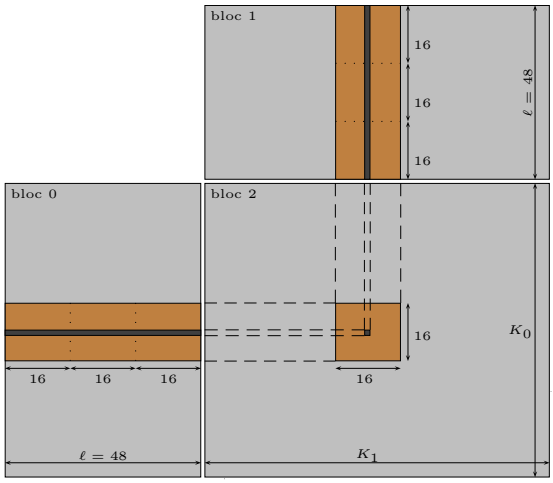


FIGURE 1.3 – Extension sans gap sur GPU

Extension sans gap

Nous accélérons également l'extension sans gap sur le GPU. Supposons que le bloc $0[K_0,\ell]$ et le bloc $1[\ell,K_1]$, avec ℓ la longueur des sous-séquences, correspondent aux deux listes de sous-séquences $BLK0_k$ et $BLK1_k$ (Figure 1.3). Pour effectuer les extensions sur le GPU, nous envoyons les deux blocs à la mémoire globale de GPU. Les scores résultats sont stockés dans le bloc $2[K_0,K_1]$. La valeur de chaque cellule[i,j] du bloc 2 est le score d'extension entre la sous-séquence i du bloc 0 (correspondant à la ligne i) et la sous-séquence j de bloc 1 (correspondant à la colonne j).

Le calcul est ensuite partitionné en threads de la manière suivante : la matrice $[K_0,K_1]$ (bloc 2) est divisée en blocs [16,16] correspondant à 256 threads qui peuvent s'exécuter indépendamment. Chaque thread calcule un élément de la matrice 16×16 en plusieurs itérations. La figure 1.3 illustre ce partitionnement qui, à l'origine, a été établi pour le calcul matriciel [14]. Nous avons repris ce schéma et nous l'avons adapté au calcul de distance entre séquences protéiques.

Dans l'implémentation du calcul des distances, la granularité est le calcul d'un bloc [16,16]. Celui-ci déclenche le transfert dans la mémoire partagée des données nécessaires au calcul (2 x 16 sous-séquences de 16 acides aminés). Les 16 sous-séquences du bloc 0 doivent être comparées avec les 16 sous-séquences du bloc 1. Il y a donc une forte réutilisation des données entre les calculs qui est gérée de la façon suivante : au début du calcul, chaque thread transfère un acide aminé de la mémoire globale vers la mémoire partagée. Cet acide aminé est ensuite partagé par plusieurs threads, mais à des instants différents. Il n'en résulte aucun conflit et une efficacité maximale. Le pseudo-code de GPU est présenté dans la figure 4 du *chapitre 5*.

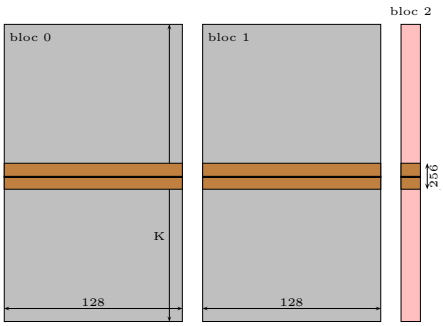


FIGURE 1.4 – Petite extension avec gap sur GPU

Petite extension avec gap

Pour exécuter les petites extensions avec gap sur le GPU, les deux blocs 0 et 1 sont construits. Un bloc correspond à $\mathcal{K}$ sous séquences de taille 128. Ensuite, les deux blocs sont transférés vers la carte graphique pour effectuer $\mathcal{K}$ extensions.

Dans cette extension, un bloc de threads est une matrice [1,256], chaque thread correspond au calcul d'une extension. Par conséquent, une grille sera composée de $\frac{\mathcal{K}}{256}$

blocs de threads. Deux blocs de sous-séquences sont mappés dans la mémoire de texture du GPU. Ce type de mémoire est partagé à tous les threads en lecture seule. Mais la taille de la mémoire cache de texture pour chaque multi-processeur est limitée à quelques dizaines de kilo octets. C'est la raison pour laquelle on ne peut pas garder 256 paires de sous-séquences dans cette zone mémoire.

Les scores résultants des $\mathcal{K}$ extensions sont stockés dans un bloc 2[1,$\mathcal{K}$] comme le montre la figure 1.4 ; le score d'extension d'une paire de sous-séquences à la position i est stocké dans la cellule 2[1,i].

1.3.4 Accélération par FPGA

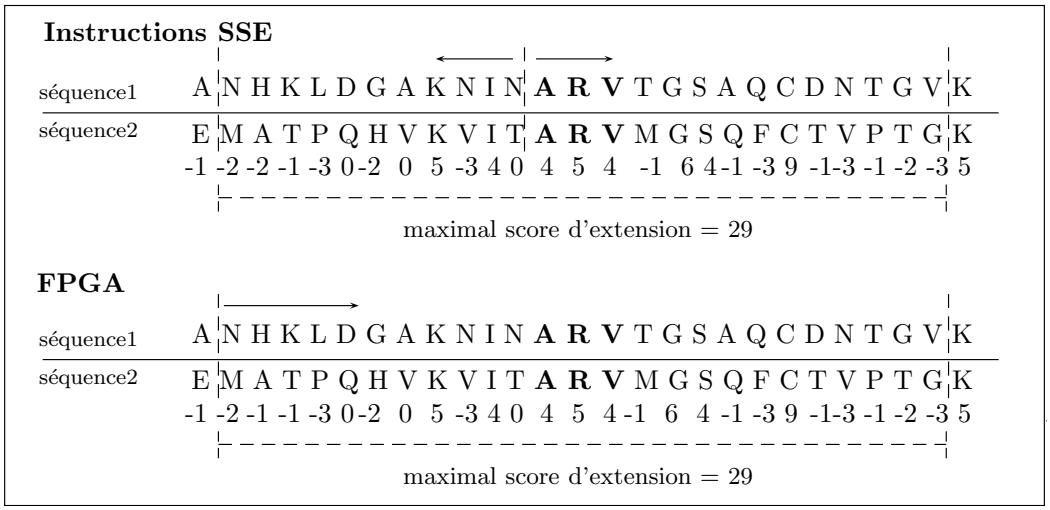


FIGURE 1.5 – Exemples de l'extension sans gap avec instructions SSE et sur FPGA. Les paramètres utilisés sont W=3, L=11, BLOSUM62. L'extension sans gap sur les composants FPGA se réalise dans une direction, en commençant à la base gauche et en se déplaçant vers la droite.

Nous avons implémenté aussi l'extension sans gap sur les composants FPGA en concevant l'architecture dédiée - *opérateur PSC* (Parallel Sequence Comparison), présenté dans la figure 1 du *chapitre 4*. L'opérateur PSC a été réalisé en collaboration avec Alexendre Cornu. L'opérateur PSC reçoit deux flux de données correspondant à deux listes de sous-séquences $BLK0_k$ et $BLK1_k$, et retourne des paires d'entiers correspondant aux numéros des sous-séquences représentant un alignement sans gap avec une similarité significative.

L'architecture de l'opérateur PSC a été conçue pour conduire à un grand nombre de processeurs élémentaires (PE). Elle est indépendant du nombre de PE et le nombre maximal de PE dépend des ressources reconfigurables disponibles. Les PE fonctionnent en mode SIMD en calculant en parallèle des scores d'une sous-séquence de $BLK0_k$ avec plusieurs sous-séquences de $BLK1_k$. Supposons que K_0 et K_1 sont les nombres de

sous-séquences respectivement de $BLK0_k$ et $BLK1_k$ et P est le nombre de PE. Pour calculer $K_0 \times K_1$ extensions, il y a K_1/P itérations. Chaque itération, P sous-séquences de $BLK1_k$ sont d'abord chargées à P PE différents puis, K_0 sous-séquences de $BLK0_k$ sont diffusées à P PE en fonction de caractère par caractère par chaque cycle d'horloge.

Plus précisément, l'extension sans gap sur les PE se réalise dans une direction, en commençant à la base gauche et en se déplaçant vers la droite. La figure 1.5 montre un exemple illustrant la différence entre l'extensions sur le PE par rapport à l'extension par les jeux d'instructions SSE. Pour chaque séquence de $BLK0_k$, les PE effectuent l'extension en ℓ cycles d'horloge, ℓ est la taille des sous-séquences.

1.4 Principaux Résultats

Nous avons implémenté le logiciel PLAST pour la recherche de similarités entre deux banques de séquences. Il se base sur l'heuristique de graine sous ensemble et sur la structure d'index des k-mots. PLAST utilise également les multi-cœurs, les jeux d'instructions SSE, les GPU, les composants FPGA ou clusters (voir le chapitre 6) pour minimiser le temps d'exécution. La qualité des résultats est similaires à BLAST.

Nous avons montré les performances de PLAST (multi-threading) sur ces trois accélérations en comparant au logiciel de référence dans le domaine : BLAST. Les performances de l'implémentation sur cluster ont été comparées à mpiBLAST [15]. Les facteurs d'accélération sont résumés dans la table 1.1.

Les performances de PLAST sont très variées en fonction de la taille des banques à traiter. PLAST obtient de meilleures performances quand la taille des jeux de donnés est importante grâce à l'architecture de l'algorithme qui regroupe des milliers de petits traitements indépendants dans deux localités de calcul, en particulier dans l'étape d'extension sans gap. Cette localité permet d'utiliser efficacement la mémoire cache ainsi que les accélérateurs. De plus, le temps consacré à l'indexation des banques est négligeable par rapport au temps passé dans l'extension pour des grands jeux de données.

	multi-cœur SSE	GPU	FGPA	cluster de PCs
Facteur d'accélération	3 - 6	5 - 10	5 - 20	5 - 7

TABLE 1.1 – Performance de PLAST par rapport à BLAST pour les quatre accélérations.

Par ailleurs, nous avons évalué la sensibilité et la sélectivité de PLAST en utilisant la courbe ROC (en anglais *Receiver Operating Characteristic curve*) [16] et le graphe de couverture par rapport à l'erreur. Les résultats, illustrés dans les figures 5 et 6 du *chapitre 3*, ont montré que PLAST est comparable à BLAST. En outre, pour mesurer la sensibilité de PLAST dans le cadre des grands jeux de données, nous avons fait une évaluation en mesurant le nombre d'alignements trouvés uniquement par PLAST ou par BLAST. Avec une E-value de 10^{-3}, les alignements trouvés par les deux programmes sont très proches (tables 4 et 5 du *chapitre 3*).

1.5 Description par chapitre

Chapitre 2. Nous dressons un état de l'art des méthodes employées de comparaison de séquences génomiques. La première section présente des concepts concernant l'alignement et le score. Le chapitre poursuit sur les méthodes de programmation dynamique et heuristique pour comparer deux séquences. Nous ferons ensuite un bref aperçu des techniques d'indexation proposées aujourd'hui afin d'optimiser l'accès aux données. Pour finir ce chapitre, nous présentons des solutions d'accélérations matérielles par les jeux d'instructions SIMD, les coprocesseurs, les composants FPGA et clusters de machines.

Chapitre 3. Nous présentons en détail l'algorithme de PLAST et l'implémentation de l'algorithme, particulièrement l'implémentation des deux extensions, sans gap et avec gap, avec l'accélération d'instructions SSE. Ce chapitre se compose de *l'article* intitulé : "*PLAST : parallel local alignment search tool for database comparison*" qui a été publié dans le journal *BMC Bioinformatics 2009 10(329)*. Dans cet article, nous évaluons la performance, la sensibilité et la sélectivité de PLAST en comparant à BLAST. Nous montrons également l'accélération des instructions SSE en calculant le profil de PLAST sans et avec les instructions SSE.

Chapitre 4. Nous présentons dans ce chapitre l'implémentation du programme de recherche TPLASTN sur les composants FPGA. Ce chapitre se compose de *l'article* intitulé : "*Implementing protein seed-based comparison algorithm on the SGI RASC-100 platform*" qui a été publié dans les actes du *Reconfigurable Architectures Workshop, IPDPS* 2009. L'article présente la déportation de l'étape d'extension sans gap sur l'accélérateur RASC-100 en concevant l'architecture dédiée - *opérateur PSC*. Cette architecture a été conçue pour conduire à un grand nombre de processeurs élémentaires (PE). Les PE fonctionnent en mode SIMD. Chaque PE calcule un score d'une extension sans gap.

Chapitre 5. Un *chapitre* intitulé : "*Seed-based parallel protein sequence comparison combining multithreading, GPU and FPGA technologies*" d'un livre publié par *Taylor & Francis/CRC Press* et édité par B. Schmidt, à paraître fin 2009. Nous aborderons dans ce chapitre l'implémentation de PLAST sur le GPU et sur les composants FPGA. Dans le cadre du GPU, les extensions sans gap et avec gap sont calculées sur le GPU. Dans l'implémentation sur les composants FPGA, le calcul le plus coûteux, l'extension sans gap, est calculé sur l'opérateur PSC. Nous comparons la différence entre les deux implémentations et évaluons la performance obtenue par ces deux implémentations.

Chapitre 6. Nous présentons dans ce chapitre l'implémentation de PLAST (multi-cœur SSE) sur un cluster de machines en utilisant la librairie MPI (Message Passing Interface), appelée mpiPLAST. Cette implémentation se base sur une segmentation des banques et l'exécution de PLAST sur un cluster. Nous évaluons les performances obtenues en comparant à mpiBLAST [15].

Chapitre 7. En conclusion, nous dressons d'abord un bilan de nos travaux et des résultats obtenus. Nous pointons ensuite les limites de PLAST et proposons différentes solutions pour y améliorer. Enfin nous présentons l'éventail des directions de recherche qui peuvent faire suite à ces travaux.

Chapitre 2

Alignement de séquences

Ce chapitre est consacré à un état de l'art sur la recherche de similarités entre séquences génomiques (ADN ou protéines). Il aborde à la fois les aspects algorithmiques et les aspects mise en œuvre avec un accent particulier sur la minimisation des temps de calcul.

La première section introduit le contexte biologique. La section suivante présente la notion d'alignement et la notion de score. L'algorithme standard de programmation dynamique pour comparer deux séquences est ensuite décrit. La quatrième section détaille les approches heuristiques à base de graines pour accélérer les traitements. La cinquième section présente des techniques d'indexation permettant d'optimiser l'accès aux données lorsque de gros volumes de données sont considérés. La dernière section présente des mises en œuvre sur des accélérateurs matériels.

2.1 Contexte biologique

2.1.1 Notion de biologie moléculaire

Le *métabolisme*, ou fonctionnement des cellules, est assuré en grand partie par des molécules appelées protéines. Les protéines sont des successions d'acides aminés qui, repliés dans une certaine structure 3D, ont une fonction au sein de la cellule. Il y a 20 acides aminés :

$$\Sigma_{20} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$$

L'information génétique codant les protéines est conservée dans les *chromosomes* constitués d'acides désoxyribonucléiques (ADN). L'ADN est une succession de nucléotides pris parmi l'alphabet à 4 lettres, les bases nucléiques :

$$\Sigma_4 = \{A, C, G, T\}$$

Un des rôles de la machinerie cellulaire est de transformer l'ADN en protéines. L'ADN est transcrit en ARN *messager*. Celui-ci est traduit en protéines par les ribosomes selon le code génétique qui associe à chaque *codon* (triplet de nucléotides de Σ_4) un acide aminé de Σ_{20}). La traduction peut se faire selon trois phases de lecture différentes par sens de lecture, soit six phases au total.

2.1.2 Évolution des données biologiques et performance des machines

Les progrès continus des biotechnologies conduisent à un accroissement exponentiel des données biologiques et leurs données associées (annotations, méta-données). La banque Genbank [17], la banque DDBJ [18] et la banque EMBL [19] mettent en commun toutes les séquences nucléiques du domaine public séquencées de l'INSDC (International Nucleotide Sequence Database Collaboration). La banque Genbank [17] contenait, en avril 1999, 2.56 milliards de bases. Aujourd'hui cette banque contient, en avril 2009, 103 milliards de bases ; sa taille a été multipliée par 40 (Figure 2.1). Pour les protéines, la situation est similaire. La taille de la banque UniProtKB/TrEMBL [20] a été multipliée par 43 de 1999 à 2009.

La croissance des banques devient alors problématique si on la compare à la puissance de calcul des processeurs qui double seulement tous les 18 mois selon la loi de Moore [21] (Figure 2.1).

2.2 Alignement et score

Étant donnée l'expansion et la taille grandissante des bases de données génomiques, la comparaison de séquences est devenue indispensable dans le domaine de la biologie moléculaire. Si deux molécules (ADN ou protéine) démontrent une forte similarité, cela signifie qu'elles partagent probablement une fonction ou une structure moléculaire semblable. Dans ce sens, les résultats d'une comparaison de séquences peuvent être utilisés pour inférer des homologies entre séquences.

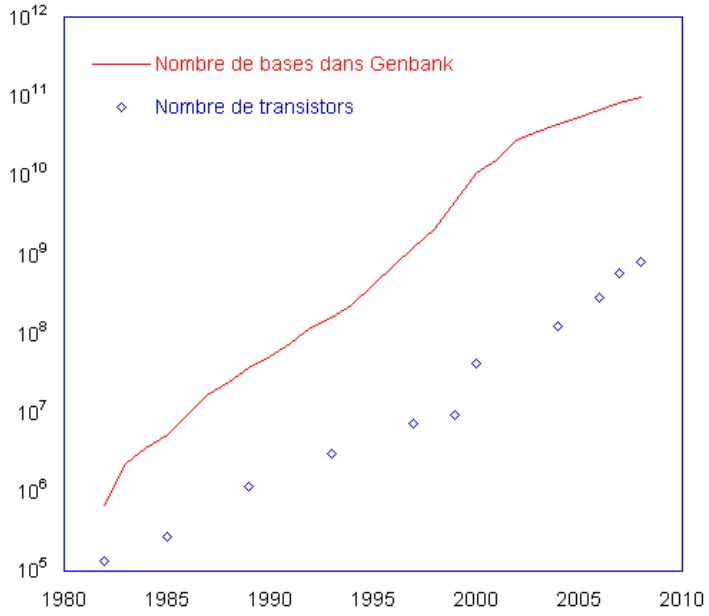


FIGURE 2.1 – Évolution de la taille de Genbank et du nombre de transistors dans les processeurs (Intel). Le nombre de transistors croît moins vite que la taille de Genbank.

Mesurer la similarité entre deux séquences génomiques revient à aligner ces séquences, c'est à dire à mettre en correspondance des régions similaires. En général, on associe un score à cette similarité, ce score représentant la somme des coûts d'opérations élémentaires pour passer d'une séquence à l'autre. Ainsi, à chaque position dans l'alignement correspond une des trois situations suivantes :

- un *appariement* ou *match* quand le même caractère apparaît dans les deux séquences ; une valeur positive est associée ;
- une *substitution* (ou *mismatch*) lorsqu'il y a deux caractères différents ; une valeur négative (éventuellement positive pour des acides aminés proches) est associée ;
- un *gap*, c'est à dire une *insertion* d'un caractère dans seulement une séquence ou symétriquement une *délétion* dans une des deux séquences ; une valeur négative est assignée.

La figure 2.2 représente un alignement entre deux séquences de protéines. Son score est calculé à partir de la matrice BLOSUM62 [22]. Les *gaps* sont symbolisés par le caractère $-$, les appariements par les caractères $|$ (*match* exact) et $+$ (*match* positif), et les substitutions par l'absence de caractère.

Les séquences biologiques se transforment au cours de l'évolution. Elles peuvent acquérir des résidus ou, au contraire, en perdre ou, plus simplement, changer leur composition (remplacement de résidus). Ces changements peuvent être modélisés par une matrice de substitution, habituellement de dimension 20×20 pour les acides aminés et de dimension 4×4 pour les nucléotides. Ces matrices traduisent la probabilité qu'un acide aminé ou qu'un nucléotide soit muté en un autre sur une période de temps d'évo-

```
Séquence 1 :  VMVDFWAEWCGPCKMLIPIIDEISKELQ--DKVKVLKMNIDENPKTPSEYGIRSIPTIML
              |+ ||+|+|||||| + |+  + +|     + +   |+|+|   +    | + ++|| +
Séquence 2 :  VVADFYADWCGPCKAIAPMYAQFAKTFSIPNFLAFAKINVDSVQQVAQHYRVSAMPTFLF
```

FIGURE 2.2 – Un alignement de séquences entre deux protéines. Dans la ligne du milieu, le caractère | indique un *match* exact tandis que le caractère + signale un *match* positif.

lution donnée. Les matrices de substitution d'acides aminés les plus utilisées sont les matrices PAM et BLOSUM. La matrice BLOSUM62 [22], calculée à partir de blocs de séquences identiques à plus de 62%, est la matrice la plus couramment utilisée.

On peut ainsi fixer une pénalité $g_{penalty}$ pour chaque insertion ou délétion (gap). Un gap de k positions successives obtient une pénalité globale valant $g(k) = g_{open} + (k-1)g_{extend}$. Dans ce cas, g_{open} représente le coût pour ouvrir un gap et g_{extend} représente le coût pour étendre un gap déjà existant.

Il est important de pouvoir déterminer si le score d'un alignement est pertinent ou s'il est obtenu purement par hasard. Lorsqu'une séquence est alignée contre toutes les séquences d'une banque, seuls les scores maximaux des alignements sont conservés. Il faut donc disposer d'un moyen pour exprimer la probabilité d'obtenir des alignements pertinents (non dus au hasard). La *E*-value [23], ou *expected value*, a été introduite par Altschul pour répondre à ce besoin. Elle se calcule de la manière suivante :

$$E = Kmne^{-\lambda S}$$

où m et n sont les longueurs des séquences (ou les longueurs de la séquence requête et de la banque de séquences) ; K et λ sont des constantes statistiques de normalisation [24, 25] qui dépendent des paramètres de calcul du score. La *E*-value d'un alignement représente donc la probabilité de générer au hasard un alignement ayant un score supérieur ou égal à S. Plus la E-value est faible, plus l'alignement est significatif.

L'alignement de séquences peut se diviser en deux catégories : l'alignement global et l'alignement local. Le premier est utilisé pour calculer la similarité totale entre deux séquences. Les séquences sont alignées sur toute leur longueur. L'algorithme qui effectue cet alignement est appelé algorithme de Needleman-Wunsch [26] et a été introduit en 1970. Le second détecte simplement les régions locales de fortes similarités entre deux séquences. Cet algorithme a été proposé par Smith-Waterman [27] en 1981.

En fonction des problématiques étudiées, on choisira le type d'alignement le mieux adapté. Par exemple, pour des études phylogénétiques qui consistent à déterminer des distances entre séquences de même nature, les alignements globaux seront considérés. Par contre, pour rechercher des zones fonctionnelles identiques entre protéines, les alignements locaux seront mieux appropriés.

2.3 Programmation dynamique

La programmation dynamique est au cœur des algorithmes d'alignement de séquences. Elle est basée sur le fait que tous les évènements sont possibles et calculables

entre des séquences. Les algorithmes de Needleman-Wunsch (NW) et Smith-Waterman (SW) se basent sur la programmation dynamique.

Dans un premier temps, nous présentons le problème de la recherche d'alignement global. Pour décrire l'algorithme NW, définissons deux séquences $A{=}a_1a_2...a_m$ et $B{=}b_1b_2...b_n$ à aligner. L'algorithme consiste à construire progressivement un alignement sur la totalité de la longueur de A et B.

Pour trouver l'alignement global optimal entre les deux séquences A et B, le principe est de calculer une matrice H de dimensions $m \times n$. La valeur $H(i,j)$ contient le score du meilleur alignement entre les i premiers caractères de A avec les j premiers caractères de B. On construit cette matrice de manière récursive. La ligne $i = 0$ et la colonne $j = 0$ sont initialisées à 0 ; les coefficients suivants sont complétés selon l'équation NW : pour $\forall i, j \neq 0$,

$$H(i,j) = max \begin{cases} H(i-1,j-1) + sub(A_i, B_j) & \text{(substitution)} \\ F(i,j) & \text{(insertion)} \\ E(i,j) & \text{(délétion)} \end{cases}$$

avec

$$E(i,j) = max \begin{cases} H(i,j-1) - g_{open} \\ E(i,j-1) - g_{extend} \end{cases}$$

$$F(i,j) = max \begin{cases} H(i-1,j) - g_{open} \\ F(i-1,j) - g_{extend} \end{cases}$$

$sub(A_i, B_j)$ est le score de substitution des bases A_i et B_j

Cependant, dans beaucoup d'applications, on préfère rechercher des similarités locales. Autrement dit, on cherche les meilleurs paires de facteurs similaires entre A et B. L'alignement local a été introduit pour surmonter la difficulté à aligner correctement des génomes d'espèces proches, ayant subies des mutations liées à leur évolution. Appelons $H(i,j)$ la similarité de la paire de facteur la plus similaire parmi elles terminant en A_i et B_j. Cette similarité peut être calculer par l'équation SW : pour $\forall i, j \neq 0$,

$$H(i,j) = max \begin{cases} H(i-1,j-1) + sub(A_i, B_j) & \text{(substitution)} \\ F(i,j) & \text{(insertion)} \\ E(i,j) & \text{(délétion)} \\ 0 \end{cases}$$

avec

$$E(i,j) = max \begin{cases} H(i,j-1) - g_{open} \\ E(i,j-1) - g_{extend} \end{cases}$$

$$F(i,j) = max \begin{cases} H(i-1,j) - g_{open} \\ F(i-1,j) - g_{extend} \end{cases}$$

$sub(A_i, B_j)$ est le score de substitution des bases A_i et B_j

La matrice H ainsi complétée donne le score du meilleur alignement entre A et B par $H(m,n)$. Cet algorithme permet aussi de déterminer l'alignement. Pour cela, il suffit de mémoriser le terme qui est maximal à chaque étape de la récursivité.

Un alignement local s'obtient par la méthode du retour sur trace (*backtracing*) :

	T	S	V	F	L	A	Y	N	G	L	K	A	A	N	R	V	P	L
L	-1	-2	1	0	4	-1	-1	-3	-4	4	-2	-1	-1	-3	-2	1	-3	4
S	1	3	-4	-1	-2	5	-3	0	-3	-6	4	-1	0	0	-4	-4	0	-5
Q	-1	1	1	-7	-3	-3	4	-3	-2	-5	-5	3	-2	0	1	-6	-5	-2
Y	-2	-3	0	4	-7	-5	4	2	-6	-3	-7	-7	1	-4	-2	0	-9	-6
L	-1	-4	-2	0	8	-3	-4	1	-2	-2	-5	-8	-8	-2	-6	-1	-3	-5
A	0	0	-4	-4	-1	12	1	0	1	-2	-3	-1	-4	-6	-3	-6	-2	-4
H	-2	-1	-3	-5	-4	1	14	3	2	1	0	-1	-2	-3	-4	-5	-6	-5
N	0	-1	-4	-6	-5	0	3	20	9	8	7	6	5	4	3	2	1	0
K	-1	0	-3	-7	-6	-1	2	9	18	7	13	6	5	5	6	1	1	-1
R	-1	-2	-3	-6	-7	-2	1	8	7	16	9	12	5	5	10	3	-1	-1
L	-1	-3	-1	-3	-2	-3	0	7	6	11	14	8	11	2	3	11	0	3
K	-1	-1	-5	-4	-5	-3	-1	6	5	4	16	13	7	11	4	1	10	-1
V	0	-3	3	-6	-3	-5	-2	5	4	6	5	16	13	4	8	8	1	11
A	0	1	-3	1	-7	1	-3	4	5	3	5	9	20	11	8	8	7	5
N	0	1	-2	-6	-2	-7	-1	3	4	2	3	4	9	26	15	14	13	12
V	0	-2	5	-3	-5	-2	-5	2	1	5	2	3	8	15	23	19	12	14
P	-1	-1	-4	1	-6	-6	-5	1	0	-1	4	2	7	14	13	21	26	15
L	-1	-3	0	-4	5	-6	-7	0	-1	4	0	3	6	13	12	14	18	30

FIGURE 2.3 – Retrouver l'alignement global de protéine. Le score de cet alignement est de 30. Dans cet exemple, la matrice BLOSUM62 est utilisée avec g_{open} de 11 et g_{extend} de 1.

à partir d'une valeur maximale de H située à l'emplacement (i, j), on détermine quel coefficient $(i-1, j)$, $(i-1, j-1)$ ou $(i, j-1)$ est maximal, puis on recommence à partir de ce dernier. L'itération se termine lorsqu'une valeur optimale nulle ou l'emplacement $(0, 0)$ est atteint. Autrement dit, on établit dans la matrice un chemin qui correspond au passage des scores les plus élevés. Cependant, un alignement global est obtenu en partant de la case $H(m, n)$ par le même processus de retour, mais l'itération se termine lorsque l'emplacement $(0, 0)$ est atteint.

La figure 2.3 montre un exemple de calcul de matrice de programmation dynamique avec l'équation NW et de calcul de retour sur trace d'alignement global entre les deux séquences données. La figure 2.4 illustre un exemple de calcul de matrice de programmation dynamique avec l'équation SW et de calcul de retour sur trace d'alignement local entre les deux séquences données.

2.4 Méthodes heuristiques

L'algorithme de Smith-Waterman est un algorithme optimal : il retourne tous les alignements de score maximal positif. Sa complexité est quadratique ($\mathcal{O}(m \times n)$, où m et n sont les longueurs respectives des deux séquences étudiés). Cette complexité est raisonnable lorsque l'on compare deux séquences de quelques milliers d'acides nucléiques ou d'acides aminés. Mais dans le cas où les séquences sont constituées de plusieurs millions de bases, son temps d'exécution devient impraticable et des algorithmes plus efficaces sont alors nécessaires.

Des méthodes heuristiques existent afin d'accélérer le processus d'alignement local.

	T	K	A	F	L	A	Y	N	G	L	K	A	A	N	I	N	Y	V
L	0	0	0	0	4	0	0	0	0	4	0	0	0	0	2	0	0	1
S	1	0	1	0	0	5	0	1	0	0	4	1	1	1	0	3	0	0
Q	0	2	0	0	0	0	4	0	0	0	1	3	0	1	0	0	2	0
Y	0	0	0	3	0	0	7	2	0	0	0	0	1	0	0	0	7	1
L	0	0	0	0	7	0	0	4	0	4	0	0	0	0	2	0	0	8
A	0	0	4	0	0	11	0	0	4	0	3	4	4	0	0	0	0	0
H	0	0	0	3	0	0	13	2	1	1	0	1	2	5	0	1	2	0
N	0	0	0	0	0	0	2	19	8	7	6	5	4	8	2	6	0	0
K	0	5	0	0	0	0	1	8	17	6	12	5	4	4	5	2	4	0
R	0	2	4	0	0	0	0	7	6	15	8	11	4	4	1	5	0	1
L	0	0	1	4	4	0	0	6	5	10	13	7	10	1	6	0	4	1
K	0	5	0	0	2	3	0	5	4	3	15	12	6	10	1	6	0	2
V	0	0	5	0	1	2	2	4	3	5	4	15	12	3	13	2	5	4
A	0	0	4	3	0	5	0	3	4	2	4	8	19	10	7	11	5	5
N	0	0	0	1	0	0	3	6	3	1	2	3	8	25	14	13	12	11
D	0	0	0	0	0	0	0	4	5	0	1	2	7	14	22	15	10	9
P	0	0	0	0	0	0	0	0	2	2	0	1	6	13	11	20	12	8
K	0	5	0	0	0	0	0	0	0	0	7	0	5	12	10	11	18	10

FIGURE 2.4 – Retrouver l'alignement local de protéine. Le score de cet alignement est de 25. Dans cet exemple, la matrice BLOSUM62 est utilisée avec g_{open} de 11 et g_{extend} de 1.

Par exemple, FASTA [28] et BLAST [1] se basent sur une heuristique efficace permettant de cibler directement de courtes zones identiques potentiellement intéressantes. Plus précisément, de manière à ne pas calculer les $m \times n$ possibilités d'alignement, ces algorithmes effectuent une sélection sur les paires de positions $(i, j) \in m \times n$. Ces algorithmes heuristiques sont conçus à l'origine pour permettre une recherche efficace sur des larges banques de données génomiques. Nous présentons maintenant le principe de ces heuristiques.

2.4.1 FASTA

FASTA [28, 29] est un programme de comparaison de séquences orienté vers la comparaison d'une séquence requête sur une banque de séquences. L'algorithme FASTA est basé sur une heuristique qui détecte des petites régions identiques, appelées k-tuples. Il se divise en quatre étapes :

Étape 1 : identification des régions de haute similarité. Il s'agit d'identifier des régions de haute similarité pour l'alignement entre la séquence requête et une séquence dans la banque. Pour cela, la taille de ces régions (paramètre k-tuple) est spécifiée, car elle commande la sensibilité et la vitesse de l'algorithme. Généralement, on spécifie k-tuple=2 pour des séquences protéiques et 6 pour des séquences d'ADN. L'utilisation d'une table de hachage (voir page 28) permet d'identifier plus rapidement les résidus identiques (k-tuple). L'algorithme détermine ensuite les 10 diagonales contenant le plus de résidus identiques.

