Algorithmique parallèle du texte : du modèle systolique au modèle CGM

Thierry Garcia

THÈSE

Présentée devant

l'Université de Picardie Jules Verne – Amiens
Faculté de Mathématiques et d'Informatique

pour obtenir

le grade de Docteur de l'Université d'Amiens

mention : INFORMATIQUE

par

Thierry GARCIA

titre de la thèse :

ALGORITHMIQUE PARALLÈLE DU TEXTE : DU MODÈLE SYSTOLIQUE AU MODÈLE CGM.

Soutenue le 27 Novembre 2003 devant la commission d'examen composée de

MM.	FERREIRA Afonso	Président
	DEHNE Frank	Rapporteur
	LECROQ Thierry	Rapporteur
	CERIN Christophe	Examinateur
	MYOUPO Jean-Frédéric	Examinateur
	SEME David	Directeur

À la mémoire de ma sœur aînée et marraine qui nous a quittés ce 03 décembre 2003,
À Nathalie, mon épouse,
À mon père et ma mère,
À mon grand-père,
À mes sœurs, mon frère, mes beau-frères, mes nièces et mes neveux, mon filleul et toute ma famille ...

Remerciements

Le verbe « supporter » a différentes significations : soit soutenir, soit tolérer. Dans un premier temps, je tiens à remercier David Semé pour m'avoir « supporté » durant ces trois années, pour son encadrement et pour m'avoir initié au monde de la recherche. Conciliant ma profession et cette thèse, David s'est toujours rendu disponible pour m'aider quel que soient l'heure et le jour de la semaine. Sans son aide, ce travail ne serait jamais parvenu à sa forme actuelle. Ses commentaires me furent d'une aide précieuse et ses suggestions ont amélioré cette thèse dans de nombreuses directions. Il m'a beaucoup appris par son approche pour aborder les problèmes de recherches, et je le remercie tout particulièrement pour la patience et la gentillesse dont il a fait preuve tout au long de ces trois années.

Je tiens également a exprimer toute ma reconnaissance aux membres du jury :

- Afonso Ferreira en sa qualité de président du jury,
- Thierry Lecroq pour avoir accepté de rapporter ce travail,
- Frank Dehne pour avoir accepté de rapporter ce travail,
- Chritophe Cérin pour avoir si volontiers accepté d'examiner cette thèse,
- Jean-Frédéric Myoupo pour m'avoir accueilli au sein de son équipe.

Un grand merci à Gilles Kassel de m'avoir accueilli au sein du laboratoire en sa qualité de directeur du LaRIA.

Je remercie chaleureusement tous celles et ceux que j'ai eu l'occasion de cotoyer durant ces 3 années : notamment les doctorants et tous les membres du laboratoire.

Je tiens également à remercier Catherine, Anne-Elisabeth, Sidney, (pour leur aide durant les répétitions) et Philippe Janssens (pour son aide durant l'expérimentation).

Je voudrais enfin remercier mon épouse Nathalie qui a « supporté » des vacances réduites, mes longues nuits devant l'ordinateur, mon caractère, mon stress et m'a soutenu pendant les conférences.

Je tiens également à avoir une pensée pour mes parents, mon grand-père, mes soeurs, mon frère, mes nièces et mes neveux, ma famille et mes amis pour le soutien et la confiance qu'ils m'ont toujours témoignés.

Table des matières

Table des figures

Chapitre 1

Introduction

L'informatique de 1946 avec la construction de l'$ENIAC$ (Electronic Numerical Integrator and Computer) composé de 19000 tubes, pesant 30 tonnes, ayant une vitesse d'horloge de 100 KHz, n'a plus rien à voir avec l'informatique actuelle où les micro-ordinateurs atteignent des vitesses supérieures à 3 GHz pour moins d'une dizaine de kilos. L'utilisation d'un ordinateur est devenue quasiment incontournable dans la vie quotidienne que l'on soit au travail, dans les transports, dans une banque, ... ou au domicile familial. Cependant, malgré l'omniprésence (ou bientôt l'omnipotence) de l'informatique, il existe encore et toujours des besoins de puissance de calculs bien supérieurs aux capacités des machines actuelles (en biologie moléculaire par exemple).

Certains jeunes ont souvent entendu la phrase : « Vous me copierez cent fois pour la semaine prochaine ... ». La sentence tombée, ils se concentraient sur la meilleure façon de coller plusieurs stylos avec du ruban adhésif afin d'exécuter la punition en plusieurs lignes parallèles ou bien, ils partaient à la recherche d'amis afin de réaliser la tâche à plusieurs.

L'écriture simultanée de plusieurs phrases ou le partage entre plusieurs personnes de la tâche à accomplir, rendait la punition moins pénible et semblait de bons moyens d'aller plus vite.

En fait, nous avons tous l'intuition qu'un travail peut être réalisé en beaucoup moins de temps s'il est réparti entre plusieurs personnes ou sur plusieurs machines. Cette notion se nomme le parallélisme qui peut se définir comme l'état de ce qui se développe dans la même direction ou en même temps.

Le parallélisme a été appliqué avec succès dans plusieurs activités humaines comme les

récoltes, la distribution du courrier, ou encore les chaînes de montage en usine. L'augmentation du nombre de travailleurs permet de terminer la tâche plus rapidement. Une limite peut, bien sûr être atteinte, de sorte qu'augmenter encore le nombre de travailleurs, n'apporte plus de gain de temps. En fait, certaines tâches sont purement séquentielles et ne peuvent être exécutées que par une seule personne à la fois. Par exemple, deux marathoniens ne peuvent se partager la distance à parcourir et réclamer la médaille d'or ([Akl85]).

C'est naturellement que la notion de parallélisme a été appliquée aux ordinateurs. De ce fait, il a été possible de répondre aux besoins de puissance nécessaire à la réalisation de projets consommateurs en temps de calculs et en taille mémoire. En fait, le parallélisme peut être défini comme une technique d'accroissement des performances d'un système informatique fondée sur l'utilisation simultanée de plusieurs ressources (processeur, mmoire, disque dur,...). Cela signifie qu'il nécessitera le découpage du problème à résoudre en plusieurs sous-problèmes qui pourront être résolus concurremment par plusieurs processeurs.

Un algorithme est un procédé de résolution de problème énoncé sous la forme d'une série d'opérations à effectuer afin d'obtenir le résultat désiré. Ce mot a comme origine le nom du mathématicien Perse Abu Ja'far Mohammed ibn Ms al-Khowrizmî, auteur d'un livre sur l'arithmétique datant de l'année 825. Le dictionnaire Webster's Ninth New College définit un algorithme comme « une procédure résolvant un problème mathématique en un nombre fini d'étapes qui implique souvent la répétition d'une opération ; ou plus largement : une procédure résolvant étape par étape un problème et aboutissant à une fin ». C'est un jeu de règles ou de procédures bien définies qu'il faut suivre pour obtenir la solution d'un problème dans un nombre fini d'étapes. Un algorithme peut comprendre des procédures et instructions algébriques, arithmétiques, logiques, et autres. Un algorithme peut être simple ou compliqué. Cependant un algorithme doit obtenir une solution en un nombre fini d'étapes. Les algorithmes sont fondamentaux dans la recherche d'une solution par voie d'ordinateur, parce que l'on doit donner à un ordinateur une série d'instructions claires pour conduire à une solution dans un temps raisonnable. Certains algorithmes utilisent des méthodes qui ont été développées bien avant la naissance des ordinateurs mais de nombreux problèmes demandent de nouvelles approches de résolution. La programmation d'un ordinateur demande plus qu'une simple traduction bien connue d'instructions en un langage compréhensible par l'ordinateur.

Le parallélisme combiné à une algorithmique performante permet de gagner du temps afin de répondre au mieux à d'importants besoins. Il rompt avec l'approche classique qui

consiste à gagner de la vitesse en effectuant plus rapidement chaque opération, approche bornée par les lois de la physique. Actuellement, l'emploi du parallélisme est d'abord justifié par des besoins de performance pour satisfaire d'une part, des contraintes de délai qui sont dues à la nature *temps réel* des problèmes tels que la prévision météorologique, l'interprétation de photographies satellites de régions critiques, etc... et d'autre part la taille des problèmes. Citons également la nature intrinsèquement parallèle de certains problèmes qui gagneront en clarté à être programmés selon un modèle parallèle offrant les constructions adéquates. Enfin, signalons le besoin de puissance en tant que phénomène économique : une machine rapide sera plus vite rentabilisée parce qu'elle réalisera plus de travail.

On distingue généralement trois formes de parallélisme. Le *parallélisme de flux* correspond au flux des données sur les unités de calcul, qui sont ainsi utilisées au maximum, et simultanément. Le *parallélisme de contrôle* consiste à découper les traitements à réaliser en entités, appelées *tâches*, et à gérer au mieux le placement de ces tâches sur les ressources de la machine, de sorte que les ressources soient utilisées simultanément et le mieux possible [Sem99a]. Selon Dekeyser et Marquet [DM96], on peut appeler langage à *parallélisme de données* tout langage dans lequel une instruction implique plusieurs traitements identiques sur les données d'un ensemble. Le parallélisme de données offre au programmeur des primitives définies sur des ensembles (vecteurs et matrices le plus couramment), qui se traduisent par la duplication du traitement autant de fois qu'il y a de données dans la structure spécifiée.

Si l'on peut trouver différentes façons de programmer une machine séquentielle, l'unicité du processeur limite cependant les modèles qui pourraient lui être associés. Ainsi, le modèle « optimal » est vite trouvé. Le parallélisme, au contraire, a très largement multiplié les possibilités : nombre de processeurs, agencement, réseau de communication, mémoire partagée ou répartie, contrôle centralisé ou distribué, etc ... La quête du modèle optimal en parallélisme reste encore du domaine de la recherche. En fait, les analyses concordent sur le fait qu'un modèle unique est illusoire, tant la diversité des problèmes et des machines (ou modèles) pour les résoudre est grande.

La notion de parallélisme a donc grandement contribué à la multiplication des modèles informatiques. Nous nous intéresserons dans cette thèse au modèle parallèle à gros grains (voir 2.3.3) baptisé Coarse Grained Multicomputers. Proposé par F. Dehne et al. [DFRC96, DDD+95], ce modèle possède des propriétés qui le rendent très intéressant d'un point de vue pratique. Celui-ci est parfaitement adapté à la modélisation des architectures existantes pour

lesquelles le nombre de processeurs peut être de plusieurs milliers et la taille des données peut atteindre plusieurs milliards d'octets. Ainsi, ces deux paramètres sont suffisants pour définir le modèle CGM (voir section 2.3.3). Un algorithme développé pour ce modèle est constitué de calculs locaux utilisant, si possible, des algorithmes séquentiels optimaux et de rondes de communication dont le nombre doit être indépendant de la taille des données à traiter. Le modèle CGM est donc très intéressant d'un point de vue économique. En effet, ce modèle est indépendant des architectures réelles et permet de réutiliser des algorithmes séquentiels efficaces, ce qui le rend très portable.

Plusieurs travaux ont été réalisés sur ce modèle. On peut citer le tri d'entiers [CD99], la résolution du problème du classement de listes [LG99], le calcul de l'enveloppe convexe et la triangulation d'un ensemble de points [DFRCU99] ou la génération de permutations aléatoires [LT00]. Le domaine des problèmes sur les graphes a également été exploré sur ce modèle. Ainsi, on peut trouver des algorithmes résolvant les problèmes de coloration [GGLGT00], de recherche de bipartition [BCDL99] et de recherche d'un stable maximal [FS99]. Dans [CDF$^+$02], les auteurs décrivent d'autres algorithmes parallèles efficaces sur le modèle CGM pour les graphes. Enfin, on commence à s'intéresser aux comparaisons de séquences biologiques [ACDS03] et à l'algorithmique du texte [ACS03, ACDS02].

Dans cette thèse nous nous intéressons à des problèmes d'algorithmique du texte. Ainsi, nous proposons des solutions CGM aux problèmes de recherche de la plus longue sous-suite croissante, de la plus longue sous-suite commune à deux mots, du plus long suffixe répété en chaque caractère d'un mot et de répétitions. Pour cela, nous sommes partis de solutions systoliques existantes que nous avons adaptées au modèle CGM. Le but de ce travail est en fait double. D'une part, nous proposons pour la première fois des solutions CGM à ces quatre problèmes. D'autre part, nous montrons comment des solutions systoliques peuvent être dérivées en algorithmes CGM. En effet, de nombreux problèmes ont été étudiés sur des architectures systoliques, c'est-à-dire des machines dédiées, non réutilisables pour d'autres problèmes. Le modèle CGM quant à lui permet de travailler avec des machines peu coûteuses et réutilisables à souhaits.

De plus, l'expérience acquise au cours de ces travaux nous permet d'avoir une bonne idée des solutions systoliques adaptables au modèle CGM. Ceci pourrait permettre de consolider le pont existant entre modèles à grain fin et modèles à gros grain. D'autres résultats ont montré comment passer du modèle PRAM aux modèles à gros grain tels que BSP ou CGM (voir [Vie96, FGL98]).

Dans les chapitres suivants, nous verrons comment s'est articulé ce travail. Pour commencer, le chapitre 2 présente des notions sur le parallélisme ainsi que des modèles à grains fin et à gros grain. Le chapitre 3 propose des solutions CGM aux problèmes de recherche de la plus longue sous-suite croissante et de la plus sous-suite commune à deux mots issues des architectures systoliques linéaires unidirectionnelles résolvant ces mêmes problèmes. Le chapitre 4 nous permet de montrer comment dériver des solutions systoliques sur réseaux linéaires bidirectionnels en solution CGM pour les problèmes de recherche du plus long suffixe répété en chaque caractère d'un mot et de répétitions. Une discussion sur l'équilibrage de charge des solutions proposées aux chapitres précédents et sur la prédictivité de l'adaptation d'autres solutions systoliques au modèle CGM est menée dans le chapitre 5. Enfin, quelques remarques et perspectives viennent conclure cette thèse au chapitre 6.

Chapitre 2

Modèles parallèles

2.1 Généralités

2.1.1 Parallélisme

Le petit Larousse 1990 ne définissait pas le parallélisme au sens informatique du terme. De nos jours, cet ouvrage le définit comme une technique d'accroissement des performances d'un système informatique fondée sur l'utilisation simultanée de plusieurs processeurs. Ainsi, le parallélisme rompt avec l'approche classique qui consiste à gagner de la vitesse en effectuant plus rapidemant chaque opération, approche bornée par les lois de la physique.

De nos jours, la technologie permet de réaliser des processeurs ayant une vitesse avoisinant les 3 GHz. Malgré cela, certaines applications restent encore coûteuse en temps de traitement. Il est donc nécessaire de concevoir des systèmes parallèles comportant un nombre important de processeurs (jusqu'à quelques centaines pour des calculateurs à usage général et jusqu'à plusieurs millions pour des processeurs spécialisés) [Kun80a, Lei91, Hwa93, Mol93, FJSD95].

L'informatique parallèle (« parallel processing » ou « high-performance computing ») regroupe à la fois la conception d'algorithmes efficaces en temps, l'architecture de l'ordinateur, la mise en œuvre des programmes et l'analyse des performances, en vue d'améliorer l'outil informatique [Sem99b].

Les problèmes liés au parallélisme peuvent être abordés tant d'un point de vue logiciel que matériel ([Hwa93]) :

1. Détermination de la concurrence (algorithmique parallèle ou « software ») par évaluation de la granularité (réfère le nombre de calculs, d'instructions ou d'opérations élémentaires), mise en place de mécanismes de contrôle assurant l'exécution des programmes (synchronisation) et la gestion des données (géométrie des communications).
2. Projection (« mapping ») des algorithmes parallèles sur des machines spécifiques (architectures parallèles ou « hardware ») : complexité de l'entité de calcul élémentaire, mode opératoire, répartition de la mémoire et réseau d'interconnexion.

Une machine parallèle est constituée d'un ensemble de processeurs capables de coopérer afin de résoudre un problème. Cette définition inclut les super-ordinateurs parallèles ayant des centaines ou des milliers de processeurs, les grappes (clusters en anglais), c'est-à-dire un ensemble de machines (stations de travail ou PC) reliées par un réseau d'interconnexion (RI), ou encore les stations de travail multiprocesseurs.
Les machines parallèles présentent essentiellement deux types de mémoire :
- Machines à mémoire partagée : les processeurs sont connectés par l'intermédiaire d'un RI à une même mémoire qu'ils partagent,
- Machines à mémoire distribuée : chaque processeur possède sa propre mémoire appelée mémoire locale, et les échanges d'informations se font le plus souvent par échanges de messages sur le RI. Les échanges d'informations peuvent aussi se faire par accès mémoire direct à distance : un processeur peut alors accéder directement et manipuler une partie de la mémoire d'un autre processeur.
Le routage détermine le chemin à prendre dans le réseau pour aller d'un processeur source à un processeur destination. Le mode de communication indique la façon dont les messages sont acheminés. On distingue les modes suivants :
- Commutation de circuits : avant d'envoyer son message, la source envoie une requête jusqu'à la destination afin de construire un circuit physique de bout-en-bout. Une fois le circuit établi, le message est alors transmis directement jusqu'à la destination,
- Commutation de paquets : chaque routeur dispose de tampons mémoires, chacun pouvant stocker un paquet. Lors de la réception d'un paquet, le routeur le stocke dans un des tampons avant de le retransmettre au routeur suivant sur la route du paquet. Chaque paquet contient des informations permettant au routeur de calculer le chemin à suivre,
- « Whormhole » : le message est découpé en petites entités appelées « flits » qui sont stockées dans les tampons mémoires des routeurs. Seul le premier « flit » contient des informations permettant au routeur de calculer le chemin à suivre, tandis que les autres

« flits » ne contiennent que des données. Ils doivent donc avancer les uns à la suite des autres dans le réseau.
La communication peut être :
- globale : chaque processeur communique avec tous les autres processeurs,
- locale : un processeur communique avec un ensemble restreint de processeurs,
- point-à-point : un processeur émetteur envoie un message à un processeur destinataire.
Son mode de fonctionnement peut être :
- Synchrone : tous les processeurs travaillent de manière synchrone sous le contrôle d'une horloge commune,
- Asynchrone : chaque processeur travaille sous une horloge spécifique. Dans un tel mode, le programmeur a la responsabilité de mettre des points de synchronisation (appelés barrières de synchronisation) appropriés chaque fois que cela est nécessaire.
Il existe plusieurs façon de programmer une machine parallèle (taxonomie de Flynn [Fly72]) :
- SISD (Single Instruction Single Data) : chaque processeur exécute en même temps la même instruction sur des données uniques. Ce sont en général des processeurs séquentiels d'architecture Von Neumann classique,
- MISD (Multiple Instruction Simple Data) : une séquence de données est envoyée à un premier processeur qui exécute une instruction avant de l'envoyer au suivant (exemple : calculateur systolique),
- SIMD (Single Instruction Multiple Data) : chaque processeur exécute en même temps la même instruction sur des données différentes,
- MIMD (Multiple Instruction Multiple Data) : chaque processeur exécute sur des données différentes ses propres instructions qui peuvent être distinctes d'un processeur à l'autre.
De plus, nous pouvons ajouter à cette liste, la manière de programmer suivante :
- SPMD (Single Program Multiple Data) : les processeurs exécutent en parallèle le même programme sur des données différentes (les instructions de condition du programme peuvent amener les processeurs à exécuter des instructions différentes).

L'algorithmique parallèle est une approche qui a permis de développer des modèles permettant de résoudre un grand nombre de problèmes. On peut classifier ces modèles en deux grandes familles que sont : les modèles à grain fin et les modèles à gros grain. Nous détaillerons plus tard ces différents modèles parallèles en section 2.2 et 2.3.

En ce qui concerne la conception de machines parallèles performantes, l'essentiel de la difficulté consiste donc à définir le « mapping » le plus adéquat possible qui résulte le

plus souvent d'un compromis entre les besoins requis par une application et les contraintes imposées par la technologie.

Les limitations imposées à la réalisation de circuits intégrés intervenant dans la conception de machines spécialisées sont exclusivement liées à l'état de la technologie VLSI [MC80, Ull84, Fru92]. Une *structure calculatoire* (« computing structure ») ne peut être implantée que si elle satisfait un certain nombre de ces contraintes technologiques [Fru92]. Une structure calculatoire désigne ici un réseau de processeurs élémentaires (PE ou cellules), c'est-à-dire un ensemble de processeurs interconnectés par des canaux de communication. Une représentation mathématique d'une telle structure est un graphe dont les nœuds sont les processeurs et les arcs, les liens de communication. Par exemple, un réseau systolique est une structure calculatoire synchrone et régulière qui ne peut contenir qu'un nombre limité de processeurs différents. Le degré d'un nœud doit être constant de même que la distance de communication entre deux processeurs, et l'architecture doit être régulière. Nous y reviendrons en détail en section 2.2.3 (page 13). Pour proposer une alternative à la classique

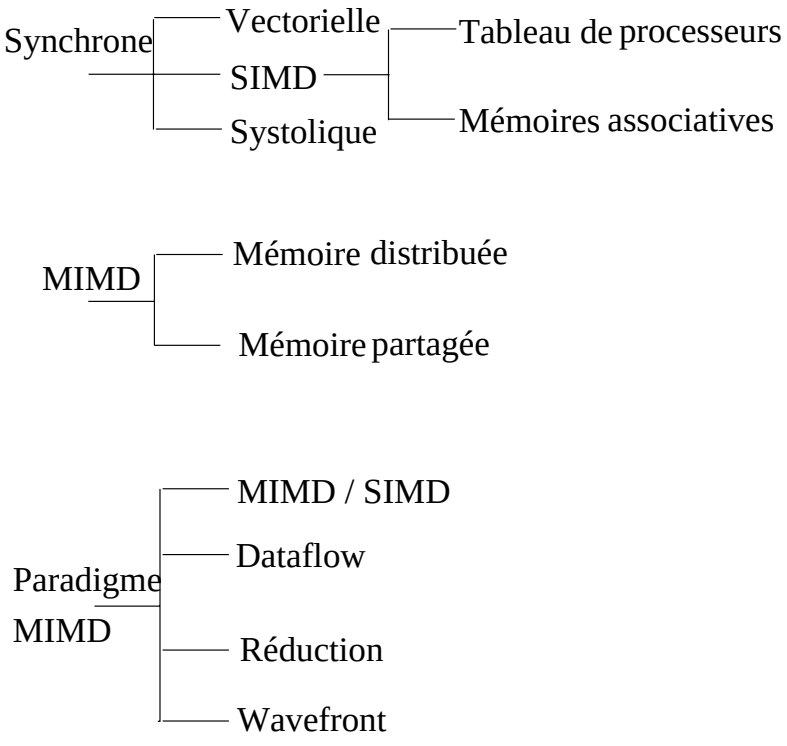


FIG. 2.1 – Paradigme possible des architectures parallèles

(et déjà ancienne) terminologie et taxonomie de Flynn, nous suggérons la lecture de l'article de R. Duncan [Dun90] qui propose une taxonomie présentée sur la figure 2.1 dont la lecture doit pouvoir se faire de façon autonome.