Étape 2 : détermination du score *init1*. En fonction de la matrice de substitution, les dix diagonales sont réévaluées en calculant le score d'alignement sans gap. Soit

init1 le score le plus élevé obtenu à cette étape.

Étape 3 : détermination du score *initn*. À cette étape, les diagonales trouvées à l'étape 1 dont le score dépasse un certain seuil *Cutoff* sont reliées entre elles en permettant les insertions/délétions. Le score de ces nouvelles régions est obtenu en combinant le score des diagonales reliées ensembles avec une pénalité de jonction des diagonales. Soit *initn* le score le plus élevé obtenu. Ce score est calculé pour chaque séquence de la banque. Le score *initn* est utilisé pour classer par ordre décroissant les différents alignements dans la banque pour la séquence requête.

Étape 4 : détermination du score *opt* **et finalisation de la recherche**. On calcule un score final *opt* en utilisant l'algorithme de Smith-Waterman sur un voisinage (de rayon prédéfini, la figure 2.5) de l'alignement correspondant à *initn*.

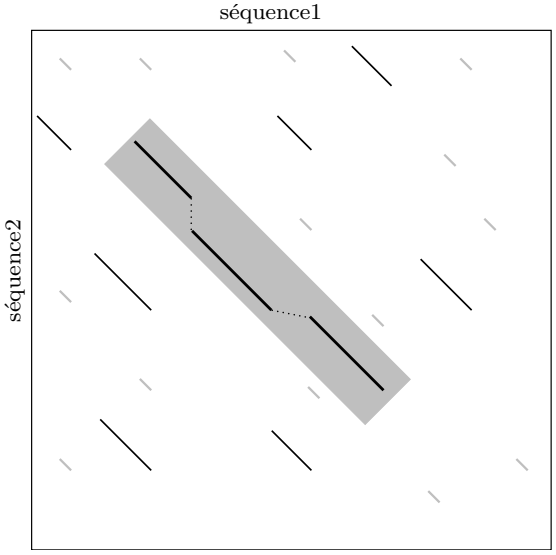


FIGURE 2.5 – Principe de FASTA : tous les k-tuples sont d'abord repérés ; on ne garde que les dix meilleurs alignements (dix diagonales en gras). On essaye ensuite de relier un maximum (ici 3) de ces dix zones. Enfin, on calcule le score final sur un voisinage par programmation dynamique.

2.4.2 BLAST

Le programme BLAST (acronyme de *Basic Local Alignment Search Tool*) est apparu en 1990 et permet aussi de calculer des alignements locaux. Il est basé sur un algorithme heuristique similaire à FASTA et permet de trouver des séquences similaires à une séquence requête dans une banque de données. La première version de BLAST [1] ne prend pas en compte les insertions/délétions. Une version améliorée avec gap, NCBI-BLAST [2], est élaborée en 1997. La rapidité et la sensibilité de BLAST en font l'un des outils d'alignement de séquences les plus utilisés. L'algorithme de BLAST se décompose en trois étapes :

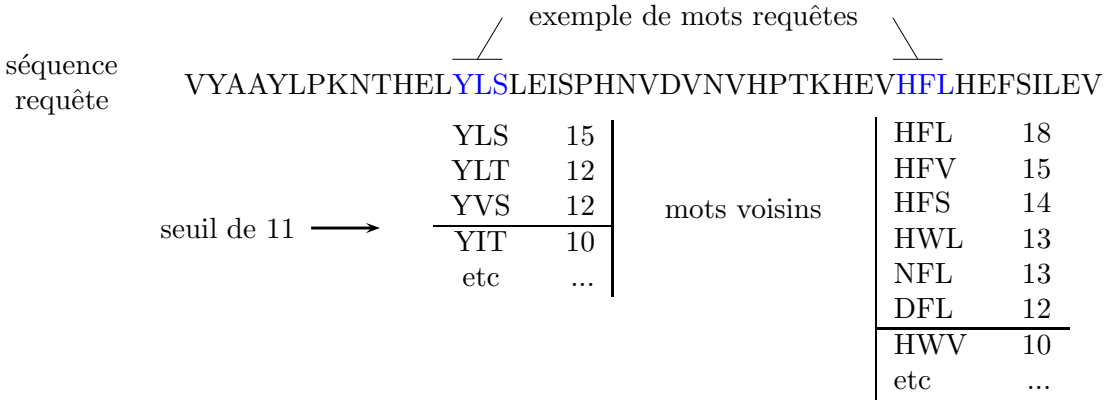


FIGURE 2.6 – Exemple de mots voisins de seuil de 11 avec la matrice BLOSUM62.

Étape 1 : recherche de points d'ancrage. Dans la séquence requête, on recense tous les mots de taille W, appelés *graines*. Pour chaque mot requête, le programme crée une liste de tous les mots voisins de même taille, ayant un score supérieur à T lorsqu'ils sont comparés au mot requête (Figure 2.6). Dans la programme BLASTN, (la version d'ADN), le paramètre T n'est pas utilisé. Les mots voisins sont stockés dans une table de hachage, ce qui permet d'effectuer un crible sur les séquences de la banque, comme suit : la banque est lue séquentiellement, en considérant chaque sous-chaîne de W caractères. Pour chacune d'elle, on recherche si elle est présente dans la table de hachage ; si c'est le cas, on obtient ce que l'on appelle un point d'ancrage.

Étape 2 : extension sans gap. À partir d'un point d'ancrage, on essaie de l'étendre à droite et à gauche pour produire une extension sans gap. Plus précisément, BLASTP, (la version de protéine), effectue une extension sans gap uniquement s'il y a deux points d'ancrage proches l'un de l'autre (Figure 2.7). Par défaut, la distance entre ces deux points d'ancrage doit être inférieure à 40 acides aminés. Un alignement sans gap incluant ces 2 points d'ancrage est alors calculé. BLASTN est différent : il réalise une extension sans gap pour chaque point d'ancrage. L'extension s'arrête lorsque le score courant devient inférieur à un certain seuil (seuil de *X-drop*) par rapport au score maximal. Si le score affecté à cet alignement dépasse un seuil prédéterminé, alors on passe à l'étape 3.

Étape 3 : extension avec gap. Dans cette étape, on cherche encore à étendre l'alignement en incluant des erreurs de gap. Cette opération est réalisée par des techniques de programmation dynamique et s'appuie sur les résultats de l'étape précédente : on part des extrémités de l'alignement sans gap et on explore, à partir de ces endroits, la possibilité d'inclure ou de supprimer certains caractères (acides aminés ou acides nucléiques). Dans cette étape, BLAST utilise également un value de *X-drop* pour limiter l'espace de recherche.

L'avantage de BLAST est sa rapidité par rapport aux techniques de programmation dynamique. Cette rapidité est d'autant plus marquée que la taille W du point d'ancrage

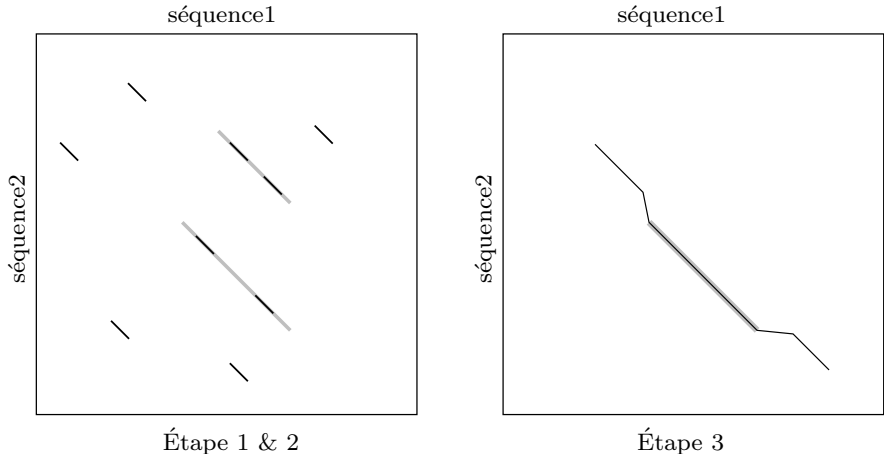


FIGURE 2.7 – Principe de BLAST : lorsque les deux points d'ancrage sont trouvés proche l'un de l'autre, une extension sans gap se réalise. Dans cet exemple, le score de l'alignement plus long dépasse un seuil prédéterminé et passe à l'étape 3 où il construira un meilleur alignement en incluant des erreurs de gap.

et la valeur T sont importantes. En effet, plus W est grand, plus la possibilité de trouver un mot de cette de taille est faible et moins les étapes 2 et 3 seront exécutées. Pour la comparaison de protéines (BLASTP), la taille est usuellement de 3 caractères et de 11 caractères pour la comparaison de génomes (BLASTN). Avec ces paramètres, BLAST est plus rapide que FASTA.

2.4.3 Type des graines

L'approche heuristique par graines, telle qu'employée par BLAST, est un moyen de trouver efficacement des points d'ancrage pour un alignement. Une variante de cette approche consiste à ne pas baser la recherche sur des graines contiguës, mais sur des graines pouvant contenir des espaces, appelées graines espacées.

Les graines espacées sont caractérisées par un mot binaire de longueur ℓ : un 1 correspond à une position sur laquelle un appariement des bases entre deux séquences est nécessaire, et un 0 n'impose aucune contrainte. Pour définir une graine espacée, nous définissons le mot $Q = q_1, ..., q_l$, ayant un poids w ($w \leqslant \ell$) défini par le nombre de uns.

Les graines espacées améliorent la première étape des algorithmes heuristiques d'alignement de séquences. Cette approche semble triviale en utilisant des méthodes de projection. Par exemple, la méthode proposée par Buhler [30] permet de projeter un espace de ℓ-mer ayant un poids w vers un espace de w-mer. Elle est particulièrement intéressante pour l'alignement d'ADN. Une graine espacée bien choisie a plus de chance d'avoir au moins une occurrence dans un alignement qu'une graine contiguë. Les graines espacées peuvent être choisies aléatoirement [31, 32] mais il est plus judicieux d'effectuer un pré-calcul pour sélectionner les graines les plus efficaces sur des modèles d'aligne-

ment. PatternHunter [3, 4] est l'un des premiers logiciels utilisant les graines espacées. Il a également introduit la notion de position sans importance (*mismatch*) dans une graine.

Kent, dans le programme BLAT [7], propose d'utiliser une graine de faible exigence, appelée graine de *mismatches* : au plus r *mismatches* sont permis (en général, r vaut 1 ou 2) dans l'alignement sans gap de k symboles consécutifs des deux séquences pour un point d'ancrage donné. La différence entre les graines espacées de l'approche PatternHunter et celle de BLAT réside dans le fait que la graine de *mismatches* est en réalité un ensemble de graines espacées. Un autre programme basé sur plusieurs détections proches d'une graine espacée est YASS [33, 34], proposé par Kucherov et Noé. Leur approche utilise aussi le modèle de deux points d'ancrage comme BLAST, mais un chevauchement est autorisé.

Pour la comparaison de séquences d'ADN, les graines espacées nécessitent w positions d'appariement dans une zone de taille ℓ. Elles permettent de détecter de manière probabiliste des régions présentant un certain pourcentage de similarité. Un avantage avéré des graines espacées est leur sensibilité élevée, comparée à celles des graines contiguës utilisées dans le programme BLASTN. L'utilisation de graines espacées multiples [4, 35, 36, 37] donne d'excellents résultats : la sensibilité est proche de celle de l'algorithme de Smith-Waterman.

De nouveaux modèles ont été définis à partir des graines espacées, afin d'améliorer leur sensibilité pour la recherche de similarités dans des séquences ADN et protéiques. Les graines vecteurs [38, 39, 40, 41] combinent à la fois des propriétés des graines espacées classiques avec des techniques de seuil minimum (issues de BLASTP) ainsi que l'approche de BLAT. Une graine vecteur est un couple $Q = (v, T)$, où v est un vecteur de nombres réels $\{v_1, v_2, ..., v_\ell\}$, et où T est un réel donnant le seuil de la graine. Pour un alignement $X = (x_1, x_2, ..., x_n)$, une graine vecteur Q détecte l'alignement X si et seulement si il existe une position $0 < i \leq n - \ell$ telle que

$$\sum_{j=1}^{\ell} (v_j \times x_{i+j-1}) \geq T$$

.

Les graines vecteurs ont été initialement proposées pour augmenter la sensibilité de la recherche des alignements de protéines [38], mais n'ont connu qu'un succès limité dans ce domaine. En effet, les régions conservées des protéines ne sont en pratique pas assez larges pour ce type d'approche. Brown [41] propose d'utiliser des graines vecteurs multiples pour l'alignement de protéines, mais ce procédé est coûteux en temps de calcul. De plus, les graines vecteurs ne peuvent être détectées à l'aide d'un index aussi simple que celui des graines espacées.

D'autres modèles, comme les graines sous-ensemble [42, 43, 6] (*subset seeds*), sont une extension des graines espacées et des graines vecteurs mais elles permettent d'utiliser une table de hachage pour localiser les points d'ancrage. Cette approche est principalement conçue pour la comparaison des séquences protéiques. Pour un alphabet Σ (d'acides aminés, par exemple), un caractère de graine sous-ensemble est une relation

binaire, symétrique et réflexive sur Σ. Soit $\mathcal{B}$ un alphabet de graine sous-ensemble, i.e. un ensemble de caractères de graine sous-ensemble. Alors, une graine sous-ensemble est un mot sur $\mathcal{B}$. La graine sous-ensemble $\pi = \pi_1, ..., \pi_k$ est une relation binaire symétrique et réflexive sur des mots de Σ^k : pour $s_1, s_2 \in \Sigma^k$, $s_1 \sim_\pi s_2$ ssi $\forall i \in [1, k]$, nous avons $\langle s_1[i], s_2[i] \rangle \in \pi_i$. Pour l'alignement de protéines, l'étude [6] montre que les graines sous-ensemble et l'approche de BLASTP sont comparables en terme de sensibilité. La figure 2.8 présente un exemple de graine sous-ensemble d'acides aminés.

$$\begin{cases} \pi_1 = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\} \\ \pi_2 = \{CFYWMLIV, GPATSNHQEDRK\} \\ \pi_3 = \{A, C, FYW, G, IV, LM, NH, P, QED, RK, TS\} \\ \pi_4 = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\} \end{cases}$$

FIGURE 2.8 – Un exemple de graine sous-ensemble d'acides aminés

2.5 Méthodes d'indexation

L'utilisation d'un index pour la recherche de similarités dans un texte est une approche relativement récente. Plusieurs méthodes ont été développées pour la recherche d'appariements exacts avec diverses structures de données (arbre des suffixes, table de suffixes, ...). Cependant ce n'est que depuis une vingtaine d'années que l'on a commencé à adapter ces techniques à la recherche de similarités dans plusieurs domaines d'applications : fouille de textes, fouille d'images, bioinformatique.

En comparant les séquences génomiques, lorsque de gros volumes de séquences sont considérés l'indexation permet réellement d'optimiser le temps d'accès aux données, de réutiliser la mémoire cache et de calculer en parallèle. Les méthodes de construction de l'index peuvent être variées. Nous pouvons distinguer ici les différents types d'index : l'index des k-mots, l'arbre des suffixes et l'index inversé.

2.5.1 Index des k-mots

L'idée de codification des mots (hash-coding ou k-tuple) a été proposée par Dumas et al. [44] et est maintenant largement utilisée, particulièrement pour la comparaison de séquences génomiques. Cette méthode se base sur la recherche de courts fragments de texte qui sont supposés conservés entre deux séquences. Cette approche, commune à FASTA et BLAST, utilise l'index par hachage pour détecter des fragments en commun. Les fragments peuvent être contigus ou non-contigus. Ces fragments sont en général de taille fixée de k sur l'alphabet des séquences Σ. Sous réserve que la plage d'entiers $[0...\Sigma^k - 1]$ puisse être codée dans un mot de machine, il est alors possible d'associer un entier *key(u)* à chaque mot u $\in \Sigma^k$.

Il y a plusieurs travaux qui utilisent l'index des k-mots pour construire l'index d'une banque de séquences en permettant de réduire le temps de recherche de matches de graines. Nous présentons les travaux qui utilisent cette approche.

RAMdb

RAMdb [45], l'acronyme de *Rapid Access Motif database*, est un algorithme qui réalise la recherche de courts motifs sur une banque de séquences génomiques. La taille des motifs est d'environ 10 caractères pour les nucléotides et 4 pour les protéines. Dans cet algorithme de Fondrat et Dessen, chaque séquence a été indexée par des mots chevauchants dans une structure de table de hachage. Pour chaque mot, les numéros de séquences contenant ce mot et ses positions dans ces séquences sont stockés dans une liste associée.

La principale fonction de RAMdb est la recherche de matches de courtes chaînes de caractères, en particulier pour localiser les motifs de taille égale ou légèrement supérieure que le mot indexé en utilisant plusieurs chevauchements de matches. Selon les auteurs, l'approche de RAMdb a montré des facteurs d'accélération de 0 à 800 fois plus rapide que les recherches exhaustives. La taille de RAMdb pour des nucléotides et des protéines prend respectivement 2 et 5.5 fois la taille de la banque de donnée d'origine.

FLASH

FLASH [46] est un outil de recherche approximative, basé sur un index redondant de données et un schéma probabiliste. Pour chaque mot de taille ℓ, FLASH stocke, dans la table de hachage, toutes les sous-séquences ordonnées contiguës et non-contiguës possibles de taille m qui commencent à la première base du mot, où $m < \ell$. Par exemple, pour une séquence nucléique ACCTGATT, la taille du premier mot $\ell = 5$ et $m = 3$, soit ACC, ACT, ACG, ACT, ACG et ATG ; chacune des sous-séquences permutées commence à la base A, la première base du mot de $\ell = 5$. Pour chaque sous-séquence permutée, la position de cette sous-séquence permutée dans la séquence et le numéro de séquence contenant cette sous-séquence permutée sont stockés dans la table de hachage. Le schéma permuté offre un modèle qui permet la recherche approximative en considérant un nombre d'insertions/délétions et substitutions dans les séquences génomiques.

Les auteurs ont déclaré que FLASH a été dix fois plus rapide que BLAST pour les petites banques de données et bien plus précis et sensible que BLAST. Cependant, l'index redondant stocké dans la table de hachage et non-compressé est un problème majeur de FLASH.

SSAHA

SSAHA [47] est un autre programme de recherche de régions similaires entre une séquence requête et les séquences d'une banque, basé sur l'organisation d'une banque de données dans une structure de table de hachage. SSAHA convertit les séquences de la banque dans une table de hachage par intervalles non chevauchants de taille k. Pour chaque intervalle, le numéro de séquence et la position de cet intervalle sont stockés dans deux structures de données : une liste de positions L et un tableau de pointeurs A.

Pour chaque intervalle chevauchant de la séquence requête, une liste de positions de cet intervalle dans les séquences de la banque est obtenue. Ensuite, les points d'ancrage

sont calculés et triés en fonction de leurs positions et diagonales. Les régions identiques sont déterminées en considérant les points d'ancrage chevauchant. En triant en fonction des diagonales, les régions de mismatch sont calculées par l'insertion/délétion du nombre de caractères correspondant. SSAHA a besoin de huit octets pour chaque entrée de l'index. La sensibilité de recherche de SSAHA est faible par rapport à BLAST et FASTA.

BLAT

BLAT (BLAST-Like Alignment Tool) [7] est un autre programme de recherche d'alignement local, basé sur la même approche que le programme BLAST, mais plutôt que de construire l'indexation d'une séquence requête, BLAT construit l'indexation des séquences d'une banque au début et scanne ensuite systématiquement la séquence de requête. Le programme BLAT construit efficacement la structure de l'index qui permet d'indexer complètement les chromosomes dans la mémoire système. De plus, BLAT peut déclencher les extensions sur un certain nombre de points d'ancrage qui soient parfaits ou presque parfaits (r erreurs).

Kent a déclaré que BLAT a été 500 fois plus rapide que les outils populaires pour l'alignement d'ADN et 50 fois plus rapide pour l'alignement de protéine. Cependant, BLAT utilise des intervalles non chevauchants pendant la construction d'indexation des séquences de la banque. En conséquence, BLAT a besoin de 0.9 GB pour indexer le génome humain, environ 30% de la taille origine. Gertz [48] a montré que BLAT a été proposé pour rechercher rapidement les alignements de haute similarité.

miBLAST

miBLAST [49] est aussi un logiciel de recherche de similarités entre des séquences requêtes et une banque de séquences (nucléotide), basé sur un index des k-mers. L'implémentation de miBLAST est composée de trois principales phases : la construction de l'index, le filtrage et la génération d'alignements.

Dans l'étape d'indexation, miBLAST indexe les séquences de la banque par intervalles chevauchants de taille k, et indexe les séquences requêtes par intervalles chevauchants de taille ℓ, où $k \leq \ell$. Cependant, pour chaque intervalle, miBLAST stocke seulement le numéro de la séquence contenant cet intervalle. En conséquence, si un intervalle apparaît plusieurs fois dans une séquence, il y a un unique intervalle stocké. Pour chaque séquence requête, la phase de filtrage construit un ensemble de séquences de la banque contenant les mots qui apparaissent dans la séquence requête. Enfin, la phase de génération d'alignement, le composant d'alignement du programme BLAST et ℓ est égale la taille du mot de BLAST, effectue une comparaison de séquences. miBLAST a été conçu pour la recherche de similarités avec des séquences requêtes courtes d'ADN, de moins de 100 bases.

2.5.2 Arbre des suffixes

L'arbre des suffixes sur un texte de longueur N possède N feuilles. Les feuilles de l'arbre contiennent un numéro qui correspond à la position de début du suffixe dans le

texte. Les branches peuvent être étiquetées de différentes manières : par une chaîne de caractères de longueur supérieure ou égale à 1 ou par un couple (p,l) qui correspond à la sous chaîne commençant à la position p, de longueur l, dans le texte. On termine le texte par un caractère spécial \$ pour éviter que certains suffixes se terminent sur des nœuds de l'arbre (Figure 2.9).

En algorithmique du texte, l'arbre des suffixes est une des structures les plus connues et fréquemment employée. Elle possède plusieurs avantages : sa taille et son temps de construction sont linéaires en taille du texte pour la version compacte [50] [51]. Dans le cadre de la recherche de similarités sur les séquences génomiques, deux logiciels ont été proposés qui utilisent cette structure de données.

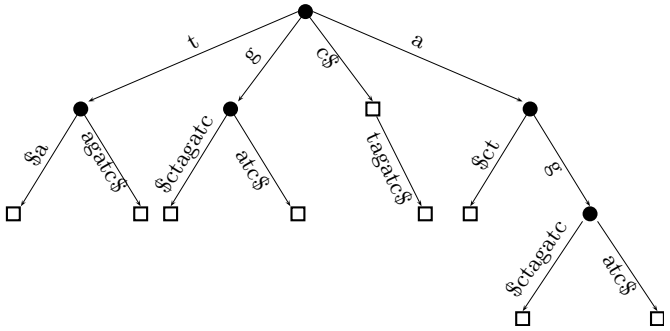


FIGURE 2.9 – Exemple d'arbre des suffixes pour la chaîne **agctagatc**. Les nœuds marqués par des carrés sont des suffixes de la chaîne initiale.

REPuter [52, 53] est un logiciel de recherche de répétitions, basé sur les travaux de Kurtz concernant la réduction de l'espace des arbres des suffixes [51]. Ce logiciel implante entre autres, deux algorithmes nommés MMR (*Maximal Mismatch Repeats*) et MDR (*Maximal Different Repeats*). Étant donnés les entiers l, k, les deux algorithmes recherchent des similarités de taille $\geq l$ ayant au plus k erreurs. Ce logiciel est extrêmement rapide, mais son inconvénient est de ne détecter que des similarités quasi-exactes avec ses paramètres par défaut.

MUMmer [54, 55] est un autre logiciel basé sur l'arbre des suffixes, qui réalise l'alignement global de deux séquences. Utilisant un arbre des suffixes généralisé sur les deux séquences à aligner, il cherche dans le premier temps des MUM (*Maximal Unique Match*) : le terme UM (*Unique Match*) désigne des fragment de texte dont une *unique copie* est présente dans chacune des deux séquences. Parmi ces UM, ceux qui sont *maximaux* sont des MUM.

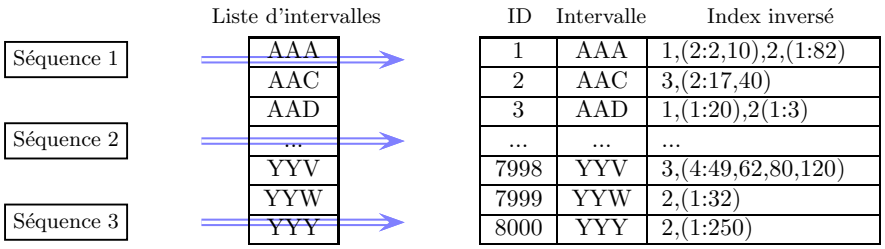
Les MUM sont ensuite classés selon leur position sur la première séquence. Un algorithme de type LIS (*Long Increasing Subsequence*) permet de localiser les MUM ordonnés de manière croissante et contiguë sur les deux séquences afin d'établir un squelette de l'alignement global. Ce squelette est enfin complété par des méthodes de programmation dynamique (comme par exemple l'algorithme de Smith-Waterman).

En fait, la construction de l'arbre des suffixes, sur un texte de longueur N, a besoin au

moins d'un espace de $12N$. Une autre méthode permettant de réduire l'espace mémoire nécessaire aux structures d'indexation est le tableau des suffixes. Cette structure est une version faible de l'arbre des suffixes qui a besoin de beaucoup moins d'espace, de quatre fois la taille d'un texte. Abouelhoda [56] a déclaré que tout algorithme qui utilise l'arbre des suffixes comme structure de données peut être remplacé par un algorithme employant le tableau des suffixes.

2.5.3 Index inversé

L'index inversé est une structure d'index stockant une cartographie des mots à leurs endroits dans un document ou un ensemble de documents. C'est la structure de données qui, pour un mot donné, nous donne directement la liste des documents où il apparaît très rapidement. Dans le cadre de la recherche de similarités sur les séquences génomiques, le logiciel CAFE [57, 58, 59] utilisant cette structure de données a été proposé.



ID	Intervalle	Index inversé
1	AAA	1,(2:2,10),2,(1:82)
2	AAC	3,(2:17,40)
3	AAD	1,(1:20),2(1:3)
...	...	...
7998	YYV	3,(4:49,62,80,120)
7999	YYW	2,(1:32)
8000	YYY	2,(1:250)

FIGURE 2.10 – Exemple de structure de l'index inversé pour trois séquences de protéine. La taille de l'intervalle est de 3. L'intervalle AAA apparaît deux fois dans la séquence 1 (positions 2 et 10) et une fois dans la séquence 2 (position 82)

Williams et Zobel extraient l'ensemble des intervalles chevauchants de taille k (3 pour les protéines et de 8 à 10 pour les nucléotides) à partir des séquences d'une banque et les stockent dans une structure d'index inversé. Il y a deux principaux composants dans l'index inversé : une structure de recherche et les listes de positions. La structure de recherche contient un ensemble d'intervalles, chaque intervalle est associé à une liste de positions. Une liste de positions contient les numéros de séquences contenant l'intervalle associé dans la structure de recherche. La liste de positions stocke également les positions de cet intervalle dans les séquences (Figure 2.10).

Afin de réduire la taille des listes de positions, la technique de compression pour l'index de texte et de chaîne de caractères est utilisée [60, 61]. L'indexation de CAFE a besoin de 2 à 2.5 fois la taille des données d'origines (k = 9 pour les nucléotides et k = 3 pour les protéines). Pour chaque intervalle de requête, CAFE groupe les points d'ancrage sur la même diagonale dans un frame. À partir des frames, les matches ou mismatches entre deux séquences sont déterminés par la fusion des points d'ancrage chevauchants ou l'insertion ou la délétion d'un certain nombre de caractères. Cette méthode est similaire à celle de SSAHA [47].

Pour l'alignement d'ADN [59], les auteurs ont déclaré que CAFE peut être plus de

quatre-vingt fois plus rapide que FASTA et huit fois plus rapide que BLAST. Cependant, CAFE n'utilise pas de programmation dynamique pour optimiser l'alignement local.

Les inconvénients de l'indexation des séquences de la banque de données sont le temps de construction de l'index et l'espace pour stocker cet index. Cependant, Les avantages de l'index est que la recherche passe mieux à l'échelle et est beaucoup plus rapide que les recherches exhaustives.

2.6 Accélérations matérielles

La recherche de similarités est une tâche fondamentale et extrêmement coûteuse en bioinformatiques. Afin de réduire le temps de calcul de cette tâche, de nombreuses équipes se sont tournées vers des solutions d'accélération matérielle, telles que les instructions SIMD, les coprocesseurs ou l'implémentation sur FPGA.

2.6.1 Instructions SIMD

Le SIMD est l'acronyme de Single Instruction Multiple Data (une seule instruction, plusieurs données). Le SIMD désigne un mode de fonctionnement des ordinateurs dotés de plusieurs unités de calcul fonctionnant en parallèle, opposition au SISD (Single Instruction on Single Data). Les instructions SIMD ont été ajoutées aux processeurs modernes pour pouvoir améliorer la vitesse de traitement. Les instructions SIMD sont composées notamment des jeux d'instructions : MMX, SSE, SSE2 et SSE3 chez Intel, 3DNow chez AMD et etc.

En biologie computationnelle, plusieurs implémentations de l'algorithme de Smith-Waterman ou semi heuristique présentées précédemment utilisent des instructions SIMD, ce qui réduit du temps de calcul.

Une des premières implémentations de l'algorithme de Smith-Waterman est proposée par Alpern et al. [62]. Cette approche consiste à diviser le registre de 64-bit Z-buffer du processeur Paragon i860 d'Intel en quatre parties. Chaque partie du registre contient une séquence différente. Un facteur d'accélération de plus de trois a été obtenu par rapport à l'implémentation d'origine. Wozniak [63] présente une autre implémentation qui utilise l'ensemble des instructions du SIMD virtuel du processeur de Sun UltraSparc. Dans cette approche, quatre opérations sont réalisées simultanément dans l'anti-diagonale. L'avantage de cette approche est l'absence de branches conditionnelles dans la boucle inférieure. Un facteur d'accélération plus de deux a été obtenu.

Rognes [9] et Farrar [10] utilisent les instructions SSE (extension du SIMD) du microprocesseur Pentium d'Intel pour calculer en parallèle la matrice de programmation dynamique de l'algorithme de Smith-Waterman. Leurs implémentations sont respectivement 6 et 13 fois plus rapide que l'implémentation SSEARCH [64], ce qui est une version standard de l'algorithme de Smith-Waterman. Leur optimisation principale est d'utiliser le profil d'une séquence requête [9]. Le profil est une matrice de substitution spécifique pour une séquence requête et calculé une fois pour toutes les séquences d'une banque. Rognes et Farrar stockent chaque score sur un entier de 8 bits. En raison de la limitation de la précision sur la valeur d'un octet, les instructions du SSE considèrent

seulement les entiers non signés. Afin d'éviter les valeurs négatives, on ajoute un offset basé sur la plus petite valeur dans la matrice de substitution utilisée. Rognes place les branches conditionnelles dans la boucle inférieure pour calculer la valeur F (l'équation SW dans la section 2.3). Dans l'implémentation de Farrar, les branches sont mises à l'extérieur de la boucle inférieure et sont corrigées si nécessaire.

Rognes a également présenté une approche semi heuristique, ParAlign [65, 66], qui exploite les avantages de la technique du SIMD pour réaliser la recherche de similarités dans une banque de séquences. ParAlign utilise également le profil de la séquence requête pour effectuer simultanément les extensions sans gap pour toutes les diagonales entre deux séquences. Ensuite, une heuristique est employée pour générer le score approximatif d'un alignement avec gap par la fusion des scores de plusieurs diagonales. ParAlign est plus sensible que BLAST, mais plus lent.

2.6.2 Coprocesseurs SIMD (GPU et CELL)

GPU

L'acronyme de GPU correspond à *Graphical Processing Unit*. Le GPU est l'élément principal d'une carte graphique. Aujourd'hui, il est plus puissant qu'un CPU lorsqu'il s'agit d'exécuter des traitements d'images sophistiqués.

Actuellement, la facilité de programmation des GPU et l'augmentation de leur précision arithmétique permettent de les utiliser dans des applications non graphiques de calcul haute performance. NVIDIA et AMD [67] ont proposé le concept de "thread processor" au lieu du concept de "stream processor". Ce concept permet aux différents processeurs de communiquer plus facilement entre eux. De plus, le CUDA [14] (le langage de programmation des cartes NVIDIA) et le Brook+ [68] (le langage de programmation des cartes AMD) représentent comme les langages de programmation standard, extension du C ou C++.

Les cartes graphiques de NVIDIA intègrent un GPU composé de nombreux multi-processeurs, chaque multi-processeur contenant de nombreux processeurs. Un bloc de diagramme de GPU est présenté dans la figure 2.11. Les processeurs accèdent tous à une mémoire partagée et une mémoire cache de texture.

Les GPU de famille Radeon de AMD reposent sur 4 gros blocs d'unités de calcul que nous appelons multi-processeurs pour les GPU de NVIDIA. Chaque bloc dispose de son propre scheduler, d'un gros fichier de registres généraux, de nombreux thread processeurs et d'une mémoire cache de texture. Chaque thread processeur contient cinq stream cœurs. Tous les threads processeurs dans un bloc exécutent le même programme.

De nombreuses implémentations de la programmation dynamique [27] sur le GPU ont été proposées :

W. Lui et al [69] et Y. Liu et al. [70] ont proposé les deux premières implémentations de l'algorithme de Smith-Waterman sur les processeurs graphiques. L'objectif de ces implémentations est d'utiliser les ressources du GPU pour accélérer le scan systématique de la banque de séquences. Les auteurs ont formulé l'algorithme de Smith-Waterman et la structure de données. L'idée de base est de calculer en parallèle la matrice de

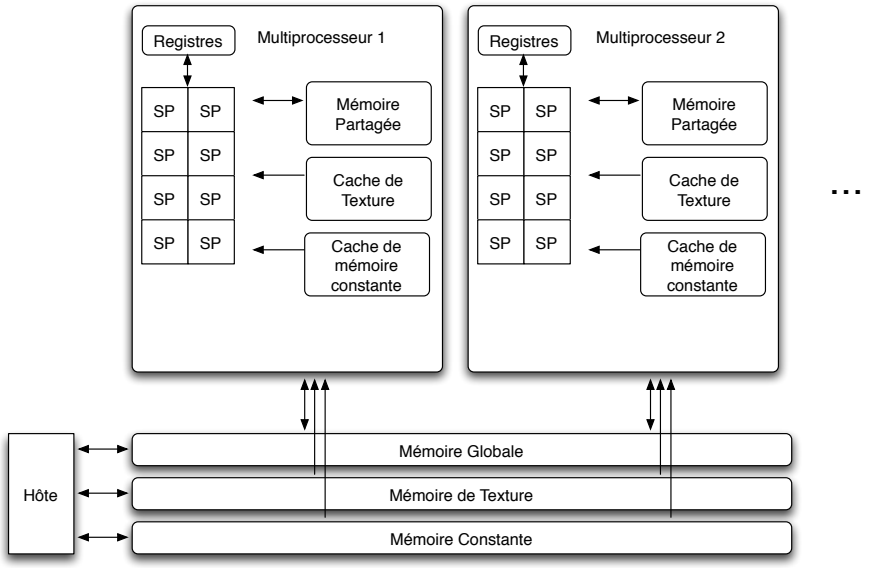


FIGURE 2.11 – Architecteur du GPU présent sur la carte GeForce 8800 GTX de NVIDIA.

programmation dynamique sur les anti-diagonales. Les résultats sont stockés dans la mémoire de texture. La séquence requête et les séquences de la banque de données sont également mises dans la mémoire de texture. Les auteurs utilisent le langage OpenGL. Les deux implémentations ont obtenu un facteur d'accélération de 2 à 4 par rapport à SSEARCH [64].

Plus récemment, Manavski et al. [11] ont parallélisé l'algorithme de Smith-Waterman sur la carte graphique GeForce 8800 de Nvidia en utilisant le CUDA. Dans cette implémentation, les auteurs utilisent le profil de la séquence requête, idée à l'origine de Rognes [9], et le stockent dans la mémoire de texture. Chaque thread de GPU calcule un alignement entre la séquence requête et une séquence de la banque. Cependant, comme les threads sont groupés dans une grille composée d'un ensemble de blocs et déroulent le même programme (*kernel*), les données traitées doivent être de la même taille. De ce fait, les séquences de la banque sont classées au préalable en fonction de leur taille. Cette implémentation obtient un facteur d'accélération de 2 à 15 par rapport à SSEARCH [64] et est équivalente à l'implémentation basée sur les instructions SSE de Farrar [10].

CELL

La puce CELL Broadbord Engine [71] se compose deux types de processeur : un cœur principal, dit PowerPC Processor Element ou PPE, et huit cœurs spécifiques, dit Synergystic Processor Element ou SPE. Le Bus d'interconnexion, dit Element Interconnection Bus ou EIB, établit la connexion entre le PPE, les SPE, la mémoire principale et les périphériques d'entrée-sortie. Un bloc-diagramme de CELL BE est présenté dans

la figure 2.12. Le PPE reste relativement proche d'un cœur classique. Un SPE contient une unité de calcul vectoriel dit Streaming Processor Unit ou SPU et une mémoire locale de 256 Ko. Le SPU est un processeur de RISC avec 128 registres SIMD de 128-bit. La mémoire locale est utilisée pour stocker des instructions et des données d'un programme de SPU.

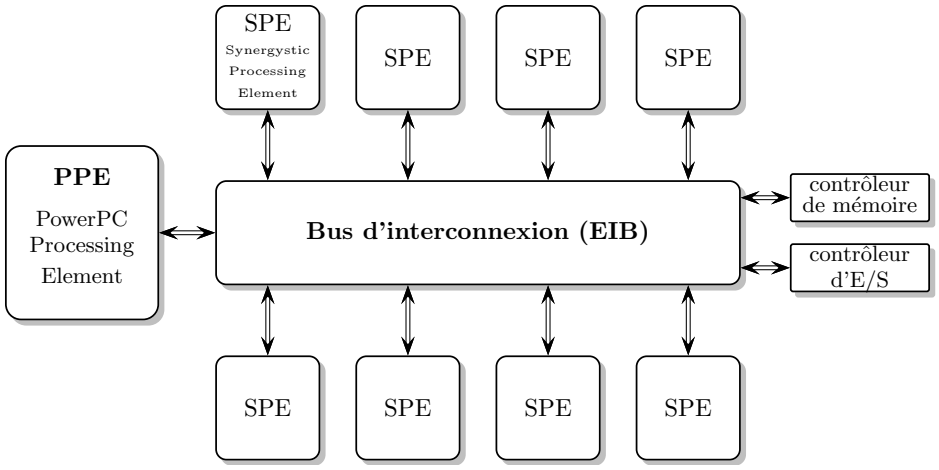


FIGURE 2.12 – Architecture du CELL.

L'algorithme de Smith-Waterman a également été implémenté sur le CELL [71, 72]. Une des premières implémentations de cet algorithme a été proposée par Sachdeva et al. [73]. Sur chaque SPE, on calcule la matrice de programmation dynamique entre la séquence requête et une séquence de la banque de données. De ce fait, 8 paires de séquence sont traitées en simultané. Cette implémentation nécessite une taille limite de deux séquences de moins de 2048 caractères.

Farrar [74] et Wirawan et al. [75] proposent une autre approche pour accélérer l'algorithme de Smith-Waterman sur le processeur CELL. Ils utilisent les registres de 128-bit des SPEs et le profil de la séquence requête. La charge de calcul - le nombre de séquences d'une banque - est connue à l'avance et est répartie de manière égale sur chaque SPE. Ces deux implémentations imposent une taille maximale de 800 caractères pour la séquence requête et obtiennent des résultats 1,6 fois plus rapide que la plus rapide que l'implémentation à base d'instructions SSE décrite dans [10].

Zhang et al. [76] proposent une implémentation du programme FSA-BLASTP [77], une variation de BLASTP, sur le CELL. Dans un premier temps, le PPE génère un DFA (Deterministic Finit-state Automaton), et le transmet à tous les SPEs. Dans le scénario le plus simple, le PPE et les SPEs scannent systématiquement la banque de séquences et traitent les trois étapes du programme FSA-BLASTP. Selon les auteurs, cette implémentation obtient un facteur d'accélération de 1.5 à 3 par rapport au programme FSA-BLAST.

2.6.3 Cluster de machines

Étant donné la croissance des banques de séquences biologiques, il est impossible de les stocker de façon permanente dans la mémoire principale d'une seule machine. Dans le cadre de la recherche d'une banque avec quelques séquences requêtes, les performances de BLAST [1, 2] peuvent décroître parce que chaque séquence requête a besoin de recharger la banque dans la mémoire principale. Pour améliorer les performances, la solution est d'exploiter la grande quantité de mémoire totale disponible dans un cluster de machines.

Dans cette approche, la banque de séquences est partitionnée et répartie dans chaque nœud contenant une partie de la banque. Les séquences requêtes sont envoyées et traitées indépendamment par chaque nœud. Les résultats partiels sont fusionnés ensuite entre les nœuds pour chaque séquence requête. La distribution des banques a été proposée par Pedretti et al. [78]. Elle a été implémentée par Darling et al. dans mpiBLAST [15]. L'algorithme de mpiBLAST se détaille dans le chapitre 6.

Les avantages de mpiBLAST sont : (1) une parallélisation efficace : chaque nœud fonctionne indépendamment et n'a pas besoin d'une communication intensive avec les autres nœuds ; (2) une bonne extensibilité : le calcul peut être déployé sur des clusters de grande taille ou sur une grille.

2.6.4 FPGA

Un FPGA (Field Programmable Gate Arrays) est un matériel qui contient de nombreuses cellules logiques librement reconfigurables. Les circuits reconfigurables FPGA sont aujourd'hui utilisés dans un large éventail d'applications comme les recherches de similarités sur des grandes banques de données. Lorsqu'un FPGA est configuré, le circuit est un intermédiaire entre les microprocesseurs dont le circuit est figé pour créer une mise en œuvre matérielle de l'application.

Un FPGA est un tableau de cellules, les tables de scrutation (*Look-Up Tables, LUT*) avec une interconnexion flexible entourée de blocs d'entrée sortie (Figure 2.13). Les signaux sont acheminés dans la matrice FPGA par des matrices de connexion programmables et les chemins des câbles. Récemment, les FPGAs sont apparus comme des accélérateurs prometteurs en biologie computationnelle. De nombreuses implémentations de la programmation dynamique ou des heuristiques sur le FPGA ont été proposées.

Programmation dynamique

Dans les implémentations de l'algorithme de Smith-Waterman sur processeur, plus de 98,6 % du temps de traitement est du calcul de la matrice de programmation dynamique [79]. Plusieurs approches proposent d'accélérer cette étape sur FPGA. Il y existe trois principaux types d'implémentation : les instructions personnalisées (custom), la reconfiguration dynamique et les réseaux systoliques.

L'implémentations d'instructions personnalisées a été présentée dans les travaux [80, 81]. Les auteurs ont d'abord décrit l'algorithme de Smith-Waterman en version logicielle. Puis ils ont remplacé la portion la plus coûteuse par une instruction personnalisée

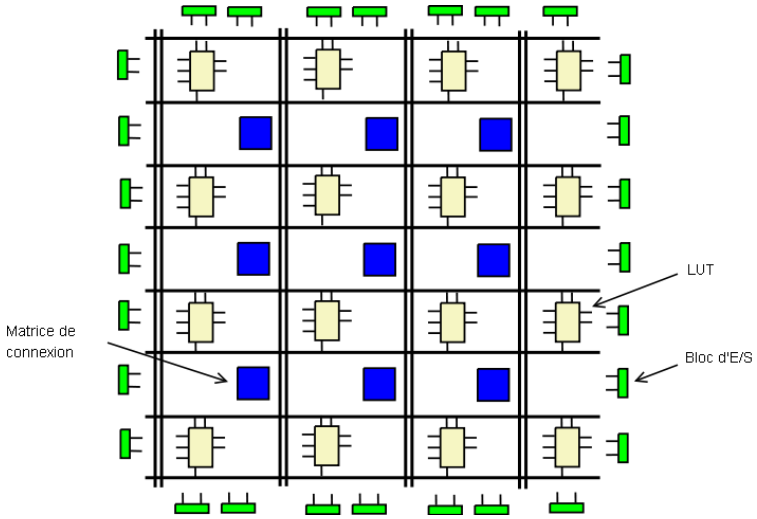


FIGURE 2.13 – Architecture simplifiée de FPGA.

modélisée sur FPGA (écrite en Verilog). L'utilisation de cette instruction personnalisée permet d'obtenir une accélération de 28% par rapport à la version logicielle de base.

La reconfiguration dynamique du FPGA augmente la densité de fonction du FPGA. L'application est divisée en plusieurs opérations indépendantes qui sont implémentées dans des configurations séparées. Pendant l'exécution, les configurations correspondants aux opérations nécessaires sont chargées sur le FPGA. Les travaux [82, 79] utilisent cette approche.

Un réseau systolique est un réseau de cellules similaires ou identiques dans lequel les données circulent de façon synchrone et selon un cheminement fixe. Kung [83] a projeté la matrice de programmation dynamique sur une architecture systolique. Lipton [84] propose une architecture bidirectionnelle dans laquelle les deux séquence se transmettent dans des directions opposées. Des architectures unidirectionnelles ont été proposées par Chown dans [85] et par Hoang [86]. Une séquence est chargée dans le réseau, puis l'autre est introduite. Après $m+n$ cycles, m et n sont respectivement la taille de deux séquences, la matrice de programmation dynamiques est calculée. Plus récemment, Hasan et al. [87] a présenté l'amélioration de performance de l'architecture systolique. Selon les auteurs, un facteur d'accélération de 2 rapport aux autres implémentations a été obtenu.

Heuristique de BLAST

Le programme BLAST étant une référence pour les biologistes, de nombreuse équipes tentent de l'accélérer. Les solutions à base de FPGA semblent très prometteuses pour répondre à cette problématique. Comme nous l'avons vu dans le paragraphe 2.4.2, l'algorithme de BLAST se décompose en trois étapes. Les différentes architectures proposées adressent une ou plusieurs de ces étapes.

Dans l'étape de recherche des points d'ancrage, Mercury BLASTN [88] et Mitrion-

Accelerated BLAST [89] implémentent un pré-filtrage en utilisant un hachage. Ainsi, ils vérifient l'existence d'un mot d'une banque de séquences dans une table de hachage construit à partir d'une séquence requête. RC-BLAST [90] et BEE2 [91] implémentent également la détection des points d'ancrage en effectuant l'indexation de la séquence requête. FPGA/FLASH Accelerator [8] emploie la structure d'index de la banque de séquences et traite les séquences requêtes pour détecter les points d'ancrage. Mercury BLASTP [92, 12] implémente un générateur de deux-graines, basé sur l'index de la séquence requête. NUDT [93] et Two-hits Method [94, 95] implémentent également un générateur de deux-graines, mais basé sur un réseau systolique qui impose une limitation sur la taille de la séquence requête.

Dans l'étape d'extension sans gap, Tree-BLAST [96] combine la recherche des points d'ancrage et l'extension sans gap en réalisant l'alignement sans gap pour toutes les diagonales entre deux séquences. Il utilise une architecture systolique qui impose une limite sur la taille de la séquence requête. Mercury BLASTN [88], Mercury BLASTP [92, 12] et FPGA-FLASH Accelerator [8] implémentent un opérateur sans gap sur une sous-séquence de taille fixée. NUDT [93] implemente un opérateur sans gap sur un réseau systolique. Two-hits Method [94, 95] implémente un bloc de ungappedExtender, qui permet de réaliser l'étape d'extension sans gap dans deux directions avec *X-drop* comme BLAST.

Dans l'étape d'extension avec gap, Mercury BLASTP [92, 12] implémente un opérateur avec gap sur une bande fixée de largeur prédéterminée. Two-hits Method [94, 95] construit un bloc gappedExtender en utilisant l'algorithme de Needleman-Wunsch avec un modèle linéaire de gap.

2.7 Conclusion

De nombreuses techniques cherchent à accélérer les algorithmes de comparaison de séquences génomiques, notamment dans le cadre de recherche dans les banques (une séquence vs un ensemble de séquences). D'un côté, il a été développé d'excellentes heuristiques et des techniques algorithmiques associées à des structures d'index efficaces. De l'autre, les accélérations matérielles se sont focalisées principalement sur les méthodes de programmation dynamique.

Dans cette thèse, nous avons cherché à accélérer la recherche de similarités :

1. dans le cadre d'une comparaison intensive entre banques de séquences ;
2. en proposant une parallélisation des heuristiques à base de graines ;
3. en développant une ossature logicielle capable de supporter plusieurs cibles technologiques parmi lesquelles les multi-cœurs, les jeux d'instructions SSE, les GPU (cartes graphiques), les composants FPGA ou clusters.

La concrétisation de ce travail est le développement du logiciel PLAST qui, comparativement au logiciel de référence dans le domaine, BLAST, accélère très significativement ce traitement.

Chapitre 3

PLAST - Version Multi-cœur SSE

Ce chapitre présente plus particulièrement la version multithreadée et vectorisée de PLAST. Cette version tire partie de l'architecture des processeurs modernes qui incluent sur une même puce plusieurs cœurs, chaque cœur étant capable d'exécuter des instructions de type SIMD : MMX, SSE, SSE2, SSE3 et SSE4 sur les processeurs Intel ; 3DNow sur les processeurs AMD ; Altivec sur les processeurs IBM.

Les performances obtenues, par rapport au logiciel BLAST, la référence du domaine, montrent une accélération de 3 à 6 suivant la taille et la nature des données traitées. En général, plus le volume de données à traiter est important, meilleure est l'accélération. Ceci s'explique par la structure du programme PLAST qui regroupe, dans une première phase, toutes les données qui partagent un traitement commun, d'où un bon usage des mémoires caches de premier niveau et des instructions de type SIMD. De plus, PLAST se comporte extrêmement bien lorsque le nombre de cœur augmente.

Ce chapitre détaille également les résultats d'un point de vue qualitatif. BLAST et PLAST ne produisent pas exactement les mêmes résultats, même s'ils sont tous les deux basés sur des heuristiques de type graine. BLAST trouve certains alignements que PLAST ne trouve pas. Inversement, PLAST détecte des alignements que BLAST ne détecte pas. Les mesures réalisées indiquent que les deux programmes possèdent une sensibilité similaire.

L'article qui supporte ce chapitre a été accepté dans le journal BMC Bioinformatics.

– Van-Hoa Nguyen and Dominique Lavenier. PLAST : parallel local alignment search tool for database comparison. *BMC Bioinformatics 2009 10(329)*.

PLAST: parallel local alignment search tool for database comparison

Van-Hoa Nguyen*[1], Dominique Lavenier*[1,2]

[1]Symbiose team-project, INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France and [2]ENS Cachan Bretagne, Campus de Ker Lann, 35170 Bruz, France

Email: Van-Hoa Nguyen - vhnguyen@irisa.fr; Dominique Lavenier - lavenier@irisa.fr;

*Corresponding author

Abstract

Background: Sequence similarity searching is an important and challenging task in molecular biology and next-generation sequencing should further strengthen the need for faster algorithms to process such vast amounts of data. At the same time, the internal architecture of current microprocessors is tending towards more parallelism, leading to the use of chips with two, four and more cores integrated on the same die. The main purpose of this work was to design an effective algorithm to fit with the parallel capabilities of modern microprocessors.

Results: A parallel algorithm for comparing large genomic banks and targeting middle-range computers has been developed and implemented in PLAST software. The algorithm exploits two key parallel features of existing and future microprocessors: the SIMD programming model (SSE instruction set) and the multithreading concept (multicore). Compared to multithreaded BLAST software, tests performed on an 8-processor server have shown speedup ranging from 3 to 6 with a similar level of accuracy.

Conclusions: A parallel algorithmic approach driven by the knowledge of the internal microprocessor architecture allows significant speedup to be obtained while preserving standard sensitivity for similarity search problems.

Background

Genomic sequence comparison is a central task in computational biology for identifying closely related protein or DNA sequences. Similarities between sequences are commonly used, for instance, to identify functionality of new genes or to annotate new genomes. Algorithms designed to identify such similarities have long been available and still represent an active research domain, since this task remains critical for many bioinformatics studies.

Two avenues of research are generally explored to improve these algorithms, depending on the target application. The first aims to increase sensitivity, while the second seeks to minimize computation time. With next generation sequencing technology, the challenge is not only to develop new algorithms capable of managing large amounts of sequences, but also to imagine new methods for processing this mass of data as quickly as possible [1].

The well-known Smith-Waterman (SW) algorithm, developed in 1981, is one of the first proposals to detect local similarities [2]. It uses a dynamic programming technique and has a quadratic complexity with respect to sequence length. A great effort has been made to obtain fast implementation on specialized hardware. Rognes [3] and Farrar [4] exploited the fine-grained parallelism of SIMD technology. Their implementations are respectively up to 6 and 13 times faster than the SSEARCH implementation [5]. More recent works use SIMD coprocessors, such as Graphics Processing Units (GPU) [6] or the CELL Broadband Engine [7].

Despite various attempts to accelerate the SW algorithm, its long computation time remains a major drawback. To increase speed, programs based on powerful heuristic methods, such as FASTA [5] or BLAST [8] have been developed. These greatly reduce execution time while maintaining a high level of sensitivity. Again, hardware coprocessors have been proposed to speed up these programs. These mostly use FPGA chips such as the SeqCruncher accelerator [9], the Mercury BLASTP implementation [10], the FPGA/FLASH board [11] or the specific FPGA-based BLAST platforms proposed in [12]. Implementation on the Cell Broadband Engine has also been experimented to make good use of the fine-grained parallelism of the BLASTP program [13].

The PLAST program is a pure software implementation designed to exploit the internal parallel features of modern microprocessors. The sequence comparison algorithm has been structured to group together the most time consuming parts inside small critical sections that have good properties for parallelism. The resulting code is both well-suited for fine-grained (SIMD programming model) and medium-grained parallelization (multithreaded programming model). The first level of parallelism is supported by SSE instructions. The second is exploited with the multicore architecture of the microprocessors.

PLAST has been primarily designed to compare large protein or DNA banks. Unlike BLAST, it is not optimized to perform large database scanning. It is intended more for use in intensive comparison processes such as bioinformatics workflows, for example, to annotate new sequenced genomes. Different versions have been developed based on the BLAST family model: PLASTP for comparing two protein banks, TPLASTN for comparing one protein bank with one translated DNA bank (or genome) and PLASTX for comparing one translated DNA bank with one protein bank. The input format is the well-known FASTA format. No pre-processing (such as formatdb) is required.

Like BLAST, the PLAST algorithm detects alignment using a seed heuristic method, but does so in a slightly different way. Consequently, it does not provide the same alignments, especially when there is little similarity between two sequences: some alignments are found by PLAST and not by BLAST, others are found by BLAST and not by PLAST. Nonetheless, comparable selectivity and sensitivity were measured using ROC curve, coverage versus error plot, and missed alignments.

Compared to BLAST (with its multithreading option activated), a speedup ranging from 3 to 6 can be obtained, depending on the amount and nature of the data to be processed. Furthermore, PLAST provides the best performance when large databases are involved.

Implementation

PLAST implements a three-step, seed-based algorithm: (1) indexing, (2) ungapped extension and (3) gapped extension. An overview of the PLAST algorithm is presented below, followed by a more detailed description of the three steps.

Overview of the PLAST algorithm

Like BLAST, the PLAST algorithm is based on a seed-based heuristic to detect similarities between two protein sequences. This heuristic supposes that two proteins sharing sufficient similarities include at least one identical common word of W amino acids. Then, from this specific word, larger similarities can be found by extending the search on the left and right hand sides. These words are called seeds because they are the starting point of the search alignment procedure.

The first step of the PLAST algorithm is to index the two protein banks using the subset seed concept [14]. Two tables of T entries are constructed, where T is the number of all possible subset seed keys. Each key entry is associated with a list of positions corresponding to all the occurrences of this subset seed in the bank.

The second step computes all the possible seed extensions. For each seed key, the two entries of the two tables are considered and each position of one list is compared with all the positions of the other list. In this context, comparing means computing small ungapped alignments by extending the subset seed on both sides.

The third step computes alignments including the gap penalty. This step is only triggered if the previous step has detected significant local similarity.

Based on these three steps, the principle of the PLAST algorithm can be described (sequentially) as follows:

Algorithm 1

1: $IT_0 \leftarrow$ Index (bank-0)
2: $IT_1 \leftarrow$ Index (bank-1)
3: for all possible seed key k
4: $IL_0 \leftarrow IT_0[k]$
5: $IL_1 \leftarrow IT_1[k]$
6: for all elements i in IL_0
7: for all elements j in IL_1
8: if ungapped_extension $(IL_0[i], IL_1[j])$
9: then gapped_extension $(IL_0[i], IL_1[j])$

Actually, this algorithm has great parallelism potential, since the computations of the 3 `for all` nested loops are independent. Basically, each seed extension can be performed in parallel. Thus, this implementation considers a first level of parallelism, called medium-grained parallelism, which is geared to multicore architectures and based on the multithreaded programming model. P threads corresponding to P available physical cores have the task of computing seed extensions simultaneously. This scheme corresponds to the parallelization of the outer `for all` loop (line 3). The algorithm is split into P+1 threads as given in Algorithm 2.

Algorithm 2 PLAST algorithm

Main thread

1: $IT_0 \leftarrow$ Index (bank-0)
2: $IT_1 \leftarrow$ Index (bank-1)
3: create P extension threads
4: K = 0
5: wait until K >= T
6: merge thread results

P extension threads

1: `while (K<T)`
2: k = K++
3: $IL_0 \leftarrow IT_0[k]$
4: $IL_1 \leftarrow IT_1[k]$
5: for all elements i in IL_0
6: for all elements j in IL_1
7: if ungapped_extension $(IL_0[i], IL_1[j])$
8: then gapped_extension $(IL_0[i], IL_1[j])$

First, the main thread constructs two indexes before creating P extension threads. It sets a shared variable

K to 0 (line 4), representing the key of the first subset seed value, and waits until all subset seed values have been processed. The extension threads increase K (line 2) and compute the extension related to K. The instruction `k=K++` is atomic in order to prevent two threads from having the same `K` value. The last action of the main thread is to merge the results provided by each extension thread.
A second level of parallelism, called fine-grained parallelism, can be found in the two nested `for all` loops (lines 5 and 6, extension threads). Again, each seed extension between all the positions of the two index lists can be carried out simultaneously. Furthermore, this computation is very *regular* in that a score is systematically computed in the seed neighborhood. The value of this score indicates whether the alignments are significant or not. This regular computation is done using the SSE instruction set (Streaming SIMD Extensions) now available on all microprocessors. In this implementation, it allows the processor to calculate 16 scores in parallel.
Each step is now described in more detail.

Step 1: bank indexing

Each protein bank is indexed using the same data structure as that shown in Figure 1. A list is made of all the positions in the protein bank of each seed key. A relative position, computed as the difference between two successive positions, is stored to minimize index size. As a result, the difference can be stored on a short integer (two bytes), rather than as an absolute position on a standard 4-byte integer. For infrequent subset seeds, however, the difference may exceed the dynamic range of short integers (2^{16}). To circumvent this problem, false positive subset seed occurrences are added between two distance positions. The overhead introduced by these extra occurrences increases the size of the list by about 2%.
A subset seed is a word of W characters built by grouping together some amino acids [14]. The following 4-character subset seed structure can be considered as an example:

- character 1: A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y
- character 2: c={C,F,Y,W,M,L,I,V}, g={G,P,A,T,S,N,H,Q,E,D,R,K}
- character 3: A,C,f={F,Y,W},G,i={I,V},m={M,L},n={N,H},P,q={Q,E,D},r={R,K},t={T,S}
- character 4: A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y

Here, the second character of a subset seed is either c or g. For example, the subset seed AcGL represents the words ACGL, AFGL, AYGL, AWGL, AMGL, ALGL, AIGL and AVGL in the 20 amino acid alphabet.

Whereas the BLAST algorithm requires two neighboring seeds of 3 amino acids to start the computation of an alignment, only one subset seed of 4 characters is used here. This offers the advantage of greatly simplifying computation by eliminating data dependencies and making it much more suitable for parallelism. An extension starts as soon as two identical subset seeds are found in two different protein sequences, thereby avoiding the need for any extra computation for managing pairs of seeds. In [14], it is shown that this subset seed structure and the BLAST approach exhibit comparable sensitivity.
PLAST requires $4 \times 20^W + 2.02 \times n$ bytes to index one sequence database, where W is the size of the subset seed being used (usually from 3 to 4) and n is the number of amino acids in the sequence database. To allow comparison of very large databases, PLAST automatically splits them into smaller fragments, which fit with the processor memory. Hence, databases of any size can be processed without further pre-processing.

Step 2: Ungapped extension

As stated earlier, BLAST ungapped extension is run when two close seeds are detected. The extension starts from one seed and extends in both directions. The extension terminates when a running score falls below a threshold value. This technique allows BLAST to limit search space efficiently. As the size of the extension regions can vary from one sequence to another, however, this technique is not suitable for regular computation targeting SSE instructions.
The approach adopted here is different, performing an extension on a predefined size L, both on the left and on the right hand sides of the subset seed. More precisely, for a seed key k in the two index tables, $IL0$ has K_0 elements and $IL1$ has K_1 elements, meaning that $K_0 \times K_1$ extensions must be processed. Thus, two blocks of subsequences $BLK0_k$ and $BLK1_k$ are constructed. Each subsequence is composed of a seed of W characters with its right and left extensions of L characters, as illustrated in Figure 2. Based on this data structure, the ungapped extension procedure between the i^{th} subsequence of $BLK0_k$ and the j^{th} subsequence of $BLK1_k$ is given in Algorithm 3.
The aim of this procedure is to compute a score related to the similarity between two protein subsequences of length $(W + 2 \times L)$. It is split into two phases. The first computes a score by extending the right neighborhood (line 4-7). The maximal value is computed and set as the initial score for the left extension (line 9-12). At the end, the maximal score is compared to a predefined threshold value τ. If it is greater than τ, the couple of subsequences ($BLK0_k[i]$,$BLK1_k[j]$) is a candidate for further processing (gapped extension).

Algorithm 3 ungapped extension procedure

1: $S_0 \leftarrow BLK0_k[i]$
2: $S_1 \leftarrow BLK1_k[j]$
3: $score \leftarrow 0$; $max\_score \leftarrow 0$
4: for $x = 1$ to $W + L$
5: $score \leftarrow score+$ Sub $(S_0[x], S_1[x])$
6: if $score > max\_score$ then $max\_score \leftarrow score$ endif
7: endfor
8: $score \leftarrow max\_score$
9: for $x = W + L + 1$ to $2 \times L + W$
10: $score \leftarrow score+$ Sub $(S_0[x], S_1[x])$
11: if $score > max\_score$ then $max\_score \leftarrow score$ endif
12: endfor
13: if $max\_score \geq \mathcal{T}$ then return true endif
14: return false

Remember that for a specific seed key k, there are $K_0 \times K_1$ extensions to process, and that all extensions can be computed in parallel (no data dependencies between these $K_0 \times K_1$ processes). Hence, SSE instructions can be advantageously used to parallelize this procedure. The idea is to compute N scores in parallel using a SIMD scheme. In this processing mode, a score fits into 1 or 2 bytes and the SIMD register of the microprocessor simultaneously contains N scores. The extension procedure can thus be run in parallel between N subsequences of $BLK0_k$ and one subsequence of $BLK1_k$.

In the implementation considered here, 16 scores are simultaneously computed on a 128-bit-wide register, forcing the score to fit between 0 and 255 (8 bits). As the score is computed on short subsequences, it rarely overflows. However, SSE instructions support saturating arithmetic on 8-bit unsigned values. Thus, if the result of an operation becomes greater than 255, it is automatically adjusted to 255.

The last point that needs to be considered is how to manage negative scores. Owing to the limited precision provided by a single byte value, SSE instructions consider only unsigned 8-bit integers. To avoid negative values, bias calculation is performed based on the smallest value of the scoring matrix. This approach is described in Rognes [3] and Farrar [4]. Figure 3 describes the pseudocode of the ungapped extension procedure for two blocks of subsequences.

Step 3: gapped extension

Ungapped alignments with significant similarities are passed in to this step to extend alignments with gap errors further. A significant amount of time can be spent on this activity, as shown by a BLASTP profiling study in [15], representing up to 30% of the total execution time. Parallelizing this step is also important to minimize overall execution time.

This is achieved as follows: the gapped extension is split into two sub-steps. In the first, *small* gapped extensions are considered. They are constrained by the number of permissible ω gaps with λ extensions to restrict the search space. The search space is also limited to a neighborhood of L amino acids on each side of the subset seed (L=64). Again, if the score exceeds a threshold value, a full gapped extension (second sub-step) is computed using the NCBI-BLAST procedure. In this way, the results are similar to the BLAST output.

The reason for splitting this step into two stages is to make the computation more regular and, in this way, exhibit greater parallelism. The first part consists in computing many small gapped alignments where the search space is identical. The strategy is the same as the *banded Smith Waterman* algorithm strategy in WU-BLAST [16] with the *band length* λ and the *band width* ω. If the score of the left and right extensions exceeds a specified threshold $\mathcal{T}_{sg}$, the second step using the full dynamic programming procedure is launched.

Small gapped extensions are also independent. SSE instructions may therefore be used again to compute a large number of them simultaneously. The ungapped alignments coming from step 2 are stored in a list. When this list contains at least K ungapped alignments, they are processed in SIMD mode.