2.1.2 Calcul de la complexité parallèle

Pour étudier la complexité parallèle, on considère des classes d'architectures parallèles très simples qui peuvent être caractérisées par un seul paramètre, P le nombre de processeurs. La complexité parallèle est alors dépendante de deux variables : la taille N du problème et le nombre de processeurs utilisés. On définit le *gain* comme le rapport entre le temps d'exécution en mode séquentiel et le temps d'exécution en parallèle sur P processeurs. On définit l'*efficacité* comme le rapport entre le gain et le nombre de processeurs utilisés. Le *travail* d'un algorithme parallèle est défini comme le produit de la complexité en temps d'un processeur par le nombre P de processeurs. Si le problème est « complètement parallélisable » alors le temps d'exécution sur P processeurs est P fois plus rapide que sur un seul. Dans ce cas, le gain est égal à P et l'efficacité maximale est donc de 1.
Dans la complexité parallèle, on a deux paramètres à gérer : N et P. Deux cas peuvent se produire :
a) Si N est plus grand que P, on place plusieurs éléments sur un même processeur et on les traite en séquence (on dit aussi de manière concurrente). Dans cette méthode dite des processeurs virtuels, la commutation de processeurs peut prendre du temps : si ce temps est une constante ne dépendant ni de N ni de P alors il n'affecte pas la complexité asymptotique. Dans le cas contraire, la perte de performance peut être importante.
b) Si N est plus petit que P, on ne peut pas utiliser tous les processeurs ; dans ce cas, seule l'efficacité est affectée.
En pratique, la taille du problème est généralement très supérieure au nombre de processeur : $N >> P$.

2.2 Modèles à grain fin

Les modèles à grain fin ont été les premiers modèles parallèles à apparaître. La notion de grain fin vient du fait que, pour ces modèles, on suppose que le nombre de processeurs est sensiblement égal au nombre de données d'entrée. Les deux grandes familles de modèles à grain fin sont : les modèles de machines à mémoire partagée et les modèles de machines

à mémoire distribuée.

2.2.1 Modèles à mémoire partagée

Le PRAM (Parallel Random Access Machine) est incontestablement le modèle le plus populaire du calcul parallèle [Akl89, GR88, FW78, KR90]. Comme décrit en figure 2.2, le modèle consiste en un certain nombre de processeurs partageant une mémoire commune. Les processeurs peuvent résoudre un problème calculable en parallèle par l'exécution simultanée des différentes étapes d'un algorithme. La mémoire partagée contient les données et les résultats, et aussi sert de moyen de communication entre les processeurs. Une unité d'interconnexion (RI) alloue aux processeurs l'accès aux emplacements de la mémoire pour lire ou pour écrire.

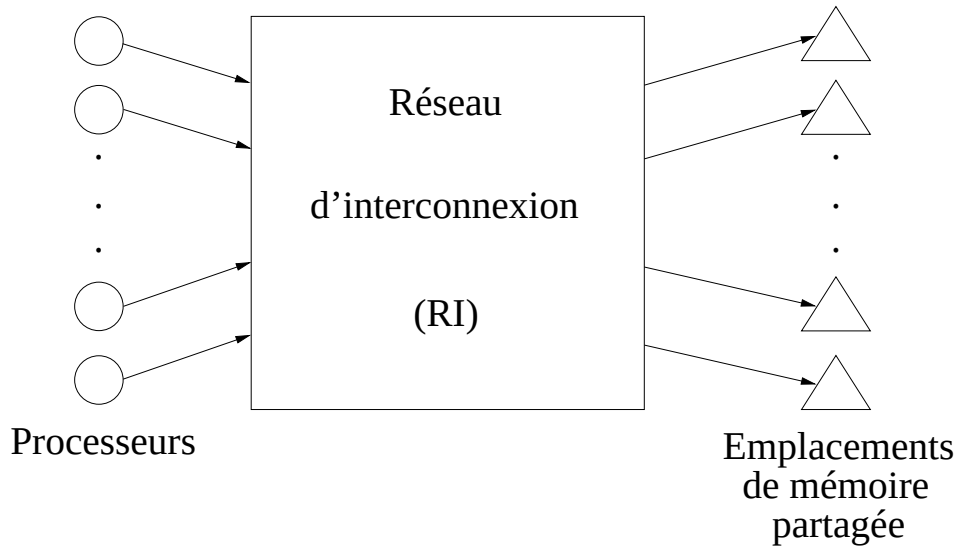


FIG. 2.2 – Le PRAM.

Ce modèle est davantage spécifié en définissant le mode d'accès à la mémoire. Quatre variantes sont le plus souvent utilisées :
- Exclusive-Read Exclusive-Write (EREW) PRAM : dans cette variante, deux processeurs ne peuvent avoir accès au même emplacement mémoire simultanément que ce soit pour lire ou bien pour écrire,

- Concurrent-Read Exclusive-Write (CREW) PRAM : cette variante alloue plus d'un processeur pour lire mais pas pour écrire dans le même emplacement mémoire en même temps,
- Concurrent-Read Concurrent-Write (CRCW) PRAM : pour cette variante, il est possible pour plusieurs processeurs d'avoir accès au même emplacement mémoire pour lire ou pour écrire,
- Exclusive-Read Concurrent-Write (ERCW) PRAM, cette variante a un nombre très restreint d'applications [Akl89].

L'avantage de ce modèle est sa simplicité. Il est, de plus, très utile pour dégager le parallélisme des problèmes étudiés. C'est souvent une première étape pour la parallélisation. Étant donné son haut niveau d'abstraction, il permet de savoir dans quelle mesure un problème peut être parallélisé ou non. Par contre, il est fortement éloigné des machines réelles. En effet, la plupart des machines sont à mémoire distribuée et non à mémoire partagée. De plus, les contraintes technologiques ne permettent pas à un grand nombre de processeurs d'accéder à une mémoire commune en temps constant. C'est pourquoi il n'existe aucune machine PRAM à ce jour.

Une propriété importante du PRAM est que, dans toutes ses variantes, l'accès mémoire est supposé s'effectuer en temps constant. La grande majorité des définitions du modèle présentes dans la littérature ne prend pas en considération le mécanisme nécessaire pour assurer une telle performance de temps d'accès à une mémoire. Par conséquent, pratiquement tous les algorithmes utilisant le PRAM ignorent aussi bien la taille du réseau d'interconnexion reliant les processeurs à la mémoire partagée, que le temps requis pour avoir accès à un emplacement mémoire arbitraire. Il est à noter que le coût d'une telle unité d'interconnexion souvent domine le coût d'exécution du programme [Akl90, Sny86]. Une des méthodes permettant de remédier à cette situation est de simuler le PRAM par un ensemble de processeurs communiquant à travers un réseau. Il n'y a plus de mémoire partagée ; à la place, la mémoire est distribuée entre les processeurs. Des exemples types de ce travail apparaissent dans [AHMP87, Ran87]. Ces travaux ne sont d'ailleurs pas étranger à l'apparition, à la fin des années 80, de modèles parallèles à gros grain à mémoire distribuée.

2.2.2 Modèles à mémoire distribuée

Dans ce modèle, chaque processeur a sa propre mémoire locale de taille constante et il n'existe pas de mémoire partagée. Les processeurs peuvent seulement communiquer grâce à un RI. Comme en PRAM, les processeurs travaillent de manière synchrone. À chaque étape,

chaque processeur peut simultanément envoyer un mot de données à un de ses voisins, recevoir un mot de données d'un de ses voisins et effectuer une opération locale sur ses données. La complexité d'un algorithme est définie comme étant le nombre d'étapes de son exécution. Chaque modèle de mémoire distribuée prend explicitement en compte la topologie du réseau d'interconnexion représentée par différentes caractéristiques :
- le *degré*, qui est le nombre maximal de voisins pour un processeur, il correspond à une sorte de limitation architecturale donnée par le nombre maximum de liens physiques associés à chaque processeur,
- le *diamètre*, qui est la distance maximale entre deux processeurs (la distance étant le plus court chemin dans le réseau d'interconnexion entre ces deux processeurs), il donne une borne inférieure sur la complexité des algorithmes dans lesquels deux processeurs arbitraires doivent communiquer,
- la *bissection*, qui est le nombre minimum de liens à enlever afin de déconnecter le réseau en deux réseaux de même taille (à plus ou moins un processeur près), elle représente une borne inférieure sur le temps d'exécution des algorithmes où il existe une phase qui fait communiquer la moitié des processeurs avec l'autre moitié.

Parmi ces modèles à mémoire distribuée, la *grille* à deux dimensions et l'*hypercube* ont été beaucoup utilisés [Lei91, Fer96]. Plus récemment d'autres réseaux d'interconnexion sont apparut tels que le *honeycomb* [Sto97, CMS99b] et son extension appelé *star-honeycomb* [CMS99a], ou bien encore le *Xmesh* [MS99].

La spécificité des algorithmes développés sur les modèles à mémoire distribuée (si possible efficaces voire optimaux) est un inconvénient majeur. Ainsi, ces algorithmes ne sont pas portables car ils sont fortement dépendants de la topologie choisie.

2.2.3 Modèle systolique

Définitions et propriétés

Les travaux de John Von Newmann [New66], où sont introduits les automates cellulaires, semblent être les premières approches explicites du calcul parallèle. Bien que Von Newmann considéra essentiellement les capacités de réplication des automates cellulaires, l'idée s'est révélée porteuse pour la compréhension de la nature des traitements parallèles. La théorie des automates cellulaires considère les possibilités de calcul d'automates élémentaires connectés par un réseau d'échange des données. Dans ce modèle, un automate peut réaliser

une opération simple et échanger des données avec des automates voisins qui lui sont connectés par les liens du réseau (*cf. e.g.* [QFR87] pour une introduction à ce modèle).

Les automates cellulaires sont aujourd'hui clairement liés au concept de *réseaux systoliques* (« systolic arrays ») [Kun80b, Ull84, QR89, Fru92]. Dans l'article [KL78] H.T. Kung définit un réseau[1] systolique comme : *« un réseau de processeurs qui calculent et échangent des données régulièrement. L'analogie est faite avec la régularité de la contraction cardiaque qui propage le sang dans le système circulatoire du corps. Chaque processeur d'un réseau systolique peut être vu comme un coeur jouant un rôle de pompe sur plusieurs flots le traversant. Le rythme régulier de ces processeurs maintient un flot de données constant à travers tout le réseau ».*

Ce modèle apparaît lorsque les technologies VLSI et WSI [LL82] ont émergées et sa réussite tient en grande partie à son adéquation entre la demande de calculateurs extrêmement rapides à faible coût et la possibilité de réduire très significativement le temps d'exécution de nombre d'algorithmes séquentiels au simple prix d'un accroissement de la complexité du matériel par réplication de structures élémentaires simples et régulières.

Une donnée introduite une seule fois dans le réseau est propagée d'un processeur à un processeur voisin et peut ainsi être utilisée un grand nombre de fois. Cette propriété autorise de gros débits de calcul, même si la cadence des entrées-sorties reste faible. En d'autres termes, on évite les engorgements des buffers d'entrées-sorties. Ceci rend le modèle systolique adéquat pour beaucoup de problèmes dont le nombre de calculs sur une même donnée est largement supérieur à son nombre d'entrées-sorties (« compute-bound problems »).

La principale réussite de H.T. Kung est d'avoir montré que de nombreux algorithmes pouvaient être modélisés à l'aide de ce type de réseaux itératifs synchrones, menant ainsi à la réalisation de circuits intégrés spécialisés qui effectuent des tâches spécifiques sur requête d'un ordinateur permettant des traitements plus généraux (machine hôte). La multiplication d'une matrice par un vecteur, le produit matriciel et la décomposition LU [Kun80b] ayant été implantés avec succès, de nouvelles implémentations ont alors concerné la réalisation de processeurs spécialisés dédiés à des applications de traitement du signal (transformée de Fourier, produits de convolution, filtrage ...).

Dès lors, d'importants efforts ont été fournis pour concevoir des architectures parallèles

[1] On entend clairement par *réseau* la donnée d'un graphe $G = (V, E)$, où V est l'ensemble des processeurs élémentaires et E les arcs associés aux connections inter-processeurs, et d'une application $d : E \to \mathbb{N}$ qui à chaque arc (v_1, v_2) fait correspondre le nombre de liens physiques nécessaires à la réalisation des canaux de communication entre les deux processeurs v_1 et v_2.

basées sur la régularité de la rythmique des données (« dataflow architectures »), pour pipeliner les processeurs ou les vectoriser, ou sur la régularité du flot d'exécution (« wavefront architectures » [Kun84, KLJH87]). Le problème principal du développement de tels ordinateurs réside dans l'organisation des flots intensifs de contrôle et de données assurant de hautes performances[2]. Les architectures pipelinées et « wavefront » sont des variantes systoliques qui sont parmi les plus efficaces. Elles ont ainsi largement contribué à la diffusion du modèle systolique et à sa familiarisation. Un grand nombre de ces structures, encore récentes, représente une solution originale pour l'exécution de certaines classes d'algorithmes complexes qui nécessitent des temps de calculs importants dans les ordinateurs classiques, pourvu que ces algorithmes puissent se décomposer en opérations simples.

Les caractéristiques dominantes d'un réseau systolique peuvent être résumées par un parallélisme massif et décentralisé, par des communications locales et régulières et par un mode opératoire synchrone. Pour décrire un réseau systolique, il est donc nécessaire, tout comme avec un langage tel que VHDL (langage normalisé de spécification pour la réalisation de circuits intégrés [ABOR90]), de spécifier :

- Le graphe d'interconnexion des processeurs (topologie du réseau),
- L'architecture d'un processeur (description des registres et canaux : nom, type, sémantique ...),
- Le programme d'un processeur[3] (lecture des valeurs sur les canaux d'entrée, combinaison d'opérateurs arithmétiques et logiques, mémorisation des résultats dans les registres et écriture sur les canaux de sortie),
- Le flot des données consommées par le réseau pour produire une solution.

Les principaux critères à considérer pour évaluer un réseau systolique seront :

- Les complexités en temps : temps d'exécution de l'algorithme (l'unité de temps étant le nombre de cycles systoliques) et temps d'exécution du programme d'un processeur (*i.e.* niveau d'imbrication maximal définissant le cycle systolique),
- Les complexités en espace (ou surfaces) des architectures envisagées : nombre de processeurs utilisés pour un problème de taille donnée, nombre de registres (et type), nombre de canaux (et leur capacités),
- La régularité et la simplicité des connexions : un réseau est *modulaire* (ou *extensible*) si seuls le temps d'exécution et le nombre de processeurs dépendent de la taille du

[2]Notion de temps réel lorsqu'un résultat est produit à chaque cycle d'horloge.
[3]Par exemple sous forme d'algorithme ou d'un diagramme logique.

problème à traiter ; un réseau est *tolérant aux pannes* si il peut encore être utilisé lorsqu'un ou plusieurs processeurs ne fonctionnent plus,
- La gestion de la synchronisation des processeurs,
- Le nombre de ports d'entrées-sorties (celui-ci devant clairement être minimisé).

Les différents réseaux utilisés

Comme nous venons de le voir une architecture systolique est en fait un réseau composé d'un grand nombre de processeurs élémentaires identiques et localement interconnectés. Les réseaux systoliques les plus couramment utilisés sont représentés Figure 2.3.

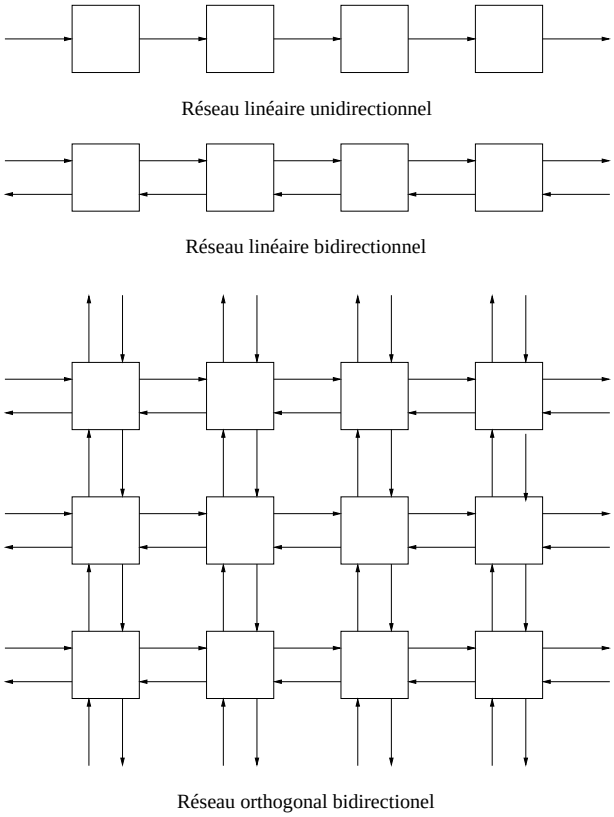


FIG. 2.3 – Architectures des réseaux systoliques les plus utilisées.

Nous utiliserons par la suite uniquement des réseaux linéaires unidirectionnels ou bidirectionnels (Figure 2.3). Mais d'autres topologies peuvent être appliquées aux architectures systoliques comme des réseaux en forme d'anneaux (Figure 2.4) ou des réseaux particuliers

(Figure 2.5). En effet, toutes topologies fortement régulières formant des réseaux d'interconnexion peuvent servir d'architectures systoliques.

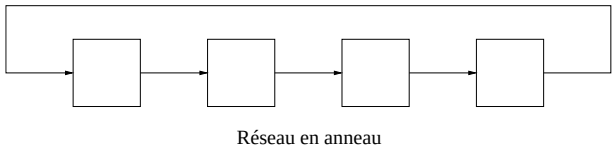


FIG. 2.4 – Réseau systolique en forme d'anneau.

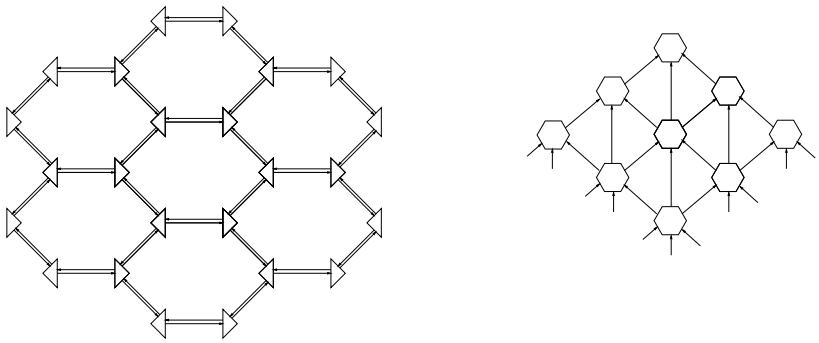


FIG. 2.5 – Autres topologies des réseaux systoliques.

2.3 Modèles à gros grain

La plupart des machines utilisées actuellement sont à gros grain de données, c'est-à-dire que la taille de chaque mémoire locale est beaucoup plus grande que la taille d'une donnée. Ainsi, de nombreux travaux ont permis de décrire des modèles prenant en compte les caractéristiques réelles des machines. Ces modèles reconnaissent, par exemple, l'existence de coût de communications, contrairement au modèle PRAM, sans pour autant spécifier la topologie du support de communications, contrairement aux modèles à grain fin de mémoire distribuée. Le modèle BSP (« Bulk Synchronous Parallel »), premier modèle du genre proposé par Valiant [Val90], formalise les caractéristiques architecturales des machines existantes au moyen de quatre paramètres. Le modèle LogP décrit par Culler et al. [CKP+93] spécifie plus de paramètres que BSP, tandis que CGM (« Coarse Grained Multicomputers »)

proposé par Dehne et al. [DFRC93] est une simplification de BSP.

Ces trois modèles sont basés sur un même modèle de machines : un ensemble de processeurs interconnectés par un réseau. Un processeur peut être une machine monoprocesseur, un processeur d'une machine multiprocesseur ou une machine multiprocesseur elle-même. Le réseau peut être constitué de n'importe quel moyen de communication entre deux processeurs : bus, réseau câblé, mémoire partagée, etc. Pour plus de détails sur tous les modèles décrits dans cette section, nous conseillons la lecture de la thèse de I. Guérin Lassous [GL99].