Unlike ungapped extensions, however, pairs of subsequences are quite similar since a significant similarity has been detected during step 2. In addition, the length of the subsequences is longer (128 amino acids). Consequently, the score is unlikely to fit the range of an 8-bit integer. Thus, in this procedure, only 8 scores are computed in parallel, each score being stored in a 16-bit signed short integer. Figure 4 shows the pseudocode of the small gapped extension procedure.

An important point to be noted is that step 2 can generate many ungapped alignments belonging to the same final alignment, especially when strong similarities occur. In this case, several subset seeds are naturally included in the same alignment. With the approach discussed here, these subset seeds are systematically processed, even if they overlap, leading to high redundancy. To generate only one final alignment, a sorted list of all alignments already computed is stored in memory. Then, before launching a full gapped extension, a check is performed to see whether the small gap alignment to be extended is not included in the final alignment list. This list is common to all the extension threads.

Statistical model

Like BLAST, PLAST uses Karlin-Altschul statistics [17,18] to evaluate the statistical significance of gapped alignments. An E-value is then associated to each alignment and is computed following the BLAST

methodology. Since PLAST manages two banks, one is considered as a list of independent queries (-i option) and the other as the database (-d option). Compositions-based statistic [19] is also available for PLASTP and TPLASTN programs.

Results and Discussion

This section presents the results of the experiments conducted on three versions of the PLAST algorithm for protein comparison: PLASTP, TPLASTN and PLASTX.

Sensitivity and selectivity were first evaluated using the receiver operating characteristic (ROC) and coverage versus errors per query (EPQ). Measurements show that results are comparable to BLAST (release 2.2.18). Execution time was then analyzed on standard multicore processors and also compared to BLAST. A speedup of 3 to 6 was achieved depending on the size and nature of the data.

Receiver operating characteristic evaluation

First, the ROC statistical measure for PLASTP was computed using the method described in [20]. The data set was the SCOP database (release 1.73) with a maximum percentage identity of 40%, downloaded from the ASTRAL SCOP website [21, 22]. This data set includes 7,678 sequences from 1,601 families. The 7,678 SCOP sequences are compared to the data set, and the results of all searches are pooled by E-value. True positives are query-subject pairs in the same family. Self-hits are ignored. For increasing E-value, the ROC score, for n false positives, is defined as:

$$ROC_n = \frac{1}{nT} \sum_{1 \leq i \leq n} t_i$$

T is the total number of true positives in the data set, i is the rank of the false positives, and t_i is the number of true positives ranked ahead of the ith false positive.

The ROC curve was calculated for both PLASTP and BLASTP with the BLOSUM62 scoring matrix and gap penalty of 11-1 and with the BLOSUM50 scoring matrix and gap penalty of 13-2. Also, in both cases, the SEG filtering was disabled. The E-value was set to 10. The ROC curves of PLASTP and BLASTP are compared in Figure 5(A).

For computing the TPLASTN ROC curve, the data set was composed of the yeast (Saccharomyces cerevisiae) genome and a set of 102 proteins [23]. We used 102 proteins as queries against the yeast genome and, again, the results of all searches are pooled by E-value. All alignments were marked as true or false

positives according to a careful human expert annotation [23]. Figure 5(B) shows the TPLASTN ROC curve.

As it can be seen, the PLAST and BLAST ROC curves are very close, but not identical. BLAST performs a little bit well than PLAST when its E-value is set to a high value. Actually, one of the main objectives of PLAST is to be included inside bioinformatics workflows to process large amount of data for automatic analysis, such as genome annotation. In that case, to increase confidence, the E-value is set to a much lower value. For example, setting the E-value to 10^{-3} in the previous ROC analysis provides identical ROC curves between PLASTP and BLASTP (see Additional file 1).

Coverage versus error plot

The coverage versus error plot was also used for evaluating the selectivity of PLAST. Instead of taking all alignments with a fixed E-value threshold, as in the ROC curve analysis, the E-value threshold was varied from 10^{-50} to 10. Then for each threshold value, two parameters were measured: the coverage and errors per query (EPQ). The coverage is the number of true positives divided by the total number of true positives available in the data set. The EPQ is the number of false positives divided by the number of queries.

The same two data sets were used for computing the coverage versus error plot for PLASTP, TPLASTN, BLASTP and TBLASTN. Figure 6 shows performance plots. Again, the plots obtained for the two program families, are very close.

Execution time

In order to evaluate the ability of PLAST to manage large amounts of data, three data sets were made to test the PLASTP, TPLASTN and PLASTX programs, i.e. one data set for each program:

Data set #1: PLASTP

- PROT-GB1-NR contains 2,977,744 protein sequences representing the first volume of the Genbank nonredundant protein database (1,000 Mega aa);
- PROT-SCOP-1K, PROT-SCOP-3K and PROT-SCOP-10K contain respectively 1,000 protein sequences (0.185 Mega aa), 3,000 protein sequences (0.434 Mega aa) and 10,000 protein sequences (1.871 Mega aa) selected from the SCOP database.

Data set #2: TPLASTN

- DNA-HUMAN-CHR1 is human chromosome 1 (NCBI Mar. 2008, 220 Mega nt);

- PROT-GB-1K, PROT-GB-3K and PROT-GB-10K contain respectively 1,000 protein sequences (0.336 Mega aa), 3,000 protein sequences (1.025 Mega aa) and 10,000 protein sequences (3.433 Mega aa) selected from the Genbank nonredundant protein database.

Data set #3: PLASTX

- SWPROT is UniProtKB/Swiss-Protis (Release 56.2, 398,181 protein sequences, 144 Mega aa);
- DNA-GB-1K, DNA-GB-3K and DNA-GB-10K contain respectively 1,000 DNA sequences (1.031 Mega nt), 3,000 DNA sequences (3.172 Mega nt) and 10,000 DNA sequences (10.175 Mega nt) selected from the gbvrl Genbank division.

The hardware platform is a 2.6 GHz Xeon Core 2 Quad processor with 8 GB of RAM running Linux Fedora 7. This platform is thus able to run 8 threads in parallel. The Xeon Core 2 processor has a standard SSE instruction set.

Comparison with BLAST

Each PLAST program was run with its specific data set. For the purpose of comparison, the BLASTP, TBLASTN and BLASTX programs (release 2.2.18) were also run with the same data set, with the multithreading option enabled (-a option). **blastall** was run as follows:

```
blastall -p program of BLAST -m 8 -a number of threads -e E-value
```

Experiments were performed on three runs:

- # threads = 2, E-value = 10^{-3} (Table 1)
- # threads = 8, E-value = 10^{-3} (Table 2)
- # threads = 2, E-value = 10 (Table 3)

In all cases, the BLOSUM62 matrix was used with gap-open penalty and gap-extension penalty set respectively to 11 and 1 (default BLAST parameters). Tables 1 to 3 show the time spent (in seconds) for each run and the speedup of PLAST compared to BLAST.

An E-value of 10^{-3} is a reasonable value when performing intensive sequence comparison. However, setting the E-value to 10 had no significant impact on the execution time.

It can be seen that for each experiment, significant speedup is obtained compared to BLAST. More precisely, the speedup obtained (each measure was performed with an identical number of threads) increased with the size of the data set.

To evaluate PLAST sensitivity for large databases, sets of alignments reported by PLAST and BLAST were compared using two large sets of data: GB1-NR versus PROT-SCOP-1K (PLASTP) and SWPROT versus DNA-GB-1K (PLASTX). Two alignments are considered equivalent if they overlap by more than 70%. An alignment A is included in an alignment B if alignment A belongs to alignment B. A misalignment occurs if an alignment found by one program is not found by the other. The E-value threshold was varied from 10 to 10^{-3}. For each threshold value, the misalignments of PLAST and BLAST were calculated for the two data sets as follows:

- $\mathsf{PLAST}_{miss} = \mathsf{BLAST}_{total}$ - $\mathsf{BLAST}_{include}$ - Identical
- $\mathsf{BLAST}_{miss} = \mathsf{PLAST}_{total}$ - $\mathsf{PLAST}_{include}$ - Identical

where BLAST_{total} and PLAST_{total} are the numbers of alignments found respectively by BLAST and PLAST; $\mathsf{BLAST}_{include}$ and $\mathsf{PLAST}_{include}$ are the numbers of alignments included respectively in PLAST_{total} and BLAST_{total} of BLAST and PLAST; Identical is the number of equivalent alignments between BLAST and PLAST. The results are shown in Tables 4 and 5. See Additional file 2 for results on the 3K and 10K data sets.

The two programs do not find exactly the same alignments. This is due to the difference between the heuristics used to discover the seeds. Nonetheless, for the small E-values generally encountered when using PLAST, the results are very close.

PLAST performance analysis

Table 6 shows the execution time (in seconds) of the three PLAST programs relative to the number of threads and data sets. A first point is that performance increases with the size the data set, whatever the number of threads. This is mainly due to the architecture of the algorithm, which presents great computational locality, especially in step 2 (ungapped extension). This locality favors the use of the memory cache system and minimizes external memory access, which is a slow process compared to the processor internal clock frequency.

A second point is the scalability of the PLAST algorithm when the number of threads increases. Figure 7 depicts speedup as a function of the number of threads. It clearly highlights limitations due to the sequential indexing part of the program as explained by Table 7, which shows the time required for step 1 as a percentage of overall execution time. As stated by Amdahl's law [24], the speedup of a program using multiple processors is limited by the time required for the sequential fraction (P) of the program. The maximum speedup is bounded by $1/(1-P)$. Here, even if the indexing part represents a small fraction of

the execution time, it represents a serious obstacle for the next generation of microprocessors, which will include a great number of cores on the same die.

To measure the benefit of the SSE accelerations, profiling was performed, as shown in Figure 8. The same data set was used. The reference (100%) was the execution time without the use of the SSE instructions. More details can be found in Additional file 3 to compare single-thread and non-SSE execution time between BLAST and PLAST.

It can be seen that the ungapped extension represents a high percentage of computation time and that it can be considerably reduced with the SSE instructions. SSE instructions have a more modest impact on gap extensions.

Conclusion

PLAST primarily focuses on intensive sequence comparison applications, unlike BLAST, which is well optimized for scanning large genomic databases. It has been designed to manage large amounts of data and provides the best performance for such applications.

PLAST is faster than BLAST, while providing comparable sensitivity with the same Karlin-Altschul statistics model. Results are not strictly identical since the heuristics for detecting alignments are different, even if both are based on seed techniques. PLAST integrates a 4-character subset seed approach while BLAST starts an extension when two 3-character seeds are located in a close neighborhood.

BLAST and PLAST do not exactly target the same bioinformatics applications, even if PLAST aims to produce identical results. BLAST performs fast and sensitive scans of genomic databases. To detect low similarities, the user can set a high E-value and then analyse and interpret alignments. In that case, BLAST is better suited than PLAST since sensitivity is a little bit better.

On the other hand, PLAST performs fast bank to bank comparison and results are expected to be piped to further automatic analysis. In this context, the E-value is generally set to a much lower value, leading PLAST to produce similar results compared to BLAST.

PLAST has been designed to target the current and next generations of microprocessors that are – and will remain – parallel machines. Two types of parallelism are taken into consideration: multithreading (targeting multi- and manycore architectures) and SIMD (use of SSE instructions). These two modes of parallelism are combined to obtain maximum performance from the architecture of current and future microprocessors. For instance, the next generation of the new Intel set of SSE instructions, called AVX [25], which extends the SIMD integer registers to 256 bits and 512 bits, will be directly operational

through the PLAST implementation. Similarly, advanced micro architectures, like the Intel Larrabee project [26] or the China Goldson-T manycore project [27], prefigure tomorrow's parallel hardware platforms, where PLAST parallelism will be fully exploited.

Since bank indexing is done on-the-fly, PLAST requires no preformatting processes (such as formatdb) before it can be run. The two banks simply need to be in the widely used FASTA format. On the other hand, PLAST does not print alignments in the default BLAST output format. The main reason for this is that PLAST is not intended for interactive use, but rather as a building block in the primary stages of computational workflows for more advanced bioinformatics studies. Hence, the default PLAST output corresponds to the "-m 8" BLAST option, which simply summarizes the features of all alignments. This format is comprehensive for humans and very easy to handle for computers.

PLAST is a 3-step algorithm where the two most time-consuming steps have been parallelized. On an 8-core architecture, corresponding to a current medium-range platform, good speedup is achieved. For larger configurations with 16 or 32 cores, speedup will be limited by the indexing part which, in the current implementation, is a purely sequential part. The next PLAST challenge is to parallelize this step.

The PLAST family programs are currently focusing on protein sequences. PLASTN is not yet included in the current package. Work is still in progress to achieve an efficient version that takes into account the specifics of DNA sequences, especially for the ungapped step extension.

Availability and requirements

Project name: PLAST

Project home page: http://www.irisa.fr/symbiose/projects/plast

Operating system(s): Linux

Programming language: C

License: CECILL

Restrictions for use by non-academics: none.

Authors' contributions

Both authors contributed to the design of the PLAST algorithm. The C implementation was mostly done by V.H. Nguyen. All authors read and approved the final manuscript.

Acknowledgements

The authors would like to thank A/Prof Laurent Noé from Lille 1 University for all his valuable comments and suggestions to improve this manuscript and for providing various subset seeds.

References

1. Pop M, Salzberg SL: **Bioinformatics challenges of new sequencing technology**. *Trends in Genetics* 2008, **24**(3):142–149.
2. Smith TF, Waterman MS: **Identification of common molecular subsequences**. *Journal of Molecular Biology* 1981, **147**:195–197.
3. Rognes T, Seeberg E: **Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors**. *Bioinformatics* 2000, **16**(2):699–706.
4. Farrar M: **Striped Smith-Waterman speeds database searches six times over other SIMD implementations.** *Bioinformatics* 2007, **23**:156–161.
5. Pearson WR: **Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms**. *Genomics* 1991, **11**:635–650.
6. Liu W, Schmidt B, Voss G, Schroeder A, Muller-Wittig W: **Bio-sequence database scanning on a GPU**. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium* 2006.
7. Sachdeva V, Kistler M, Speight E, Tzeng TK: **Exploring the viability of the Cell Broadband Engine for bioinformatics applications**. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium, CA, USA* 2007:1–8.
8. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: **Basic local alignment search tool.** *J Mol Biol* 1990, **215**(3):403–410.
9. **Timelogic SeqCruncher^{TM} PCIe accelerator card** [http://www.timelogic.com/seqcruncher.html].
10. Jacob A, Lancaster J, Buhler J, Chamberlain RD: **FPGA-accelerated seed generation in Mercury BLASTP**. In *Proceedings of 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* 2007:95–106.
11. Lavenier D, Georges G, Liu X: **A reconfigurable index FLASH memory tailored to seed-based genomic sequence somparison algorithms**. *VLSI Signal Processing* 2007, **48**(3):255–269.
12. Fei X, Yong D, Jinbo X: **FPGA-based accelerators for BLAST families with multi-seeds detection and parallel extension**. In *Proceedings of the 2nd International Conference in Bioinformatics and Biomedical Engineering* 2008:58–62.
13. Zhang H, Schmidt B, Mueller-Wittig W: **Accelerating BLASTP on the Cell Broadband Engines**. In *Proceedings of Pattern Recognition in Bioinformatics, Third IAPR International Conference* 2008:460–470.
14. Roytberg M, Gambin A, Noe L, Lasota S, Furletova E, Szczurek E, Kucherov G: **On subset seeds for protein alignment**. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 2009, **6**(3):483–494.
15. Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ: **Gapped BLAST and PSI-BLAST: a new generation of protein database search programs**. *Nucleic Acids Research* 1997, **25**(17):3389–3402.
16. Chao KM, Pearson WR, Miller WC: **Aligning two sequences within a specified diagonal band**. *Computer Applications in the Biosciences* 1992, **8**(5):481–487.
17. Altschul SF, Gish W: **Local alignment statistics**. *Methods Enzymol* 1996, **266**:460–480.
18. Karlin S, Altschul SF: **Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes**. *Proc Natl Acad Sci USA* 1990, **87**(6):2264–2268.
19. Yu YK, Altschul SF: **The construction of amino acid substitution matrices for the comparison of proteins with non-standard compositions**. *Bioinformatics* 2005, **21**(7):902–911.

20. Yu YK, Gertz EM, Agarwala R, Schaffer AA, Altschul SF: **Retrieval accuracy, statistical significance and compositional similarity in protein sequence database searches**. *Nucleic Acids Research* 2006, **34**(20):5966–5973.

21. **ASTRAL SCOP release 1.73** [http://astral.berkeley.edu/scopseq-1.73.html].

22. Chandonia JM, Hon G, Walker NS, Conte LL, Koehl P, Levitt M, Brenner SE: **The ASTRAL compendium in 2004.** *Nucleic Acids Research* 2004, **32**(Database issue):D189–192.

23. Gertz M, Yu YK, Agarwala R, Schaffer AA, Altschul SF: **Composition-based statistics and translated nucleotide searches: Improving the TBLASTN module of BLAST**. *BMC Biology* 2006.

24. Amdahl GM: **Limits of Expectation**. *International Journal of High Performance Computing Applications* 1988, **2**:88–94.

25. Firasta N, Buxton M, Jinbo P, Nasri K, Kuo S: **Intel ® AVX: new frontiers in performance improvements and energy efficiency** 2008, [http://isdlibrary.intel-dispatch.com/isd/1563/AVX.pdf].

26. **Larrabee microarchitecture** [http://www.intel.com/technology/visual/microarch.htm].

27. Hu W, Zhang FX, Li LS: **Microarchitecture of the Godson-2 processor**. *J. Comput. SAstralAstralci. & Techno.* 2005, **20**:243–249.

Figures

Figure 1 - Bank indexing

	seed	relative offset
size	*not stored*	*2 bytes*
	HCAS	1346
		22318
		18236
	HCAT	5689
		31701
		11021
	HCAW	9870
		4038

Fragment of indexing scheme. For each seed key, a list of relative occurrence positions is stored on short integers.

Figure 2 - Subsequence block

	seed	seed	right neighbor	left neighbor
size	*not stored*	*4 bytes*	*L bytes*	*L bytes*
	HCAS	HLAS	EDSA . . SDKH	HKES . . WIMQ
		HFAS	KCVA . . GIWK	ESAT . . GNYN
		HCAS	LMSA . . SATW	RSWS . . YPNC
	HCAT	HCAT	HFGY . . PMAS	PKHS . . GVHA
		HVAT	HGES . . GDEH	HKSQ . . VNER
		HYAT	GFSE . . YTRM	NCLS . . ASMN
	HCAW	HCAW	ASDE . . CVBN	DHSA . . GNCV
		HLAW	MQID . . GHSE	TSGH . . VNWA

Fragment of subsequence block. For each seed key, a list of subsequences is constructed. Each subsequence contains a seed and its right and left neighborhood.

Figure 3 - pseudocode for *ungapped extension*

```
//Outer loop to process subsequneces in BLK0,
// iterator value = N
for k := 1 to sizeof(BLK0)
  // Initialize scoring profile for N sequences of BLK0
  vProfil := <…>;
  // Compute scores between N subsequences
  // of all subsequences in BLK1

  //Inner loop to process subsequneces in BLK1
  for i := 1 to sizeof(BLK1)
    //Initialize S1 from subsequence BLK1[i];
    S1 :=  BLK1[i];
    // Initialize MaxScore value to zero
    vMaxScore := <0,…,0>;
    // Initialize Score value to zero
    vScore := <0,…,0>;

    //Inner loop to process subsequnece length
    for j :=1 to 2 * L + W
      // Add the scoring profil to vSvore
      vScore := vScore + vProfil[S1[j]][j];
      // Save any vMaxScore values
      // greater than MaxScore
      vMaxScore := max(vMaxScore, vscore);
      // Load the vScore from vMaxscroe before
      // to process the left neigbor
      if(j = W + L)
        vScore :=  vMaxScore;
      endif
    endfor
  endfor
endfor
```

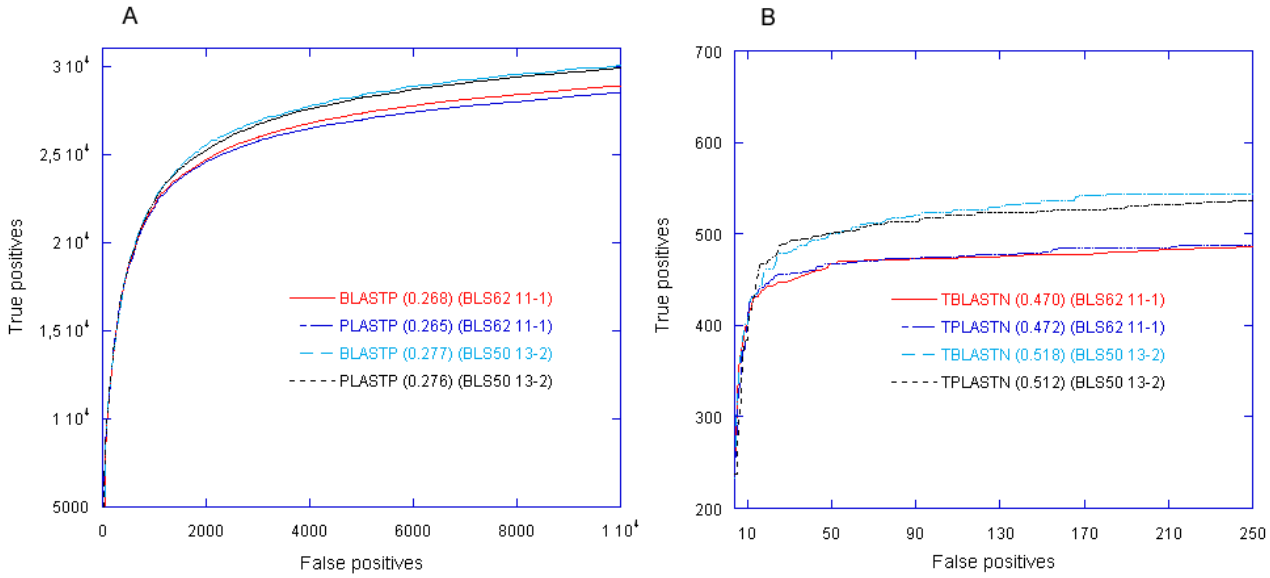
The pseudocode of ungapped extension procedure for 2 blocks of subsequences. Sixteen extensions are simultaneously processed and a score is stored on an 8-bit unsigned byte integer.

Figure 4 - pseudocode for *small gapped extension*

```
// Initialize MaxScore value to zero
vMaxScore = <0,...,0>;
// Initialize first position to zero
first = 1;

// Outer loop to process length λ
for i :=1 to λ
  // Initialize Score and GapRow to zero
  vScore := <INT2_MIN ,..., INT2_MIN>;
  vGapRow := <INT2_MIN ,..., INT2_MIN>;

  // Inner loop to process width ω
  for j : =  first to ω
    // Load GapCol from BestGap[j]
    vGapCol := vBestGap[j];
    // Load sequentially Sub from scoring matrix
    vSub := Sub([S0],[S1]);
    // Save the NextScore from Sub and Best[j]
    vNextScore := vSub + vBest[j];
    // Save any greatest values to Score
    vScore := max(vScore, vGapCol);
    vScore := max(vScore, vGapRow);
    vMaxScore := max (vMaxScore, vScore);

    // Calculate the vGapCol and vGapRow
    // based on the gap penalties
    vGapCol := vGapCol - vGapExtend;
    vGapRow := vGapRow - vGapExtend;
    vGapRow:= max(vScore-vGapOpen,vGapRow);
    vBestGap[j] := max(vScore-vGapOpen,vGapCol);
    // Save the Best value off
    vBest[j] := vScore;
    // Load the next Score value to process
    vScore := vNextScore;
  endfor
  // Increase the first position to process
  if(i> (ω /2)) first++;
  // Correct the Best and BestGap element at
  // position ω  and increase ω value
  if(ω <  λ)
    vBest[ω] := vScoreGRow;
    vBestGap[ω] := vGapRow - vGapOpen;
    ω ++;
  endif
enfor
```

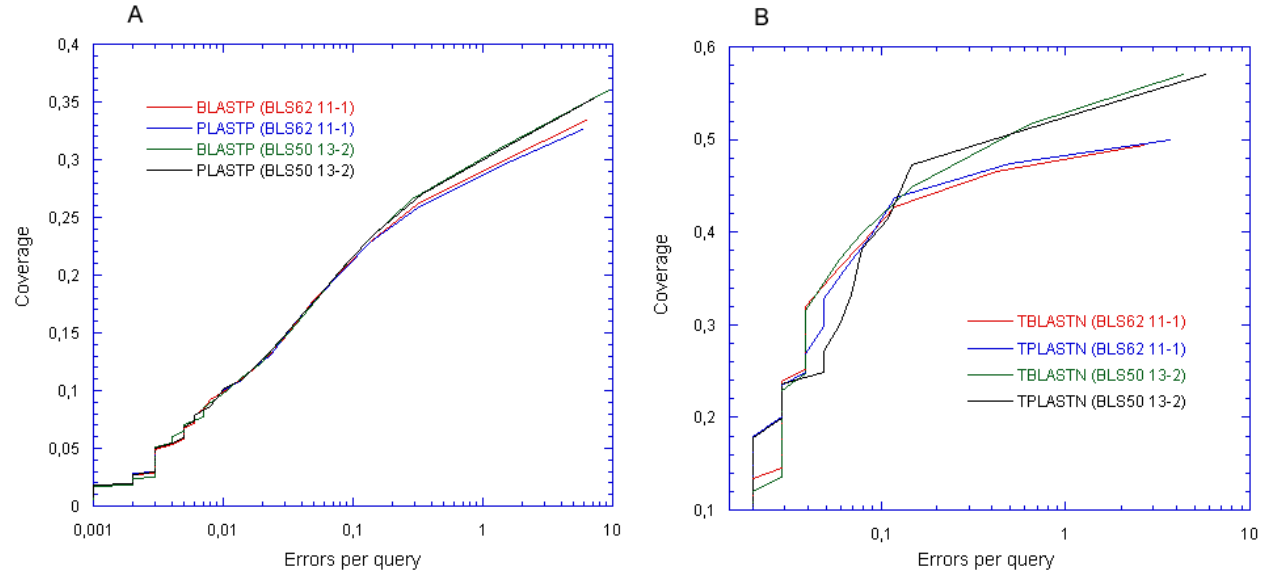
The pseudocode of small gapped extension procedure. Eight extensions are simultaneously processed and a score is stored on a 16-bit signed short integer.

Figure 5 - ROC curve



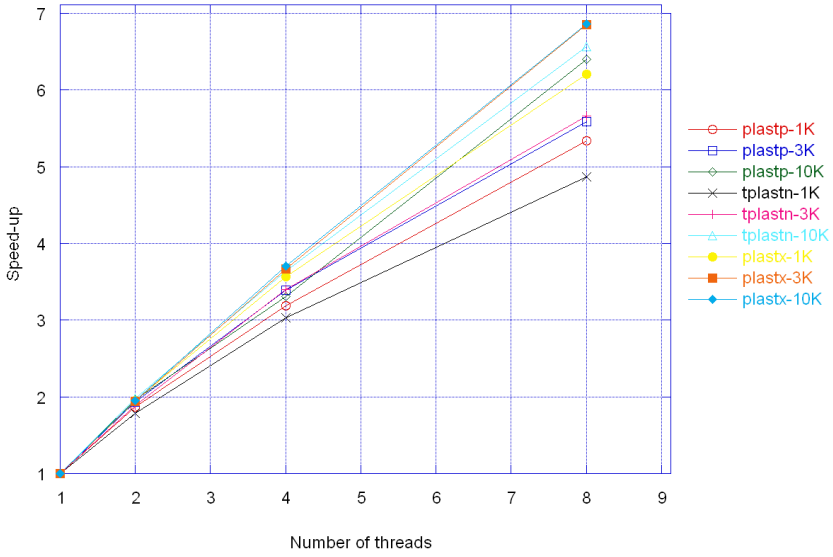
(A) The ROC curves for the SCOP/ASTRAL40 data set of PLASTP and BLASTP. (B) The ROC curves for the Yeast data set of TPLASTN and TBLASTN. The ROC_{10000} score in (A) and ROC_{250} score in (B) for each program are shown in parentheses after the program name.

Figure 6 - Coverage versus error plot



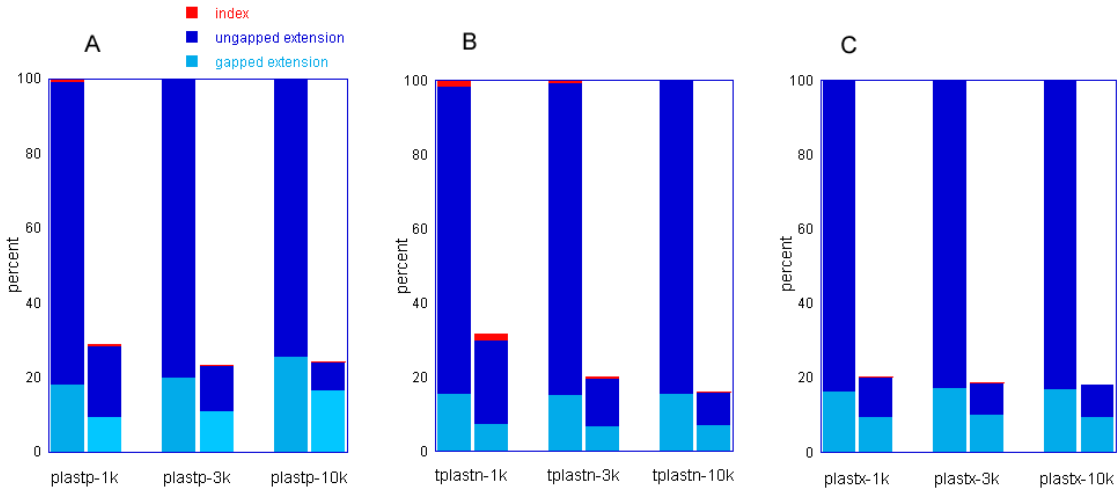
(A) The coverage versus error plots for the SCOP/ASTRAL40 data set of PLASTP and BLASTP. (B) The coverage versus error plots for the Yeast data set of TPLASTN and TBLASTN.

Figure 7 - Speedup of the three PLAST programs



Speedup of the three PLAST programs relative to the number of threads and data sets. The E-value cutoff is set to 10^{-3}

Figure 8 - PLAST profile



The profiles of the three PLAST programs, with and without SSE instructions. Each PLAST program was run with its specific data set: (A) PLASTP; (B) TPLASTN; (C) PLASTX.

Tables

Table 1 - Multicore with 2 threads and E-value equal to 10^{-3}

query	protein vs protein			protein vs DNA			DNA vs protein		
bank	BLASTP	PLASTP	speedup	TBLASTN	TPLASTN	speedup	BLASTX	PLASTX	speedup
1K	4380	1446	3.02	805	319	2.52	2261	554	4.08
3K	10860	2602	4.17	2344	556	4.21	6989	1591	4.39
10K	52131	12415	4.19	7971	1416	5.26	21667	4981	4.34

Comparison of performance of BLAST and PLAST families running with 2 threads. The E-value cutoff is set to 10^{-3} and option "-m8" of BLAST is enabled. BLAST is run in multithread mode (-a 2). Execution times are given in seconds.

Table 2 - Multicore with 8 threads and E-value equal to 10^{-3}

query	protein vs protein			protein vs DNA			DNA vs protein		
bank	BLASTP	PLASTP	speedup	TBLASTN	TPLASTN	speedup	BLASTX	PLASTX	speedup
1K	1530	506	3.02	384	117	3.28	651	174	3.74
3K	4206	898	4.46	1068	186	5.74	1999	451	4.43
10K	21450	3807	5.60	3659	428	8.54	6237	1418	4.39

Comparison of performance of BLAST and PLAST families running with 8 threads. The E-value cutoff is set to 10^{-3} and option "-m8" of BLAST is enabled. BLAST is run in multithread mode (-a 8). Execution times are given in seconds.

Table 3 - Multicore with 2 threads and E-value equal to 10

query	protein vs protein			protein vs DNA			DNA vs protein		
bank	BLASTP	PLASTP	speedup	TBLASTN	TPLASTN	speedup	BLASTX	PLASTX	speedup
1K	4836	1521	3.17	1003	360	2.78	2286	558	4.08
3K	12298	2861	4.29	2881	632	4.55	7010	1631	4.29
10K	58145	14004	4.15	9480	1631	5.81	21774	5002	4.35

Comparison of performance of BLAST and PLAST families running with 2 threads. The E-value cutoff is set to 10 and option "-m8" of BLAST is enabled. BLAST is run in multithread mode (-a 2). Execution times are given in seconds.

Table 4 - Misalignments of PLASTP and BLASTP

E-value	10	1	10^{-1}	10^{-2}	10^{-3}
BLASTP_{total}	556570	507225	462673	423919	394887
PLASTP_{total}	537892	497933	464238	422466	394636
Identical	513096	477982	442854	409437	383746
$\text{BLASTP}_{include}$	11227	9135	7586	6181	5259
$\text{PLASTP}_{include}$	3880	2868	2277	1570	1250
BLASTP_{miss}	20916 (3.9%)	17083 (3.4%)	19271 (4.1%)	9640 (2.2%)	10890 (2.7%)
PLASTP_{miss}	32247 (5.9%)	20108 (4.0%)	12232 (2.6%)	8301 (1.9%)	5882 (1.4%)

Misalignments of PLASTP and BLASTP for GB1-NR versus PROT-SCOP-1K for different E-values.

Table 5 - Misalignments of PLASTX and BLASTX

E-value	10	1	10^{-1}	10^{-2}	10^{-3}
BLASTX_{total}	127124	104474	96559	91760	88127
PLASTX_{total}	123425	101789	96051	90736	87085
Identical	113336	98660	93267	89285	85982
$\text{BLASTX}_{include}$	1694	1240	972	794	655
$\text{PLASTX}_{include}$	1317	823	591	398	268
BLASTX_{miss}	8772 (7.5%)	2306 (2.2%)	2193 (2.2%)	1053 (1.0%)	835 (0.9%)
PLASTX_{miss}	12094 (9.5%)	4574 (4.3%)	2701 (2.8%)	1681 (1.8%)	1490 (1.6%)

Misalignments of PLASTX and BLASTX for SWPROT versus DNA-GB-1K for different E-values.

Table 6 - Execution time of the three PLAST programs

program	PLASTP			TPLASTN			PLASTX		
query bank	1K	3K	10K	1K	3K	10K	1K	3K	10K
1 thread	2704	5024	24374	570	1053	2810	1081	3090	9730
2 threads	1446	2602	12415	319	556	1416	554	1591	4981
4 threads	847	1480	7370	188	310	773	303	842	2620
8 threads	506	898	3807	117	186	428	174	451	1418

Performance of the three PLAST programs running with multithreading mode. The E-value cutoff is set to 10^{-3}. Execution times are given in seconds.

Table 7 - Percentage of indexing time overall in the three PLAST programs

program	PLASTP			TPLASTN			PLASTX		
query bank	1K	3K	10K	1K	3K	10K	1K	3K	10K
Indexing time (1 thread)	3.0%	1.6%	0.3%	6.5%	3.5%	1.4%	1.2%	0.4%	0.1%
Indexing time (8 threads)	15.8%	8.9%	2.1%	31.6%	19.9%	8.9%	7.5%	2.9%	1.0%

Percentage of indexing time overall in the three PLAST programs with number of threads equal to 1 and 8.

Additional Files

Additional file 1

Title: Supplementary ROC curve

Description: The ROC curves for the SCOP/ASTRAL40 data set of PLASTP and BLASTP with E-value of 10^{-3} and ROC curves for the Yeast data set of TPLASTN and TBLASTN with E-value of 1.

File formate: PDF

Additional file 2

Title: Misalignments of PLAST and BLAST

Description: The sensitivity results of BLAST and PLAST for four large sets of data: GB1-NR versus PROT-SCOP-3K, GB1-NR versus PROT-SCOP-10K, SWPROT versus DNA-GB-3K and SWPROT versus DNA-GB-10K.

File formate: PDF

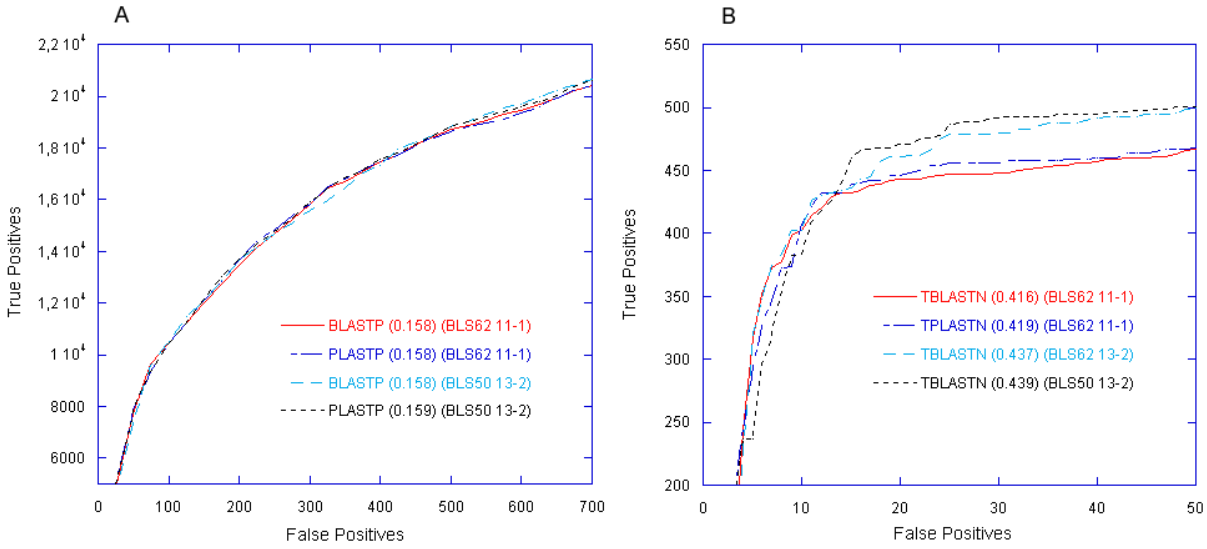
Additional file 3

Title: Single-threaded performance

Description: The comparison of performance of BLAST and PLAST families running with single-threaded and non-SSE.

File formate: PDF

Additional File 1



(A) The ROC curves for the SCOP/ASTRAL40 data set of PLASTP and BLASTP, the E-value was set to 10^{-3}. (B) The ROC curves for the Yeast data set of TPLASTN and TBLASTN, the E-value was set to 1. The ROC_{700} score in (A) and ROC_{50} score in (B) for each program are shown in parentheses after the program name.

Additional File 2

Evalue	10	1	10^{-1}	10^{-2}	10^{-3}
$BLASTP_{total}$	2265188	1957969	1681873	1458895	1311728
$PLASTP_{total}$	2134996	1891731	1682760	1444123	1304392
Identical	2042719	1821964	1599772	1403789	1270949
$BLASTP_{include}$	42583	34500	27492	22156	18369
$PLASTP_{include}$	9674	7009	5467	3787	2990
$BLASTP_{miss}$	82603 (3.8%)	62758 (3.3%)	77521 (4.6%)	36547 (2.5%)	30453 (2.3%)
$PLASTP_{miss}$	179886 (7.9%)	101505 (5.1%)	54609 (3.2%)	32950 (2.2%)	22410 (1.8%)

Table 1: Misalignments of PLASTP and BLASTP for GB1-NR versus PROT-SCOP-3K for different E-values

Evalue	10	1	10^{-1}	10^{-2}	10^{-3}
$BLASTP_{total}$	11614806	10550719	9574744	8747939	8144404
$PLASTP_{total}$	11333685	10434198	9659926	8744933	8162232
Identical	10838876	10037458	9231407	8492530	7944855
$BLASTP_{include}$	217981	180888	148387	120024	100852
$PLASTP_{include}$	65476	51777	42181	31738	25901
$BLASTP_{miss}$	429333 (3.8%)	344963 (3.3%)	386338 (3.9%)	220665 (2.5%)	191476 (2.3%)
$PLASTP_{miss}$	557949 (4.8%)	332373 (3.1%)	194950 (2.0%)	135385 (1.5%)	98697 (1.3%)

Table 2: Misalignments of PLASTP and BLASTP for GB1-NR versus PROT-SCOP-10K for different E-values

Evalue	10	1	10^{-1}	10^{-2}	10^{-3}
$BLASTX_{total}$	371880	306345	282946	268854	258804
$PLASTX_{total}$	363491	299675	282350	266740	257147
Identical	331268	289037	273265	261872	253115
$BLASTX_{included}$	5055	3682	2832	2211	1970
$PLASTX_{included}$	4491	2829	2034	1355	1067
$BLASTX_{miss}$	27732 (7.6%)	7809 (2.6%)	7051 (2.5%)	3513 (1.3%)	2965 (1.1%)
$PLASTX_{miss}$	35557 (9.5%)	13626 (4.4%)	6849 (2.4%)	4771 (1.7%)	3719 (1.4%)

Table 3: Misalignments of PLASTX and BLASTX for SWPROT versus DNA-GB-3K for different E-values

Evalue	10	1	10^{-1}	10^{-2}	10^{-3}
$BLASTX_{total}$	1236542	1017329	940507	893046	856538
$PLASTX_{total}$	1205960	987950	932160	881113	847576
Identical	1100825	956029	904680	866478	835016
$BLASTX_{include}$	22292	16728	13202	10497	8735
$PLASTX_{include}$	14581	9215	6693	4477	3306
$BLASTX_{miss}$	90554 (7.5%)	22706 (2.2%)	20787 (2.2%)	10158 (1.0%)	9254 (1.1%)
$PLASTX_{miss}$	113425 (9.1%)	44572 (4.3%)	22625 (2.4%)	16071 (1.8%)	12787 (1.5%)

Table 4: Misalignments of PLASTX and BLASTX for SWPROT versus DNA-GB-10K for different E-values

Additional File 3

	protein vs protein			protein vs DNA			DNA vs protein		
query bank	BLASTP	PLASTP	speedup	TBLASTN	TPLASTN	speedup	BLASTX	PLASTX	speedup
1K	8151	2704	3.01	1444	573	2.53	4458	1074	4.15
3K	19910	5024	3.96	4256	1053	4.04	13716	3085	4.44
10K	92973	24359	3.81	14380	2810	5.11	42569	9710	4.38

Table 1: Comparison of performance of BLAST and PLAST families running with single-threaded. The E-value cutoff is set to 10^{-3} and option "-m8" of BLAST is enabled. Execution times are given in seconds.

	protein vs protein			protein vs DNA			DNA vs protein		
query bank	BLASTP	PLASTP*	speedup	TBLASTN	TPLASTN*	speedup	BLASTX	PLASTX*	speedup
1K	8151	9737	0.84	1444	1904	0.76	4458	5553	0.80
3K	19910	22161	0.90	4256	5502	0.77	13716	17424	0.79
10K	92973	104124	0.89	14380	18323	0.78	42569	56055	0.76

Table 2: Comparison of performance of BLAST and PLAST (**without SSE instructions**) families running with single-threaded. The E-value cutoff is set to 10^{-3} and option "-m8" of BLAST is enabled. Execution times are given in seconds. (*) represents PLAST without SSE instructions. It's clear that speedup is bought by SSE instructions and that the structure of the algorithm is interesting only if such instructions are available.

Chapitre 4

PLAST - Version FPGA

Ce chapitre détaille la mise en œuvre de PLAST sur une architecture reconfigurable. La plate-forme matérielle est l'accélérateur RASC-100 de SGI couplé à un processeur hôte ALTIX 350 (bi-cœur, Intanium 2). L'accélérateur comporte deux composants FPGA qui peuvent être sollicités indépendamment par les 2 cœurs du processeur. Cet ensemble supporte donc efficacement la mise en œuvre multithreadée de PLAST, où chaque thread pilote un co-processeur FPGA.

Seule la partie extension sans gap a été implémentée sur l'accélérateur car elle représente la grande majorité du temps de calcul. La partie extension avec gap, quant à elle, est parallélisée avec le jeu d'instructions SIMD, comme décrit au chapitre précédent.

La parallélisation du traitement s'effectue sur un réseau de processeurs spécialisés dans la comparaison de courtes chaînes d'acides aminés. Chaque FPGA comporte 192 processeurs 8 bits fonctionnant à 100 MHz. Les résultats produits par tous les processeurs sont filtrés pour ne retourner vers le processeur hôte que les paires de chaînes présentant des scores significatifs.

Les performances mesurées sur TPLASTN indiquent un gain de 20 par rapport au logiciel TBLASTN. Elles illustrent les gains de ce type de matériel sur une application typique de la biologie moléculaire (comparaison d'un protéome avec un génome) par rapport à un logiciel très optimisé.

L'article qui supporte ce chapitre a été présenté lors du workshop RAW 2009 (Reconfigurable Architecture Workshop) qui se tient chaque année en marge de IPDPS.

– Van-Hoa Nguyen, Alexandre Cornu and Dominique Lavenier. Implementing Protein Seed-based Comparison Algorithm on the SGI RASC-100 Platform. *16th Reconfigurable Architecture Workshop*, Rome, Italy, May 2009.

Implementing Protein Seed-Based Comparison Algorithm on the SGI RASC-100 Platform

Van-Hoa Nguyen
IRISA/INRIA
Rennes, France
vhnguyen@irisa.fr

Alexandre Cornu
IRISA/INRIA
Rennes, France
acornu@irisa.fr

Dominique Lavenier
ENS Cachan Bretagne/IRISA
Rennes, France
lavenier@irisa.fr

Abstract

This paper describes a parallel FPGA implementation of a genomic sequence comparison algorithm for finding similarities between a large set of protein sequences and full genomes. Results comparable to the `tblastn` *program from the BLAST family are provided while the computation is improved by a factor 19. The performances are mainly due to the parallelization of a critical code section on the SGI RASC-100 accelerator.*

1. Introduction

Genomic treatments are good candidates for FPGA accelerators since they only need integer computation and small data paths. Data are basically DNA or protein sequences coming from all living organisms from which new knowledge are extracted by comparing intensively their genes and/or genomes.

With the rapid progresses of new biotechnology processes, and especially the next generation sequencing able to generate millions of small DNA sequences in a single run, the bioinformatics discipline is now facing new challenges. The short read sequencing (SRS) technology, for example, opens the door to new possibilities and requires to reconsider basic bioinformatics treatments such as genome and metagenomic annotation, genome resequencing, assembly of closely related species, de novo assembly, etc. [10] [12]. All these domains have in common to manipulate an increasing number of genomic sequences (DNA or protein).

This paper focuses on one specific treatment: the comparison of a large set of protein sequences against full genomes. Typically, it is included in bioinformatics workflows for annotating new sequenced genomes. From a set of known proteins, the aim is to locate in the genome regions having significant similarities. Thus, using the genetic code, the genome is first translated into its 6 possible protein frames. As a result, two large sets of protein sequences need to be compared together.

A well known program to perform this task is the `tblastn` program from the NCBI BLAST family [4]. Actually, there are only a few implementations onto FPGA accelerator targeting specifically this program. We can cite the SeqCruncher PCIe board from TimeLogic which implements the Tera-TBLASTN software [2], the Cube from CLC bioinformatics which accelerates a sensitive version called Smith and Waterman tblastn [3], the FPGA/FLASH prototype from IRISA which combines FLASH memory and reconfigurable computing [9], and a systolic approach from NUDT [6].

Implementing the NCBI BLAST programs onto FPGA for comparing two large sets of sequences is not straightforward for the following reasons:

- the BLAST programs have been first designed for scanning purpose: querying large banks (millions of sequences) with a single sequence;
- the internal BLAST algorithm is fundamentally sequential, even if a multithreaded option is available;
- the BLAST programs are highly optimized. It is thus difficult to find new tricks for improving significantly the existing code.

We propose a slightly different way to find similarities between protein sequences. We use the same heuristics as BLAST programs, but the code is structured differently: it considers two large sets of data to process, and not one sequence versus many sequences. As a result, we can reorganize the computation in such a way that the most time consuming part can be localized on a small critical section.

This critical section has been implemented on the SGI RASC-100 platform and performances have been compared with the NCBI `tblastn` program. Speedup values ranging

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

from 6 to 19 has been observed depending on the size of the protein sets.

The rest of the paper is organized as follows: the next section describes briefly our algorithm. Section 3 presents the genomic operator implemented onto the RASC-100 accelerator. Section 4 details the performances and section 5 concludes the paper.

2 Seed-based algorithm

2.1 Algorithm overview

The algorithm is based on a well known and powerful heuristic to detect similarities between two protein sequences. It supposes that two protein sequences sharing sufficient similarities include at least one identical common word of W amino acids. Thus, instead of systematically scanning all sequences, as it is done in the dynamic programming method, only sequences (or part of sequences) with common words are considered. From this common word, extensions on both sides are performed to find larger similarity. These words are called *seeds* since they are the starting point to find regions of interest.

To be efficient, this method supposes first to index the sequences according to words of W characters (amino acids). This index is then used to locate potential seeds from which extensions are performed. Our algorithm follows this idea. More precisely, it is split into 3 distinct steps:

- **step 1:** indexing
- **step 2:** ungapped extension
- **step 3:** gap extension

The first step indexes the sequences of the two banks. If the size of the seed is W, then we construct two W^α entry tables $T0$ and $T1$ (one for each bank) with α equal to the alphabet size (20 for protein). The number of entries reflects the number of possible words of W characters. Each entry k of the table points to an index list (IL_k) of sequence offsets where such a word occurs.

The second step corresponds to the following nested loops:

```
for k = 1 to W^α
   for i = 1 to len(IL0_k)
      for j = 1 to len(IL1_k)
         ungapped_extension(IL0_k[i],IL1_k[j])
```

For all entries of tables $T0$ and $T1$, all elements of the two associated index lists $IL0$ and $IL1$ are considered as potential seeds to start a similarity region. If, for an entry k, $IL0_k$ has K_0 elements and $IK1_k$ has K_1 elements, then there are $K_0 \times K_1$ extensions to proceed. At this stage, the extension procedure is a very simple treatment for deciding if it is worth to start a complete – and expensive – computation. It is named *ungapped_extension* because the similarity computation doesn't consider the possibility to lose or include extra characters (gap) in the solution.

The third step is much more complex. The search space is augmented by the possibility to consider gaps. This operation is triggered only if the neighbouring of a seed (computed in the previous step) presents enough similarity.

2.2 Ungapped extension

Table 1 shows the percentage of time spent in the different steps when comparing 30,000 proteins against the Human chromosome 1. It clearly indicates that the ungapped extension step represents the majority of the execution time. Thus, speeding up the whole algorithm must first target this critical section.

step 1	step 2	step 3
0.3%	97 %	2.7 %

Table 1. Percentage of time spent in the different steps of the algorithm

The ungapped extension procedure aims at rapidly computing a raw similarity to decide if the seed neighbouring has a good probability to generate relevant similarity on a larger region. This is simply done by computing a score depending on the similarity between amino acids. More precisely, a maximum score is computed from the substitution costs between pairs of independent amino acids surrounding the seed.

If we consider two substrings `S0` and `S1` of length $2 \times N+W$ composed of a seed of W characters with its left and right extensions of N characters, then the maximum score is computed as follows:

```
score = max_score = 0;
for (k=1; k<=2*N+W; k++) {
   score = max(score,
               score + Sub[S0[k]][S1[k]])
   max_score = max(score,max_score);
}
```

where `Sub[x][y]` is the cost for substituting `x` by `y`, and `S[k]` is the k^{th} character of `S`. The matrix `Sub` is generally determined by genomic considerations based on protein evolution theory, for example, the BLOSUM62 matrix [8].

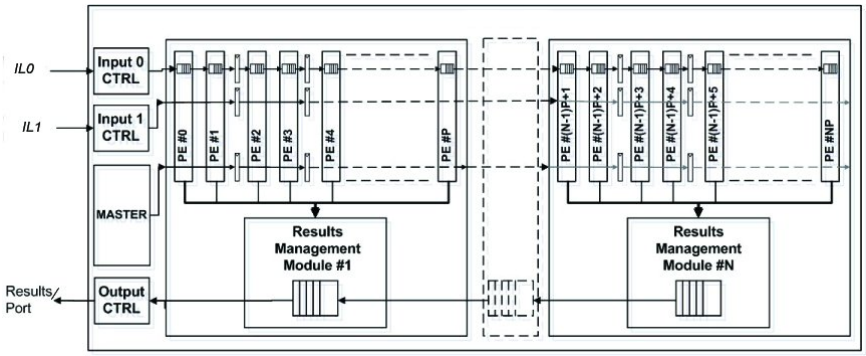


Figure 1. PSC operator architecture

When the score exceeds a given threshold value, the couple (`S0,S1`) is transmitted to the next stage for further processing (gap extension).

The great advantages of this ungapped extension step are:

- the simplicity of the computation: a score is a sum of small integers;
- the locality of the data: including the score calculation, we have a 4 level nested loops, suggesting a strong reuse of data;
- the regularity: all the substrings have the same length.

These interesting features make this step a good candidate for a parallel implementation on a processor array. The next section describes the architecture of a Parallel Sequence Comparison operator (*PSC operator*) dedicated to protein score computation.

3 Architecture and implementation

3.1 PSC operator architecture

Basically, the PSC operator receives two data flows corresponding to the IL_0 and IL_1 index lists (described in the previous section) and output pairs of integers corresponding to the numbers of the 2 sub-sequences presenting strong similarity.

The architecture has been designed to drive a large number of processing elements (PE). These PEs work in a SIMD fashion and are specialized to compute in parallel the score between one sub-sequence from $IL0$ with several sub-sequences from $IL1$.

A pipeline structure has been chosen for optimizing the clock frequency. Indeed, short and parallel data paths – instead of long and shared data paths – imply shorter delays and makes the Place and Route process easier, especially when the design is using a consequent amount of FPGA resources. Thus, slots (or clusters) of several PEs are separated by registers barriers, which delay and reenforce the signals (data and control) as shown below:

In this architecture, the control is independent of the number of PEs. This is a great advantage for, at least, two reasons: (1) validation and test: a single PE can be used first for simulation, software development, etc. Then, gradually, the number of PEs can be increased to the pipeline length and, finally, set to its maximal size; (2) the design can target different array size depending on the available reconfigurable resources.

The PSC operator architecture is divided into 5 main components:

- *Input Controller 0*: it reads sub-sequences from the $IL0$ port and pushes them into the $IL0$ pipeline;
- *Input Controller 1*: it reads sub-sequences from the $IL1$ port and pushes them into the $IL1$ pipeline;
- *PE Slots*: they are groups of PE with a common result management module made of:
  - *Processing Element (PE)*. One PE stores an $IL0$ sub-sequences. Its main task is to compute a score between the $IL0$ sub-sequence and many $IL1$ sub-sequences;
  - *Result Management Module*. This module scans results from a slot of PE and stores them into a FIFO if the computed score is higher than a threshold value. These FIFOs are cascaded to asynchronously transfer the results to the output port.
- *Output Controller*: it reads data from the cascaded FIFOs and writes them to the Result port;
- *Master controller*: it manages the global architecture: process start, data loading, score computation, results recovering, process end.

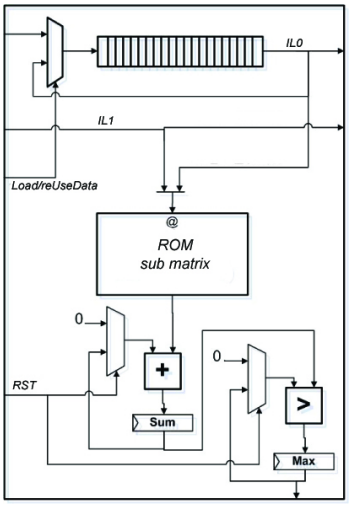


Figure 2. Processing element architecture

3.2 PE architecture

Figure 2 represents the PE architecture. A complete treatment is split into two phases:

- *initialization*. This phase loads an $IL0$ sub-sequence of $W + 2 \times N$ amino acids into the shift register. A feedback loop allows the sub-sequence stored in the shift register to be reused for several computations. The size of the shift register correspond to the size of the sub-sequences ($W + 2 \times N$);

- *computation*. During the computation process, the $IL0$ sub-sequence is sequentially sent, amino acid by amino acid, to the score computation unit together with amino acids coming from the $IL1$ data path.

A computation is performed in $W + 2 \times N$ clock cycles. On each clock cycle, a PE processes one amino acid coming from the $IL0$ sub-sequences and one amino acid coming from the $IL1$ sub-sequence. They are first sent a ROM which output the substitution cost of these 2 amino acids. The result is added to the current score and a maximum value is computed.

After $W+2\times N$ cycles, the maximum is sent to the result management module. It is thus compared with a threshold value. If it is larger, it is sent to the cascaded FIFOs.

3.3 RASC-100 architecture

The PSC operator has been implemented on the RASC-100 (Reconfigurable Application-Specific Computing) accelerator from SGI. This reconfigurable platform is interconnected to the host system through a NUMAlink bus. The RASC-100 is made of two Xilinx Virtex-4 FPGA components, two TIO modules (for connecting the FPGA components to the Altix system), a SRAM memory and a loader module for initializing the FPGA with new bitstreams. In addition, SGI provides a user-configurable interface (SGI Core) for managing DMA transfer, memory access and user registers (Algorithm Defined Registers : ADR).

Figure 3 details the RASC-100 architecture and the way the PSC operator has been integrated in this environment.

4 Performances

We implement our algorithm on the Altix 350 platform composed of an Intel Itanium2 Core2 (1.6 GHz) with 1 MB cache L2, 4 GB RAM, and running SUSE Linux. Steps 1 and 3 are performed on the Altix 350 while step 2 is deported on the RASC-100 accelerator.

Computation time is compared with the NCBI `tblastn` program (Version: 2.2.18) run on the Altix 350 with an E-value set to 10^{-3}, which is a recommended value in the context of intensive sequence comparison. The other parameters are set to their default values. The execution time is calculated using the Linux command, `time`. The following data set has been considered:

- the Human chromosome 1 (220×10^6 nucleotides) translated into its 6 reading frames (NCBI Mar. 2008);

- 4 protein banks selected from the non-redundant protein data bank (NCBI Aug. 2008) including respectively 1,000, 3,000, 10,000, and 30,000 proteins and representing respectively 336,232, 1,025,835, 3,433,471 and 10,335,365 amino acids.

4.1 Overall performances

Table 2 gives the execution times when comparing the different protein banks against the Human chromosome 1. It compares the execution times of NCBI BLAST and the RASC implementation with respectively 64, 128 and 192 PEs running at 100 MHz.

Note that this experimentation uses only half of the resources of the ALTIX 350 / RASC-100 platform: only one FPGA is used, and steps 1 and 3 are run sequentially onto one core. To be fair, the NCBI `tblastn` program is also run in a sequential mode (one core). It can be seen that for small protein banks the performance ratio between the RASC implementation and NCBI BLAST ranges from 5 to 10. This is mainly due to the PE array which is not used at its maximal capacity: there are not enough sub-sequences related to one specific seed to feed entirely the array. Another factor is the time for indexing the banks: it remains high compared to the execution time of steps 2 and 3. But,

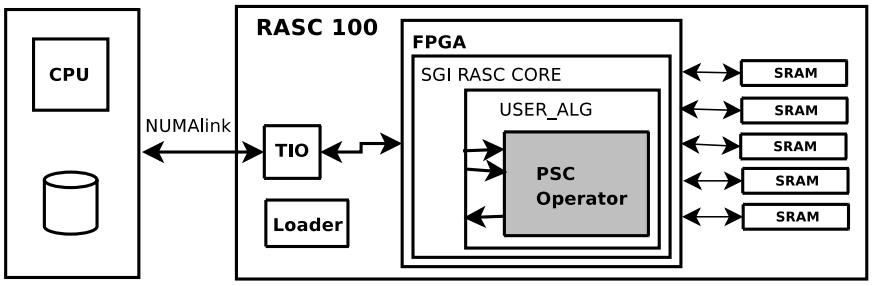


Figure 3. RASC-100 FPGA architecture

	NCBI tblastn	RASC 64 PE	Speedup	RASC 128 PE	Speedup	RASC 192 PE	Speedup
1K protein	2,379	506	4.70	451	5.27	443	5.37
3K protein	7,089	873	8.10	689	10.20	631	11.23
10K protein	24,017	2,220	10.81	1,661	14.45	1,450	16.56
30K protein	70,891	6,031	11.75	4,312	16.44	3,667	19.33

Table 2. Performance comparison of NCBI BLAST and our FPGA implementation. Time is given in second.

as the bank size becomes larger, the ratio becomes much better.

To test the RASC-100 system with its 2 FPGAs, a parallel pthread version using the 2 cores has been developed. Unfortunately, due to communication synchronization problems between the FPGA accelerator board and the Altix 350 platform, we have not been able to perform intensive tests. Some problems are encountered when results are sent back to the host. To overcome these problems, we increase the ungapped threshold value, leading to report much less results. Note that this modification do not reduce the amount of calculation which need to be performed on the accelerator board. It just aims to lighten the traffic between the FPGA board and the host to permit a complete execution of the program. Table 3 indicates the execution time (in second) when using one or two FPGAs in these specific situation, each FPGA implementing 192 PEs. Again, it can be seen that a fairly good speedup (1.8) is obtained when the size of the protein bank increase. This experimentation also shows that a multithreading implementation using two independent processes driving separate FPGA is valid, and that the full computational power of the RASC-100 board can be exploited.

4.2 Step 2 analysis

To understand the parallelization efficiency of the ungapped extension implementation, Table 4 reports the time (in second) and speed up measured with and without the RASC-100 accelerator. It clearly depends of the number of PEs and of the size of the data set: larger the amount of data, better the efficiency.

Protein bank	1K	3K	10K	30K
1 FPGA	168	223	510	1,373
2 FPGAs	148	175	330	759
Speedup	1.14	1.27	1.54	1.80

Table 3. Performance comparison of 1 FPGA and 2 FPGAs for 192 PEs and the 4 protein banks

An interesting point to highlight is that the sequential execution time of step 2 of our implementation is slower than the overall execution time of the NCBI tblastn program (first column of the 2 Tables 2 and 4). But this section of code, which represents a very high percentage of time over the total execution time, has been primarily designed to have an optimal efficiency on a parallel support. Hence, executed on the RASC-100 accelerator, the execution time of step 2 is thus strongly divided, leading to significant speedups over optimized sequential implementation such as the NCBI tblastn software.

4.3 Comparison with other tblastn FPGA implementation

One criteria for comparing different implementations can be the amount of data process per second. In the case of the `tblastn` program it is given by the product of the number of *Kilo Amino Acids (Kaa)* and the number of *Mega nucleotides (Mnt)* divided by the processing time. Based on

	Sequential	RASC 64 PE	Speedup	RASC 128 PE	Speedup	RASC 192 PE	Speedup
1K protein	2,368	220	10.76	176	13.45	169	14.01
3K protein	7,577	462	16.40	280	27.06	223	33.97
10K protein	24,687	1,366	18.07	720	34.28	510	48.38
30K protein	73,492	3,932	18.68	2,015	36.47	1,373	53.52

Table 4. Performance comparison of step 2 only for the 3 different sizes of PE array and the 4 protein banks

DeCypher	CLC	FLASH/FPGA	Systolic	1/2 RASC-100
182	2	451	863	620

Table 5. Number of Kilo amino acids x Mega nucleotides processed par second (*KaaMnt/sec*)

this ratio, Table 5 compares various implementations.

DeCypher engine has been benchmarked [1] for comparing 4289 proteins (1,358,990 aa) against 192 bacterial genomes (775,191,168 aa) in 1 hour and 36 minutes. The next version, called SeqCrunch, provides probably better performance, but no data are available. For CLC, values are extrapolated from [3] where performance are given in GCUPS (Giga Comparison per second) which are similar to our measure. The comparison is strongly biased since the CLC implementation is very sensitive and based on the dynamic programming method. The FLASH/FPGA board, also developed in our team, provides similar results but requires specific hardware not available on the market [9]. The performance of the Systolic approach given in [6] are peak performance measured on a FPGA prototype when the protein sequence length exactly match the size of the array (3072 PE). A standard protein (330 aa) search gives an average ratio of 258. In addition, the performance of Systolic implementation doesn't include gap extension stage.

4.4 Sensitivity and selectivity

Another important issue is the quality of the results produced by the RASC-100 implementation. We use the same seed heuristics as the BLAST algorithm but in a rather slightly different way. In the NCBI BLAST algorithm, the ungapped extension is started when two seeds of 3 amino acids are detected in a closed neighbouring. In our implementation we consider only one seed of 4 amino acids, but based on the subset seed approach [11]. The main reason is that this approach is very efficient for indexing the protein sequences. Theoretically, both approaches have the same sensitivity.

Table 6 reports the receiver operating characteristic (ROC_{50}) and the average precision (*AP-Mean*) scores of both RASC and NCBI BLAST, in which all parameters are set to their default values. The ROC curves and AP-Mean were generated by analyzing the results of aligning 102 queries against the yeast genome in [7]. They correspond to standard procedures to evaluate sensitivity and selectivity of sequence comparison algorithms. Similar values indicate similar sensitivity and selectivity.

	FPGA-RASC	NBCI-BLAST
ROC_{50}	0.468	0.479
AP-Mean	0.447	0.441

Table 6. ROC_{50} and AP-Mean scores of RASC and NCBI BLAST

More precisely, the ROC_{50} value is calculated as follows: each protein sequence is compared with the yeast genome and the first 100 best hits are marked as true or false positives according to a careful human expert annotation. True positives are sequences of the same family. Then, for each of the first 50 false positives, the number of true positives with a higher score is get. These numbers are added and the sum is divided by $50 \times P$, P being the number of sequences of the family. The average of these ROC_{50} scores gives the final ROC_{50} score.

The average precision (AP) criterion is borrowed from information retrieval research as described in [5]. For calculating the average precision, the 50 best alignments per query are marked as either true or false positives. For each true positive found by the comparison algorithm, the true positive rank is divided by its position. All these numbers are summed up and divided by the total number of true positives, giving one AP value per query. The Mean-AP is the average of all the APs.