2.3.1 Modèle BSP

Le modèle BSP formalise les caractéristiques architecturales des machines existantes à l'aide de quatre paramètres. Un algorithme écrit dans le modèle BSP est constitué d'une séquence de super-étapes. Lors d'une super-étape, un processeur peut faire des calculs locaux et un certain nombre de communications (composées d'envois et de réceptions). Entre deux super-étapes consécutives, une barrière de synchronisation, permet comme son nom l'indique, de synchroniser l'exécution sur les processeurs. Le modèle BSP est spécifié à l'aide des paramètres suivants :
- L : période de synchronisation qui correspond à L unités de temps nécessaires pour synchroniser tous les processeurs,
- g : coût pour envoyer un mot à travers le réseau,
- h : nombre maximal de messages que peut envoyer ou recevoir chaque processeur,
- s : surcoût fixe dû à la mise en place d'une communication, aussi petit que soit le message à envoyer.

Ainsi, le coût des communications se calcule à l'aide de ces paramètres. Une phase de communication demande donc un temps $gh + s$. Le coût des calculs locaux correspond à la somme des coûts unitaires de chaque opération élémentaire. La complexité d'une super-étape correspond à la somme du coût de la phase de communication, du coût des calculs locaux et de la période de synchronisation. La complexité d'un algorithme BSP est la somme des coûts des super-étapes le constituant.

2.3.2 Modèle LogP

Le modèle LogP tente de refléter plus précisément les caractéristiques des machines réelles. Le modèle LogP est spécifié à l'aide des paramètres suivants :

- L : latence,
- o : surcoût fixe d'une communication (temps de traitement d'un message par un processeur),
- g : pas (intervalle de temps minimum entre deux envois ou deux réceptions consécutives de messages sur un processeur,
- P : nombre de processeurs.

Pour ce modèle, on suppose qu'au plus $\lceil \frac{L}{g} \rceil$ messages peuvent être envoyés ou reçus par un processeur à chaque étape.

Les processeurs travaillent de manière asynchrone. Contrairement à BSP, la synchronisation est abandonnée et les opérations pour les messages ne sont pas spécifiées (ce ne sont pas nécessairement des envois et des réceptions). Les communications entre les processeurs sont de type point-à-point.

Comme le modèle LogP suppose que seuls des messages élémentaires peuvent être échangés entre les processeurs, plusieurs extensions ont été proposées, comme LogGP [A. 95] ou LogP étendu [LZE97, KAP95], afin de permettre des envois de messages plus longs, ce qui permet d'amortir les coûts d'initialisation des communications.

Dans [BHP+96], les auteurs montrent que les modèles BSP et LogP sont équivalents d'un point de vue asymptotique, bien que le modèle BSP soit plus simple d'utilisation.

2.3.3 Modèle CGM

Le modèle CGM (Coarse Grained Multicomputers) a été introduit en 1993 [DFRC93]. CGM a été largement étudié dans [BCDL99, CD99, DFRC96, DDD+95, DFRCU99, FS99, KP00]. Par rapport au modèle BSP, il s'affranchit des paramètres L,g,h et s, ainsi que de l'étape de synchronisation. En effet, ce modèle n'utilise que deux paramètres : P qui est le nombre de processeurs utilisés et N qui représente le nombre de données d'entrée du problème. Dans le modèle CGM, P doit être nettement inférieur à N ($P << N$). Ce modèle représente beaucoup mieux les architectures existantes composées de plusieurs milliers de processeurs et qui peuvent traiter des millions ou des milliards de données.

De plus, le modèle donne explicitement le nombre de données par processeur. Par la suite, nous considèrerons le nombre de données fournies aux processeurs dans nos problèmes, de taille $\frac{N}{P} \geq P$. Les algorithme écrits dans ce modèle sont composés d'une succession de deux phases :

- une phase où chaque processeur effectue un calcul sur ses données locales,

- une phase où les processeurs échangent des données afin de les redistribuer.

Pendant cette seconde phase, chaque processeur peut envoyer $O(\frac{N}{P})$ données et en recevoir $O(\frac{N}{P})$. Ainsi, dans un modèle CGM à P processeurs, chacun des processeurs a une mémoire locale de taille en $O(\frac{N}{P})$.

La communication est globale entre les processeurs et n'importe quel réseau d'interconnexion peut être utilisé. Le modèle cherche à obtenir un nombre de rondes de communications qui soit le plus petit possible ($\log P$). En aucun cas ce nombre ne doit dépendre de la taille du problème car les performances s'en trouveraient dégradées.

En application du modèle PRAM, on considère que le nombre de super-étapes devrait être polylogarithmique en P, mais cela semble être loin de la réalité. En fait, les algorithmes qui assurent simplement un nombre de super-étapes en fonction de P (et non de N) fonctionnent bien dans la pratique, voir Goudreau et al. [GKLST96]. En effet, comme le nombre de super-étapes dépent du nombre de rondes de communication et que chaque processeur doit, dans le pire des cas, travailler sur l'ensemble des données, ce nombre de super-étapes doit être au plus polylogarithmique en P ou fonction de P.

2.4 Choix du modèle CGM

L'aspect peu contraignant, réaliste et portable du modèle CGM permet d'implanter nos algorithmes et de les expérimenter sur des grappes de machines diverses et variées. En effet, le modèle CGM permet de s'affranchir des paramètres présents dans BSP et LogP, d'obtenir des solutions facilement implantables et de concevoir des algorithmes qui ne sont pas dépendants d'une architecture donnée.

De plus, il permet de réutiliser des algorithmes optimaux lors des calculs locaux. Ceci permet d'abaisser le coût du développement qui n'est pas négligeable lors d'implantations parallèles. Si la machine ne contient que des monoprocesseurs, les algorithmes séquentiels optimaux pourront être utilisés.

C'est donc un modèle « moderne » adapté aux différentes plate-formes existantes. Loin d'être un modèle purement théorique, il permet, en effet, de réconcilier théorie et pratique.

Chapitre 3

Problèmes sur réseau linéaire unidirectionnel

3.1 Introduction

Ce chapitre propose des solutions CGM aux problèmes de recherche de la plus longue sous-suite croissante et de la plus longue sous-suite commune à deux mots issues des architectures systoliques linéaires unidirectionnelles résolvant ces mêmes problèmes. Ce type d'architecture systolique est représenté sous la forme d'un réseau linéaire de processeurs à travers lesquels les communications se font dans un seul sens (voir figure 3.1) :

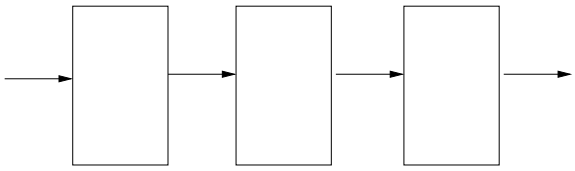


FIG. 3.1 – Réseau systolique linéaire unidirectionnel

Chaque processeur contient des registres de taille constante, lit les donnés provenant des canaux d'entrée, exécute son programme interne composé d'opérations simples et envoie les données calculées vers les canaux de sortie (voir Section 2.2.3).

Chaque problème sera présenté de la manière suivante : nous présenterons l'intérêt pratique du problème, la complexité des solutions séquentielles, la description de la solution

systolique, la description de la solution CGM, la complexité de la solution CGM et enfin, nous donnerons des résultats expérimentaux.

3.2 Plus longue sous-suite croissante

Ce chapitre présente une solution de la plus longue sous-suite croissante sur le modèle CGM. L'algorithme peut être implanté en CGM sur P processeurs avec une complexité de $O(\frac{N^2}{P})$ en temps et P rondes de communication. Après la description du problème, nous donnerons la complexité des solutions séquentielles et systolique. Nous proposerons notre solution CGM et nous expliquerons sa complexité.

3.2.1 Description du problème

Introduction

Le problème de la plus longue sous-suite croissante (Longest Increasing Subsequence - LIS) a intéressé beaucoup de scientifiques [CDM93, P. 35, Fre75, JV92, Mis78, Szy75, GMS01]. En juillet 1978, E.W. Dijkstra qui enseignait à Marktoberdof's School, demanda à ses étudiants de trouver la longueur de la plus longue sous-suite croissante d'une séquence d'entiers. Bien que ce problème semblait relativement simple, peu d'entre eux avaient été capables de le résoudre. Aujourd'hui, cet exercice reste un exemple didactique utile en méthodologie de programmation séquentielle. En particulier, il montre comment renforcer une hypothèse d'induction d'une façon très explicite [Gra88, Gri89, Man89].

Intérêt du problème

L'extraction d'une plus longue sous-suite croissante peut améliorer la compression de données qui est très utilisée dans la transmission, le traitement d'images ou le traitement de données médicales.

Au paragraphe suivant, nous donnerons d'une part, les définitions de bases du problème LIS et d'autre part, nous poserons les fondements de notre solution CGM par une définition du problème basée sur la programmation dynamique.

Définitions de base et notations

Définition 3.1 *Étant donnée une suite A de N entiers distincts, une sous-suite de A est une suite L qui peut être obtenue à partir de A en supprimant zéro ou plusieurs entiers (non nécessairement consécutifs).*

Définition 3.2 *Une sous-suite est croissante si chaque entier de cette suite est plus grand que l'entier précédent. Étant donnée une suite $A = \{x_1, x_2, \ldots, x_N\}$ de N entiers distincts, nous définissons une sous-suite croissante de longueur l comme une sous-suite croissante de A : $\{x_{i_1}, x_{i_2}, \ldots, x_{i_l}\}$ avec $\forall$ j,k : $1 \leq j < k \leq l => i_j < i_k$ et $x_{i_j} < x_{i_k}$.*

Définition 3.3 *Une plus longue sous-suite croissante est une sous-suite croissante de longueur maximale. Il est à noter qu'une plus longue sous-suite croissante n'est pas nécessairement unique.*

Programmation dynamique

Les définitions et les propositions suivantes vont nous permettre de résoudre le problème LIS par la programmation dynamique.

Définition 3.4 *Une sous-suite est décroissante si chaque entier de cette suite est plus petit que l'entier précédent.*

Définition 3.5 *Une couverture de A est un ensemble de sous-suites décroissantes de A qui contient tous les nombres de A.*

Définition 3.6 *La taille d'une couverture est le nombre de sous-suites décroissantes la composant, et une plus petite couverture est une couverture de taille minimum parmi toutes les couvertures.*

Lemme 3.1 *Si I est une sous-suite croissante de A de longueur égale à la taille de la couverture de A, nommée C, alors I est une plus longue sous-suite croissante de A et C est un couverture minimale de A.*

Preuve : voir [Gus97]. □

Nous pouvons définir un algorithme séquentiel à partir du lemme 3.1, qui sera la base de notre algorithme CGM. Soit A un ensemble de N entiers. Nous voulons construire une couverture décroissante de A. L'idée est la suivante : en démarrant de la gauche de A (contenu

dans un tableau), on examine chaque entier de A et on le place à la fin de la première sous-suite décroissante la plus à gauche qu'il peut étendre. S'il ne peut étendre aucune sous-suite décroissante, on commence une nouvelle sous-suite décroissante. Cet algorithme produit une couverture de A qui est appelée la « greedy » couverture dans [Gus97]. Après que cette couverture soit trouvée, une LIS de A peut être trouvée facilement comme il est décrit dans [Gus97]. À la fin de l'algorithme décrit dans [Gus97], I contient une LIS de A. La « greedy » couverture de A est trouvée en $O(N^2)$ en temps et la LIS est trouvée en $O(N)$ en temps grâce à la « greedy » couverture.

Théorème 3.1 *La i-ème sous-suite de la « greedy » couverture contient tout élément de A qui a la i-ème position dans la plus longue sous-suite croissante le contenant.*

Preuve : voir [Gus97]. □

Trouver la position d'un élément dans la plus longue sous-suite croissante le contenant peut être réalisé par la proposition suivante. Au final, le vecteur $Major$ contiendra toutes les longueurs des plus longues sous-suite croissante en chaque caractère de la suite A.

Proposition 3.1
$\forall i$, $Major[i]$=*1*
$\forall i, j/j < i$, $Major[i]$=*max(*$Major[j] + 1$*) si* $A[j] < A[i]$.

Preuve :
$Major[i_k] = Major[i_{k-1}] + 1$ avec $A[i_{k-1}] < A[i_k]$ et $\exists i_k, i_{k-1}/i_{k-1} < i_k$
$Major[i_{k-1}] = Major[i_{k-2}] + 1$ avec $A[i_{k-2}] < A[i_{k-1}]$ et $\exists i_{k-1}, i_{k-2}/i_{k-2} < i_{k-1}$
$\ldots$ $\ldots$
$\ldots$ $\ldots$
$\ldots$ $\ldots$
$Major[i_2] = Major[i_1] + 1$ avec $A[i_1] < A[i_2]$ et $\exists i_2, i_1/i_1 < i_2$
Donc,
$Major[i_k] = k$ avec $A[i_1] < A[i_1] < \cdots < A[i_{k-2}] < A[i_{k-1}] < A[i_k]$ et $i_1 < i_2 < \cdots < i_{k-2} < i_{k-1} < i_k$
Comme nous cherchons une plus longue sous-suite croissante, k doit être maximum (i.e. on cherche à étendre une sous-suite croissante de longueur $k-1$). Ainsi, la position de l'élément $A[i_k]$, dans la plus longue sous-suite croissante le contenant, est k. □

Un exemple de calcul du vecteur $Major$ est donné ci-dessous (table 3.2.1).

i	1	2	3	4	5	6	7	8
$A[i]$	10	2	5	6	1	13	15	1
$Major[i]$	1	1	2	3	1	4	5	1

TAB. 3.1 – Exemple de construction du vecteur $Major$ pour la suite A=10,2,5,6,1,13,15,1.

3.2.2 Complexité des solutions séquentielles

Le problème consistant à trouver la longueur d'une LIS peut être résolu de manière séquentielle en $O(N \log N)$ en temps (où N est la longueur de la sous-suite) [Man89]. Pour le problème LIS, il existe un algorithme séquentiel qui fonctionne en $O(N^2)$ en temps [Gus97] et en $O(N \log N)$ en utilisant les arbres binaires de recherche équilibrés.

3.2.3 Description de la solution systolique

Un réseau systolique efficace pour résoudre le problème LIS a été proposé dans [CDM92]. Cette solution est basée sur les propositions définies au paragraphe 3.2.1. Elle est basée sur une architecture systolique linéaire unidirectionnelle (voir figure 3.2) composée de cellules contenant trois registres et quatre canaux de communication (voir figure 3.3).

Algorithme

Soit une sous-suite de N entiers, on considère un réseau linéaire de N cellules nommées $C(1) \ldots C(N)$, reliées par quatre canaux unidirectionnels (voir figure 3.2). Trois canaux sont utilisés pour transmettre les entiers et le quatrième est utilisé pour transmettre un signal de contrôle. Ce signal peut prendre une des valeurs suivantes : (M, S, L, I) qui seront détaillées par la suite. À travers ces canaux, les informations voyagent d'un processeur à son voisin en un cycle d'horloge. Enfin, chaque cellule a une mémoire locale composée de trois registres de type entier (voir figure 3.3). En fait, chaque cellule $C(i)$ correspond au i-ième élément d'une liste appelée $Major$ (voir 3.2.1).

Les éléments x_i avec $i = 1, 2, ..., N$ sont introduits dans la cellule $C(1)$ un par un à chaque cycle d'horloge. Deux canaux de communication sont utilisés : $In.Val$ pour la

FIG. 3.2 – Réseau linéaire unidirectionnel de N processeurs

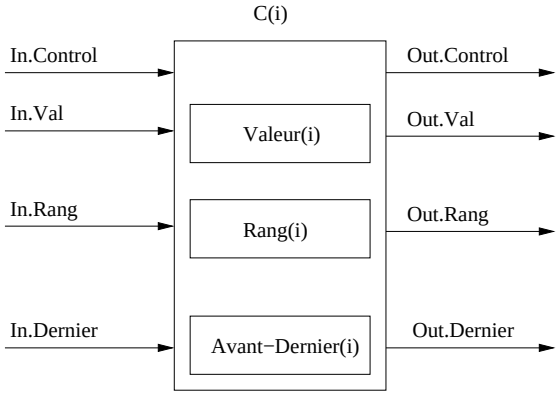


FIG. 3.3 – Processeur élémentaire

valeur et $In.Rang$ pour le rang. En même temps, un élément de contrôle est envoyé par le canal $In.Control$ à $C(1)$ afin d'avertir la cellule que c'est le premier élément du tableau. $C(1)$ est la seule cellule qui recevra ce signal. La construction du $Major$ est effectuée en trois étapes. Premièrement, la position d'insertion de l'élément courant dans $Major$ doit être déterminée, puis cette position doit être localisée dans la séquence et enfin, l'élément doit être inséré au bon endroit. De plus, le premier processeur a un travail d'initialisation supplémentaire à réaliser. Trois algorithmes ont été développés : Recherche, Pointage et Insertion et un algorithme spécial est développé pour le premier processeur : Maître. On considére les opérations effectuées sur un élément qui accroit une sous-suite croissante de longueur l. Cet élément devra être inséré à droite après $Dernier(l-1)$ dans la liste des $Major$.

Insertion

Ce calcul est initialisé avec la cellule correspondante à $Dernier(l-1)$. On définit un signal

de contrôle appelé I. On suppose que :

- $Valeur(i)$ avec $i > 1$ est initialisé à 0 ;
- $C(i)$ reçoit la valeur et le rang d'un élément provenant des canaux $In.Valeur$ et $In.Rang$.

Insertion :
dans le cas où $In.Control$ **est**
« I » :
si $Valeur(i) \neq 0$ **alors**
$Out.Valeur = Valeur(i)$
$Out.Rang = Rang(i)$
$Out.Control =$« I »
finsi
$Valeur(i) = In.Valeur$
$Rang(i) = In.Rang$
Ainsi, tous les éléments de la liste sont décalés vers la droite aussitôt qu'une insertion est réalisée.

Pointage
Dans le paragraphe précédent, on a vu que l'insertion est demandée par la cellule contenant l'élément $Dernier(l-1)$. On décrit dans cette partie comment une cellule reconnait son éligibilité. $C(i)$ comprend que la séquence essaye de localiser $Dernier(l-1)$ quand il reçoit le signal de contrôle « L ». À cet instant, la cellule peut déterminer si elle correspond à la position demandée. On suppose que :

1. $C(i)$ reçoit la valeur $Dernier(i)$ provenant du canal $In.Dernier$;
2. $C(i)$ reçoit la valeur et le rang d'un élément provenant des canaux $In.Valeur$ et $In.Rang$.

Pointage :
dans le cas où $In.Control$ **est**
« L » :
si $Valeur(i) = In.Dernier$ **alors**
($C(i)$ est l'endroit souhaité)
$Out.Control =$« I »
sinon

(Essayer la prochaine cellule)
$Out.Control =$« L »
$Out.Dernier = In.Dernier$
finsi
$Out.Valeur = In.Valeur$
$Out, Rang = In.Rang$

Recherche

En premier, le réseau doit déterminer l, la longueur de la sous-suite croissante en examinant les différents $Dernier(i)$ jusqu'à $Dernier(l)$. Puis, un signal « L » est envoyé avec la valeur de $Dernier(l-1)$ pour retrouver la position de l'insertion. Malheureusement, $Dernier(l)$ arrive après $Dernier(l-1)$ dans la liste $Major$. En effet, les canaux sont unidirectionnels et fonctionnent de la gauche vers la droite. Il est impossible d'envoyer un élément à travers la cellule correspondante au $Dernier(l)$, revenir en arrière jusqu'à trouver la cellule correspondante à $Dernier(l-1)$ et réaliser l'insertion. Cependant, trouver l est réalisé avant que l'élément ne dépasse la cellule correspondante à $Dernier(l-1)$ et en outre, strictement avant que l'élément n'atteigne la cellule correspondante à $Dernier(l)$. On introduit donc une nouvelle variable $Avant\text{-}Dernier(i)$ qui contient la valeur de $Dernier(i+1)$. Maintenant, quand l'élément atteint $C(l-1)$, il est comparé avec $Avant\text{-}Dernier(l-1)$ qui est la valeur de $Dernier(l)$. On définit un signal de contrôle appelé « S ». On suppose que :

1. $Avant\text{-}Dernier(i)$ avec $i > 1$ est initialisé à l'infini ;
2. $C(i)$ reçoit la valeur et le rang d'un élément provenant des canaux $In.Valeur$ et $In.Rang$;
3. $C(i)$ reçoit $Avant\text{-}Dernier(i-1) \equiv Dernier(i)$ du canal $In.Dernier$.

Recherche :
dans le cas où $In.Control$ **est**
« S » :
si $In.Val > Avant\text{-}Dernier(i)$ **alors**
(Rechercher encore)
$Out.Control =$« S »
$Out.Dernier = Avant\text{-}Dernier(i)$
sinon (Nous accroissons la liste finissant par $Avant\text{-}Dernier(i) \equiv Dernier(i+1)$)
$Avant\text{-}Dernier(i) = In.Valeur$

Si $Valeur(i) = In.Dernier$ **alors**
Insertion
$Out.Control =$« I »
sinon
Pointage
$Out.Control =$« L »
$Out.Dernier = In.Dernier$
finsi
finsi
$Out.Val = In.Valeur$
$Out.Rang = In.Rang$

Maître

Le premier processeur doit s'occuper de $Lis(1)$ et $Lis(2)$. $C(1)$ reçoit un signal de contrôle « M » et un élément en entrée. Nous supposons que :

1. $Valeur(1)$ et $Avant\text{-}Dernier(1)$ sont initialisés à l'infini ;
2. $C(1)$ reçoit la valeur et le rang d'un élément provenant des canaux $In.Valeur$ et $In.Rang$.