5 Conclusion

We have proposed an FPGA implementation of the `tblastn` algorithm on the SGI RASC-100 accelerator. Compared to the NCBI BLAST software version, the

Protein bank	1K	3K	10K	30K
step 1	43 %	31 %	14 %	6 %
step 2	38 %	35 %	35 %	37 %
step 3	19 %	34 %	51 %	57 %

Table 7. Percentage of time spent in the different steps of RASC with 192 PEs for 4 protein banks

RASC-100 implementation provides a speed up of 19 for processing large amount of data. This has been achieved by rewriting the code in a way suitable for parallel processing on hardware accelerators.

The heart of the hardware architecture is based on the parallel score calculation between two short amino acid sequences. In that way, this design can be directly reused for implementing `blastp`, `blastx`, and `tblastx` BLAST family programs.

Improving the parallelization of step 2 (*ungapped_extension*) would provide better overall performances, but would be limited by the execution time of step 3 as shown on the code profiling, Table 7, when using the RASC-100 with 192 PEs.

Now, step 3 has the largest execution time. Hence, optimizing global performances implies now to consider a larger array with faster PEs for ungapped extension together with the design of another reconfigurable operator dedicated to the computation of similarities including gap penalty. The RASC-100 architecture would perfectly support this double activity since it allows two different designs to run concurrently on its two FPGAs.

Also, another way to further optimize would be to consider the next processor generation which will include 4, 8 or more cores. As a matter of fact, when such processors will be linked to reconfigurable resources, the question will be how to dispatch the overall computation between cores and FPGA to get optimal performances. The next platforms involved in reconfigurable super computing will have to deal with this matter to find the best compromise and to decide which part of the application will be more suited for reconfigurable implementation.

References

[1] Decypher performance BLAST http://www.timelogic.com/benchmark_blast.html.

[2] Timelogic seqcruncher^{TM} PCIe accelerator card, http://www.timelogic.com/seqcruncher.html.

[3] White paper on CLC bioinformatics cube 1.03, clc bio, 2007, http://www.clcbio.com.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

[5] Z. Chean. Assessing sequence comparison methods with the average precision criterion. *Bioinformatics*, 19:2456–2460, 2003.

[6] X. Fei, D. Yong, and X. Jinbo. Fpga-based accelerators for blast families with multi-seeds detection and parallel extension. In *Bioinformatics and Biomedical Engineering in The 2nd International Conference*, pages 58–62, 2008.

[7] M. Gertz, Y. K. Yu, R. Agarwala, A. Schaffer, and S. Altschul. Composition-based statistics and translated nucleotide searches: Improving the tblastn module of blast. *BMC Biology*, 2006.

[8] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89(22):10915–10919, 1992.

[9] D. Lavenier, G. Georges, and X. Liu. A reconfigurable index flash memory tailored to seed-based genomic sequence somparison algorithms. *VLSI Signal Processing*, 48(3):255–269, 2007.

[10] E. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, 24(3):133–141, 2008.

[11] P. Peterlongo, L. Noe, D. Lavenier, G. Georges, J. Jacques, G. Kucherov, and M. Giraud. Protein similarity search with subset seeds on a dedicated reconfigurable hardware. In *Parallel Bio-Computing (PBC-07)*.

[12] M. Pop and S. L. Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, 24(3):142–149, 2008.

Chapitre 5

PLAST - Comparaison GPU/FPGA

Ce chapitre compare l'implémentation de PLAST sur deux types d'accélérateurs matériels : les cartes graphiques (GPU) et les architectures reconfigurables (FPGA). La mise en oeuvre sur FPGA est celle décrite au chapitre précédent : seule la partie extension sans gap est parallélisée sur FPGA, la partie extension avec gap étant parallélisée à l'aide d'instructions SIMD.

À l'opposée, la mise en œuvre GPU considère les parallélisations des extensions avec et sans gap. Les deux fonctions critiques (UNGAPP et SMALL-GAP) ont été décrites en CUDA et compilées sur des cartes NVIDIA (Tesla C870 et GTX 280). Le multithreading est également activé pour piloter deux cartes graphiques en parallèle.

Les performances mesurées, toujours par rapport au logiciel de référence BLAST (en exécution multithreadée), varient en fonction des tailles des banques et peuvent se résumer par le tableau suivant :

	GPU	FPGA
P1K	5.4	3.2
P3K	8.4	8.6
P10K	10	18.3
P30K	10	28

Ces accélérations ont été obtenues avec le logiciel TPLASTN en comparant une banque croissante de protéines (de 1000 à 30000 séquences) avec le chromosome humain numéro 1. La mise en œuvre FPGA a été optimisée par rapport au chapitre précédent en filtrant plus sévèrement les résultats à l'issue de l'extension sans gap.

L'article support est un chapitre d'un livre intitulé : *Emerging parallel architectures for Bioinformatics applications* publié par *Taylor & Francis/CRC Press* et édité par B. Schmidt, À paraitre fin 2009.

– Dominique Lavenier and Van-Hoa Nguyen. Seed-based Parallel Protein Sequence Comparison Combining Multithreading, GPU and FPGA Technologies. B. Schmidt editor, chapter *8*, Taylor & Francis/CRC Press, 2009.

Chapter 8

Seed-Based Parallel Protein Sequence Comparison Combining Multithreading, GPU and FPGA Technologies

Dominique Lavenier[1], Van-Hoa Nguyen[2]

[1]ENS Cachan / IRISA, France
[2]INRIA / IRISA, France

1. Introduction

Despite the increasing diversity of genomic data now available through many different biotechnologies, DNA or protein sequences remain one of the main materials for bioinformatics studies. The basic treatment performed on these data is often a comparison process for detecting any kind of similarities. Traditionally, given a single request, the scan of large databases aims to report all related sequences. One the other hand, more specific applications, such as genome annotation, for instance, have a large set of sequences (proteome) to compare with a complete genome. In both cases, the heart of the algorithms, from a computational point of view, is the same: the detection of similarities between strings of characters.

For almost two decades, the sizes of the genomic banks have steadily increased, nearly doubling every 18 months. From 1999 to 2009, for example, the size of UniProtKB/TrEMBL database [1] has been multiplied by 43 (release 11, July 1999, 199794 entries). Today it contains 8594382 sequence entries comprising 2774832018 amino acids (release 40.4, 16 June 2009). Practically, during the last 10 years, TrEMB has grown by a factor 1.9 every 18 months. From the DNA size, the situation is similar. In 1999, GenBank [2] (release 111, April 1999) contained 2.56 billions of nucleotides. Today, Genbank (release 171, April 2009) contains 103 billions of nucleotides. Its size has been multiplied by 40.

Furthermore, recent progresses in biotechnologies, and specifically the fast improvements of sequencing machines, have revolutionized the genomic research field [3]. The equivalent (in raw data) of the human genome can now be generated in a single day. Billions of nucleotides spread in millions of very short fragments (25 to 70 nucleotides) are thus available allowing a large spectrum of new large scale applications to be set up: genome re-sequencing, meta-genomic analysis, molecular bar-coding, etc. Bioinformatics treatments related to these new types of data often deals, in their earlier steps, with intensive sequence comparison.

Hence, together, the exponential growth of the databases and the next generation sequencing technology (NGS) make the processing of this avalanche of data a more and more challenging task. This is also currently strengthened by the relative stagnation of the microprocessor clock frequencies which cannot longer help, as it was the case for more than 20

years, to compensate the exponential growth of the genomic data. Today, to keep things on track, the use of parallelism is essential.

As a main task in bioinformatics, the parallelization of genomic sequence comparison algorithms has been widely investigated. When large volumes of data need to be processed, a straightforward way is to split the data into smaller packets and to dispatch the computation on independent processing units. This parallelization scheme fits well with platforms of cluster and is commonly implemented in most bioinformatics research centers. The main advantages of this approach are (1) an efficient parallelization: each node works independently and does not require intensive communication with other nodes; (2) a good scalability: the computation may be deployed on large clusters or targets grid environments.

Nonetheless, other alternatives exist for parallelizing this basic bioinformatics treatment. They rely on the internal potential parallelism of the algorithms. As opposed to the cluster or grid implementations, where each processing unit works independently on different data, the comparison of two sequences – or a group of sequences – is shared between different processing units tightly interconnected. This fine-grained implementation targets specific hardware platforms such as reconfigurable accelerators (FPGA) or graphical processing units (GPU). Their great advantages, compared to cluster machines, are their lower cost and their high performance. A standard computer enhanced, for example, with two recent GPU boards or one medium level FPGA board can then be 10 to 20 times faster.

These two schemes of parallelization (cluster/grid versus GPU/FPGA), however, are not antagonist and can be combined to provide optimal use of computer resources. A few nodes of a general purpose cluster can be advantageously equipped with such accelerators. When intensive comparisons are required, the system automatically assigns these nodes to these specific processes, freeing the rest of the machines for other tasks.

The class of genomic sequence comparison algorithms implemented on these accelerators is mainly related to dynamic programming methods. Two main reasons can be emphasized: (1) algorithms are very time-consuming, yielding a real need for speeding them up; (2) computations are very regular and fit well with highly parallel hardware structures. Description of such parallelization techniques can be found in [4].

Another class of algorithms, designs with a powerful heuristic based on the use of seeds to limit the space search, allows the computation to be drastically reduced, compared to dynamic programming algorithms. The two famous software widely adopted by the scientific community are FASTA [5] and BLAST [6-7]. Unfortunately, very few attempts to parallelize them onto hardware accelerators have been done [8-13]. Again, two reasons can be proposed: (1) these algorithms are very fast, and the pressure to speed them up is lower; (2) computations are not regular and are much better suited to sequential processors than to parallel machines. With the NGS data surge, the first reason will rapidly become obsolete and fast solutions are now needed to parallelize these software at any levels, from transistor to grid! The second reason may represent a serious bottleneck for parallelization: algorithms have been designed for sequential machines and cannot be directly mapped to parallel hardware. They need to be redesigned, at a fine-grained level, to benefit from current technologies such as GPU of FPGA.

This chapter presents a parallel seed-based algorithm for comparing protein banks, and its instantiation into two technologies: GPU boards and reconfigurable accelerators. The algorithm has been thought to express the maximum of parallelism and to be easily speeded up by specific hardware platforms. As opposed to BLAST or FASTA, it does not aim to scan databases. It takes as input two and performs an all-by-all sequence comparison. Speed up from 10 (GPU) to 30 (FPGA) are measured compared to the latest optimized NCBI BLAST version.

The chapter is organized as follows: Section 2 presents the principle of the parallel seed algorithm, called PLAST (Parallel Local Alignment Search Tool). Section 3 details implementations on GPU and FPGA. Section 4 compares performance of both technologies. Section 5 concludes the chapter.

2. Principles of the algorithm

Overview

For detecting similarities, PLAST assumes that two protein sequences sharing sufficient similarities include, at least, one common word of W residues. From these specific words, larger similarities can be computed by extending the search on their left and right hand sides. These words are called *seeds* since they are the starting point of the alignment procedure.

To anchor two sequences with common seeds, the two banks are first indexed into two separate index-tables having exactly the same structure. The number of entries represents the number of all possible seeds (20^W). The content of one specific entry memorizes all the positions where the associated seed appears in the bank. As an example, suppose a bank composed of the two following sequences s1 and s2:

```
s1  = AGGTGCTAGCTCT     s2 = TCTGCATCTGCAT
```

The content of the entry associated to the seed `TGC` will be (s1,4) ; (s2,3) ; (s2,9) since the word `TGC` appears in position 4 in sequence s1 and in positions 3 and 9 in sequence s2.

Taking the same entry of the two index-tables immediately gives the positions where the sequences have a common word (a hit) and, thus, potential local similarity. The next step is then to extend the similarity search in the hit neighborhood. This is done within two distinct phases: the first phase performs a simple extension by only considering substitution errors (ungap extension). A score is calculated regarding the number of matches and mismatches in the immediate neighborhood. If the score exceeds a threshold value, then the second phase is activated. This phase is more complex and considers insertion and deletion errors (gap extension). Again, a score is computed. If it exceeds a threshold value, the alignment is reported as a significant one.

Practically, the PLAST algorithm can be described as follows:

Algorithm 1: PLAST principle

```
 0: GapAlignList = Ø
 1: IndexTable1 = index_bank(Bank1)
 2: IndexTable2 = index_bank(Bank2)
 3: for all possible seed sk
 4:   AAStringList1 = make_string(IndexTable1[sk])
 5:   AAStringList2 = make_string(IndexTable2[sk])
 6:   for all s1 in AAStringList1
 7:     for all s2 in AAStringList2
 8:       UngapAlign = ungap_extension(s1,s2)
 9:       if UngapAlign.score > T1 and UngapAlign not in GapAlignList
10:         then GapAlign = gap_extension(UngapAlign)
11:              if GapAlign.score > T2
12:                then GapAlignList.add_and_sort(GapAlign)
```

Lines 1 and 2 build the two bank indexes. Line 3 iterates on all possible seeds. Then, for each seed, two lists of short amino acid strings are constructed (lines 4, 5). These strings are made from the left and right neighborhoods of the seeds, and have a fixed length. Pairwise extensions of all the elements of the two lists are performed (lines 6, 7, 8). If the ungap alignment resulting from the ungap extension procedure has a score greater than a threshold value (T1) and if it is not included in an alignment already computed (line 9), then the gap procedure is launched. If the score of this new alignment exceeds a new threshold value (T2) then it is added and sorted in the final list of alignments.

The test checking if an ungap alignment is included in the final list of alignments (line 9) is essential: usually, significant alignments contain several anchoring sites from where final alignments can be generated. This test avoids the duplication (and the computation) of gap alignments. To speed up the inclusion search (line 9) the final list of alignments is sorted by their diagonal number (line 12).

Actually, this algorithm has great potentiality for parallelism since the 3 `for all` nested loops are independent. Basically, each seed extension can be performed concurrently. A first medium-grain parallelism, oriented to multicore architecture, is thus to consider a multithreading programming model for the outer `for all` loop (line 3). N Threads can thus be associated to N different seed extensions. The parallel multithreaded version of the algorithm is the following:

Algorithm 2: Parallel scheme

Main Thread

```
0: GapAlignList = Ø
1: IndexTable1 = index_bank(bank1)
2: IndexTable2 = index_bank(bank2)
3: create N extension threads
4: SK = 0
5: wait until SK >= MAX_SK
```

Extension Thread

```
 1: while (SK<MAX_SK)
 2:   sk = SK++
 3:   AAStringList1 = make_string(IndexTable1[sk])
 4:   AAStringList2 = make_string(IndexTable2[sk])
 5:   for all s1 in AAStringList1
 6:     for all s2 in AAStringList2
 7:       UngapAlign = ungap_extension(s1,s2)
 8:       if UngapAlign.score > T1 and UngapAlign not in GapAlignList
 9:         then GapAlign = gap_extension(UngapAlign)
10:              if GapAlign.score > T2
11:                then GapAlignList.add_and_sort(GapAlign)
```

The main thread constructs 2 index tables before creating N threads dedicated to the computation of the alignments. It sets a share variable SK to 0 (line 4) representing the first seed and wait until all the seeds have been processed. The extension threads increment the variable SK and compute the alignments associated to this specific seed. The instruction `sk = SK++` is an atomic operation to avoid 2 threads to get the same SK value.

A second level of parallelism is brought by the two inner `for all` loops (lines 5 and 6). If `i` is the number of elements of IndesxList1 and `j` the number of elements of IndexList2, then there are systematically `i × j` ungap independent extensions to compute. To exploit the regularity of the computation, the lines 5 to 11 can be decomposed as follows:

```
 5:   for all s1 in AAStringList1
 6:     for all s2 in AAStringList2
 7:       UngapAlign = ungap_extension(s1,s2)
 8:       if UngapAlign.score > T1
 9:         then UngapAlignList.add(UngapAlign)
10:   for all x in UngapAlignList
11:     if x not in GapAlignList
12:       then GapAlign = gap_extension(UngapAlign)
13:            if GapAlign.score > T2
14:              then GapAlignList.add_and_sort(GapAlign)
```

The computation is split into 2 distinct parts: lines 5 to 9 compute ungap extensions and store the successful ungap alignments into the ungap alignment list. This list is then scanned for the gap extension procedure (line 10 to 14). For large databases, it appears that most of the computation time is spent in the first part. Hence, the computation performs by these 2 nested loops (lines 5 to 9) can be deported on specific hardware able to support a very high parallelization of this task.

Bank indexing

The bank indexing process consists in modifying raw genomic data structures (sequence of characters) into more complex structures favoring the fast location of hits between sequences. The protein indexing scheme is based on the concept of subset seeds [14-15]. A subset seed is a word of W characters over an extended alphabet: the extra characters represent a specific set of amino acids. Below, a subset seed of size 4 is presented:

- character 1: A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y
- character 2: c={C,F,Y,W,M,L,I,V}, g={G,P,A,T,S,N,H,Q,E,D,R,K}
- character 3: A,C,f={F,Y,W},G, i={I,V}, m={M,L}, n={N,H}, P, q={Q,E,D}, r={R,K}, t={T,S}
- character 4: A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y

As an example, the subset seed AcGL represents the words ACGL, AFGL, AYGL, AWGL, AMGL, ALGL, AIGL and AVGL in the amino acid alphabet. Compared to the BLAST algorithm which requires two neighboring seeds of 3 amino acids to start the computation of an alignment, we use only one subset seed of 4 characters. The great advantage is that the computation is highly simplified by eliminating data dependencies and making it much more suitable for parallelism. An extension immediately starts when two identical subset seeds are found in two different protein sequences, avoiding extra computation for managing couple of seeds. In [16], it is shown that this subset seed structure and the BLAST approach have comparable sensitivity.

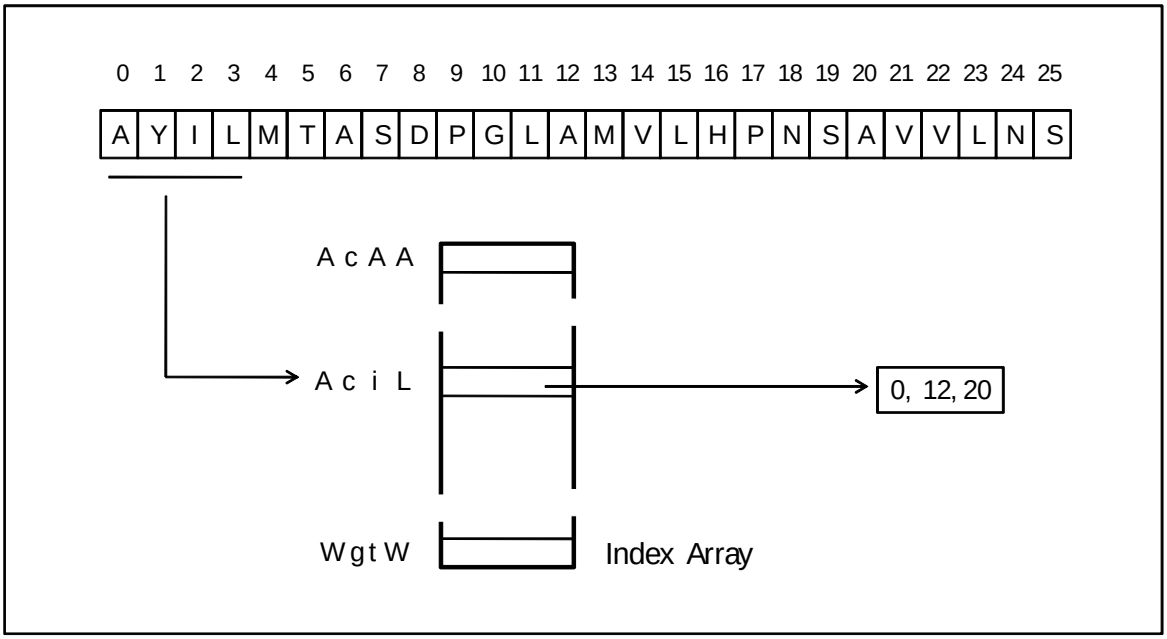


Figure 1: principle of the indexing with subset seeds.

The principle of the bank indexing with subset seed is illustrated Figure 1. Each entry of an index table points to a list of subset seed positions. Each word of 4 amino acids in the bank needs to be translated into its equivalent subset seed. For example, the words AYIL, AMVL and AVVL respectively at position 0, 12 and 20 are translated into the subset seed word AciL. The entry AciL points to a list of integers where such words occur in the bank. Actually, for memory optimization purpose, the positions are encoded in a relative way: only the difference between two consecutive positions is reported, leading to a 16-bit encoding. For comparing a protein bank and a DNA bank, the DNA bank is translated into its six reading frames, and then indexed in the same way.

The advantage of this index structure is that it provides immediately all the hits between two protein sequences. Coming back to Algorithm 1, line 3, it can be seen that building the two index lists is straightforward.

Ungap extension

The ungap extension procedure aims to rapidly check if a hit can give raise to a significant alignment. Thus, starting from the hit, left and right investigations are done to measure the similarity of the close neighborhood. In our approach, the neighborhood is fixed to a predefined length of L1 amino acids in both directions, and the score of an ungap alignment is only computed on this restricted area as follows:

Algorithm 3: ungap extension

```
 1: ungap_extension(s1,s2)
 2:   score = 0
 3:   max_score = 0
 4:   for x = 1 to L1+W
 5:     score = score + SUB(s1[x],s2[x])
 6:     max_score = max(score,max_score)
 7:   score = max_score
 8:   for x = L1+W+1 to 2*L1+W
 9:     score = score + SUB(s1[x],s2[x])
10:     max_score = max(score,max_score)
11:   return score
```

The ungap extension procedure takes as input two strings of amino acids. Their sizes are equal to 2*L1 + W with L1 the length of the neighborhood and W the length of the seed. The first W characters represent the seed, the L1 following ones represent the right neighborhood, and the last L1 characters represent the left neighborhood. Hence, lines 4 to 6 compute the right extension (including the seed) and lines 8 to 10 compute the left extension. After various tests, L1 has been set to 22, (1) for practical implementation issues and (2) because it provides satisfactory results, but the size of the neighborhood could be set to any other values.

This computation is very regular (no if statement) and, consequently, well suited for an implementation on highly parallel hardware.

Gap extension

The gap extension procedure increases the search space by allowing gaps to be included in the final alignment. It is launched only if the previous step detects enough similarity near the hits. Algorithms for computing alignments with gaps are based on dynamic programming techniques, which are time consuming procedures. And, as shown in [17], in some cases, this step may represents up to 30% of the total computation time. Parallelizing this procedure is thus sometime interesting to minimize the overall computation time.

Again, with the objective of making the computation as regular as possible, this step is split into two phases. The first phase, called small gap extension, restricts the search both on a close hit neighborhood and on a specific number of allowed gaps. A dynamic programming algorithm is run, starting from both sides of the hit, but on a limited number of diagonals, as

shown Figure 2.

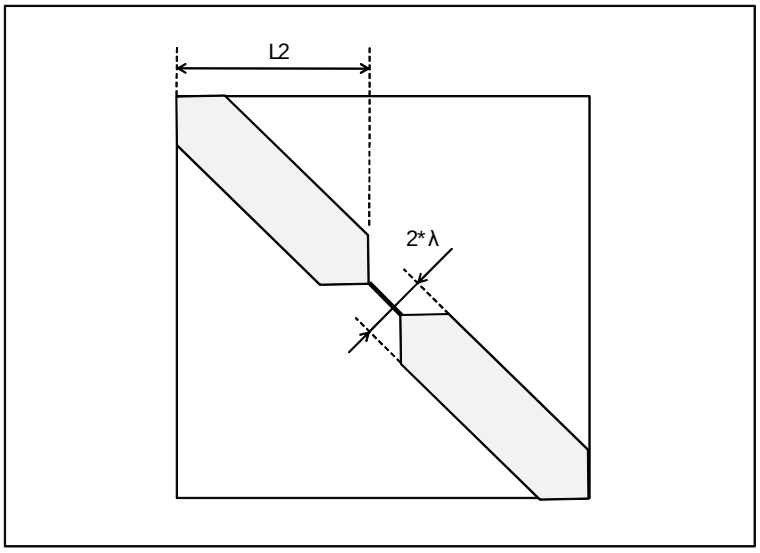


Figure 2: search space for the first phase of the gap extension procedure

L2 is the length of the neighborhood and λ the size of the banded diagonals. The search space is represented by the shadow polygon. If the score of the restricted gap alignment exceeds a threshold value (T3), then a full gap extension (2^{nd} phase) is computed using the standard NCBI-BLAST procedure, leading to similar results with this software.

The main reason to break this step into two phases is that the first step exhibit high potential parallelism. As a matter of fact, the small gap extension can be done concurrently on many different sequences of size W + 2*L2 since each computation requires identical search space.

Again, this phase can be computed in a parallel way.

Generic Hardware Implementation

The implementation of PLAST combines the multithreaded level approach with the fine grained FPGA or GPU parallelization. The extension thread of algorithm 2 is modified as shown below, the main thread staying the same. For a given seed, two lists of amino acid strings are built form the index tables (lines 3 & 4). These two lists are processed by the **`UNGAP`** function which sent back a list of ungap alignments exceeding a threshold value T1 (line 5). The **`UNGAP`** function can be parallelized using two different technologies: FPGA accelerator or Graphic Processing Units.

Elements of the list which are not included in the list of final alignments are put on a temporary list (lines 6 to 8). Actually, this list contains couples of sequences of size W+2*L2 ready for the next step. When the size of this list overcomes its capacity, the first phase of the gap extension is activated. All the ungap alignments inside the temporary list are processed by the **`SMALL_GAP`** function. Again, this function can be parallelized on specific hardware taking as input a large set of sequences and sending back a list of alignments having a score greater than a threshold value T3. These alignments are then extended using the standard NCBI BLAST

procedure.

Algorithm 4: Multithreaded and fine grained parallelism

Extension Thread

```
 0: TmpList = Ø
 1: while (SK<MAX_SK)
 2:   sk = SK++
 3:   AAStringList1 = make_string(IndexTable1[sk])
 4:   AAStringList2 = make_string(IndexTable2[sk])
 5:   UngapAlignList = UNGAP (AAStringList1,AAStringList2,T1)
 6:   for all UngapAlign in UngapAlignList
 7:     if UngapAlign not in GapAlignList
 8:       then add UngapAlign in TmpList
 9:            if size(TmpList) >= N
10:              then SmallGapAlignList = SMALL_GAP (TmpList,T3)
11:                   for SmallGapAlign in SmallGapAlignList
12:                     GapAlign = NCBI_BLAST_ALIGN (SmallGapAlign)
13:                     if GapAlign.score > T2
14:                       then GapAlignList.add_and_sort(GapAlign)
13:                   TmpList = Ø
```

This generic hardware implementation allows the PLAST software to adapt itself regarding the available parallel resources.

3. Parallelization

Whatever the target technology, the **UNGAP** function takes as input two lists of short amino acid strings and detect the couple of sequences having an ungap alignment score above a threshold value. An all-by-all pairwise comparison is done between all sequences of the two lists, as explained above. This function has been parallelized both on GPU and FPGA platforms.

The **SMALL_GAP** function acts as a pre-processing step for computing small gap alignments. It takes as input a list of pairs of sequences and computes a score including gap errors. The search space is however limited by the length of the sequences and by the number of allowed gaps. Parallelization of this function has only been done on GPU.

UNGAP parallelization on GPU

The parallelization of the **UNGAP** function on Graphics Processing Units is an adaptation of the matrix multiplication algorithm proposed in the CUDA documentation [18]. Matrices of numbers are simply replaced by blocks of strings of amino acids. More precisely, for each function call, there are two lists of amino acid sequences to process: List1 and List2. Suppose that block B1[N1, L] and block B2[N2,L] correspond respectively to List1 and List2, with L the length of the amino acid sequences and N1 (N2) the number of sequences in List 1 (List2). The result of the computation is a third block C[N1,N2] which stores the scores of all the computation between block B1 and block B2. In other words, C[i][j] hold the score of the i^{th} sequence of List1 and the j^{th} sequence of List2.

The overall treatment is done by partitioning the computation into blocks of threads computing only a sub block of C, called C_{sub}. Each thread within the block processes one

element of C_{sub} dimensioned as a 16.x16 square matrix. This size has been chosen to optimize the memory accesses, allowing the GPU internal fast memory to store 2 x 16 amino acid sequences which can simultaneously be shared by 256 threads. Figure 4 gives the CUDA kernel code optimized for sequence of length equal to 48 amino acids (`BLOCK_SIZE` = 16).

```
UNGAP_kernel(char* C, char* B1, char* B2, int N1, int N2)
{
  int bx = blockIdx.x;                  // block index
  int by = blockIdx.y;

  int tx = threadIdx.x;                 // thread index
  int ty = threadIdx.y;

  int Begin1 = N1 * BLOCK_SIZE * by;    // B1 index
  Begin1 += N1 * ty + tx;

  int Step1 = BLOCK_SIZE;               // B1 iteration step

  int Begin2 = __mul24(BLOCK_SIZE,bx);  // B2 index
  Begin2 += N2 * ty + tx;
  int Step2 = __mul24(BLOCK_SIZE,N2);   // B2 iteration step

  int Csub = 0;                         // initialize results block
  int CsubMaxi = 0;

  __shared__ int SB1[BLOCK_SIZE][BLOCK_SIZE]; // to store sub-block of B1
  __shared__ int SB2[BLOCK_SIZE][BLOCK_SIZE]; // to store sub-block of B2

  for (int j=0; j<3; j++)
    {
      SB1(ty, tx) = B1[Begin1 + j*Step1];        // load the matrices from
      SB2(ty, tx) = B2[Begin2 + j*Step2];        // device to shared memory

      __syncthreads();                  // make sure the blocks are loaded

      for (int k=0; k<BLOCK_SIZE; k++)  // score computation
        {
          Csub = Csub + texfetch(matrix, SB1(ty, k), SB2(k, tx));
          if(Csub>CsubMaxi) CsubMaxi = Csub;
        }
      __syncthreads();
     }

  int c = Step2 * by + Begin2;         // write the block to global memory,
  C[c] = CsubMaxi;                     // each thread writes one element
}
```

Figure 4: CUDA code for the **UNGAP** function

Each score is computed by first loading the two corresponding 16×16 sub-blocks from global memory to shared memory with one thread loading one element of each block and by having each thread getting one substitution cost. Each thread accumulates this cost to the current score and performs a maximum operation. When it is done, the result is written to the global memory. By doing the computation in such a way, the shared memory is highly solicited, saving a lot of global memory bandwidth since blocks B1 and B2 are read from global memory only 3 times. For a maximal efficiency, the substitution matrix is stored in the texture memory.

Practically, the **UNGAP** function consist in sending to the GPU board two lists of amino acid sequences, and getting back an N1xN2 matrix of scores. A sequential post processing is however required to extract significant scores.

UNGAP parallelization on FPGA

The reconfigurable architecture implementing the computation of the **UNGAP** procedure is a linear array of processing elements (PEs) dedicated to the calculation of a score between two amino acid sequences. If P is the number of PEs, then P scores can be computed simultaneously between one sequence and P sequences. The figure 5 depicts the architecture principle of the accelerator. More details can be found in [19]. It works as follows: If N2 is the number of sequences of List2, then N2/P iterations are required. One iteration loads P sequences into P different PEs in a systolic way. Then all sequences of List1 are broadcasted to all PEs, character by character, every clock cycle. Each time a PE receives a new amino acid, it updates its score. P scores are then available after L cycles (L is the length of the amino acid string). The scores are sent to a Result Management Module which selects the PEs having scores greater than a predefined threshold value (T1). These score are pushed through a FIFO to the output channel.

For efficiency purpose, the array has been split into sub array of fixed size which can be pipelined together. The advantages of this structure are twofold: (1) the architecture can be adapted to many FPGA platforms according to the available reconfigurable resources; (2) the performance of the system only depends of the number of PEs; the frequency remains identical whatever the number of sub arrays.

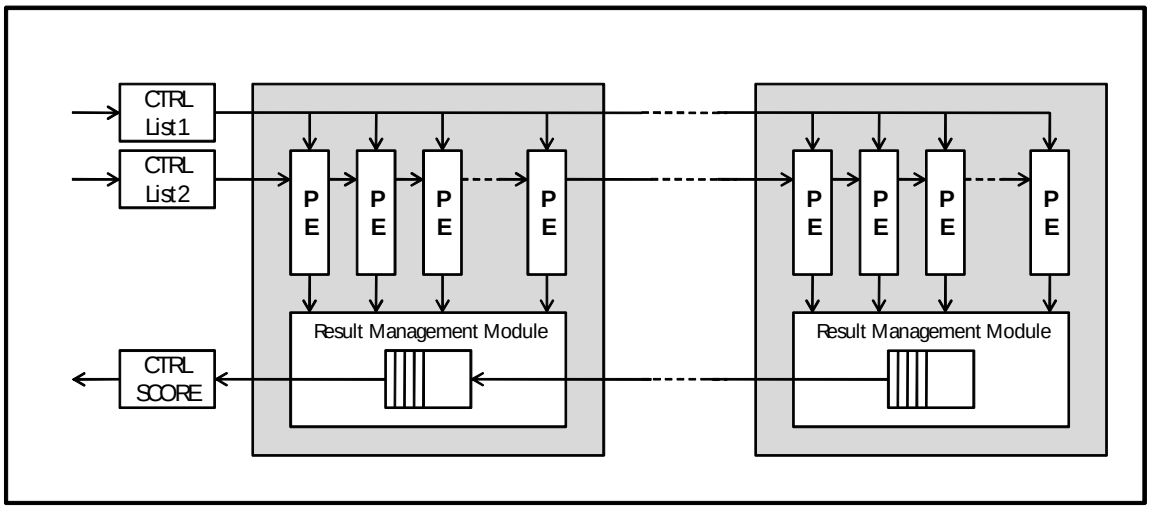


Figure 5: Principle of the FPGA architecture.

Compared to the GPU approaches, the host doesn't need to extract the highest scores. This is done online by the Result Management Module. Instead, the host receives a couple of integers indicating which pair of amino acid sequence has generated a significant score. This mechanism contributes to significantly decrease the need for a high data bandwidth since a small amount of information need to be transfer from the FPGA accelerator to the host memory.

SMALL GAP parallelization on GPU

The parallelization of the **SMALL_GAP** function on GPU is straightforward. The host downloads the GPU memory with couples of string of identical size (2*L2+W). Then, a thread is devoted to the computation of one score between couples of strings. Each thread performs a dynamic programming treatment using a banded Smith & Waterman algorithm. All the scores are sent back to the host which needs to post process these data before triggering, if necessary, a full gap extension.

4. Comparison of the GPU / FPGA technologies

The aim of this section is to evaluate the different approaches on a common base, and to discuss the advantages and the drawbacks for each of them having in mind the minimization of the execution time. Here, the sensitivity aspect won't be discussed. A detailed study showing that PLAST and BLAST have an equivalent sensitivity can be found in [16]. Briefly, both algorithms use the same family of heuristics. They slightly differ on the seed choice and, consequently, do not exactly generate the same list of alignments. A few percentages of alignments are found by BLAST and not by PLAST. Inversely, a few are found by PLAST and not by BLAST. The difference is mainly expressed by alignments of weak similarity and represents less than 2% for an e-value of 10^{-3}.

The reference is the execution time of the BLAST software running on a multithreaded mode. It should allow the readers (1) to measure the contributions of the various technologies over a conventional but highly optimized implementation and (2) to appreciate the difference of performances between GPU and FPGA technologies. Two hardware platforms are considered: a GPU platform and a FPGA platform.

GPU platform

The GPU platform is a Dell Server, 2.6 GHz Xeon Core 2 Quad processor with 8 GB of RAM running Linux Fedora 7. It is equipped with **two** NVIDIA Tesla C870 boards interconnected through PCI express buses. Each board houses 1.5 GB of GDDR3 memory and a graphic chip including 128 multithreaded processors. The programming language is CUDA. The **UNGAP** and **SMALL_GAP** functions are run on the GPU boards.

FPGA Platform

The FPGA platform is a SGI ALTIX 350 machine composed of an Intel Itanium2 Core2 (1.6 GHz) with 4 GB of RAM, running SUSE Linux, and equipped with a RASC-100 accelerator (Reconfigurable Application-Specific Computing). This device is interconnected to the host system through a NUMAlink bus and is made of **two** Xilinx Virtex-4 FPGA components. The programming language is VHDL. Only the **UNGAP** function is run on the FPGA board. The **SMALL_GAP** function is processed by the host.

Software and data set

Like BLAST, PLAST is declined into several programs targeting various data sets. Here, the comparison between TBLASTN/TPLASTN is only presented. Reasons are:

- Sequence comparison perform by this program is time-consuming due to the translation of the DNA bank into its six reading frames, and consequently very well suited for demonstrating the efficient contributions of GPU or FPGA accelerators.

- The gap extension step, in these two programs, represents generally a minor percentage of the execution time. As an FPGA implementation does not exist for the **`SMALL_GAP`** function, the FPGA approach will not be too disadvantaged.

The GPU and FPGA version of TPLASTN are respectively referenced as GPU-TPLASTN (Graphical Processing Unit) and RCC-TPLASTN (Re-Configurable Computing).

The data set is composed as follows:

- The human chromosome 1 (220 x10^6 bp) : hchr1
- 4 protein banks randomly constructed from the GenBank non redundant protein database:
  - P1K : 1000 protein sequences (0.336 × 10^6 aa)
  - P3K : 3000 protein sequences (1.025 × 10^6 aa)
  - P10K : 10,000 protein sequences (3.433 × 10^6 aa)
  - P30K : 30,000 protein sequences (10.335 × 10^6 aa)

The BLAST (release 2.2.18) options have been set as follows:

```
blastall –p tblastn –d hchr1 –i pxxK –o rxxK –m 8 –a 2 –e 0.001
```

The –m 8 option provides a tabulated output synthesizing the features of the alignments. The –a 2 option runs BLAST in a multithreaded mode (2 threads). The –e 0.001 option sets the e-value to 10^{-3}.

Comparison of the execution times

Table 1 and Table 2 report the execution times of the NCBI TBLASTN, GPU-TPLASTN and RC-TPLASTN.

	NCBI TBLASTN (2 threads)	GPU-TPLASTN (2 boards)	speed up
P1K	754	140	5.38
P3K	2172	258	8.41
P10K	7436	744	9.99
P30K	21951	2165	10.13

Table 1: Execution time (in second) of NCBI TBLASTN and GPU-TPLASTN on the GPU platform (2 × C870 TESLA NVIDIA boards – 128 PEs per chip)

	NCBI TBLASTN (2 threads)	RCC-TPLASTN (2 FPGA)	speed up
P1K	1162	363	3.20
P3K	3441	398	8.64
P10K	11733	643	18.29
P30K	37088	1323	28.03

Table 2: Execution time (in second) of NCBI TBLASTN and RCC-TPLASTN on the FPGA platform (SGI RASC-100 – 2 × Xilinx Virtex 4 – 192 PEs per chip)

Note that the NCBI TBLASTN execution time is different for the two platforms. However, this time serves as a reference to compare the speed up with accelerators (GPU or FPGA). To be fair, for each experiment, NCBI TBLASTN, GPU-TPLASTN and RCC-TPLASTN have been run in a multithreaded mode with two threads. In the GPU mode, each thread drives a NVIDIA Tesla C870 board and progresses independently as explain in section 2. Similarly, in the RCC mode, each thread controls a separate FPGA Virtex-4 device.

Globally, it can be seen that performances increase with the size of the data, whatever the technology or the platform used. A plateau is however reached for huge computations (a few hours). This can be explained by the fact that, in that case, the **`UNGAP`** and **`SMALL_GAP`** functions represent a very high percentage of the total execution time which is efficiently parallelized on the accelerators. On the other hand, when the volume of data is low, the ratio between the sequential part and the parallel part increases and, following the Amdahl's law, limits the potential speed up.

GPU implementation

Adding two NVIDIA C870 Tesla boards provide a speed up factor of 10 for intensive protein sequence comparison compared to the NCBI BLAST multithreaded version. Each board integrates a GPU chip (G80) housing 128 programmable processing units. A question is: can we do better? A first answer is to take the following generation of graphic boards to test the scalability of this approach. Experimentations with the NVIDIA GTX-280 board (T10 chip - 240 processing units) are reported Table 3.

	NCBI TBLASTN (1 thread)	GPU Tesla C870 TPLASTN		GPU GTX-280 TPLASTN		NCBI speed up		GTX-280 speed up	
		Total	UNGAP	Total	UNGAP	Tesla	GTX 280	Total	UNGAP
P1K	1369	250	114	216	80	5.47	6.33	1.15	1.42
P3K	4009	474	306	383	215	8.45	10.36	1.23	1.42
P10K	13391	1341	971	1053	681	9.98	12.71	1.27	1.42
P30K	40444	3932	2917	3077	2057	10.38	13.14	1.27	1.42

Table 3: performance comparison between the NVIDIA Telsa C870 board and the NVIDIA GTX-280 board (time is given in second).

In this experiment, the multithreaded mode is disabled for only highlighting the difference of performance between two successive generations of graphic boards. Several comments can be made:

- The threads overhead is negligible. The speed up with 2 threads and 2 boards (Table 1) is very closed of the speed up with 1 thread and 1 board (Table 3, column 7).

- Moving to the next board generation provide an immediate increasing of performance (columns 7 and 8) without any modification of the CUDA code.

- The **UNGAP** function represents an important percentage of the total execution time. Thus, there are still some rooms for further speed up improvements. In the P30K configuration, the theoretical maximum speed-up compared to the NCBI TBLASTN software is about 40 (col2 / (col5 – col6)). This value is estimated as the NCBI-BLASTN execution time divided by the sequential part of GPU-TPLASTN.

FPGA implementation

Table 2 reports the results for 2 × 192-PE arrays implemented on both FPGA devices. This is the maximum of PEs we were able to fit inside the FPGA device. However, we experiment the performances on various array sizes as shown Table 4.

	NCBI TBLASTN (1 thread)	RCC-TPLASTN **64 PEs**			RCC-TPLASTN **128 PEs**			RCC-TPLASTN **192 PEs**		
		Total	UNGAP	Speed up	Total	UNGAP	Speed up	Total	UNGAP	Speed up
P1K	2185	476	220	4.59	421	176	5.19	414	169	5.27
P3K	6448	738	462	8.73	554	280	11.63	496	223	13.00
P10K	21888	1763	1366	12.41	1104	720	16.02	890	510	24.59
P30K	65461	4463	3932	14.66	2744	2015	23.85	2099	1373	31.86

Table 4: Execution time (in second) of RCC-TPLASTN with different numbers of PEs.

Again, the multithreaded mode is disabled to only measure the contribution of the FPGA accelerator. It can be seen that the number of PEs is inversely proportional to the execution time of the **UNGAP** function. For example, if the **UNGAP** speedup is measured relatively to 64 PEs, we get:

# proc	64 PEs	128 PEs	192 PEs
UNGAP *(second)*	3992	2015	1373
Speed up *(relatively to 64 PEs)*	1	1. 98	2.9

Larger arrays are thus still possible to decrease significantly the overall execution time. The SGI RASC-100 accelerator houses Virtex 4 Xilinx components of 200 K logic cells with 336 × 18Kb RAM Blocks (Virtex-4 LX 200). With the next generation of Xilinx components, a faster 384 PE array could be easily implemented in a single FPGA (Virtex6: XC6VLX550T) and

would at least provide a speedup ranging from 5 to 6 compared to a 100 MHz 64 PE array. In that case, the overall speedup would be somewhere between 45 and 50.

5. Conclusion

PLAST is a parallel software for intensive protein comparison. Unlike BLAST, it does not target the scan of genomic databases. It has been designed for processing two large banks of sequences. Different versions are available depending of the nature of the data: PLASTP (protein/protein), PLASTX (DNA/protein), TPLASTN (protein/DNA) and TPLASTX (DNA/DNA). DNA sequences are translated into 6 reading frames. The heart of these programs and their parallelization scheme are however identical.

Like FASTA and BLAST, PLAST uses the concept of seeds to reduce the search space. The main difference is that two index tables are built, allowing groups of identical hits to be immediately identified. Each group can be processed independently on a multithreaded architecture (first level of parallelism), and the computation of each group can be deported on a GPU or FPGA accelerator (second level of parallelism). The combination of these two levels of parallelism fit well with current machines made of multi-core processors and which can easily be enhanced with hardware accelerators connected through fast interfaces, like PCI express buses.

The originality of PLAST is that its design has been thought, in its earlier steps, as a parallel algorithm able to target the current and the next generations of computer systems. To compensate the end of systematical increase of the microprocessor clock frequency, to optimize the electric power consumption, and to continue to follow the Moore's law, the future chips will be highly parallel systems. The GPGPU architectures are probably an intermediate (and necessary) phase before more flexible parallel structures of hundreds of processing elements. Bioinformatics algorithms need to be revisited to benefit from maximal efficiency provided by these new architectures in order to face the exponential demand in terms of genomic data processing.

6. Bibliography

[1] The Universal Protein Resource (UniProt), The UniProt Consortium, Nucleic Acids Research 37 (Database issue): D169-D174, 2009.

[2] Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Wheeler DL, GenBank, Nucleic Acids Research, 36 (Database issue): D25-30, 2008.

[3] Shendure, J. & Hanlee, J., Next-generation DNA sequencing, *Nature Biotechnology*, vol. 26, no. 10, pp.1135-1145, 2008

[4] Lavenier D, Giraud M, Bioinformatics Applications, in Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays, M.B. Gokhale, P.S. Graham editor,chapter 9, Springer, 2005.

[5] Pearson W, Lipman D, Improved tools for biological sequence comparison. Proc. National Academy of Science, vol. 85, no. 8, pp. 2444–2448, 1988.

[6] Altschul S, Gish W, Miller W, Myers E, Lipman D, Basic local alignment search tool, J. Mol. Biol., vol. 215, no. 3, pp. 403–410, 1990.

[7] Altschul S, Madden T, Schäffer A, Zhang J, Zhang Z, Miller W, Lipman D, Gapped

BLAST and PSI-BLAST: a new generation of protein database search programs, Nucleic Acids Res, vol. 25, pp. 3389-3402, 1997

[8] Muriki K, Underwood K, Sass R, RC-BLAST: Towards a portable, cost-effective open source hardware implementation. In 19th International Parallel and Distributed Processing Symposium IPDPS05, 2005.

[9] Lancaster J, Jacob A, Buhler J, Harris B, Chamberlain R, Mercury BLASTP: accelerating protein sequence alignment, ACM Transactions on Reconfigurable Technology and System, 1(2), 2008.

[10] Lavenier D, Gille Georges G, Xinchu L, A recofigurable index flash memory tailored to seed-based genomic sequence comparison algorithms. VLSI Signal Processing, 48(3), pp. 255-269, 2007.

[11] Fei Xia, Yong Dou, and Jinbo Xu. Hardware BLAST algorithms with multi-seeds detection and parallel extension. In Reconfigurable Computing: Architectures, Tools and Applications, 4th International Workshop, ARC 2008, pp. 39-50, 2008.

[12] Kasap S, Ying L, Benkrid K, High performance FPGA-based core for BLAST sequence alignment with the two-hit method. In 8th IEEE International Conference on BioInformatics and BioEngineering, pp. 1-7, 2008.

[13] Herbordt M, Model J, Gu Y, Sukhwani B, Van-Court T. Single pass, BLAST-Like, approximate string matching on FPGAs. In Field-Programmable Custom Computing Machines, pp. 217-226, 2006.

[14] Roytberg M, Gambin A, Noé L, Lasota S, Furletova E, Szczurek E, Kucherov G, On Subset Seeds for Protein Alignment, IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 99, no. 1, 2009.

[15] Peterlongo P, Noe L, Lavenier D, Georges G, Jacques J, Kucherov G, Giraud M, Protein similarity search with subset seeds on a dedicated reconfigurable hardware. In Parallel Bio-Computing, 2007.

[16] Nguyen V, Lavenier D, PLAST: Parallel Local Alignment search Tool for Database comparison, BMC bioinformatics, to appear, 2009.

[17] Nguyen V, Lavenier D, Fine-grained parallelization of similarity search between protein sequences, INRIA Report, RR-6513, 2008.

[18] NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.0, 23/6/2007.

[19] Nguyen V, Cornu A, Lavenier D. Implementing Protein Seed-Based Comparison Algorithm on the SGI RASC-100 Platform, 16th Reconfigurable Architectures Workshop, May 25-26, Rome, Italy, 2009.

Chapitre 6

PLAST - Version MPI

Ce chapitre présente la mise en œuvre de PLAST (version multi-cœur SSE) sur un cluster de machines en utilisant la librairie MPI. Cette implémentation utilise l'approche de mpiBLAST [15] qui partitionne et répartit une banque de séquences sur les nœuds d'un cluster.

La première section introduit le contexte de l'implémentation de BLAST sur des architectures à mémoire partagée et distribuée. L'algorithme de mpiBLAST est décrit dans la section suivante. La troisième section présente l'implémentation de mpiPLAST et évalue ses performances.

6.1 Introduction

Le programme BLAST [1, 2] est un logiciel de référence dans le domaine de la bioinformatique. Étant donnée la croissance des données génomiques, de nombreuses approches utilisent la parallélisation pour améliorer ses performances. La parallélisation de BLAST sur des architectures à mémoire partagée ou distribuée délivre un bon facteur d'accélération.

Pour paralléliser sur une architecture à mémoire partagée, comme les multi processeurs ou les multi-cœurs, BLAST exploite le modèle de programmation multi-thread pour comparer simultanément des séquences requêtes avec des morceaux de banque. Chaque thread est chargé de scanner un fragment de la banque. La parallélisation de BLAST sur ce modèle d'architecture a donc besoin d'un dispositif d'E/S intensif.

La duplication d'une banque de séquences sur des supports de stockage distribué constitue une autre possibilité [78]. Cette approche utilise le paradigme maître/esclaves. Les séquences requêtes sont partitionnées et assignées à chaque esclave. Le point faible de cette approche est la duplication de la banque.

L'approche de distribution d'une banque a été proposée par Pedretti et al. [78]. La version MPI a été implémentée par Darling et al. dans mpiBLAST [15].

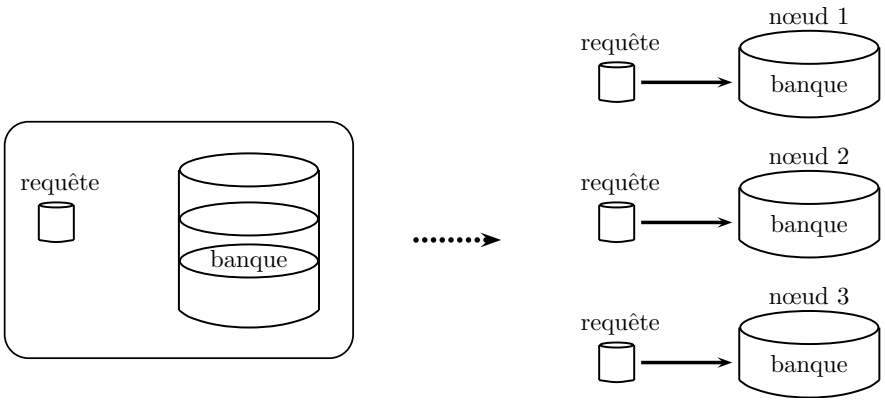


FIGURE 6.1 – Principe de mpiBLAST : segmentation de la banque et recherche dans les fragments de la banque.

6.2 Algorithme de mpiBLAST

La parallélisation de mpiBLAST se base sur la segmentation d'une banque. Avant la recherche, la banque est formatée et partitionnée en plusieurs fragments. L'algorithme de mpiBLAST se décompose en deux étapes principales. Dans la première étape, la banque est segmentée et stockée dans un support de stockage partagé. mpiBLAST effectue, dans la deuxième étape, une recherche indépendante dans les fragments sur chaque nœud. Si un nœud ne contient pas le fragment dans son disque dur, il le copie à

partir du support de stockage partagé. L'affectation des fragments à chaque nœud est déterminée par un algorithme minimisant le nombre de copies des fragments.

6.2.1 Segmentation et distribution de la banque

Algorithm 6.1 Maître

Soit Q les séquences requêtes
Soit F = $\{f_1, f_2, \dots\}$ l'ensemble des fragments de la banque
Soit **Unsearched** $\subseteq$ F l'ensemble des fragments pas encore recherchés
Soit **Unassigned** $\subseteq$ F l'ensemble des fragments pas encore assignés
Soit W = $\{w_1, w_2, \dots\}$ l'ensemble des esclaves participant
Soit $D_i \subseteq$ W l'ensemble des esclaves ayant le fragment f_i sur leur disque dur
Résultat $\leftarrow \emptyset$
Diffuser Q aux esclaves
Unsearched $\leftarrow$ F
Unassigned $\leftarrow$ F
tant que |**Unsearched**| $\neq 0$
  Recevoir un message d'un esclave w_j
  si message est une demande d'état
    si |**Unassigned**| = 0
      Envoyer l'état SEARCH_COMPLETE à w_j
    sinon
      Envoyer l'état SEARCH_INCOMPLETE à w_j
    fin si
  sinon si message est une demande d'un fragment
    si ($\neg\exists f_i \in$ **Unassigned** et $w_j \in D_i$)
      Ajouter w_j dans D_i
    fin si
    Supprimer f_i dans **Unassigned**
    Envoyer fragment f_i à w_j
  sinon si message est l'ensemble de résultats pour le fragment f_i
    Fusionner le message à *résultat*
    Supprimer f_i dans **Unsearched**
  fin si
fin tant

mpiBLAST fournit un outil *mpiformatdb* qui est une variante d'outil *formatdb* et qui se combine avec la tâche de formatage et la tâche de partition. mpiformatdb divise la banque en plusieurs fragments de taille approximative. Ensuite, les fragments formatés sont stockés dans le support de stockage partagé (Figure 6.1)

6.2.2 Algorithme de mpiBLAST

mpiBLAST organise des processus parallèles en un maître et plusieurs esclaves (Algorithmes 6.1 et 6.2). Le maître utilise l'algorithme "greedy" pour assigner des fragments pas encore recherchés aux esclaves. Les esclaves copient des fragments assignés à leur stockage local et réalisent simultanément une recherche. Après avoir fini la recherche dans un fragment, l'esclave rend compte de ses résultats au maître pour fusionner. Ce processus se répète jusqu'à ce que tous les fragments soient recherchés. mpiBLAST obtient un bon facteur d'accélération, en particulier pour des clusters de taille moyenne.

Algorithm 6.2 Esclave

```
requête ← recevoir les séquences requêtes du maître
ÉtatCourant ← recevoir l'État du maître
tant que ÉtatCourant ≠ SEARCH_COMPLETE
   FragmentCourant ← recevoir un fragment du maître
   si FragmentCourant n'est pas dans le stockage local
     Copier FragmentCourant au stockage local
   fin si
   résultat ← BLAST (requête, FragmentCourant)
   Envoyer résultat au maître
   ÉtatCourant ← recevoir l'État du maître
fin tant
```

Le temps d'exécution de mpiBLAST peut être décomposé en deux parties : le temps de recherche de BLAST et le temps de non-recherche. Le temps de non-recherche contient le temps de copie des fragments, le temps de communication et le temps de fusion des résultats.

Lorsque le ratio fragment/esclaves est petit, il peut influencer le temps d'exécution de mpiBLAST, parce que quelques esclaves finissent tandis que les autres réalisent la recherche des fragments restants. De plus, dans certains cas, quelques fragments peuvent prendre beaucoup plus de temps que d'autres. Ce ratio peut donc provoquer un problème de déséquilibre sur les esclaves. La solution potentielle est d'augmenter le nombre de fragments. Cependant, Lin et al. [97] ont montré que l'augmentation de ce ratio peut augmenter le coût des E/S. La version mpiBLAST-PIO utilisant des E/S parallèles pour accéder aux données partagées a été proposé par Lin et al. [97] pour améliorer la performance de mipBLAST.

Le paradigme d'un maître et plusieurs esclaves obtient un facteur d'accélération linéaire sur un cluster de cent processeurs [15]. Ensuite, plus le nombre de processeurs est important, moins mpiBLAST est performant, parce que le maître a beaucoup de travail administratif. Cela peut conduire les esclaves à attendre plus longtemps pour une requête chaque fois qu'ils veulent plus de travail. Pour fonctionner efficacement sur un système parallèle massif, comme IBM Blue Gen/L [98], la version multi-maîtres de mpiBLAST a été proposée par Thorsen et al. [98]. Dans cette version, le nombre d'esclaves par rapport à un maître est limité en créant des groupes de nœuds. Chaque groupe contient un maître et 64 esclaves [98].

6.3 Implémentation et évaluation de mpiPLAST

6.3.1 Implémentation

Comme mpiBLAST, l'algorithme de mpiPLAST se décompose en deux principaux composants : la segmentation de la banque et la recherche dans les fragments.

La banque est divisée en plusieurs fragments de taille approximative par un outil séquentiel, *segmentdb*. Les fragments sont stockés également dans un support de stockage partagé. L'outil mpiformatdb de mpiBLAST utilise le programme de *formatdb* pour formater la banque avant de la diviser en plusieurs fragments. Cependant, *segmentdb* de mpiPLAST partitionne seulement la banque.

PLAST a été conçu et été optimisé pour la comparaison intensive de séquences sur des nouvelles architectures multi-cœurs. Pour l'instant, mpiPLAST utilise l'algorithme d'origine de Darling [15] : un maître et plusieurs esclaves. Dans le cadre de la comparaison intensive, le cœur de mpiPLAST est le programme PLAST effectuant des comparaisons entre des fragments d'une banque et une banque requête. Dans cette implémentation, nous avons utilisé la version multi-cœur SSE.

6.3.2 Plate-forme et jeux de données

Pour évaluer les performances de mpiPLAST et mpiBLAST (version1.5.0-PIO) [99], nous avons utilisé un cluster dont les nœuds ont les caractéristiques suivantes : 2.8 GHz Xeon Core 2 Quad processeur avec 64 GB de RAM. Les deux programmes utilisent le logiciel MPICH (version 2-1.1) utilisant la librairie MPI pour lancer des calculs sur ce cluster.

Les trois jeux de données considérés sont :

- PLASTP/BLASTP : 1000 protéines sélectionnées dans ASTRAL/SCOP (version 1.73) contre GenBank non redundant protein database (version Jan-2009, 2.6 GB) ;
- TPLASTN/TBLASTN : 1000 protéines sélectionnées dans Genbank non redundant protein database contre Genbank Plant (version 1.72, 4 GB) ;
- PLASTX/BLASTX : 1000 séquences ADN sélectionnées dans Genbank gbest230 contre GenBank non redundant protein database (version Jan-2009, 2.6 GB).

No de proc.	BLASTP	PLASTP	TBLASTN	TPLASTN	BLASTX	PLASTX
8	113m15	13m57 (7)	183m37	24m33 (7)	157m11	16m53 (9)
16	47m18	7m04 (6)	90m45	13m18 (6)	78m43	8m37 (9)
32	23m39	3m48 (6)	45m02	7m44 (5)	39m24	4m34 (8)
64	11m56	1m52 (6)	22m35	4m03 (5)	15m50	2m20 (6)

TABLE 6.1 – Performances de mpiBLAST et mpiPLAST en fonction du nombre de processeurs sur trois jeux de données.

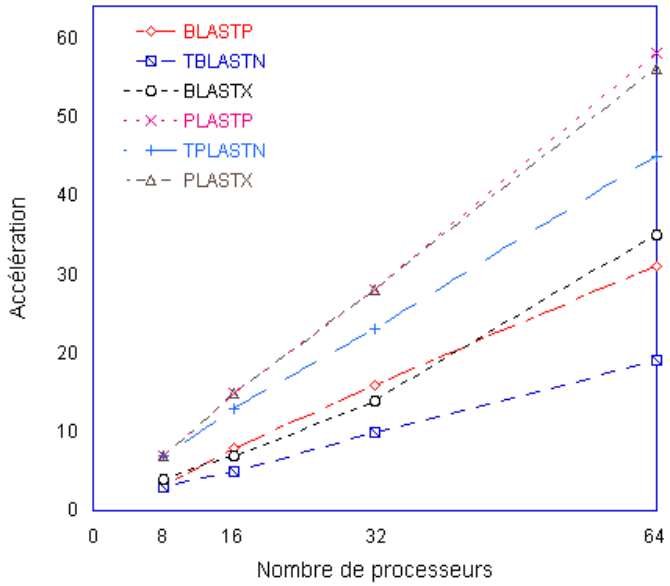


FIGURE 6.2 – Accélération obtenue de mpiBLAST et mpiPLAST par rapport à la version séquentielle sur trois jeux de données.

6.3.3 Performances

Le nombre de processeurs utilisés varie de 8 à 64. Le nombre de fragments des banques est de 64. L'E-value est positionnée à 10^{-10} et l'option "-m8" de BLAST est activée. Les performances de mpiBLAST et mpiPLAST sur les trois jeux de données se présentent dans la table 6.1. Le numéro entre parenthèses présente l'accélération de mpiPLAST par rapport à mpiBLAST. La figure 6.2 montre l'accélération obtenue de mpiBLAST et mpiPLAST en comparant à la version séquentielle sur un processeur.

Le ratio fragment/esclaves provoque également le problème de déséquilibre sur des processeurs. La performance de mpiPLAST peut être améliorée quand ce ratio est augmenté car les résultats partiels sont stockés dans la mémoire locale. Par contre, PLAST utilise les jeux d'instructions SSE pour accélérer les deux étapes d'extension (gap et sans gap). Avec plusieurs petits fragments, le jeu d'instructions n'est pas vraiment exploité.

6.4 Conclusion

Nous avons présenté l'implémentation de mpiPLAST sur un cluster de machines en utilisant la librairie MPI. Cette implémentation utilise l'approche de mpiBLAST [15], la référence du domaine de la bioinformatique. Les performances obtenues, par rapport à mpiBLAST, montrent une accélération significative suivant le nombre de processeurs. De plus, elles montrent que l'accélération de mpiPLAST est presque linéaire en fonction du nombre de processeurs rapport à la version séquentielle (PLAST).

Chapitre 7

Conclusion et perspectives

Ce chapitre est divisé en trois parties. La première rappelle les contributions de la thèse. La seconde pointe les améliorations potentielles du logiciel PLAST. La troisième dresse quelques perspectives de recherche à plus long terme.

7.1 Travaux réalisés

Dans cette thèse, nous nous sommes intéressés à l'accélération de la recherche de similarités entre séquences protéiques et, plus précisément, à la comparaison banque à banque. La parallélisation des calculs a été effectuée en tenant compte de l'architecture des microprocesseurs actuels. Concrètement, nos travaux ont porté sur les points suivants :

- un état de l'art sur : (1) les techniques heuristiques à base de graines ; (2) les techniques algorithmiques associées à des structures d'index efficaces ; (3) les accélérateurs matériels relatifs à la comparaison de séquences génomiques.
- une proposition d'algorithme parallèle pour la comparaison de séquences protéiques. Cet algorithme utilise le concept de graine pour réduire l'espace de calcul et se base sur une structure d'index qui permet de regrouper des traitements indépendants. L'algorithme a été imaginé en tenant compte des technologies actuelles capables de supporter une mise en œuvre parallèle de la comparaison entre banques de séquences.
- le développement du logiciel PLAST en plusieurs versions suivant les cibles technologiques visées : multi-cœur SSE, cartes graphiques, accélérateurs reconfigurables (FPGA) et clusters. Différents programmes ont été implémentés : PLASTP pour la comparaison entre protéines, TPLASTN et PLASTX pour la comparaison protéine / ADN et TPLASTX pour la comparaison ADN / ADN via la transcription suivant les 6 phases de lecture.
- l'évaluation des performances des différentes versions de PLAST pour le traitement des comparaisons entre banques en comparant avec le programme BLAST. Nous avons également mesuré la sensibilité et sélectivité de PLAST par rapport à BLAST. Les résultats ont montré que la sensibilité de PLAST est comparable

à celle de BLAST.

7.2 Amélioration du logiciel PLAST

Comme nous l'avons exprimé dans les diverses conclusions des articles, le logiciel PLAST pourrait être amélioré selon plusieurs axes. Nous proposons, ci-dessous, quatre pistes qui, potentiellement, peuvent encore accroître les performances.

Indexation

Au cours de l'étape d'indexation, pour des graines rares, de faux appariements sont volontairement ajoutés entre deux positions dont l'intervalle dépasse 65536 (codage sur 16 bits). Cette situation pourrait être améliorée en considérant plutôt l'ajout de mots proches. Cela permettrait d'augmenter indirectement la sensibilité des traitements, sans perte d'efficacité.

Par ailleurs, les tests avec huit processeurs (version multi-cœur SSE) ont montré que le pourcentage de temps d'indexation devient significatif par rapport au temps d'exécution total. Cette étape, rappelons le, est calculée séquentiellement. D'après la loi Amdahl [100], l'accélération maximum d'un programme est de $1/(1-P)$ où P représente la fraction parallèle. La non parallélisation de cette partie deviendra donc un facteur limitant dès lors que le nombre de processeurs augmentera. Il faut alors envisager une version parallèle de cette étape pour bénéficier des nombreux cœurs qui seront à notre disposition dans les processeurs de demain.

Extensions avec gap

Dans la version actuelle de l'algorithme, l'extension avec gap est découpée en 2 phases. Seule la première est accélérée, la seconde reprenant la procédure de BLAST développée par le NCBI. Lorsque les accélérateurs sont activés, le pourcentage de temps consacré à cette partie devient donc prépondérant. Plusieurs solutions sont possibles :

- pour la version multi-cœur SSE, il pourrait être combiné l'implémentation de Wozniak [63] et l'optimisation de *X-drop* de BLAST [2]. De cette manière, huit anti-diagonales pourraient être calculées simultanément à l'aide d'instructions SSE ;
- du côté des GPU, une stratégie consistant à calculer en parallèle un grand nombre de séquences triées suivant leur taille pourrait être mise en œuvre et apporter des gains intéressants comme il a été montré dans [11] ;
- du côté des FPGA, deux implémentations systoliques pourraient être employées : (1) une programmation dynamique implémentée sur une bande de largeur prédéterminée [12] ; (2) une programmation dynamique lancée dans deux directions différentes à partir du point d'ancrage [95].