Maître :
dans le cas où $In.Control$ **est**
"M" :
si $In.Val < Valeur(i)$ **alors**
(Nouveau $Lis(1)$)
$Out.Val = Valeur(i)$
$Out.Rang = Rang(i)$
$Valeur(i) = In.Val$
$Rang(i) = In.Rang$
$Out.Control =$« I«
sinon
si $In.Val < Avant - Dernier(i)$ **alors**
(Nouveau $Lis(2)$)
$Avant\text{-}Dernier(i) = In.Val$
$Out.Control =$« I »

sinon
Recherche d'une nouvelle sous-suite plus prométeuse
$Out.Last$=$Avant\text{-}Dernier(i)$
$Out.Control$ =« S »
finsi
finsi
$Out.Val = In.Valeur$
$Out.Rang = In.Rang$

Algorithme d'un processeur
Nous pouvons grouper les différentes procédures citées ci-dessus afin d'obtenir le programme exécuté par une cellule $C(i)$ à chaque cycle d'horloge.

Lire$(In.Control, In.Val, In.Rang, In.Dernier)$;
dans le cas où $In.Control$ **est**
« M » : **Maître** ;
« S » : **Recherche** ;
« L » : **Pointage** ;
« I » : **Insertion** ;
Envoyer$(Out.Control, Out.Val, Out.Rang, Out.Dernier)$;

Complexité

La complexité de cette solution systolique est de $O(N)$ en temps avec N processeurs dotés de 4 registres et de 4 canaux de communication unidirectionnels. Cette complexité induit un travail en $O(N^2)$. Notre but va être de proposer une solution CGM ayant le même travail également basée sur les propositions de programmation dynamique du paragraphe 3.2.1.

3.2.4 Description de la solution CGM

La solution CGM que nous présentons dans ce chapitre est issue de l'article [GMS01]. Notre algorithme CGM est basé sur trois algorithmes séquentiels. Chaque algorithme séquentiel représente une phase de l'algorithme CGM.

Algorithmes séquentiels

L'Algorithme 1 est basé sur la proposition 3.1 et trouve la position de chaque élément $A[i]$ de A dans la plus longue sous-suite croissante le contenant.

Algorithme 1

(1) **pour** $(i = 0)$ *à* $(i = N - 1)$
  $Major[i] = 1$
 finpour

(2) **pour** $(i = 1)$ *à* $(i = N - 1)$
  pour $(j = 0)$ *à* $(j = i)$
   si $(A[j] < A[i]$ **et** $Major[i] < Major[j] + 1)$ **alors**
   $Major[i] = Major[j] + 1$
   finsi
  finpour
 finpour

Algorithme 2

L'algorithme séquentiel suivant calcule une LIS à partir du résultat de l'algorithme 1 qui calcule pour chaque élément $A[i]$ $(\forall i / 0 \leq i < N)$ de A sa position (i.e. $Major[i]$) dans la LIS le contenant (i.e. $A[i]$). Ensuite, nous devons trouver le maximum, appelé Max, entre tous les $Major[i]$ $(\forall i / 0 \leq i < N)$. Nous appelons Ind l'index minimum tel que $Max = Major[Ind]$. Ces opérations sont réalisées par la première partie de l'algorithme 2. En démarrant de $Major[Ind]$, qui représente à la fois la longueur de la LIS et la position de l'élément $A[Ind]$ dans la LIS, nous pouvons facilement extraire une LIS (sauvegardée dans le tableau $RLIS$) en ordre inverse de la sous-suite A. Cette opération est réalisée par la seconde partie de l'algorithme 2.

(1) $Max = 0$
 $Ind = 0$
 pour $(i = 0)$ *à* $(i = N - 1)$
  si $(Major[i] >= Max)$ **alors**
   $Max = Major[i]$

$Ind = i$
finsi
finpour

(2) $j = 0$
pour $(i = N - 1)$ *à* $(i = 0)$
si $(Major[i] = Max')$ **alors**
$RLIS[j] = A[i]$
$Max = Max - 1$
$j = j + 1$
finsi
finpour

Remarque : Le tableau $RLIS$ contient une LIS en ordre inverse. Il est simple de construire une LIS à partir du tableau $RLIS$ en temps linéaire.

Algorithme 3
En partageant la sous-suite A, nous pouvons définir un autre algorithme séquentiel basé sur les précédents (algorithmes 1 et 2). La sous-suite A est partagée en $\frac{N}{P}$ parties (avec $P = \frac{N}{P}$). Aussi, nous devons ajouter des boucles dans les précédents algorithmes 1 et 2 pour obtenir l'algorithme séquentiel 3. Il est évident qu'en partageant la sous-suite A en $\frac{N}{P}$ parties, nous pourrons écrire un algorithme CGM à partir de cet algorithme séquentiel.

(1) **pour** $(num = 0)$ *à* $(num = P - 1)$
pour $(i = 0)$ *à* $(i = N/P - 1)$
$Major[num * N/P + i] = 1$
finpour
finpour

(2) **pour** $(k = 0)$ *à* $(k = P - 2)$
pour $(num = 0)$ *à* $(num = P - 1)$
si $(num = k)$ **alors**
pour $(i = 1)$ *à* $(i = N/P - 1)$
pour $(j = 0)$ *à* $(j = i)$

si $(A[num * N/P + j] < A[num * N/P + i])$
et $Major[num * N/P + i] < Major[num * N/P + j] + 1)$ **alors**
$Major[num * N/P + i] = Major[num * N/P + j] + 1$
finsi
finpour
finpour
sinon
si $(num > k)$ **alors**
pour $(i = 0$ *à* $(i = N/P - 1)$
pour $(j = 0)$ *à* $(j = N/P - 1)$
si $(A[k * N/P + j] < A[num * N/P + i]$
et $Major[num * N/P + i] < Major[k * N/P + j] + 1)$ **alors**
$Major[num * N/P + i] = Major[k * N/P + j] + 1$
finsi
finpour
finpour
finsi
finsi
finpour
finpour

(3) $Max = 0$
pour $(num = 0)$ *à* $(num = P - 1)$
pour $(i = 0)$ *à* $(i = N/P - 1)$
si $(Major[num * N/P + i] >= Max)$ **alors**
$Max = Major[num * N/P + i]$
finsi
finpour
finpour

(4) $j = 0$
pour $(num = P - 1)$ *à* $(num = 0)$
pour $(i = N/P - 1)$ *à* $(i = 0)$

si $(Major[num * N/P + i] = Max)$ **alors**
$RLIS[j] = A[num * N/P + i]$
$Max = Max - 1$
$j = j + 1$
finsi
finpour
finpour

Algorithme CGM

L'Algorithme LIS_CGM présente la solution CGM pour le problème de la LIS. Chaque processeur num $(1 \leq num \leq P)$ a la num-ième partie de $\frac{N}{P}$ éléments de la suite A. L'algorithme CGM suivant présente le programme pour chaque processeur num. Cet algorithme utilise l'algorithme séquentiel 3 décrit précédemment. La figure 3.4 décrit les rondes de communication utilisées dans l'algorithme CGM pour le calcul des positions d'une LIS. La figure 3.5 décrit les rondes de communication utilisées pour le calcul du Max et enfin, la figure 3.6 décrit les rondes de communication nécessaire pour retrouver une LIS en ordre inverse. En fait, on parcourt le réseau en sens inverse (de droite à gauche).

Algorithme LIS_CGM

(1) **pour** $(i = 0)$ *à* $(i = N/P - 1)$
$Major[num * N/P + i] = 1$
finpour

(2) **pour** $(k = 0)$ *à* $(k = P - 2)$
si $(num = k)$ **alors**
pour $(i = 1)$ *à* $(i = N/P - 1)$
pour $(j = 0)$ *à* $(j = i)$
si $(A[j] < A[i]$ **et** $Major[i] < Major[j] + 1)$ **alors**
$Major[i] = Major[j] + 1$
finsi
finpour
finpour
Envoyer $(num, A, tous\_les\_autres)$

Envoyer (num,$Major$,$tous\_les\_autres$)
sinon
Recevoir (num',A')
Recevoir (num',$Major'$)
si ($num > num'$) **alors**
pour ($i = 0$) à ($i = N/P - 1$)
pour ($j = 0$) à ($j = N/P - 1$)
si ($A'[j] < A[i]$ **et** $Major[i] < Major'[j] + 1$) **alors**
$Major[i] = Major'[j] + 1$
finsi
finpour
finpour
finsi
finsi
finpour

(3) $Max = 0$
pour ($i = 0$) à ($i = N/P - 1$)
si ($Major[i] >= Max$) **alors**
$Max = Major[i]$
finsi
finpour
Envoyer (num,Max,$tous$)
pour ($i = 0$) à ($i = P - 1$)
Recevoir (num',Max')
$Max\_proc[num'] = Max'$
finpour
$Max' = Max\_proc[0]$
pour($i = 1$) à ($i = P - 1$)
si ($Max' < Max\_proc[i]$) **alors**
$Max' = Max\_proc[i]$
finsi
finpour

(4) **si** $(num = P - 1)$ **alors**
  $j = 0$
  pour $(i = N/P - 1)$ *à* $(i = 0)$
    si $(Major[i] = Max')$ **alors**
      $RLIS[j] = A[i]$
      $Max' = Max' - 1$
      $j = j + 1$
    finsi
  finpour
  Envoyer $(num, Max', num - 1)$
sinon
  Recevoir(num', Max')
  $j = 0$
  pour $(i = N/P - 1)$ *à* $(i = 0)$
    si $(Major[i] = Max')$ **alors**
      $RLIS[j] = A[i]$
      $Max' = Max' - 1$
      $j = j + 1$
    finsi
  finpour
  Envoyer $(num, Max', num - 1)$
finsi

Remarque : L'algorithme LIS_CGM utilise deux fonctions nommées Envoyer et Recevoir qui sont définies de la façon suivante :
– **Envoyer** $(num, Max, tous)$ où les valeurs num (le numéro du processeur) et Max sont envoyées à tous les processeurs,
– **Envoyer** $(num, A, tous\_les\_autres)$ où les valeurs num et A sont envoyées à tous les processeurs excepté le processeur num,
– **Recevoir** (num', A') où les valeurs num' et A' sont reçues par le processeur num'.

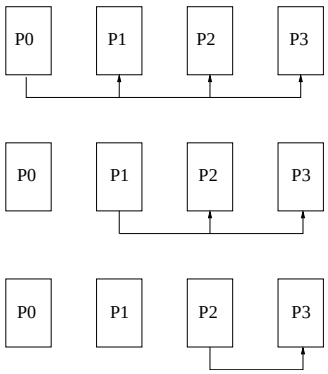


FIG. 3.4 – Rondes de communication 1 pour le problème LIS (pour 4 processeurs).

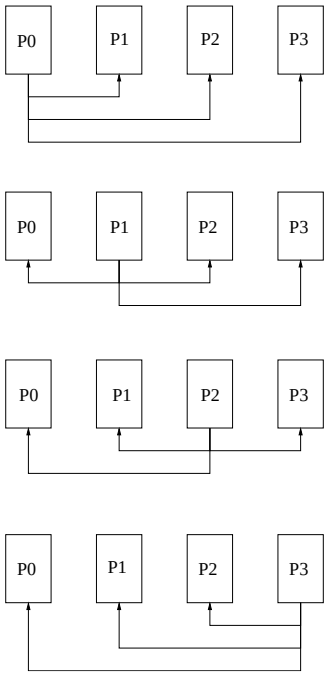


FIG. 3.5 – Rondes de communication 2 pour le problème LIS (pour 4 processeurs).

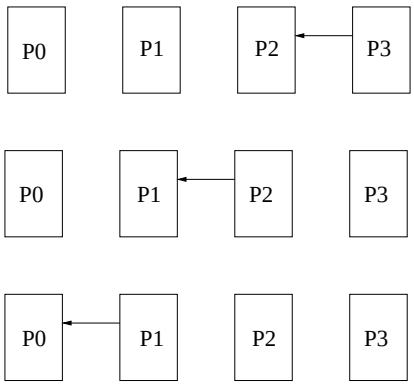


FIG. 3.6 – Rondes de communication 3 pour le problème LIS (pour 4 processeurs).

3.2.5 Complexité de la solution CGM

L'algorithme 3 de la section précédente 3.2.4 est composé de quatre parties. La première partie est une initialisation, elle effectue N opérations. La seconde partie est issue de l'algorithme 1 basé sur la proposition 3.1. Cette partie effectue $N \times N$ opérations. Les troisième et quatrième parties, qui représentent respectivement la recherche du maximum des positions d'un élément dans la plus longue sous-suite croissante le contenant et la construction d'une LIS en ordre inverse effectuent également N opérations. L'algorithme CGM_LIS a été developpé pour le modèle CGM utilisant P processeurs et $\frac{N}{P}$ données. Nous avons donc pour la première partie une complexité de $O(\frac{N}{P})$ en temps, pour la seconde partie une complexité de $O((\frac{N}{P})^2)$ et pour les deux dernières parties une complexité de $O(\frac{N}{P})$. L'algorithme CGM_LIS a donc une complexité globale de $O(\frac{N^2}{P})$ en temps. De plus, un processeur va effectuer au maximum P rondes de communication.
Cette complexité induit donc un travail en $O(N^2)$ (avec P processeurs) qui est équivalent au travail de la solution systolique citée au paragraphe 3.2.3.

3.3 Plus longue sous-suite commune à deux mots

Ce chapitre présente une solution de la plus longue sous-suite commune à deux mots sur le modèle CGM. L'algorithme peut être implanté en CGM sur P processeurs avec une complexité de $O(\frac{N^2}{P})$ en temps et P rondes de communication. Après la description du problème, nous donnerons la complexité des solutions séquentielles et systolique. Nous

proposerons notre solution CGM et nous expliquerons sa complexité.

3.3.1 Description et intérêt du problème

Introduction

Le problème de la plus longue sous-suite commune à deux mots (Longest Common Subsequence - LCS) peut lui aussi être résolu par la programmation dynamique. Étant données deux suites, le problème LCS consiste à trouver une séquence commune à ces deux suites, de longueur L maximale. Le calcul de L résout seulement le problème de la détermination de la longueur d'une LCS (problème LLCS). De façon plus formelle, le problème peut être énoncé comme suit : étant donnée une suite A sur un alphabet Σ (ensemble fini de symboles), une sous-suite de A est une suite C pouvant être obtenue à partir de A en effaçant zéro ou quelques symboles (non nécessairement consécutifs). Le problème de la plus longue sous-suite commune à deux mots pour deux suites $A = A_1A_2 \cdots A_N$ et $B = B_1B_2 \cdots B_M$ ($M \leq N$) consiste à trouver une autre suite $C = C_1C_2 \cdots C_L$ telle que C soit une sous-suite à la fois de A et de B, et de longueur maximale.

Intérêt du problème

L'extraction d'une plus longue sous-suite commune à deux mots peut améliorer la compression de données [Sto88], la correction automatique d'erreurs syntaxiques dans des textes ou des programmes [Lav93], le traitement du signal (analyse de spectres vocaux). En biologie moléculaire, il permet la comparaison d'une séquence d'acide nucléiques (ADN/ARN) ou d'acides aminés (protéines) à une ou plusieurs séquences (voire une banque de données entière) pour déterminer et/ou localiser certaines similarités [SK83, SW89]. De plus, c'est un cas d'école [CLR90], intensivement étudié depuis une trentaine d'années, en témoigne le volume de la bibliographie (cf. [HL97]) et le nombre de thèses de doctorat et d'habilitations [Eis85, Lop87, Lav89, Epp89, Kec91, Stu91, Dan94, Fra95, Sch96]. Il y a donc une forte motivation dans la conception d'algorithmes performants [Wat85].

Au paragraphe suivant, nous donnerons d'une part, les définitions de bases du problème LCS et d'autre part, nous poserons les fondements de notre solution CGM par une définition du problème basée sur la programmation dynamique.

Définitions de base et notations

Définition 3.7 *Soit Σ un ensemble fini de symboles appelé alphabet. Une suite (ou chaîne ou mot) de Σ est une séquence finie d'éléments de Σ. Σ^* est l'ensemble de toutes les suites de Σ, incluant la suite vide ε. Σ^+ représente $\Sigma^* - \{\varepsilon\}$. Pour toute suite A, $|A|$ représente la longueur de A (i.e. le nombre de symboles de A). Notons que $|\varepsilon| = 0$.*

Définition 3.8 *Une suite $A \in \Sigma^+$ est totalement décrite en écrivant $A = A[1] \dots A[N]$, où $A[i] \in \Sigma$ $(1 \leq i \leq N)$. La suite $A' = A[i_1] \dots A[i_k]$ $(1 \leq k \leq N)$, où $1 \leq i_1 < \dots < i_k \leq N$, est appelée* sous-suite *de A. ϵ est aussi une sous-suite de A.*

Définition 3.9 *Soit A une suite telle que $A = A[1] \dots A[N]$.*
$A[i : j]$ représente une sous-suite de A telle que $A[i : j] = A[i] \dots A[j]$ $(1 \leq i \leq j \leq N)$.

Définition 3.10 *Soient A et B deux suites de Σ. La suite U est une* sous-suite commune *à A et à B si c'est une sous-suite de A et de B.*

Définition 3.11 *U est la* plus longue sous-suite commune *(LCS) de A et de B si U est une sous-suite commune de A et de B de longueur maximale.*

Nous noterons que $llcs(i, j)$ représente la longueur de la plus longue sous-suite commune de $A[1 : i]$ et de $B[1 : j]$.

Programmation dynamique

Les propositions suivantes vont nous permettre de résoudre le problème LCS par la programmation dynamique. La proposition 3.2 va permettre de construire un tableau *llcs* á deux dimensions contenant les valeurs des longueurs des sous-suites communes à A et B en chaque caractère de A et de B.

Proposition 3.2 *Pour $1 \leq i \leq N$ et $1 \leq j \leq M$:*

$$llcs(i,0) = llcs(0,j) = llcs(0,0) = 0$$

$$llcs(i,j) = \begin{cases} llcs(i-1,j-1)+1, & si\ A[i]{=}B[j] \\ max(llcs(i-1,j), llcs(i,j-1)), & sinon \end{cases}$$

Preuve : voir [RT85]. □

La table 3.3.1 illustre ce calcul pour deux suites A et B données. Les valeurs en gras correspondent aux longueurs lorsque un él'ement de la suite A est gal un él'ement de la suite B (si aucun él'ement de la suite A ne correspond avec la suite B, on prendra la valeur 0). La valeur soulignée est la première longueur maximale parmi toutes les valeurs en gras. La proposition 3.3 va permettre de construire un vecteur $LLCS$ contenant les valeurs maximales des longueurs des sous-suites communes à A et B en chaque caractère B.

Proposition 3.3 *Pour* $1 \leq j \leq m$ *:*

$$LLCS(j) = max\left\{ \ llcs(i,j) \quad | \ \forall\ i : 1 \leq i \leq n \ \right\}$$

Preuve : La ligne j de la table *llcs* contient les valeurs de toutes les plus longues sous-suites communes de $B[0:j]$. La plus grande de ces valeurs donne la valeur correcte pour $LLCS[j]$. □

		b	c	a	c	b	d	a	b	c
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	**1**	1	1	1	**1**	1	1
c	0	0	**1**	1	**2**	2	2	2	2	**2**
b	0	**1**	1	1	2	**3**	3	3	**3**	3
a	0	1	1	**2**	2	3	3	**4**	4	4
d	0	1	1	2	2	3	**4**	4	4	4
c	0	1	**2**	2	**3**	3	4	4	4	**5**
d	0	1	2	2	3	3	**4**	4	4	5
a	0	1	2	**3**	3	3	4	**5**	5	5
b	0	**1**	2	3	3	**4**	4	5	**6**	6

TAB. 3.2 – Exemple de table *llcs* pour les suites A=acbadcdab et B=bcacbdabc.

La proposition 3.4 va permettre de construire un vecteur $PRED$ contenant, pour chaque caractère de B, l'indice minimum du caractère qui le précède dans la plus longue sous-suite commune le contenant.

Proposition 3.4 *Pour* $1 \leq j \leq m$ *et* $1 \leq i \leq n$ *:*

$$PRED(j) = \begin{cases} min\{k \quad | \exists\ k < i,\ l < j\ :\ LLCS[j]{=}llcs(k,l){+}1\}, & si\ LLCS[j] \neq 0 \\ -1, & sinon \end{cases}$$

Preuve : La proposition 3.3 a pour conséquence que $\forall\ j$, $\exists\ k$, l tels que :

$$\text{LLCS[j]} = \text{llcs(k,l)} + 1 \quad \text{si } LLCS[j] \neq 0 \quad (1)$$

Parmi tous les k qui vérifient l'égalité (1) nous gardons le minimum.
Puis, cette valeur k est l'index du caractère de B qui précède $B[j]$ dans la plus longue sous-suite commune qui le contient. □

Théorème 3.2 *Soit B[k] le dernier caractère de la plus longue sous-suite commune.*
La LCS est construite par $B[PRED^{p-1}[k]] \ldots B[PRED^2[k]]B[PRED[k]]B[k]$ *(p étant la longueur de la LCS et* $PRED^n[k] = \underbrace{PRED[PRED[\ldots PRED[k]\ldots]}_{n\ termes}$ *).*

Preuve : Nous avons vu dans la proposition 3.4 que $PRED[j]$ est l'index du caractère de B qui précède $B[j]$ dans la plus longue sous-suite commune le contenant.
Démarrant de $B[k]$, étant le dernier caractère de la plus longue sous-suite commune (i.e. k est l'index minimum tel que $LLCS[k]{=}\max LLCS[j] | 1 \leq j \leq m$), le caractère de B qui précède $B[k]$ dans la plus longue sous-suite commune est $B[PRED[k]]$. En appliquant cette opération récursivement, nous avons :
$LCS = B[PRED^{p-1}[k]] \ldots B[PRED^2[k]]B[PRED[k]]B[k]$ avec p étant la longueur de la plus longue sous-suite commune et $PRED^n[k] = \underbrace{PRED[PRED[\ldots PRED[k]\ldots]}_{n\ \text{termes}}$. □

Un exemple de calcul des tables $B, LLCS$ et $PRED$ est donné ci-dessous (table 3.3.1).

j	1	2	3	4	5	6	7	8	9
$B[j]$	b	c	a	c	b	d	a	b	c
$LLCS[j]$	1	2	3	3	4	4	5	6	6
$PRED[j]$	0	1	2	3	4	5	6	7	7

TAB. 3.3 – Exemple pour les suites A=acbadcdab et B=bcacbdabc.

La proposition suivante peut permettre de construire une plus longue sous-suite commune en utilisant le vecteur $LLCS$. Cette proposition est utilisée dans la solution systolique.

Proposition 3.5 *Pour* $1 \leq i \leq n$ *et* $1 \leq j \leq m$ *:*
$LCS(i,0) = LCS(0,j) = LCS(0,0) = \varepsilon$,

$$LCS(i,j) = \begin{cases} LCS(i-1,j-1)A[i] & si\ A[i] = B[j], \\ LCS(i-1,j) & si\ LLCS(i-1,j) \geq LLCS(i,j-1), \\ LCS(i,j-1) & sinon. \end{cases}$$

Preuve : Par induction sur i et j.
Les conditions initiales de la proposition 3.5 nous permettent de supposer la propriété vraie pour $LCS(i-1,j-1), LCS(i-1,j)$ et $LCS(i,j-1)$. Notons H_1, H_2 et H_3 ces trois hypothèses. Deux cas sont à considérer :
a) détection d'une égalité $A[i] = B[j]$: on a alors $LLCS(i,j) = LLCS(i-1,j-1)+1$ et $LCS(i,j) = LCS(i-1,j-1)A[i]$ est un plus long sous-mot commun aux deux mots $A[1:i]$ et $B[1:j]$ qui est de longueur $LLCS(i,j)$.
b) si $A[i] \neq B[j]$) : deux cas sont à considérer :

i) $LLCS(i-1,j) \geq LLCS(i,j-1)$ et donc $LCS(i-1,j)$ étant un sous-mot commun de $A[1:i-1]$ et $B[1:j]$ (H_2) de longueur $LLCS(i,j)$ c'est aussi un plus long sous-mot commun de $A[1:i]$ et $B[1:j]$ de longueur $LLCS(i,j)$.

ii) $LLCS(i-1,j) < LLCS(i,j-1)$ et donc $LLCS(i,j) = LLCS(i,j-1)$ et $LCS(i,j-1)$ étant un sous-mot commun de $A[1:i]$ et $B[1:j-1]$ (H_3) de longueur $LLCS(i,j)$ c'est aussi un sous-mot commun de $A[1:i]$ et $B[1:j]$ de longueur $LLCS(i,j)$. □

3.3.2 Complexité des solutions séquentielles

Solutions séquentielles

Les problèmes de la LLCS et de la LCS ont été largement étudiés dans la littérature pour développer des algorithmes séquentiels, par exemple dans [A. 76, ABG92, DH84, LLM97, LM97, Hir77, HS77, KNY82, MP80], mais la complexité $O(NM)$ en temps du problème LCS peut seulement être améliorée sous des conditions restrictives. D'autres algorithmes séquentiels sont rappelés dans [DP94].

3.3.3 Description de la solution systolique

Ces dernières années, la parallélisation de ce problème a suscité beaucoup d'intérêts, lire par exemple [AALM90, LL94] sur les modèles PRAM. Plus récemment, un réseau systolique linéaire efficace pour résoudre les problèmes LLCS et LCS a été proposé dans [LLM97, Luc97]. Cette solution est basée sur les propositions définies au paragraphe 3.3.1. Elle est basée sur une architecture systolique linéaire unidirectionnelle composée des cellules décrites par la figure 3.7 contenant m registres et $m+3$ canaux de communication (voir figure 3.8).

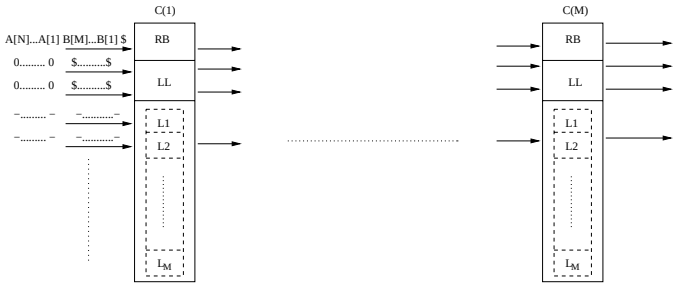


FIG. 3.7 – Réseau linéaire unidirectionnel de M processeurs

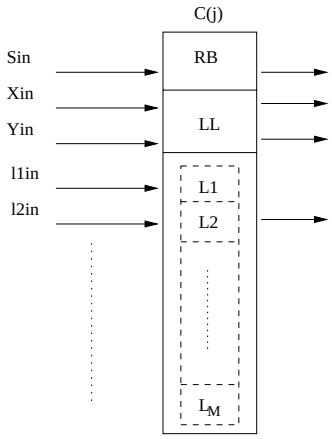


FIG. 3.8 – Processeur élémentaire

Algorithme

Chaque processeur PE_j $(1 \leq j \leq M)$ est doté des registres suivants :

– un registre RB pour mémoriser $B[j]$,
– un registre LL pour mémoriser $LLCS(i,j)$ après le traitement de $A[i]$,
– une zone L de M registres, disons $L = L_1 L_2 \dots L_M$, pour mémoriser $LCS(i,j)$ après le traitement de $A[i]$.

La mémoire locale L sert de substitut aux piles systoliques de [RT85] et aux CAMs de [Lin94].

Les connexions entre deux processeurs voisins sont :
– les canaux S, X et Y du réseau linéaire pour le problème LLCS,
– M canaux lk $(1 \leq k \leq M)$ permettant le transport de M symboles de Σ ou de symboles $-$ ($- \notin \Sigma$ est un symbole spécial utilisé pour initialiser la zone mémoire L).

L'introduction de M canaux d'entrées-sorties rend possible le transport du mot $LCS(i,j)$ selon la proposition 3.5 avec la valeur $LLCS(i,j)$ calculée par PE_j après le traitement de $A[i]$ selon la proposition 3.2, supprimant par là même le besoin d'opérer une seconde phase. Dans la suite, l_{in} et l_{out} désigneront respectivement $l1_{in} \dots lM_{in}$ et $l1_{out} \dots lM_{out}$, où lk_{in} et lk_{out} sont les valeurs lues en entrée et écrites en sortie du canal lk $(1 \leq k \leq M)$.

Le flot des données requis par l'algorithme systolique suivant qui implémente la proposition 3.5 sur le réseau décrit ci-dessus est donné figure 3.7.

Algorithme d'un processeur

```
si (S_in = $) alors
    RB ← $
    L ⟵ l_in                                   ℒ1
    l_out ⟵ L
sinon
    si (X_in = $) alors
        X_out ← $
        si (Y_in = $) et (RB = $) alors
            RB ← S_in
            LL ← 0
            Y_out ← 0
        sinon
            Y_out ← Y_in
        finsi
    sinon                                      ℒ2
```

$X_{out} \leftarrow LL$
si $(Y_{in} > LL)$ **alors**
$LL \leftarrow Y_{in}$
$L \longleftarrow l_{in}$
sinon
si $(S_{in} = RB)$ et $(LL < 1 + X_{in})$ **alors**
$LL \leftarrow 1 + X_{in}$
$L \longleftarrow l_{in} \oplus S_{in}$ $\mathcal{L}3$
finsi
finsi
$Y_{out} \leftarrow LL$
$l_{out} \longleftarrow L$
finsi
finsi
$S_{out} \leftarrow S_{in}$

Le principe d'initialisation du réseau est celui décrit par Lin dans [Lin94]. De plus, la zone L est initialisée par le mot vide $\epsilon = --\cdots-$. Dand la partie $\mathcal{L}1$, l'instruction $L \longleftarrow l_{in}$ signifie $L_k \leftarrow lk_{in}$ pour tout $1 \leq k \leq M$.

Les $A[i]$ $(1 \leq i \leq N)$ sont ensuite introduits. Comme $A[i]$ traverse le réseau de gauche à droite, le processeur PE_j $(1 \leq j \leq M)$ considère le symbole $A[i]$ à l'instant $M + i + j$ et calcule, d'une part $LLCS(i, j)$ selon la proposition 3.2, d'autre part $LCS(i, j)$ selon la proposition 3.5. Les configurations d'un processeur, avant puis après exécution de l'algorithme pour le calcul LLCS ($\mathcal{L}2$), sont les suivantes :

Avant que PE_j considère $A[i]$:

$$RB = B[j]$$
$$LL = LLCS(i-1, j)$$
$$L = L_1 L_2 \ldots L_M = LCS(i-1, j)$$
$$S_{in} = A[i]$$
$$X_{in} = LLCS(i-1, j-1)$$
$$Y_{in} = LLCS(i, j-1)$$
$$l_{in} = l1_{in} l2_{in} \ldots lM_{in} = LCS(i, j-1)$$

Après que PE_j ait considéré $A[i]$:

$$RB = B[j$$
$$LL = LLCS(i,j)$$
$$L = L_1 L_2 \dots L_M = LCS(i,j)$$
$$S_{out} = A[i]$$
$$X_{out} = LLCS(i-1,j)$$
$$Y_{out} = LLCS(i,j)$$
$$l_{out} = l1_{out} l2_{out} \dots lM_{out} = LCS(i,j)$$

Précisément, à l'instant $m+i+j$, $A[i]$ est en entrée du processeur PE_j. Soit $A[i-1]$ a été considéré par PE_j au top précédent (si $i > 1$), soit on a $LL = 0$ et $L = \epsilon$. Le registre LL, dont le contenu est $LLCS(i-1,j)$, est tout d'abord envoyé au processeur voisin par le canal X. Puis, si $Y_{in} > LL$ (c'est-à-dire si $LLCS(i,j-1) > LLCS(i-1,j)$) alors LL mémorise $LLCS(i,j-1)$ et L sauvegarde le mot l_{in} (*i.e.* $LCS(i,j) = LCS(i,j-1)$) en utilisant les canaux d'entrée lk cablés aux registres L_k ($1 \leq k \leq M$). Sinon si une égalité est détectée et la condition $LL < 1 + X_{in}$ respectée (*i.e.* $LLCS(i-1,j) < 1 + LLCS(i-1,j-1)$) alors LL est incrémenté de 1 et L mémorise l_{in} puis sauvegarde S_{in} dans L_{LL} pour avoir alors $L = LCS(i,j) = LCS(i-1,j-1)A[i]$ ($\mathcal{L}3$ où $\oplus$ désigne l'opération de concaténation sur les registres de L). Ces nouvelles valeurs calculées sont alors transmises au processeur voisin. Comme $A[i]$ ($1 \leq i \leq N$) est introduit dans le réseau par le processeur PE_1 avec des valeurs initiales nulles sur les canaux X et Y ($LLCS(i-1,0) = LLCS(i,0) = 0$) et sur les canaux lk ($LCS(i-1,0) = LCS(i,0) = \epsilon$), il est en sortie du processeur PE_j avec $Y_{out} = LLCS(i,j)$ et $l_{out} = L = LCS(i,j)$, en accord avec les propositions 3.2 et 3.5.

Complexité

La complexité de la solution systolique est de $O(N)$ en temps avec N processeurs dotés de N registres et de $N+3$ canaux de communications unidirectionnels. Cette complexité induit un travail en $O(N^2)$. Notre but va être de proposer une solution CGM ayant le même travail également basée sur les propositions de programmation dynamique du paragraphe 3.3.1.

3.3.4 Description de la solution CGM

Nous décrivons dans ce chapitre la solution CGM présentée dans [GMS03]. Notre algorithme CGM est basé sur trois algorithmes séquentiels. Chaque algorithme séquentiel

représente une phase de l'algorithme CGM.

Algorithmes séquentiels

L'Algorithme 1 calcule, pour chaque caractère de B, la longueur de la plus longue sous-suite commune le contenant, et l'index de son prédécesseur dans cette sous-suite. Pour chaque j, le calcul de $LLCS[j]$ (i.e. la longueur de la plus longue sous-suite commune contenant $B[j]$), est basé sur la proposition 3.3. Pour cela, nous devons utiliser la proposition 3.2 qui permet de calculer $llcs(i, j)$ (avec $1 \leq i \leq n$). De plus, les variables $LL[j]$, $X[j]$, et $Y[j]$ sont utilisées pour stocker respectivement $llcs(i-1, j)$, $llcs(i-1, j-1)$, et $llcs(i, j-1)$. Le calcul du prédecesseur de $B[j]$ dans la LCS contenant $B[j]$ est directement issu de la proposition 3.4. Le résultat de cette opération est stocké dans $PRED[j]$. Afin de ne pas écraser le vecteur $PRED$ local, on utilise le vecteur LLP en réception du vecteur $PRED$ envoyé par un autre processeur.

Algorithme 1

pour $(i = 1)$ à $(i = N)$
  pour $(j = 1)$ à $(j = N)$
    $TEMP = LL[j]$
    si $(Y[i] > LL[j])$ **alors**
      si $((B[j] = A[i])$ **et** $(LL[j] < 1 + X[i]))$ **alors**
        $LLCS[j] = 1 + X[i]$
        $PRED[j] = LLP[i]$
      finsi
      $LL[j] = Y[i]$
      $LLP[i] = j$
    sinon
      si $((B[j] = A[i])$ **et** $(LL[j] < 1 + X[i]))$ **alors**
        $LL[j] = 1 + X[i]$
        $LLCS[j] = LL[j]$
        $PRED[j] = LLP[i]$
      sinon
        si $((LLCS[j] = LL[j])$ **et** $(LLCS[j] \neq 0))$ **alors**
          $LLP[i] = j$

finsi
finsi
finsi
$Y[i] = LL[j]$
$X[i] = TEMP$
finpour
finpour

L'Algorithme 2 détermine la longueur de la plus longue sous-suite commune de A et de B et la stocke dans la variable Max. La variable ind contient la position dans B du dernier caractère de la LCS. Il est facile de calculer ind et Max puisque nous avons la longueur des plus longues sous-suites communes finissant en chaque caractère de B.

Algorithme 2
$Max = LLCS[1]$
pour $(i = 2)$ à $(i = N)$
si $LLCS[i] > Max$ **alors**
$Max = LLCS[i]$
$ind = i$
finsi
finpour

L'Algorithme 3 présente l'extraction d'une plus longue sous-suite commune de A et de B à partir du Théorème 3.2 et travaille de la façon suivante : la construction de la LCS commence par l'élément $B[ind]$ tel que $LLCS[ind]$ est égal à Max. $PRED[ind]$ représente la position de l'élément qui précède $B[ind]$ dans la LCS. Puis, nous considérons l'élément précédant $B[ind]$ et nous décrémentons la variable Max. Nous répétons ces opérations tant que Max n'est pas égal à 0.

Algorithme 3
faire
$LCS[Max] = B[ind]$
$ind = PRED[ind]$

$Max = Max - 1$
tant que $(Max \neq 0)$

Algorithme CGM

L'Algorithme LCS_CGM présente la solution CGM du problème LCS. Chaque processeur num $(1 \leq num \leq P)$ a la num-ième partie de $\frac{N}{P}$ éléments des suites A et B. L'algorithme CGM suivant présente le programme pour chaque processeur num. Cet algorithme utilise les algorithmes sequentiels 1, 2 et 3 décrits précédemment. Les figures 3.9, 3.10, et 3.11 décrivent les rondes de communication 1, 2, et 3 respectivement utilisées dans l'algorithme CGM.

Algorithme LCS_CGM
pour (k=1) *à* (k=P)
si k $\leq$ num **alors**
Calcul local en utilisant l'Algorithme 1
Rondes de communication 1
finsi
finpour
Calcul local en utilisant l'Algorithme 2
Rondes de communication 2
Calcul local en utilisant l'Algorithme 3
Rondes de communication 3

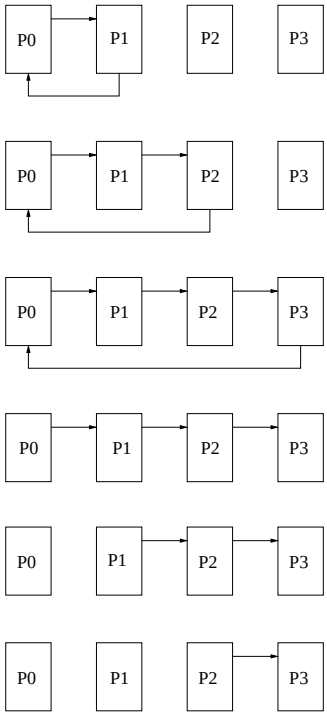


FIG. 3.9 – Rondes de communication 1 pour le problème LCS (pour 4 processeurs).

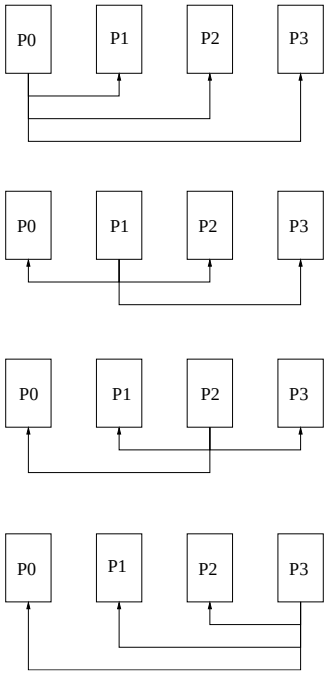


FIG. 3.10 – Rondes de communication 2 pour le problème LCS (pour 4 processeurs).

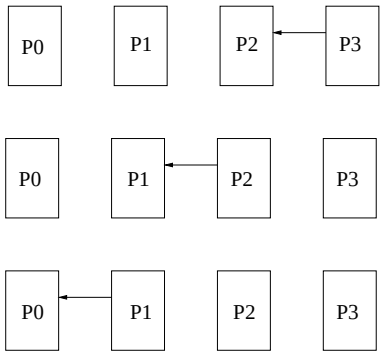


FIG. 3.11 – Rondes de communication 3 pour le problème LCS (pour 4 processeurs).

3.3.5 Complexité de la solution CGM

L'algorithme 1 de la section précédente réalise $N \times N$ opérations. Les algorithmes 2 et 3, réalisent quant à eux, N opérations. L'algorithme CGM_LCS a été developpé pour le modèle CGM utilisant P processeurs et $\frac{N}{P}$ données. Le premier algorithme a donc une complexité de $O((\frac{N}{P})^2)$ en temps et les deux autres algorithmes, une complexité de $O(\frac{N}{P})$. L'algorithme CGM_LCS a donc une complexité globale de $O(\frac{N^2}{P})$ en temps. De plus, un processeur va effectuer au maximum P rondes de communication. Cette complexité induit donc un travail en $O(N^2)$ (avec P processeurs) qui est équivalent au travail de la solution systolique citée au paragraphe 3.3.3.

3.4 Conclusion

Nous avons présenté deux algorithmes CGM pour le problème de la recherche de la plus longue sous-suite croissante et de la plus longue sous-suite commune à deux mots. Ces algorithmes utilisent P processeurs, ont une complexité en temps de $O(\frac{N^2}{P})$ et utilisent P rondes de communication. Nous présentons aussi des résultats expérimentaux qui montrent que ces algorithmes sont très efficaces, pour des grandes suites de données, quand nous augmentons le nombre de processeurs. À partir des solutions systoliques linéaires unidirectionnelles, nous avons donc réussi à dériver deux solutions CGM en conservant le même travail.

Chapitre 4

Problèmes sur réseau linéaire bidirectionnel

4.1 Introduction

Ce chapitre propose des solutions CGM aux problèmes de recherche du plus long suffixe répété en chaque caractère d'un mot et de répétitions issues des architectures systoliques linéaires bidirectionnelles résolvant ces mêmes problèmes. Ce type d'architecture systolique est représenté sous la forme d'un réseau linéaire de processeurs à travers lesquels les communications se font dans les deux sens (voir figure 4.1) :

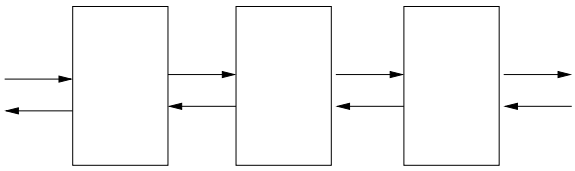


FIG. 4.1 – Réseau systolique linéaire bidirectionnel

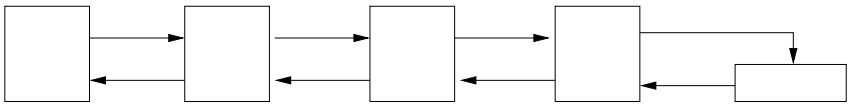


FIG. 4.2 – Réseau systolique linéaire bidirectionnel de N processeurs

Plus particulièrement, nos algorithmes vont utiliser le réseau décrit à la figure 4.2.

Chaque problème sera présenté de la manière suivante : nous présenterons l'intérêt pratique du problème, la complexité des solutions séquentielles, la description de la solution systolique, la description de la solution CGM, la complexité de la solution CGM et enfin, nous donnerons des résultats expérimentaux.

4.2 Plus long suffixe répété en chaque caractère d'un mot

Ce chapitre présente une solution du plus long suffixe répété en chaque caractère d'un mot sur le modèle CGM. L'algorithme peut être implanté en CGM sur P processeurs avec une complexité de $(\frac{2N^2}{P} - \frac{N^2}{P^2}) = 0(\frac{N^2}{P})$ en temps et $2P - 1$ rondes de communication. Après la description du problème, nous donnerons la complexité des solutions séquentielles et systolique. Nous proposerons notre solution CGM et nous expliquerons sa complexité.