Portabilité

La version GPU de PLAST utilise actuellement CUDA pour décrire les traitements à déporter sur les cartes graphiques. Le langage CUDA est malheureusement lié aux

cartes graphiques NVIDIA. Pour briser cette dépendance, il conviendrait de programmer ces traitements avec OpenCL [101], le futur standard qui permettra de s'affranchir des langages spécifiques aux fabricants.

La version FPGA de PLAST, quant à elle, à recours au langage VHDL pour décrire l'architecture reconfigurable. Le problème de ce langage est qu'il reste difficile à maîtriser pour des non experts en architecture. Aujourd'hui, il existe cependant des outils de synthèse matérielle (Mitrion-C [102], Gaut [103], Impulse-C [104], etc.) opérant à partir de spécifications haut niveau (C/C++). Ceux-ci ne permettent pas d'atteindre les niveaux de performances par rapport à un code VHDL écrit par un initié, mais ils permettent de raccourcir les cycles de design tout en offrant une certaine pérennité.

PLASTN

La famille actuelle des programmes PLAST se focalise sur les séquences de protéines. Le programme PLASTN, pour la comparaison ADN/ADN, n'est pas inclus dans le paquet courant.

Les techniques imaginées pour le traitement des protéines ne peuvent s'appliquer directement à l'indexation de séquences d'ADN, notamment l'usage de *subset seeds* propre aux acides aminés. Des techniques d'indexation innovantes et visant le parallélisme intrinsèque des microprocesseurs doivent donc encore être imaginées. Cette partie est encore en phase exploratoire.

7.3 Perspectives

La thèse ouvre aussi des perspectives pour utiliser la même stratégie de parallélisation à plusieurs autres types d'algorithmes d'analyse de séquences, notamment les algorithmes basés sur l'indexation des banques génomiques. De plus, pour être capable de traiter des données massives, les applications bioinformatiques ont toujours besoin d'algorithmes parallèles pour s'adapter aux nouvelles architectures. La solution sera des nouveaux algorithmes et des nouvelles structures de données qui seront intrinsèquement parallèle afin d'être exécutés sur des accélérateurs matériels variés.

De nombreuses implémentations parallèles d'algorithmes d'analyse de séquences sur les accélérateurs matériels (SIMD, GPU, CELL et FPGA) ou clusters de machines sont disponibles : Smith-Waterman sur GPU et CELL ; BLAST sur FPGA et CELL ; MPI-BLAST [15], CLUSTAL-MPI [105] sur cluster de PCs ; PLAST sur multi-cœur SSE, CUDA, FPGA ou cluster ; etc. La plupart des implémentations supportent seulement un accélérateur matériel à la fois. EXOCHI [106] et HMPP [107] permettent de générer automatiquement du code pour les accélérateurs en utilisant des environnements de compilation. La génération automatique d'un algorithme peut être difficile, en particulier en favorisant les différents accélérateurs. En attendant de nouvelle programmation hybride, que pourra utiliser des libraires comme : MPI, pthread, fonctions intrinsèques SSE, etc ; et des langages comme : OpenCL et VHDL ou Handel-C, la solution pour accélérer des algorithmes d'analyse de séquences.

Annexe A

Analyse de sensibilité

La sensibilité a été mesurée par la couverture des alignements. Les alignements rapportés par PLASTP et BLASTP sont comparés aux alignements rapportés par SSEARCH (l'implémentation de l'algorithme Smith-Waterman dans le paquet Fasta-35.4.7). Les alignements rapportés par SSEARCH qui ne sont pas trouvés par l'un des alignements rapportés par PLASTP et BLASTP sont considérés respectivement comme les alignements perdus de PLASTP et BLASTP. La matrice BLOSUM62 [22] est utilisée avec une pénalité (g_{open}, g_{extend}) de 11-1. L'option des paramètres statistiques de Karlin-Altschul [25] a été utilisée pour SSEARCH (-z 3). Pour PLASTP et BLASTP, l'ajustement de score (composition-based) et le filtre des séquences requêtes sont désactivés.

La sensibilité de PLASTP (SSE) et BLASTP (version 2.2.18) a été analysée par les quatre jeux de données ci-dessous :

1. 1000 protéines sélectionnées dans Genbank non redundant protein database contre PIR (version 80, 283416 proteins) ;
2. 1000 protéines sélectionnées dans ASTRAL/SCOP (version 1.73) contre PIR (version 80, 283416 proteins) ;
3. 1000 protéines sélectionnées dans ASTRAL/SCOP (version 1.73) contre UniprotKB (version 56.2, 398181 proteins) ;
4. 1000 protéines sélectionnées dans ASTRAL/SCOP (version 1.73) contre le premier volume de NR (Genbank non redundant protein database, 2977744 proteins).

Les deux graines sous-ensembles (ci-dessous) ont été utilisées pour évaluer la sensibilité de PLASTP. Les résultats de sensibilité et le temps d'exécution (en second) sont présentés dans les quatre tables ci-dessous. (*) représente PLASTP avec la deuxième graine sous-ensemble.

$$\begin{cases} \pi_1 = \{H, FY, W, IV, LM, C, RK, Q, E, N, D, A, S, T, G, P\} \\ \pi_2 = \{HFYWIVLMC, RKQENDASTGP\} \\ \pi_3 = \{H, FYW, IVLM, C, RK, QE, ND, A, ST, G, P\} \\ \pi_4 = \{H, FY, W, IV, LM, C, R, K, Q, E, N, D, A, S, T, G, P\} \end{cases}$$

FIGURE A.1 – Première graine sous-ensemble d'acides aminés.

$$\begin{cases} \pi_1 = \{AS, RQK, N, DE, C, G, H, IV, LM, FWY, P, T\} \\ \pi_2 = \{AGST, RNDQEHK, CPW, ILMFYV\} \\ \pi_3 = \{C, FYW, MLIV, GPATS, HQERKDN\} \\ \pi_4 = \{AST, RK, ND, C, QE, G, H, IL, MV, FWY, P\} \end{cases}$$

FIGURE A.2 – Deuxième graine sous-ensemble d'acides aminés.

	Alignements perdus			Temps
	E-value=10^{-3}	E-value=10^{-4}	E-value=10^{-5}	d'exécution
BLASTP	2467 (1,8%)	2064 (1,6%)	1842 (1,6%)	846
PLASTP	3174 (2,3%)	2501 (2,0%)	2099 (1,8%)	294
PLASTP*	2514 (1,8%)	2065 (1,6%)	1823 (1,6%)	398

TABLE A.1 – Analyse de sensibilité de BLASTP et PLASTP pour le premier jeu de données.

	Alignements perdus			Temps
	E-value=10^{-3}	E-value=10^{-4}	E-value=10^{-5}	d'exécution
BLASTP	1783 (2,1%)	1392 (1,7%)	1045 (1,3%)	1543
PLASTP	2048 (2,4%)	1324 (1,6%)	912 (1,2%)	530
PLASTP*	1837 (2,1%)	1286 (1,6%)	882 (1,2%)	698

TABLE A.2 – Analyse de sensibilité de BLASTP et PLASTP pour le deuxième jeu de données.

	Alignements perdus			Temps
	E-value=10^{-3}	E-value=10^{-4}	E-value=10^{-5}	d'exécution
BLASTP	3458 (2,5%)	2890 (2,2%)	2455 (2,0%)	1281
PLASTP	3919 (2,8%)	3009 (2,3%)	2434 (2,0%)	449
PLASTP*	3416 (2,5%)	2773 (2,1%)	2324 (1,9%)	610

TABLE A.3 – Analyse de sensibilité de BLASTP et PLASTP pour le troisième jeu de données.

	Alignements perdus			Temps d'exécution
	E-value=10^{-3}	E-value=10^{-4}	E-value=10^{-5}	
BLASTP	5519 (1,3%)	4239 (1,1%)	3412 (0,9%)	8151
PLASTP	7556 (1,8%)	5465 (1,4%)	4218 (1,1%)	2704
PLASTP*	5612 (1,3%)	4257 (1,1%)	3469 (0,9%)	3698

TABLE A.4 – Analyse de sensibilité de BLASTP et PLASTP pour le quatrième jeu de données.

Annexe B

Annexe technique du logiciel PLAST

Le logiciel PLAST recherche d'alignements locaux entre deux banques de séquences génomiques (séquences protéiques et séquences d'ADN). Le logiciel PLAST se décline en plusieurs versions suivant les cibles technologiques visées : multi-coeur SSE, cartes graphiques, accélérateurs reconfigurables (FPGA) ou cluster des machines.

Programmes de PLAST

PLAST se décline en 4 programmes
- PLASTP pour la recherche d'alignements entre 2 banques protéiques ;
- TPLASTN pour la recherche d'alignements entre 1 banque protéique et un génome transcrit dans les 6 phases de lecture ;
- PLASTX pour la recherche d'alignement entre 1 banque d'ADN transcrite dans les 6 phases de lecture et une banque protéique ;
- TPLASTX pour comparer deux banques d'ADN traduites en 6 phases.

Cibles technologiques visées

- Multi-cœur SSE utilise les technologies de multi-threading et d'instructions de type SSE ;
- CudaPLAST utilise les cartes graphiques (Nvidia) ;
- RCC-PLAST utilise les accélérateurs reconfigurables (SGI-RASC).
- mpiPLAST utilise un cluster de PCs.

Format d'entrée

Format d'entrée : les banques doivent être au format FASTA [108]. Elles ne demandent pas de pré formatage préalable comme le logiciel BLAST (formatdb).

Format de sortie

Format de sortie : pour chaque alignement trouvé, PLAST indique sa position, son score, son e-value, et son pourcentage d'identité. Ce format est équivalent à l'option "-m 8" du logiciel BLAST.

Langages de programmation et librairie

- PLAST multi-cœur SSE : langage C
- CudaPLAST : langage C et CUDA (Nvidia)
- RCC-PLAST : langage C et VHDL, librairie RASC (SGI)
- mpiPLAST : langage C et librairie MPI

Code source disponible

Le code source de PLAST est disponible sur le site de PLAST :
- Multi-cœur SE : http ://www.irisa.fr/symbiose/projects/plast
- CudaPLAST : http ://www.irisa.fr/symbiose/projects/plast/cudaplast-1.0.tgz
- RCC-PLAST : pas encore disponible
- mpiPLAST : http ://www.irisa.fr/symbiose/projects/plast/mpiplast-1.0.tgz

Utiliser un outil pour décompresser le package de PLAST, par exemple :
$ tar -zxvf plast-1.x.tar.gz

Nombre de lignes de code

- Multi-cœur SSE : 15163
- CudaPLAST : 15328
- RCC-PLAST : 13229
- mpiPLAST : 15566

Pré-requis techniques

- Multi-cœur SSE : jeux d'instructions SSE2 ;
- CudaPLAST : au moins un GPU ;
- RCC-PLAST : une carte SGI-RASC.
- mpiPLAST : un cluster de PCs.

Système d'exploitation

- Multi-cœur SSE : Linux, Windows, Mac
- CudaPLAST : Linux
- RCC-PLAST : Linux
- mpiPLAST : Linux

Compilation

1. Multi-cœur SSE
   - Linux et Mac (GNU)
     Changer dans le répertoire contenant PLAST
     Compiler : make
   - Windows (MinGW - Minimalist GNU for Windows)
     Changer dans le répertoire contenant PLAST
     Compiler : mingw-make
2. CudaPLAST
   - Installer CUDA (CUDA Toolkit et CUDA SDK)
   - Changer dans le répertoire contenant cudaPLAST
   - Modifier Makefile
     CUDA_INSTALL_PATH=/path/to/cuda/install
     CUDA_LIBDIR=/path/to/cuda/lib
     CUDA_COMMONDIR=/path/to/cuda/common
   - Compiler : make
3. RCC-PLAST
   - Installer SGI-RASC
   - Charger le bitstrem de PLAST
   - Changer dans le répertoire contenant RCC-PLAST
   - Modifier Makefile
     RASC_INCLUDE=/path/to/rasc/include
   - Complier : make
4. mpiPLAST
   - Installer MPICH ou LAM/MPI
   - Changer dans le répertoire contenant mpiPLAST
   - Complier : make

Exécution

$ plastall -p <prog> -d </path/to/database1> -i </path/to/database2> [options]
database1 est considéré comme une banque de données
database2 est considéré comme des séquences requêtes

Options

Pour afficher toutes les options de commande du programme PLAST, tapez $plastall

Flag	Description	Type	Défaut
-p	Program Name : plastp, tplastn, plastx and tplastx	String	
-d	Database	File In	
-i	Query file	File In	
-o	PLAST report Output File	File Out	stdout
-e	Expectation value (E)	Real	10.0
-n	Size of neighbourhood peforming ungapped extension	Integer	22
-s	Ungapped threshold trigger a small gapped extension	Integer	38
-z	Effective length of the database	Real	Real size
-a	Number of processors to use for PLAST multi-cœur SSE or number of GPU to use for Cuda-PLAST or number of FPGA to use for RCC-PLAST	Integer	1
-G	Cost to open a gap	Integer	11
-E	Cost to extend a gap	Integer	1
-X	X dropoff value for gapped alignment in bits	Integer	0 for tplastx, all others 15
-Z	X dropoff value for final gapped alignment in bits	Integer	0 for tplastx, all others 25
-F	Filter query sequence	T/F	T
-M	Matrix	String	BLOSUM62
-S	Subset seed	File In	
-R	Query strands to search against database (for plastx and tplastx) : 3 is both, 1 is top, 2 is bottom	Integer	3
-C	Use composition-based score adjustments as in Bioinformatics 21 :902-911 for plastp or tplastn	T/F	T

Glossaire

ADN	Acide Désoxyribo Nucléique
BLAST	Basic Local Alignment Search Tool
BLOSUM	BLOcks of Amino Acid SUbstitution Matrix
CUDA	Compute Unified Device Architecture
DFA	Deterministic Finit-state Automaton
EIB	Element Interconnection Bus
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
LIS	Long Increasing Subsequence
MDR	Maximal Different Repeats
MMR	Maximal Mismatch Repeats
MMX	MultiMedia eXtensions
MPI	Massage Passing Interface
MUM	Maximal Unique Match
NCBI	National Center for Biotechnology Information
PAM	Point Accepted Mutation
PE	Processing Element
PLAST	Parallel Local Alignment Search Tool
PPE	PowerPC Processor Element
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction, Multiple Data
SPE	Synergystic Processor Element
SPU	Streaming Processor Unit
SSE	Streaming SIMD Extensions
PSC	Parallel Sequence Comparison
UM	Unique Match

Bibliographie

[1] Stephent F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3) :403–410, 1990.

[2] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST : a new generation of protein database search programs. *Nucleic Acids Research*, 25(17) :3389–3402, 1997.

[3] Bin Ma, John Tromp, and Ming Li. PatternHunter : faster and more sensitive homology search. *Bioinformatics*, 18(3) :440–445, 2002.

[4] Ming Li, Bin Ma, Derek Kisman, and John Tromp. PatternHunter II : highly sensitive and fast homology search. *Journal of Bioinformatics and Computational Biology*, 2(3) :417–439, 2004.

[5] Mihkail Roytberg, Anna Gambin, Laurent Noe, Slawomir Lasota, Eugenia Furletova, Ewa Szczurek, and Gregory Kucherov. Efficient seeding techniques for protein similarity search. In *Bioinformatics Research and Development, Second International Conference, Vienna, Austria*, pages 466–478, 2008.

[6] Mihkail Roytberg, Anna Gambin, Laurent Noe, Slawomir Lasota, Eugenia Furletova, Ewa Szczurek, and Gregory Kucherov. On subset seeds for protein alignment. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 6(3) :483–494, 2009.

[7] W. James Kent. BLAT–the BLAST-like alignment tool. *Genome Res*, 12(4) :656–664, 2002.

[8] Dominique Lavenier, Gille Georges, and Xinchu Liu. A reconfigurable index flash memory tailored to seed-based genomic sequence comparison algorithms. *VLSI Signal Processing*, 48(3) :255–269, 2007.

[9] Torbjøn Rognes and Erling Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(2) :699–706, 2000.

[10] Michael Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23 :156–161, 2007.

[11] Svetlin A. Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[12] Joseph M. Lancaster Arpith C. Jacob and, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP : accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and System*, 1(2), 2008.

[13] Michael Cameron and Adam Cannane Hugh E. Williams and. Improved gapped alignment in BLAST. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 1(3) :116–129, 2004.

[14] NVIDIA CUDA compute unified device architecture, programming guide, version 2.0.

[15] Aaron E. Darling, Lucas Carey, and Wu-Chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of Conference on Linux Cluster : The HPC Revolution*, 2003.

[16] C. E. Metz. Basic principles of ROC analysis. *Seminars in nuclear medicine*, 8(4) :283–298, October 1978.

[17] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank. *Nucleic Acids Research*, 36 :D25–30, 2008.

[18] Hideaki Sugawara, Takashi Abe, Takashi Gojobori, and Yoshio Tateno. DDBJ working on evaluation and classification of bacterial genes in INSDC. *Nucleic Acids Research*, 35 :D13–15, 2007.

[19] Tamara Kulikova et al. EMBL nucleotide seequence database in 2006. *Nucleic Acids Research*, 35(Database issue) :D16–20, 2007.

[20] The UniProt Consortium. The universal protein resource. *Nucleic Acids Research*, 37 :D169–174, 2009.

[21] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[22] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89(22) :10915–10919, 1992.

[23] Stephent F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *J Mol Biol*, 219(3) :555–565, 1991.

[24] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87(6) :2264–2268, March 1990.

[25] S. F Altschul and W. Gish. Local alignment statistics. *Methods Enzymol*, 266 :460–480, 1996.

[26] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3) :443–453, 1970.

[27] Thomas F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1) :195–197, 1981.

[28] David J. Lipman and William R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693) :1435–1441, 1985.

[29] William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8) :2444–2448, 1988.

[30] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. In *Proceedings of the Fifth Annual International Conference on Computational Biology*, pages 69–76, 2001.

[31] Jeremy Buhler. Provably sensitive indexing strategies for biosequence similarity search. In *RECOMB, Washington, DC (USA)*, pages 90–99. ACM Press, April 2002.

[32] Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. In *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM)*, volume 2089 of *Lecture Notes in Computer Science*, pages 73–85. Springer-Verlag, 2001.

[33] Laurent Noé and Gregory Kucherov. Improved hit criteria for DNA local alignment. *BMC Bioinformatics*, 2004.

[34] Laurent Noé and Gregory Kucherov. Yass : enhancing the sensitivity of DNA similarity. *Nucleic Acids Research*, 33 :540–543, 2005.

[35] Yanni Sun and Jeremy Buhler. Designing multiple simultaneous seeds for DNA similarity search. In *Proceedings of the 8th Annual International Conference on Computational Molecular Biology (RECOMB), San Diego (California)*, pages 76–84, 2004.

[36] Jinbo Xu, Daniel G. Brown, Ming Li, and Bin Ma. Optimizing multiple spaced seeds for homology search. In *Proceedings of the 15th Symposium on Combinatorial Pattern Matching (CPM), Istambul (Turkey)*, pages 47–58, 2004.

[37] I. Yang, S. Wang, Y. Chen, P. Huang, L. Ye, X. Huang, and K. Chao. Efficient methods for generating optimal single and multiple spaced seeds. In *Proceedings of the IEEE 4th Symposium on Bioinformatics and Bioengineering (BIBE), Taichung (Taiwan)*, pages 411–416, 2004.

[38] B. Brejová, D. Brown, and T. Vinar. Vector seeds : an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In *WABI*, volume 2812 of *Lecture Notes in Computer Science*, pages 39–54. Springer-Verlag, 2003.

[39] Daniel G. Brown. Multiple vector seeds for protein alignment. In I. Jonassen and J. Kim, editors, *Proceedings of the 4th International Workshop in Algorithms in Bioinformatics (WABI), Bergen (Norway)*, volume 3240, pages 170–181. Springer-Verlag, 2004.

[40] Brejová, Daniel G. Brown, and T. Vinar. Vector seeds : An extension to spaced seeds. *Journal of Computer and System Sciences*, 70(3) :364–380, 2005.

[41] Daniel G. Brown. Optimizing multiple seed for protein homology search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1) :29–38, 2005.

[42] Gregory Kucherov, Laurent Noé, and Mikhail A. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *Journal of Bioinformatics and Computational Biology*, 4(2) :553–569, 2006.

[43] Pierre Peterlongo, Laurent Noé, Dominique Lavenier, Gille Georges, Julien Jacques, Gregory Kucherov, and M. Giraud. Protein similarity search with subset seeds on a dedicated reconfigurable hardware. In *Parallel Bio-Computing*, 2007.

[44] Jean-Pierre Dumas and Jacques Ninio. Efficient algorithms for folding and comparing nucleic acid sequences. 10(1) :197–206, 1982.

[45] Christian Fondrat and Phillippe Dessen. A rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. *Bioinformatics*, 11(3) :273–279, 1995.

[46] Andrea Califano and Isidore Rigoutsos. Flash : A fast look-up algorithm for string homology. In *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64. AAAI Press, 1993.

[47] Zemin Ning, Anthony J. Cox, and James C. Mullikin. SSAHA : A fast search method for large DNA databases. *Genome Res.*, 11(10) :1725–1729, 2001.

[48] E Michael Gertz, Yi-Kuo Yu, Richa Agarwala, Alejandro A. Schäffer, and Stephen F. Altschul. Composition-based statistics and translated nucleotide searches : Improving the TBLASTN module of BLAST. *BMC Biology*, 2006.

[49] You J. Kim, Andrew Boyd, Brian D. Athey, and Jignesh M.Patel. miBLAST : scalable evaluation of a batch of nucleotide sequence queries with BLAST. *Nucleic Acids Research*, 33(13) :4335–44, 2005.

[50] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[51] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13) :1149–1171, 1999.

[52] Stefan Kurtz and Chris Schleiermacher. REPuter : fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5) :426–427, 1999.

[53] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. REPuter : the manifold applications of repeat analysis on a genomic scale. *Nucl. Acids Res.*, 29(22) :4633–4642, 2001.

[54] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res*, 30(11) :2369–76, 1999.

[55] Arthur L. Delcher, Adam Phillippy, Jane Carlton, and Steven L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res*, 27(11) :2478–83, 2002.

[56] Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1) :53–86, March 2004.

[57] Hugh E. Williams. Compressed indexing for genomic retrieval. *Journal of Mathematical. Modelling and Scientific Computing*, 9(2) :144–154, 1998.

[58] Hugh E. Williams. CAFE : An indexed approach to searching genomic database. *SIGIR*, page 389, 1998.

[59] Hugh E. Williams and Justin Zobel. Indexing and retrieval for genomic databases. *IEEE Trans. Knowl. Data Eng.*, 14(1) :63–78, 2002.

[60] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4) :349–379, 1996.

[61] Justin Zobel and Philip W. Dart. Finding approximate matches in large lexicons. *Softw., Pract. Exper.*, 25(3) :331–345, 1995.

[62] Bowen Alpern, Larry Carter, and Kang Su Gatlin. Microparallelism and high-performance protein matching. In *Proceeding of the Supercomputing Conference*, pages 3–8, 1995.

[63] Andrzej Wozniak. Using video-oriented instructions to speed up sequence comparison. *Comput Appl. Biosci.*, 13(2) :145–150, 1997.

[64] William R. Pearson. Searching protein sequence libraries : Comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms. *Genomics*, 11 :635–650, 1991.

[65] Torbjøn Rognes. ParAlign : a parallel sequence alignment algorithm for rapid and sensitive database searches. *Nucleic Acids Res*, 29(7) :1647–1652, 2001.

[66] Per Eystein Sæbø, Sten Morten Andersen, Jon Myrseth1, Jon K. Laerdahl, and Torbjørn Rognes. ParAlign : rapid and sensitive sequence similarity searches powered by parallel computing technology. *Nucleic Acids Res*, 33 :W535–9, 2005.

[67] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5) :879–899, 2008.

[68] Technical overview ATI stream computing. 2009.

[69] Weiguo Liu, Bertil Schmidt, Gerrit Voss, Andre Schroeder, and W. Muller-Wittig. Bio-sequence database scanning on GPU. In *Proceeding of the 20th IEEE International Parallel & Distributed Processing Symposium (HICOMB Workshop)*, 2006.

[70] Yang Liu, Wayne Huang, John Johnson, and Sheila Vaidya. GPU accelerated smith-waterman. In *International Conference on Computational Science*, pages 188–195, 2006.

[71] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the CELL multiprocessor. *IBM Journal of Research and Development*, 2005.

[72] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network : Built for speed. *IEEE Micro*, 26(3) :10–23, 2006.

[73] Vipin Sachdeva, Michael Kistler, William E. Speight, and Tzy-Hwa K. Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. In *Sixth IEEE International Workshop on High Performance Computational Biology*, pages 1–8, 2007.

[74] Michael S. Farrar. Optimizing smith-waterman for the cell broadband engine. `http://farrar.michael.googlepages.com/smith-watermanfortheibmcellbe.Accessed2008Sep18`.

[75] Adrianto Wirawan, Kwoh C. Keongand, Nim T. Hieu, and Bertil Schmidt. CBESW : Sequence alignment on the playstation 3. *BMC Bioinformatics*, 9, 2008.

[76] Huiliang Zhang and, Bertil Schmidt, and W. Mueller-Wittig. Accelerating BLASTP on the cell broadband engines. In *Pattern Recognition in Bioinformatics, PRIB 2008, Melbourne, Australia*, pages 460–470, 2008.

[77] Michael Cameron, Hugh E. Williams, and Adam Cannane. A deterministic finite automaton for faster protein hit detection in BLAST. *Journal of Computational Biology*, 13(4) :965–978, 2006.

[78] Kevin T. Pedretti, Thomas L. Casavant, R. C. Braun, Todd E. Scheetz, C. L. Birkett, and Chad A. Roberts. Three complementary approaches to parallelization of local BLAST service on workstation clusters. In *PaCT '999 : Proceedings of the 5th International Conference on Parallel Computing Technologies*, pages 271–282, London, UK, 1999. Springer-Verlag.

[79] Steve Margerm, Cray, Inc. Reconfigurable computing in real-world applications. *FPGA and Programmable Logic*, 2006.

[80] Jason Chiang, Michael Studniberg, Jack Shaw, and Kevin Truong Stephen Seto. Hardware accelerator for genomic sequence alignment. In *28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5787–5789, 2006.

[81] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC Bioinformatics*, 8, 2007.

[82] Yoshiki Yamaguchi, Yosuke Miyajima, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search using run-time reconfiguration. In *12th International Conference on Field-Programmable Logic and Applications, Reconfigurable Computing Is Going Mainstream*, pages 281–291, 2002.

[83] H. T. Kung and C. Leiserson. *Algorithms for VLSI processors arrays.* Addison-Wesley, 1980.

[84] R.J. Lipton and D. P. Lopresti. A systolic array for rapid string comparison. In *Chapel Hill Conf. on VLSI, H. Fuchs, ed., Rockville, Md. : Computer Science Press*, pages 363–376, 1985.

[85] E. Chow, T. Hunkapiller, J. Peterson, and M.S. Waterman. Biological information signal processor. In *Proceedings of the International on Application Specific Array Processors*, pages 144–160, 1991.

[86] Dzung T. Hoang. Searching genetic databases on Splash 2. In *Proceedings. IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[87] Laiq Hasan, Yahya M. Khawaja, and Abdul Bais. A systolic array architecture for the Smith-Waterman algorithm with high performance cell design. In *Proceedings of IADIS European Conference on Data Mining*, pages 35–42, 2008.

[88] Jeremy D. Buhler, Joseph M. Lancaster, Arpith C. Jacob, and Roger D. Chamberlain. Mercury BLASTN : faster DNA sequence comparison using a streaming hardware architecture. In *Reconfigurable Systems Summer Institute*, 2007.

[89] Mitrionics.ins. : Mitron-accelerated BLAST 2007. `http://www.mitrionics.com`.

[90] Krishna Muriki, Keith D. Underwood, and Ron Sass. RC-BLAST : Towards a portable, cost-effective open source hardware implementation. In *19th International Parallel and Distributed Processing Symposium IPDPS*, 2005.

[91] Chan Chang. BLAST implementation on BEE2. Electrical Engineering and Computer ScienceUniversity of California at Berkeley, http ://www.cs.berkeley.edu.

[92] Brandon Harris, Arpith C. Jacob, Joseph M. Lancaster, Jeremy Buhler, and Roger D. Chamberlain. A banded smith-waterman FPGA accelerator for Mercury BLASTP. In *International Conference on Field Programmable Logic and Applications - FPL*, pages 765–769, 2007.

[93] Fei Xia, Yong Dou, and Jinbo Xu. Hardware BLAST algorithms with multi-seeds detection and parallel extension. In *Reconfigurable Computing : Architectures, Tools and Applications, 4th International Workshop*, pages 39–50, 2008.

[94] Server Kasap and andYing Liu Khaled Benkrid. High performance FPGA-based core for BLAST sequence alignment with the two-hit method. In *8th IEEE International Conference on BioInformatics and BioEngineering*, pages 1–7, 2008.

[95] Server Kasap, Khaled Benkrid, and Ying Liu. Design and implementation of an FPGA-based core for gapped BLAST sequence alignment with the two-hit method. *International Association Of Engineers*, August 2008.

[96] Martin C. Herbordt, Josh Model, Yongfeng Gu, Bharat Sukhwani, and Tom VanCourt. Single pass, BLAST-Like, approximate string matching on FPGAs. In *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 217–226, 2006.

[97] Heshan Lin, Xiaosong Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of 19th IEEE International on Parallel and Distributed Processing Symposium*, 2005.

[98] Oystein Thorsen, Brian Smith, Carlos P. Sosa, Karl Jiang, Heshan Lin, Amanda Peters, and Wu-Chun Feng. Parallel genomic sequence-search on a massively parallel system. In *Proceedings of the 4th international conference on Computing frontiers*, pages 59–68. ACM, 2007.

[99] mpiBLAST. `http://www.mpiblast.org`.

[100] G. Amdahl. Limits of expectation. *Supercomputer Appln USA*, pages 88–94, 1988.

[101] Aaftab Munshi. OpenCL specification v1.0. Technical report, 2008.

[102] Volodymyr V. Kindratenko, Robert J. Brunner, and Adam D. Myers. Mitrion-C application development on SGI Altix 350/RC100. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 239–250, 2007.

[103] David Elléouet, Yannig Savary, and Nathalie Julien. An FPGA power aware design flow. In *Power and Timing Modeling, Optimization and Simulation, 16th International Workshop*, pages 415–424, 2006.

[104] Impulse CoDeveloper C-to-FPGA tools. `http://www.impulseaccelerated.com/products_universal.htm`.

[105] Kuo-Bin Li. ClustalW-MPI : ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19(12) :1585–1586, 2003.

[106] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge : a programming model for heterogeneous multi-core systems. In *ASPLOS XIII : Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296. ACM, 2008.

[107] Romain Dolbeau, Stéphene Bihan, and François Bodin. HMPP : A hybrid multi-core parallel programming environment. `http://www.caps-entreprise.com/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf`, 2007.

[108] FASTA format description. `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`.

Table des figures

Résumé

La comparaison de séquences est une des tâches fondamentales de la bioinformatique. Les nouvelles technologies de séquençage conduisent à une production accélérée des données génomiques et renforcent les besoins en outils rapides et efficaces pour effectuer cette tâche.

Dans cette thèse, nous proposons un nouvel algorithme de comparaison intensive de séquences, explicitement conçu pour exploiter toutes les formes de parallélisme présentes dans les microprocesseurs de dernière génération (instruction SIMD, architecture multi-cœurs). Cet algorithme s'adapte également à un parallélisme massif que l'on peut trouver sur des accélérateurs de type FPGA ou GPU.

Cet algorithme a été mis en œuvre à travers le logiciel PLAST (Parallel Local Alignment Search Tool). Différentes versions sont disponibles suivant les données à traiter (protéine et/ou ADN). Une version MPI a également été mise au point pour un déploiement sur un cluster de PCs.

En fonction de la nature des données et des technologies employées des accélérations de 3 à 20 ont été mesurées par rapport à la référence du domaine, le logiciel BLAST, pour un niveau de qualité équivalent.

Mots-clés : comparaison de séquences, indexation, graine sous-ensemble, parallélisation, accélérateur, multi-cœur, instructions SSE, GPU, FPGA.

Abstract

The sequence comparison process is one of the main bioinformatics task. The new sequencing technologies lead to a fast increasing of genomic data and strengthen the need of fast and efficient tools to perform this task.

In this thesis, a new algorithm for intensive sequence comparison is proposed. It has been specifically designed to exploit all forms of parallelism of today microprocessors (SIMD instructions, multi-core architecture). This algorithm is also well suited for hardware accelerators such as FPGA or GPU boards.

The algorithm has been implemented into the PLAST software (Parallel Local Alignment Search Tool). Different versions are available according to the data to process (protein and/or DNA). A MPI version has also been developed.

According to the nature of the data and the type of technologies, speedup from 3 to 20 has been measured compared with the reference software, BLAST, with the same level of quality.

Keywords : sequence comparison, index, subset seed, parallel computing, accelerator, multi-core, SSE instructions, GPU, FPGA.