4.2.1 Description et intérêt du problème

Introduction

Le problème du plus long suffixe répété (Longest Repeated Suffix Ending at each point of a word - LRSE) peut être une bonne illustration de la programmation dynamique. Il existe un grand nombre d'études sur la détection de répétitions dans un mot donné A de longueur N (voir [Smy00]). Par exemple, la fonction « shift » de Morris et Pratt [MP70] donne pour chaque préfixe de p la longueur de ses plus longues bordures. La fonction « matching shift » de Boyer-Moore (voir [BM77], [KMP77] et [Ryt80]) (aussi appelée fonction « good suffix shift ») donne pour chaque suffixe ses positions les plus à droite dans A. Crochemore [Cro81] donna un algorithme en $O(N \log N)$ en temps qui trouve les répétitions maximales de A. Plus récemment, Kolpakov et Kucherov [KK00] ont présenté un algorithme en temps linéaire qui trouve toutes les répétitions d'un mot. Dans ce chapitre, nous nous sommes intéressé

au problème de trouver le plus long suffixe répété en chaque caractère d'un mot de A.

Intérêt du problème

L'extraction du plus long suffixe répété peut améliorer la compression de données, être utilisée dans la combinatoire de mots, la théorie du langage formel, la réécriture de termes ou en bio-informatique.

Au paragraphe suivant, nous donnerons d'une part, les définitions de bases du problème LRSE et d'autre part, nous poserons les fondements de notre solution CGM par une définition de la solution de ce problème basée sur la programmation dynamique.

Définitions de base et notations

Définition 4.1 *Une chaîne W est une sous-chaîne de A si $A = UWV$ pour $U, V \in \Sigma^*$; nous pouvont également dire que la chaîne W occupe la position $|U|$ dans la chaîne A.*

Définition 4.2 *La position $|U|$ est la position de démarrage de W dans A et la position $|U| + |W|$ la position finale de W dans A. Nous noterons $W = A[|U| + 1 \dots |U| + |W|]$.*

Définition 4.3 *Une chaîne W est un préfixe de A si $A = WU$ pour $U \in \Sigma^*$. De même, W est un suffixe de A si $A = UW$ pour $U \in \Sigma^*$.*

Programmation dynamique

Nous allons maintenant présenter une solution décrite en termes de programmation dynamique se trouvant dans [LMS02].
Soit $A = A[0 \dots N-1]$ un mot de longueur N sur un alphabet Σ. Nous définissons une table à deux dimentions T de la manière suivante :

$$T[i,j] = \max\{\ell \mid A[i-\ell+1 \dots i] = A[j-\ell+1 \dots j]\}$$

pour $0 \leq j \leq N-1$ et $0 \leq i < j$.
$T[i,j]$ est la longueur du suffixe commun le plus long de $A[0 \dots i]$ et $A[0 \dots j]$. La Figure 4.3 donne un exemple d'une table T.
Le lemme suivant montre comment calculer les valeurs de la table T.

	j	0	1	2	3	4	5	6	7	8	9	10	11
i		a	b	b	c	a	b	c	d	a	b	c	d
-1	ε	0	0	0	0	0	0	0	0	0	0	0	0
0	a		0	0	0	1	0	0	0	1	0	0	0
1	b			1	0	0	2	0	0	0	2	0	0
2	b				0	0	1	0	0	0	1	0	0
3	c					0	0	2	0	0	0	2	0
4	a						0	0	0	1	0	0	0
5	b							0	0	0	2	0	0
6	c								0	0	0	3	0
7	d									0	0	0	4
8	a										0	0	0
9	b											0	0
10	d												0

FIG. 4.3 – Valeurs de la table T pour $A = abbcabcdabcd$.

Lemme 4.1 *Les valeurs de la table T peuvent être calculées en utilisant la formule récursive suivante :*

$$T[-1, j] = 0 \text{ for } 0 \leq j \leq N - 1 \tag{4.1}$$

et

$$T[i, j] = \begin{cases} T[i-1, j-1] + 1 & si\ A[i] = A[j] \\ 0 & sinon \end{cases} \tag{4.2}$$

pour $0 \leq j \leq N - 1$ et $0 \leq i < j$.

Preuve : Voir [LMS02].

□

Maintenant, nous allons définir une table S de la manière suivante :

$$S[j] = \max\{|W| \mid W \text{ est un suffixe de } A[0 \dots j] \text{ et } W \text{ est un facteur de } A[0 \dots j-1]\}$$

pour $0 \leq j \leq N - 1$
$S[j]$ est la longueur du plus long suffixe de $A[0 \dots j]$ qui apparait au moins deux fois dans $A[0 \dots j]$. La Figure 4.4 donne un exemple de la table S.
Le lemme suivant montre comment calculer les valeurs de la table S.

$A[i]$	a	b	b	c	a	b	c	d	a	b	c	d
$S[i]$	0	0	1	0	1	2	2	0	1	2	3	4

FIG. 4.4 – Valeurs de la S pour l'exemple de la Figure 4.3 .

Lemme 4.2 *Les valeurs de la table S peuvent être calculées en utilisant la table T de la manière suivante :*

$$S[j] = \max\{T[i,j] \mid 0 \leq i < j\} \tag{4.3}$$

pour $0 \leq j \leq N-1$.

Preuve : Voir [LMS02]. □

Il est trivial de trouver un algorithme pour calculer les valeurs de la table S qui est décrit par le corollaire suivant.

Corollaire 4.1 *Il existe un algorithme qui calcule les valeurs de la table T et S en programmation dynamique. Ils fonctionnent en $\Theta(N^2)$ en temps et ils requièrent $O(N)$ d'espace mémoire.*

Preuve : Voir [LMS02]. □

L'idée est de couper la table T en P colonnes de $\frac{N}{P}$ éléments. La Figure 4.5 donne un exemple de cette table coupée T'.
La table T' contient deux sortes de parties : des sous-tables triangulaires (TT) et des sous-tables carrées (ST). Le lemme 4.1 peut être appliqué sur les sous-tables triangulaires parce que nous comparons une sous-chaîne de A avec elle-même. Le lemme suivant montre comment calculer les valeurs des sous-tables carrées ST qui permettent la comparaison entre deux sous-chaînes différentes de A.

Lemme 4.3 *Les valeurs de la sous-table ST peuvent être calculées en utilisant la formule récursive suivante :*

$$ST[-1,j] = 0 \text{ pour } -1 \leq j \leq N-1 \tag{4.4}$$

$$ST[i,-1] = 0 \text{ pour } 0 \leq i \leq N-1 \tag{4.5}$$

	j	0	1	2	3	4	5	6	7	8	9	10	11
i		a	b	b	c	a	b	c	d	a	b	c	d
0	a	0	0	0	0	1	0	0	0	1	0	0	0
1	b		0	1	0	0	2	0	0	0	2	0	0
2	b			0	0	0	1	0	0	0	1	0	0
3	c				0	0	0	2	0	0	0	2	0
4	a					0	0	0	0	1	0	0	0
5	b						0	0	0	0	2	0	0
6	c							0	0	0	0	3	0
7	d								0	0	0	0	4
8	a									0	0	0	0
9	b										0	0	0
10	c											0	0
11	d												0

FIG. 4.5 – Table coupée T' pour $A = abbcabcdabcd$

et

$$ST[i,j] = \begin{cases} ST[i-1,j-1]+1 & si\ A1[i] = A2[j] \\ 0 & sinon \end{cases} \tag{4.6}$$

pour $0 \leq j \leq N-1$ *et* $0 \leq i < N-1$.

Preuve : Nous prouvons la formule par récurrence sur j. Il est clair que quand $j = 0$, puisque la seule valeur qui est définie dans la table ST avec $j = 0$ est $ST[-1,0]$ qui est initialisé à 0 par l'équation 4.4. Maintenant supposons que la propriété soit vraie en $j-1$ et montrons qu'elle est encore vraie pour j. Pour $0 \leq i < N-1$, si $A1[i] \neq A2[j]$ alors le plus long suffixe commun de $A1[0 \ldots i]$ et $A2[0 \ldots j]$ se terminant en $A1[i]$ est le mot vide, qui est calculé dans le second cas de l'équation 4.6. Si $A1[i] = A2[j]$ alors la longueur du plus long suffixe commun de $A1[0 \ldots i]$ et $A2[0 \ldots j]$ est égal à la longueur du plus long suffixe commun de $A1[0 \ldots i-1]$ et $A2[0 \ldots j-1]$ se terminant en $A1[i-1]$ plus 1, qui est, du fait de l'hypothèse de récurrence $ST[i-1,j-1]+1$ le premier cas de l'équation 4.6. Fin de la preuve. □

Chaque table triangulaire devra être initialisée par :

$$TT[-1,j] = L[j] \text{ pour } 0 \leq j \leq N-1 \tag{4.7}$$

et
chaque table carrée devra être initialisée par :

$$ST[-1, j] = L[j] \text{ pour } -1 \leq j \leq N-1 \tag{4.8}$$

$$ST[i, -1] = C[i] \text{ pour } 0 \leq i \leq N-1 \tag{4.9}$$

où C et L sont respectivement la dernière colonne et la dernière ligne de la précédente partie de la table coupée.

4.2.2 Complexité des solutions séquentielles

Solutions séquentielles

Lefebvre and Lecroq [LL00] ont utilisé l'oracle des facteurs de A pour obtenir un calcul linéaire en ligne donnant une bonne approximation de ces valeurs. Lecroq, Myoupo et Semé [LMS02] ont donné un algorithme séquentiel en $O(N^2)$ en temps, qui est le premier à calculer la valeur exacte du plus long suffixe répété en chaque caractère d'un mot.

4.2.3 Description de la solution systolique

Lecroq, Myoupo et Semé [LMS02] ont aussi donné un algorithme parallèle. Cette solution est basée sur une architecture systolique linéaire bidirectionnelle définie dans l'introduction de ce chapitre (voir figure 4.2) composée de cellules, comme celle décrite par la figure 4.6, contenant six registres et six canaux de communication.

Algorithme

Avant le calcul dans la cellule i :
$X1_{in}$ est le j-ième caractère de la sous-suite ;
$X2_{in}$ est le k-ième (avec $k = j + i$) caractère de la sous-suite ;
$S1_{in} = max(T[j-1, k]) \mid 0 \leq j < k-1$;
$T1_{in} = T[j-1, k-1]$;
$S2_{in} = max(T[i, j]) \mid 0 \leq i < j$;
$T2_{in} = T[j-1, k-2]$;

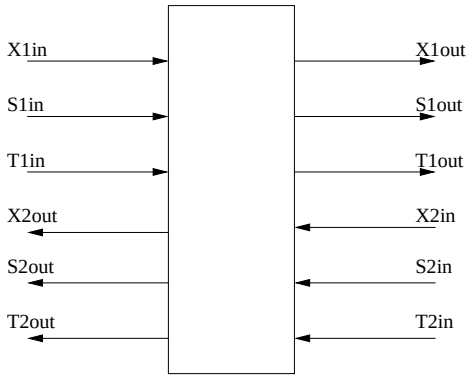


FIG. 4.6 – Processeur élémentaire

Après le calcul dans la cellule i :
$X1_{out} = X1_{in}$;
$X2_{out} = X2_{in}$;
$S1_{out} = max(T[j,k]) \mid 0 \leq j < k$;
$T1_{out} = T[j,k]$;
$S2_{out} = S2_{in}$;
$T2_{out} = T1_{in}$;

Chaque processeur i de ce réseau, sauf la cellule 0, est un comparateur de caractères qui compare les caractères j et $k = j + i$ avec $(i \in \{1, 2, \ldots, N-1\})$. Le rôle de la cellule 0 est de fermer la boucle du flux de données du côté droit. L'algorithme exécuté par un processeur i est décrit ci-après. Pour l'initialisation du réseau, les caractères doivent être introduit tous les deux cycles d'horloge. Dans le cas contraire, des caractères pourraient ne pas être comparés.

Algorithme d'un processeur
$T2_{out} \leftarrow T1_{in}$
$S2_{out} \leftarrow S2_{in}$
$S1_{out} \leftarrow S1_{in}$
si $X1_{in} = X2_{in}$
  alors $X1_{out} \leftarrow X2_{out} + 1$

sinon $T1_{out} \leftarrow 0$
si $T1_{out} > S1_{out}$
alors $S1_{out} \leftarrow T1_{out}$
$X1_{out} \leftarrow X1_{in}$
$X2_{out} \leftarrow X2_{in}$

Complexité

La complexité de cette solution systolique est en $O(N)$ en temps avec N processeurs dotés de 6 registres et de 6 canaux de communications unidirectionnels. Cette complexité induit un travail en $O(N^2)$. Notre but va être de proposer une solution CGM ayant le même travail également basée sur les propositions de programmation dynamique du paragraphe 4.2.1.

4.2.4 Description de la solution CGM

Dans l'article [GS03] nous décrivons la solution CGM du problème LRSE proposée dans ce chapitre. Notre algorithme CGM est basé sur deux algorithmes séquentiels. Chaque algorithme séquentiel représente une phase de l'algorithme CGM.

Algorithmes séquentiels

L'Algorithme 1 calcule la table triangulaire TT, colonne par colonne et construit la table S en utilisant les lemmes 4.1 et 4.2. À la fin, l'algorithme sauvegarde la dernière colonne dans une table C.

Algorithme 1
pour $(i = 2)$ *à* $(i = N)$
pour $(j = 1)$ *à* $(j = i - 1)$
$TEMPC[j] = 0$
si $(A[i] = A[j])$ **alors**
si $(j = 1)$ **alors** $TEMPC[j] = L[i - 1] + 1$
sinon $TEMPC[j] = C[j - 1] + 1$
finsi

si $(TEMPC[j] > S[i])$ **alors** $S[i] = TEMPC[j]$ **finsi**
finsi
finpour
pour $(j = 1)$ *à* $(j = i - 1)$ $C[j] = TEMPC[j]$ **finpour**
finpour

L'Algorithme 2 calcule la table carrée ST, colonne par colonne et construit la table S en utilisant les lemmes 4.1 et 4.3. À la fin, l'algorithme sauvegarde la dernière colonne dans une table C et la dernière ligne dans une table L. On note que le premier élément de la table L est le dernier élément de la colonne précédente C.

Algorithme 2
$TEMPL = L$
$L[0] = C[N]$
pour $(i = 1)$ *à* $(i = N)$
pour (j=1) *à* $(j = N)$
$TEMPC[j] = 0$
si $(A[i] = A[j])$ **alors**
si $(j = 1)$ **alors** $TEMPC[j] = TEMPL[i - 1] + 1$
sinon $TEMPC[j] = C[j - 1] + 1$
finsi
si $(TEMPC[j] > S[i])$ **alors** $S[i] = TEMPC[j]$ **finsi**
finsi
finpour
$L[i] = TEMPC[N]$
pour $(j = 1)$ *à* $(j = N)$ $C[j] = TEMPC[j]$ **finpour**
finpour

Algorithme CGM

L'Algorithme LRSE_CGM présente la solution CGM pour le problème de la LRSE. Chaque processeur num $(1 \leq num \leq P)$ contient la num-ième partie de $\frac{N}{P}$ éléments de la suite A. L'algorithme CGM suivant présente le programme pour chaque processeur num. Cet algorithme utilise les algorithmes sequentiels 1 et 2 décrits précédemment. La figure 4.7

décrit les rondes de communication utilisées dans cet algorithme CGM.

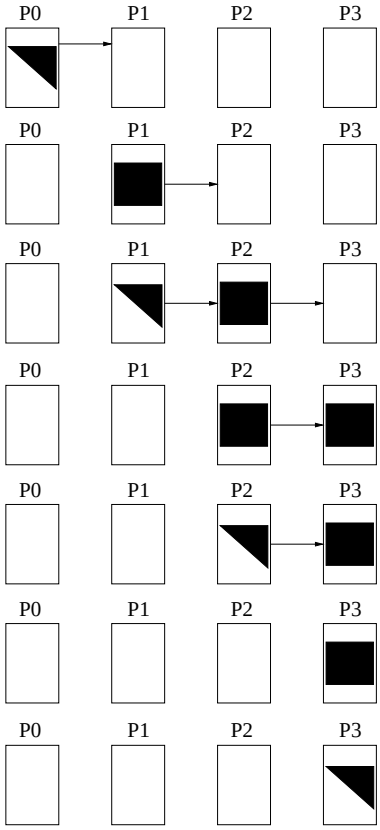


FIG. 4.7 – Rondes de communication pour le problème LRSE (pour 4 processeurs).

Deux fonctions sont utilisées pour les rondes de communication :
envoyer(X,num) : un vecteur X est envoyé au processeur num.
recevoir(Y,num) : un vecteur Y est reçu du processeur num.

Algorithme LRSE_CGM
pour $(k = 1)$ *à* $(k \leq num)$
  si $(k = num)$ **alors**
    Calcul local en utilisant l'Algorithme 1
    si $(num < P)$ **alors**
      envoyer $(A, num+1)$
      envoyer $(C, num+1)$
    finsi
  sinon
    recevoir $(A, num\text{-}1)$
    recevoir $(C, num\text{-}1)$
    Calcul local en utilisant l'Algorithme 2
    si $(num < P)$ **alors**
      envoyer $(A, num+1)$
      envoyer $(C, num+1)$
    finsi
  finsi
finpour

4.2.5 Complexité de la solution CGM

L'Algorithme 1 et l'Algorithme 2 de la section précédente 4.2.4 réalisent $N \times N$ opérations. L'algorithme CGM_LRSE a été developpé pour le modèle CGM utilisant P processeurs et $\frac{N}{P}$ données. Ainsi, nous avons des calculs locaux ayant une complexité en $O(\frac{N^2}{P})$ puisqu'il sont basés sur les algorithmes 1 et 2. Le dernier processeur doit attendre $P-1$ rondes de communication avant de commencer à travailler. Il réalise ensuite P fois son calcul local de complexité $O(\frac{N}{P}^2)$ (ceci est dû au calcul de la table triangulaire T). En tout, il faut $2P-1$ rondes pour que le dernier processeur (ainsi que tous les autres) ait terminé ses calculs. Le nombre de rondes incluant les communications n'est quant à lui que de $2P-3$. La complexité en temps de notre algorithme CGM_LRSE est donc de $(\frac{2N^2}{P} - \frac{N^2}{P^2}) = 0(\frac{N^2}{P})$ (la complexité des calculs locaux multipliée par le nombre de rondes). Ceci induit donc un travail en $O(N^2)$ (avec P processeurs) qui est équivalent au travail de la solution systolique citée au paragraphe 4.2.3.

4.3 Recherche de répétitions

Ce chapitre présente la recherche de répétitions sur le modèle CGM. L'algorithme peut être implanté en CGM sur P processeurs avec une complexité de $O(\frac{N^2}{P})$ en temps et $2P-1$ rondes de communication. Après la description du problème, nous donnerons la complexité des solutions séquentielles et systolique. Nous proposerons notre solution CGM et nous expliquerons sa complexité.

4.3.1 Description du problème

Introduction

La détection des répétitions (Detection of Repetitions - DR)dans un mot ainsi que d'autres variantes du problème ont été bien étudiées dans la littérature à cause de leurs applications importantes dans de nombreux domaines [Aho90, Lee93]. Une variante de ce problème, l'étude des mots sans carrés (« square free »), a attiré l'attention des chercheurs depuis longtemps. Sa première apparition est peut-être datée du début du siècle avec les travaux de A. Thue [Thu12]. Plus récemment [Thu77], A. Thue a découvert des chaînes de symboles arbitrairement longues qui ne contiennent aucun carré (« square free ») sur un alphabet de trois lettres. Plusieurs autres papiers ont été dédiés à la construction de telles chaînes [ML84, Ber92], ainsi que des chaînes ayant d'autres contraintes liées aux répétitions [Bra80, Har78, Hed67, Lot97]. Une liste étendue de références sur des notions complémentaires de périodicité et de chevauchement se trouve dans [Duv78, Ber92, BS93]. Une autre variante du problème de détection de répétitions, la recherche de carrés dans un mot, est aussi bien abordée dans la littérature [Cro83, Lot97].

Un autre problème lié à la reconnaissance de motifs est celui de calculer les statistiques des sous-chaînes d'une chaîne de symboles donnée. Ce problème s'applique également dans plusieurs autres domaines et c'est d'ailleurs un sujet de recherche bien abordé (voir les références de [AN82]). Étant donné un mot x sur un alphabet fini, calculer les statistiques des sous-chaînes de x consiste à calculer le nombre de fois où chacune d'elles se retrouve dans x. Ceci est particulièrement utile dans le domaine de la compression de données [Wab98].

Intérêt du problème

La recherche de répétitions peut être appliquée en biologie moléculaire pour étudier les séquences d'ADN ainsi qu'à d'autres domaines comme le traitement de textes, la linguistique

et la reconnaissance de motifs. On peut aussi l'utiliser dans le domaine de la compression de données. En effet, une répétition peut être remplacée par la sous-chaîne répétée et le nombre d'occurrences de celle-ci. Ainsi, de longues chaînes peuvent être remplacées par un codage qui occupe moins de place.

Au paragraphe suivant, nous donnerons d'une part, les définitions de bases du problème DR et d'autre part, nous poserons les fondements de notre solution CGM par une définition du problème en programmation dynamique.

Définitions de base et notations

Définition 4.4 *Un facteur d'une chaîne $X \in \Sigma^+$ est une sous-chaîne de X dont la première lettre a sa position dans $\{1, 2, \cdots, N\}$.*
La notation $X[k:l]$ représente le facteur de X suivant : $x_k x_{k+1} \cdots x_l$.
Un facteur gauche (respectivement droit) de X est un préfixe (respectivement suffixe) de X.

Définition 4.5 *Une chaîne $X \in \Sigma^+$ est primitive si supposer que $X = u^k$ implique que $u = X$ et $k = 1$.*

Définition 4.6 *Un carré dans une chaîne X est une sous-chaîne non vide de X de la forme uu.*

Définition 4.7 *Une chaîne X est sans-carré si aucune sous-chaîne de X n'est un carré. De façon équivalente, X est sans-carré si chaque sous-chaîne de X est primitive.*

Définition 4.8 *Une répétition dans X est un facteur $X[k:m]$ pour lequel il existe les indices l, d ($k < d \leq l \leq m$) tels que :*

1. *$X[k:l]$ est égal à $X[d:m]$,*
2. *$X[k:d-1]$ correspond à un mot primitif,*
3. *$X_{l+1} \neq X_{m+1}$.*

Définition 4.9 *p est une période d'une répétition $X[k:m]$ de X si $x_i = x_{i+p}$ ($\forall\ i = k, k+1, \cdots, |X[k:m]| - p$). Ce qui implique que $1 \leq p \leq \frac{|X[k:m]|}{2}$.*

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		a	b	b	b	b	a	a	c	a	c	a	d	d	a	c	a
0	a																
1	b	1															
2	b	2	1														
3	b	3	2	1													
4	b	4	3	2	1												
5	a	5	4	3	2	1											
6	a	6	5	4	3	2	1										
7	c	7	6	5	4	3	2	1									
8	a	8	7	6	5	4	3	2	1								
9	c		8	7	6	5	4	3	2	1							
10	a			8	7	6	5	4	3	2	1						
11	d				8	7	6	5	4	3	2	1					
12	d					8	7	6	5	4	3	2	1				
13	a						8	7	6	5	4	3	2	1			
14	c							8	7	6	5	4	3	2	1		
15	a								8	7	6	5	4	3	2	1	

FIG. *4.8 – Valeurs de toutes les périodes p pour* $X = abbbbaacacad$.

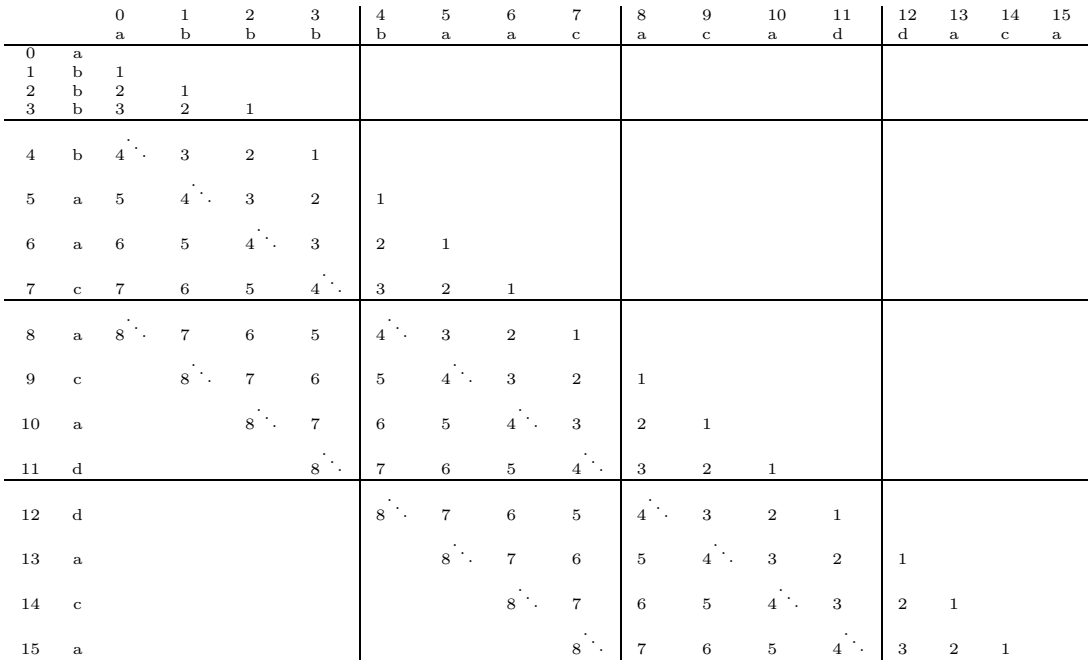
Programmation dynamique

Soit une chaîne X de longueur N et une période p, nous désirons détecter toutes les répétitions dans X pour une période donnée p en utilisant les définitions 4.8 et 4.9. La definition 4.9 montre que la plus grande période possible est $\frac{N}{2}$. La figure 4.8 donne un exemple de toutes les périodes calculées pour la chaîne X de longueur N =16.
L'idée est de couper cette table en P colonnes de $\frac{N}{P}$ éléments, avec $P << N$ et $P \neq 0$. La Figure 4.9 donne un exemple de cette coupure pour quatre processeurs. La Figure 4.9 est composée de carrés contenants deux parties égales de X. Chaque carré est composé d'une partie triangulaire inférieure et une partie triangulaire supérieure. Notre approche dynamique est basée sur les différentes solutions systoliques trouvées dans [Wab98, AN84, MW97, Sem99a].

4.3.2 Complexité des solutions séquentielles

Solutions séquentielles

Des algorithmes de complexité $O(N^2)$ détectant les répétitions dans un mot de longueur N ont été facilement développés sur la base d'outils et de techniques de reconnaissance de



		0 a	1 b	2 b	3 b	4 b	5 a	6 a	7 c	8 a	9 c	10 a	11 d	12 d	13 a	14 c	15 a
0	a																
1	b	1															
2	b	2	1														
3	b	3	2	1													
4	b	4	3	2	1												
5	a	5	4	3	2	1											
6	a	6	5	4	3	2	1										
7	c	7	6	5	4	3	2	1									
8	a	8	7	6	5	4	3	2	1								
9	c		8	7	6	5	4	3	2	1							
10	a			8	7	6	5	4	3	2	1						
11	d				8	7	6	5	4	3	2	1					
12	d					8	7	6	5	4	3	2	1				
13	a						8	7	6	5	4	3	2	1			
14	c							8	7	6	5	4	3	2	1		
15	a								8	7	6	5	4	3	2	1	

FIG. 4.9 – Figure coupée pour $X = abbbbaacacad$

motifs (« pattern matching ») déjà existants. Un algorithme de complexité $O(N \log N)$ qui détermine si un mot sur un alphabet fini contient ou pas (au moins) une répétition a été proposé dans [ML79].

M. Crochemore [Cro81] a développé un algorithme en $O(N \log N)$ pour déterminer toutes les répétitions dans un mot. Cet algorithme est optimal car il existe des mots qui ont effectivement $O(N \log N)$ répétitions. il s'agit des mots de Fibonacci qui sont des cas particuliers des mots sturmiens [Ber81, Séé91]. Un autre algorithme optimal a été présenté dans [AP83]. Ces deux algorithmes effectuent des calculs hors-ligne (« off-line ») sur des structures de données spécifiques dans une mémoire. La différence entre les deux est que le premier est basé sur un algorithme de minimisation des automates finis [AHU74] tandis que le second combine les propriétés des arbres de suffixes associés aux mots et celles d'une nouvelle structure de données appelée « leaf-tree ». L'utilisation des « transducteur » pour la détection de répétitions et de facteurs dans un mot a été étudiée par J. Berstel [Ber79] puis par M. Crochemore [Cro84, Cro86]. Un *transducteur* est un automate déterministe qui produit en sortie une valeur qui peut être la position d'un facteur dans un mot.

4.3.3 Description de la solution systolique

[AN84] présente un algorithme systolique pour la détection des répétitions ayant une complexité en temps de $O(N)$ (plus exactement fonctionnant en $4N - 3$ tops d'horloge) et utilisant N processeurs, tandis que le réseau systolique linéaire résolvant le même problème présenté dans [MW97] demande $(5N/4)$ tops d'horloge avec $N/4$ processeurs. A. Wabbi [Wab98] propose une amélioration de celui présenté par E. Thomas dans [Tho95]. Cette solution est basée sur les propositions définies au paragraphe 4.3.1. Elle est basée sur une architecture systolique linéaire bidirectionnelle définie dans l'introduction de ce chapitre (voir figure 4.2) composée de cellules décrites par la figure 4.10 contenant quatre registres et deux canaux de communication.

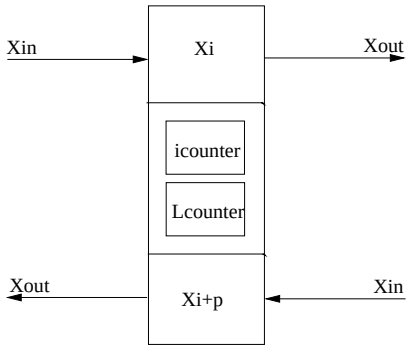


FIG. 4.10 – Processeur élémentaire

Algorithme

Afin de bien comprendre l'algorithme systolique de A. Wabbi [Wab98] il est bon d'expliquer l'algorithme séquentiel sur lequel il est basé. Étant donné un mot x de longueur n et une période p, l'algorithme séquentiel présenté dans [AN84] détecte toutes les répétitions de x pour la période p en temps quadratique.

Algorithme séquentiel

$count \leftarrow p$
pour $i = 1$ **jusqu'**$\grave{a}$ $(n - p)$ **faire**
  si $x(i) \neq x(i + p)$ **alors**
    $count \leftarrow p$
  sinon

```
            count ← count + 1
            si count ≥ 2p alors
                afficher R(i − count + p + 1, p, count)
            finsi
        finsi
    finpour
```

L'exécution de cet algorithme pour toutes les valeurs de p (l'ensemble des périodes) nécessite un temps de $O(n^2)$. On peut ainsi remarquer que l'exécution de la boucle de l'algorithme 4.3 pour une valeur de p est indépendante des autres valeurs que peut prendre p. Ceci a donné l'idée à A. Apostolico et A. Negro [AN84] d'utiliser un réseau linéaire systolique bidirectionnel de n processeurs, chacun dédié à une période donnée afin d'améliorer la performance en terme de temps par un facteur de n. Dans le réseau systolique, le processeur le plus à gauche est à la fois l'entrée et la sortie du réseau. Chaque processeur P_p de ce réseau, sauf la cellule P_0, est un comparateur de caractères et prend donc les caractères i et $i + p$ comme arguments ($i \in \{1, 2, \ldots, n - p\}$). Le rôle de la cellule P_0 est de fermer la boucle du flux de données du côté droit. L'algorithme exécuté par un processeur P_p est décrit ci-après. *Lcounter* et *icounter* sont des compteurs locaux initialisés par la valeur de p et 1 respectivement.

Algorithme d'un processeur

```
si x ≠ y alors
    icounter ← icounter + Lcounter − p + 1
        Lcounter ← p
    sinon
        Lcounter ← Lcounter + 1
        si Lcounter ≥ 2p alors
            afficher R(icounter, p, Lcounter)
        finsi
    finsi
finsi
```

Complexité

La complexité en temps de cette solution est de $3n + 1$ tops d'horloge utilisant N processeurs. Cette complexité induit un travail en $O(N^2)$. Notre but va être de proposer une solution CGM ayant le même travail également basée sur les propositions de programmation dynamique du paragraphe 4.3.1.

4.3.4 Description de la solution CGM

La solution CGM du problème de détection de répétitions décrite dans [GSa] est reprise et est entièrement détaillée dans ce chapitre.

Notre algorithme CGM est basé sur deux algorithmes séquentiels. Chaque algorithme séquentiel représente une phase de l'algorithme CGM.

Algorithmes séquentiels

L'Algorithme 1 détecte toutes les répétitions pour toute période p incluse entre 1 et $N-1$ en parcourant la chaîne X depuis le premier caractère jusqu'au $(N-p)$ième caractère. Á la fin, deux tables Ic et Lc sont créées ou modifiées. Ces tables représentent respectivement la position de départ et la dernière position d'une répétition dans la chaîne X pour une période p. Les différentes répétitions sont données en temps réel pendant l'exécution. Pour réaliser cela, la fonction suivante est nécessaire :
R(Ic,p,Lc) : écrit la répétition qui démarre en Ic pour une période p et de longueur Lc.

Algorithme 1
pour $(p = 1)$ *à* $(p = N - 1)$
  pour $(i = 1)$ *à* $(i = N - p)$
    si $(X[i] \neq X[i + p])$ **alors**
      $Ic[p] = Ic[p] + Lc[p] - p + 1$
      $Lc[p] = p$
    else
      $Lc[p] = Lc[p] + 1$
      si $(Lc[p] \geq 2p)$ **alors** R($Ic[p]$,p,$Lc[p]$) **finsi**
    finsi
  finpour

finpour

L'Algorithme 2 détecte toutes les répétitions pour toute période p incluse entre 1 et N en parcourant la chaîne X depuis le$(N-p+1)$ième caractère jusqu'au (N)ième caractère. Les mêmes tables Ic et Lc ainsi que la fonction R(Ic,p,Lc) sont utilisées.

Algorithme 2
pour $(p = 1)$ *à* $(p = N)$
  pour $(i = N - p + 1)$ *à* $(i = N)$
    si $(X[i] \neq X[i + p])$ **alors**
      $Ic[p] = Ic[p] + Lc[p] - p + 1$
      $Lc[p] = p$
    else
      $Lc[p] = Lc[p] + 1$
      si $(Lc[p] \geq 2p)$ **alors** R(Ic,p,Lc) **finsi**
    finsi
  finpour
finpour

Algorithme CGM

L'Algorithme REPET_CGM présente la solution CGM du problème de détection de répétitions. Chaque processeur num $(1 \leq num \leq P)$ a la num-ième partie de $\frac{N}{P}$ éléments de la chaîne X. L'algorithme CGM suivant présente le programme pour chaque processeur num.
Cet algorithme utilise les algorithmes sequentiels 1 et 2 décrits précédemment. La figure 4.11 décrit les rondes de communication utilisées dans l'algorithme CGM.
Deux fonctions sont utilisées pour les rondes de communication :
envoyer(X,num) : un vecteur X est envoyé au processeur num.
recevoir(Y,num) : un vecteur Y est reçu du processeur num.

Algorithme REPET_CGM
pour $(ii = 0)$ *à* $(ii \leq (P-1) - num)$

si $(ii = 0)$ **et** $(num \neq 0)$ **alors envoyer** $(X,0)$
si $(ii \neq 0)$ **et** $(num = 0)$ **alors recevoir** (Y,ii)
si $(num \neq 0)$ **alors recevoir** $(Y,num - 1)$
si $(ii \neq 0)$ **alors**
    Calcul local en utilisant l'Algorithme 2
    envoyer $(Y, num + 1)$
    envoyer $(Lc, num + 1)$
    envoyer $(Ic, num + 1)$
finsi
si $(num \neq 0)$ **alors**
    recevoir $(Lc,num - 1)$
    recevoir $(Ic,num - 1)$
finsi
Calcul local en utilisant l'Algorithme 1
finpour

4.3.5 Complexité de la solution CGM

Les algorithmes 1 et 2 de la section précédente 4.3.4 réalisent $N \times N$ opérations. L'algorithme CGM_REPET a été developpé pour le modèle CGM utilisant P processeurs et $\frac{N}{P}$ données. Nous avons donc pour les deux algorithmes précédents une complexité de $O((\frac{N}{P})^2)$ en temps.

L'algorithme CGM_REPET a donc une complexité globale de $O(\frac{N^2}{P})$ en temps. De plus, au niveau communication, un processeur va effectuer au maximum $2P - 1$ rondes de communication. Cette complexité induit donc un travail en $O(N^2)$ (avec P processeurs) qui est équivalent au travail de la solution systolique citée au paragraphe 4.3.3.

4.4 Conclusion

Nous avons présenté deux algorithmes CGM pour les problèmes de la recherche du plus long suffixe répété en chaque caractère d'un mot et de détection de répétitions. Ces algorithmes utilisent P processeurs, ont une complexité en temps de $O(\frac{N^2}{P})$ et utilisent $2P - 1$ rondes de communication. Nous présentons aussi des résultats expérimentaux qui montrent que ces algorithmes sont très efficaces, pour des grandes suites de données, quand

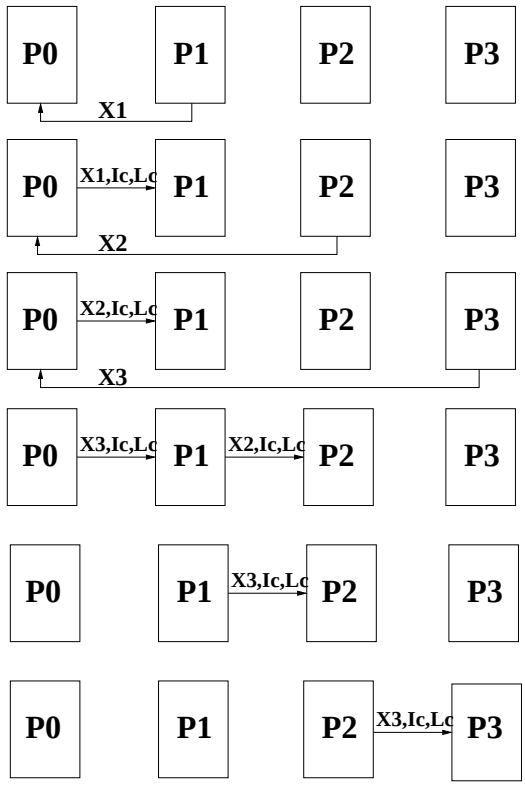


FIG. 4.11 – Rondes de communication pour le problème DR (pour 4 processeurs).

nous augmentons le nombre de processeurs. Á partir des solutions systoliques linéaires bidirectionnelles, nous avons donc réussi à dériver deux solutions CGM qui conservent le même travail.

Chapitre 5

Discussion

5.1 Introduction

Le travail developpé durant cette thèse, nous a conduit à nous poser un certain nombre de questions auxquelles nous avons tenté de trouver des réponses. Ainsi, ce chapitre est dédié à une discussion portant sur deux points : l'équilibrage de charge et la prédictivité.

Nous venons de voir dans les deux précédents chapitres qu'il était possible de dériver des architectures systoliques en solutions CGM. Cependant, nous constatons que la charge de travail n'est pas la même sur chaque processeur. Ce déséquilibre de charge est intrinsèquement lié aux problèmes étudiés. Nous nous sommes donc intéressés à ce problème d'équilibrage de charge et une méthode simple mais efficace nous a permis de résoudre ce problème. Ainsi le détail de cette méthode se trouve dans la section 5.2 de ce chapitre.

La prédictivité est à la fois un concept facile à définir et difficile à cerner. En effet, on peut la définir de la façon suivante : dire ce qu'on prévoit, par raisonnement et par conjecture, devoir arriver. La difficulté réside dans la sélection des paramètres que l'on souhaite prédire. Dans ce chapitre, nous tentons de faire une extrapolation des résultats de nos travaux afin de prédire quelles sont les adaptations envisageables des architectures systoliques au modèle CGM.

5.2 Équilibrage de charge sur le modèle CGM

Ce chapitre présente une méthode d'équilibrage de charge pour les algorithmes CGM présentés dans les chapitres précédents. Dans ces algorithmes, P processeurs travaillent sur

$\frac{N}{P}$ données. Cependant, la charge de travail des processeurs n'est pas optimale. Par exemple, dans le problème LRSE, le premier processeur travaille une fois, le second processeur travaille deux fois et ainsi de suite. Notre but dans ce chapitre est de proposer une méthode qui permettrait d'obtenir une bonne répartition de charge pour chaque processeur. Dans [ACDS03], Alves et al., se sont aussi intéressés au problème de l'équilibrage de charge dans le modèle CGM. Ils ont tenté d'assigner la charge de travail d'un processeur de manière optimale. Pour ce faire, ils proposent de diminuer la taille des messages que le processeur P_i envoie au processeur P_{i+1}. Au lieu d'une taille de message de l'ordre de $\frac{N}{P}$, ils considèrent une taille $\alpha \times \frac{N}{P}$ $(0 < \alpha \leq 1)$. Notre méthode va consister à mieux distribuer les rondes de communication plutôt que de travailler sur la taille des messages. Cette meilleure distribution conduit à une réduction du nombre de processeurs.

5.2.1 Description de la méthode d'équilibrage de charge

La méthode d'équilibrage de charge présentée dans ce chapitre est également disponible dans [GSb]. Notre but est de définir une technique qui va permettre d'améliorer la charge des processeurs. En fait, nos algorithmes CGM décrits précédemment sur P processeurs sont capables de travailler sur $\frac{P}{2}$ processeurs. Nous noterons lbP le nombre de processeurs réellement utilisés (avec lbP=$\frac{P}{2}$).

Les quatres problèmes que nous avons résolus (LIS,LCS,LRSE,DR - voir chapitre 3.2, 3.3, 4.2 et 4.3) dérivent de solutions systoliques. Ces solutions systoliques peuvent se partager en deux classes : les solutions basées sur un réseau linéaire unidirectionnel et celles basées sur un réseau linéaire bidirectionnel (voir 2.2.3).

Malgré l'efficacité de nos solutions, la charge de travail de nos processeurs est très mauvaise. Nous allons donc proposer une meilleure distribution des données avec une réduction du nombre de processeurs.

Définitions et Propositions

Propriété 5.1 *Comme notre approche consiste en une distribution optimale des données, les calculs locaux de tous nos algorithmes CGM ne seront jamais modifiés.*

Définition 5.1 *Chaque processeur i possède un numéro d'identification physique lbP_i et deux ensembles de $\frac{N}{P}$ données : le i-ième ensemble et le $(lbP + i)$-ième ensemble.*

Définition 5.2 *Pour tout $i < lbP (= \frac{P}{2})$, le processeur i pourra recevoir tous les messages destiné au processsseur virtuel $lbP + i (= \frac{P}{2} + i)$.*

Propriété 5.2 *Les définitions 5.1 et 5.2 impliquent qu'un processeur i peut recevoir des messages dans deux cas, ceux destiné au processeur i et ceux destiné au processeur virtuel $lbP (= \frac{P}{2} + i)$.*

Définition 5.3 *Nous appelerons M_i et M_{lbP+i} les messages reçus par le processeur i destinés respectivement au processeur i et au processeur virtuel $lbP + i$.*

Lemme 5.1 *Un processeur i doit terminer le calcul des données provenant du message M_i avant de calculer celles du message M_{lbP+i}.*

Preuve : Du fait que les problèmes traités soient issus de la programmation dynamique, il est évident que les données du message M_i doivent être traitées avant les données du message M_{lbP+i}. □

Définition 5.4 *Toutes les rondes de communication doivent être déterministes.*

Propriété 5.3 *Les rondes de communication des algorithmes CGM pourront être modifiées afin de respecter toutes nos suppositions décrites ci-dessus.*

Équilibrage de charge pour les algorithmes CGM issus de solutions systoliques linéaires unidirectionnelles

Les algorithmes CGM basés sur des solutions systoliques linéaires unidirectionnels (voir figure 5.1) ont des rondes de communications dans une seule direction.

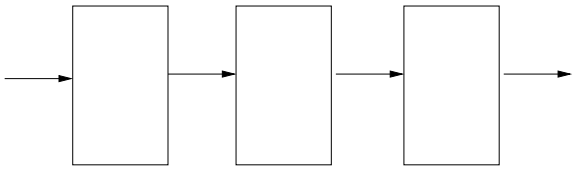


FIG. 5.1 – Réseau systolique linéaire unidirectionnel

Le problème LIS Nous avons décrit une solution au problème LIS dans la section 3.2 à partir d'une solution systolique linéaire unidirectionnelle. La partie droite de la figure 5.2 représente les rondes de communication utilisées pour 4 processeurs dans la solution CGM présentée dans la chapitre 3.2.4. La partie gauche de la figure 5.2 montre les rondes de communication utilisées dans notre méthode d'équilibrage de charge.
Nous représentons en noir les processeurs qui effectuent des calculs locaux et en blanc les processeurs qui ne sont pas utilisés pour des calculs locaux. Notons que le nombre de rondes de la solution CGM classique qui utilise 4 processeurs est 4 et que nous avons le même nombre de rondes pour la solution CGM avec équilibrage de charges sur 2 processeurs.
Quand nous observons toutes les rondes de communication, le nombre de processeurs qui effectue des calculs locaux (processeurs noirs sur la figure 5.2) est seulement de 43% dans la solution CGM classique (7 processeurs noirs et 9 processeurs blancs) et 75% pour la solution CGM avec équilibrage de charge (6 processeurs noirs et 2 processeurs blancs).

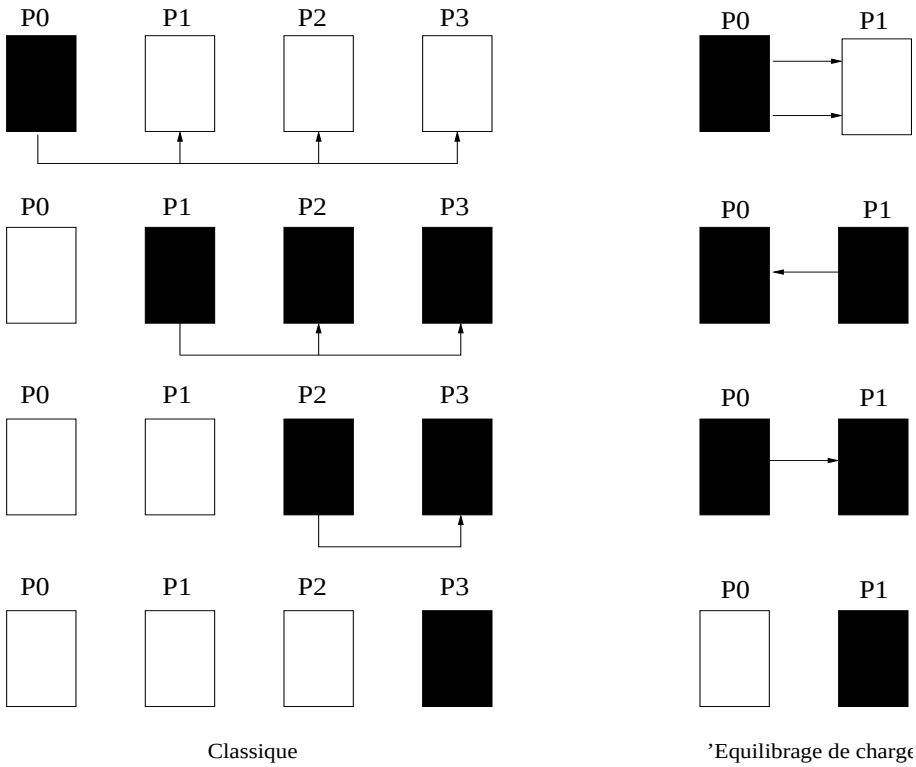


FIG. 5.2 – Rondes de communication pour la méthode classique et avec équilibrage de charge pour le problème LIS (avec P=4 et lbP=2 respectivement).

Plus généralement, la charge de travail pour la solution CGM classique est $\frac{P^2-P+2}{2P^2}$ et la charge de travail globale pour la solution avec équilibrage de charge est $\frac{3lbP^2-lbP+2}{4lbP^2}$. En fait, la charge de travail globale pour la solution CGM classique est approximativement égale à 50% et la charge de travail globale pour la solution avec équilibrage de charge est approximativement égale à 75% pour $P \geq 20$ et $lbP \geq 10$.

Néanmoins, la solution avec équilibrage de charge entraine une surcharge de quelques processeurs. En fait, ces processeurs exécutent deux fois les calculs locaux. Le nombre de processeurs possédant une charge supplémentaire est $\frac{lbP-1}{4lbP}$ pour $lbP \geq 2$. La charge de travail supplémentaire globale est approximativement égale à 25% pour $lbP \geq 10$.

Le problème LCS Nous avons décrit une solution au problème LCS dans la section 3.3 à partir d'une solution systolique linéaire unidirectionnelle. La partie droite de la figure 5.3 représente les rondes de communication utilisés pour 4 processeurs dans la solution CGM présentée dans la chapitre 3.3.4. La partie gauche de la figure 5.3 montre les rondes de communication utilisées dans notre méthode d'équilibrage de charge.

Notons que le nombre de rondes de la solution CGM classique qui utilise 4 processeurs et le nombre de rondes pour la solution CGM avec équilibrage de charge sur 2 processeurs est 7.

Quand nous observons toutes les rondes de communication, le nombre de processeurs qui effectue des calculs locaux (processeurs noirs dans la figure 5.3) est seulement de 57% dans la solution CGM classique (16 processeurs noirs et 12 processeurs blancs) et 85% pour la solution CGM avec équilibrage de charge (12 processeurs noirs et 2 processeurs blancs).

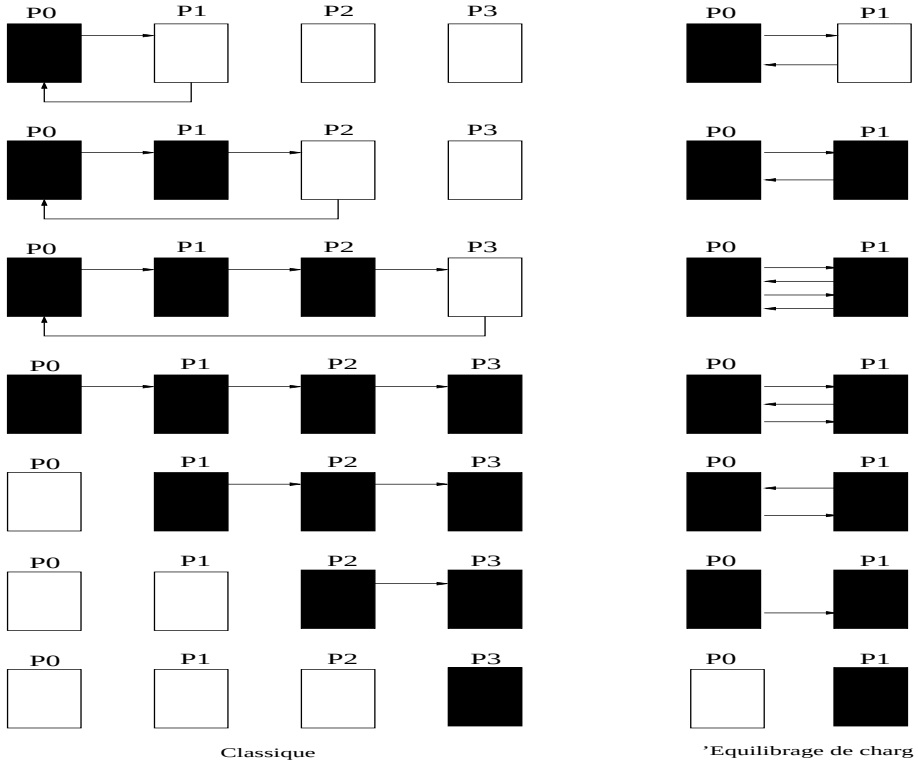


FIG. 5.3 – Rondes de communication pour la méthode classique et avec équilibrage de charge pour le problème LCS (avec P=4 et lbP=2 respectivement).

La charge de travail pour la solution CGM classique est $\frac{P^2}{2P^2-P}$ et la charge de travail globale pour la solution avec équilibrage de charge est $\frac{3lbP^2}{4lbP^2-lbP}$. Nous avons donc une charge de travail globale pour la solution CGM classique approximativement égale à 50% et une charge de travail globale pour la solution avec équilibrage de charge approximativement égale à 75% pour $P \geq 20$ et $lbP \geq 10$.

Néanmoins, dans la solution avec équilibrage de charge, nous observons que quelques processeurs ont une charge supplémentaire. En fait, ces processeurs exécutent deux fois les calculs locaux. Le nombre de processeurs possédant une charge supplémentaire est $\frac{lbP^2}{4lbP^2-lbP}$ pour $lbP \geq 2$. La charge de travail supplémentaire globale pour la solution est approximativement égale à 25% pour $lbP \geq 10$.

Équilibrage de charge pour les algorithmes CGM issus de solutions systoliques linéaires bidirectionnelles

Les algorithmes CGM basés sur des solutions systoliques linéaires bidirectionnels (voir figure 5.4) ont des rondes de communication dans deux directions.

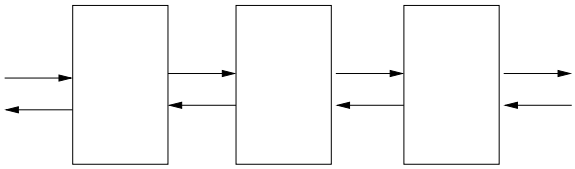


FIG. 5.4 – Réseau systolique linéaire bidirectionnel

Le problème LRSE Nous avons décrit une solution au problème LRSE dans la section 4.2 à partir d'une solution systolique linéaire bidirectionnelle. La partie droite de la figure 5.5 représente les rondes de communication utilisées pour 4 processeurs dans la solution CGM présentée dans la chapitre 4.2.4. La partie gauche de la figure 5.5 montre les rondes de communication utilisées dans notre méthode d'équilibrage de charge.

Notons que le nombre de rondes de la solution CGM classique qui utilise 4 processeurs

et le nombre de rondes pour la solution CGM avec équilibrage de charge sur 2 processeurs est le même, c'est-à-dire 6.

Quand nous observons toutes les rondes de communication, le nombre de processeurs qui effectue des calculs locaux (processeurs noirs sur la figure 5.5) est seulement de 35% pour la solution CGM classique (7 processeurs noirs et 9 processeurs blancs) et 71% pour la solution CGM avec équilibrage de charge (6 processeurs noirs et 2 processeurs blancs).

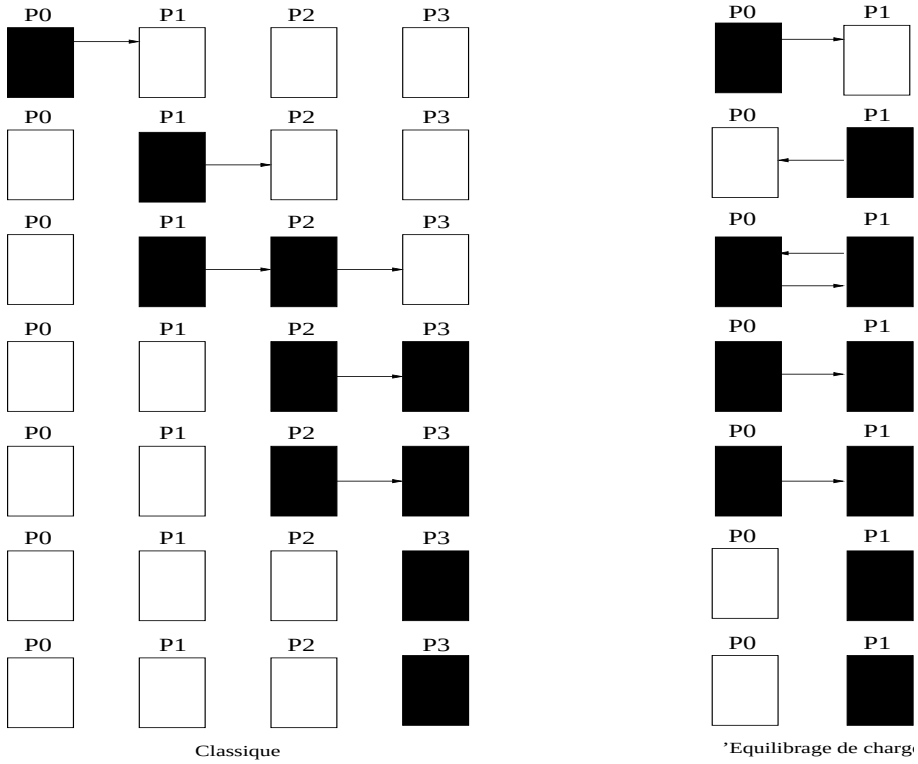


FIG. 5.5 – Rondes de communication pour la méthode classique et avec équilibrage de charge pour le problème LRSE (avec P=4 et lbP=2 respectivement).

La charge de travail pour la solution CGM classique est $\frac{P+1}{4P-2}$ et la charge de travail globale pour la solution avec équilibrage de charge est $\frac{2P+1}{4P-1}$. En fait, la charge de travail globale pour la solution CGM classique est approximativement égale à 25% et la charge de travail globale pour la solution avec équilibrage de charge est approximativement égale à 50% pour $P \geq 20$ et $lbP \geq 10$. Notons qu'aucun processeur n'a de charge supplémentaire pour la solution avec équilibrage de charge du problème LRSE.

Le problème DR Nous avons décrit une solution au problème DR dans la section 4.3 à partir d'une solution systolique linéaire bidirectionnelle. La partie droite de la figure 5.6 représente les rondes de communication utilisées pour 4 processeurs dans la solution CGM présentée dans la chapitre 4.3.4. La partie gauche de la figure 5.6 montre les rondes de communication utilisées dans notre méthode d'équilibrage de charge.

Notons que le nombre de rondes de la solution CGM classique qui utilise 4 processeurs et le nombre de rondes pour la solution CGM avec équilibrage de charge sur 2 processeurs est le même, c'est-à-dire 7.

Quand nous observons toutes les rondes de communication, le nombre de processeurs qui effectue des calculs locaux (processeurs noirs dans la figure 5.6) est seulement de 32% dans la solution CGM classique (9 processeurs noirs et 19 processeurs blancs) et 64% pour la solution CGM avec équilibrage de charge (9 processeurs noirs et 5 processeurs blancs).

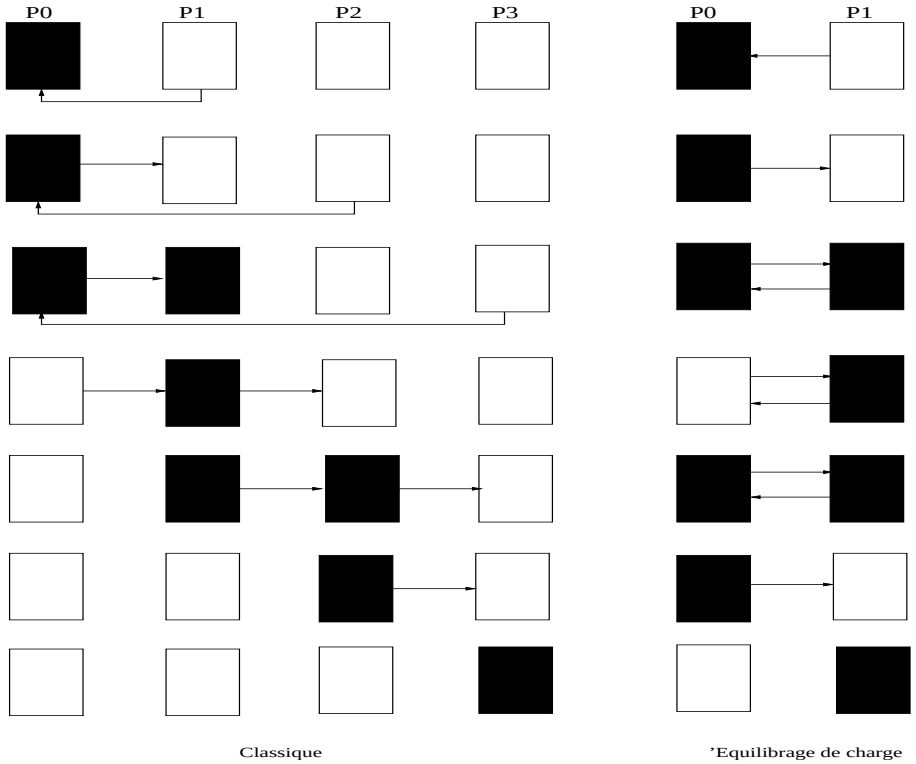


FIG. 5.6 – Rondes de communication pour la méthode classique et avec équilibrage de charge pour le problème DR (avec P=4 et lbP=2 respectivement).

La charge de travail pour la solution CGM classique est $\frac{P^2+P-2}{4P^2-2P}$ et la charge de travail globale pour la solution avec équilibrage de charge est $\frac{2lbP^2+lbP-1}{4lbP^2-lbP}$. Ainsi, la charge de travail globale pour la solution CGM classique est approximativement égale à 25% et la charge de travail globale pour la solution avec équilibrage de charge est approximativement égale à 50% pour $P \geq 20$ et $lbP \geq 10$. Notons qu'aucun processeur n'a de charge supplémentaire pour la solution avec équilibrage de charge du problème DR.

5.3 Du systolique au CGM

L'idée est maintenant d'essayer de prédire quelles architectures systoliques peuvent être dérivées en algorithmes CGM en conservant le même travail. La notion de prédictivité peut être définie comme l'annonce de ce qu'on prévoit par raisonnement et par conjecture. Les résultats présentés dans les chapitres précédents montrent qu'il a été possible d'écrire quatre algorithmes CGM efficaces à partir de quatre architectures systoliques. Dans un premier temps, nous allons examiner quelles caractéristiques de ces architectures systoliques ont permis d'obtenir des algorithmes efficaces sur le modèle CGM. Dans un second temps, nous essayerons de prédire quels problèmes pourraient se voir appliquer le même procédé.

5.3.1 Observations

On peut d'abord noter que les quatre problèmes étudiés dans les chapitres précédents peuvent être résolus grâce à la programmation dynamique. Ainsi, du fait de leur caractère récursif, on peut leur trouver naturellement des solutions systoliques. Les algorithmes séquentiels résolvant ces problèmes par la programmation dynamique ont une complexité quadratique. Il est donc logique que les architectures systoliques associées soient basées sur des réseaux linéaires. Comme leur complexité en temps (nombre de cycles systoliques) est également linéaire, nous retrouvons un travail quadratique cohérent avec la complexité des solutions séquentielles. Notons également, que les quatre architectures systoliques sont modulaires. En fait, seuls les calculs locaux et les communications entre les processeurs diffèrent d'une solution à l'autre.

L'adaptation au modèle CGM des solutions systoliques étudiées dans cette thèse est basée sur le principe suivant : chaque processeur du modèle CGM simule un groupe de processeurs systoliques. En effet, le nombre de processeurs dans le modèle CGM étant de $P \leq \frac{N}{P}$ (où N représente la taille du problème), on regroupe sur chacun de ces processeurs

les données contenues dans $\frac{N}{P}$ processeurs systoliques. Ainsi, le calcul local réalisé sur ces $\frac{N}{P}$ données correspond au travail des $\frac{N}{P}$ processeurs systoliques et sa complexité est donc de $\frac{N^2}{P^2}$. La difficulté réside alors dans le choix du schéma de communication entre les processeurs du modèle CGM. Celui-ci est bien évidemment lié à l'architecture systolique de laquelle est dérivée la solution CGM. Comme il a été rappelé précédemment, les architectures systoliques auxquelles nous nous sommes intéréssées sont basées sur des réseaux linéaires. Ainsi, le nombre de rondes de communication dans nos solutions CGM est de $O(P)$, ce qui conduit à un travail de $O(N^2)$ pour chacune de nos solutions.

5.3.2 Prédictivité

Nous pensons que le procédé utilisé pour les quatre architectures systoliques présentées dans les chapitres précédents peut être réutilisé de façon générale pour d'autres architectures systoliques.

Théorème 5.1 *Toute architecture systolique linéaire de N processeurs ayant une complexité en temps de N peut être dérivée en solution CGM utilisant $P \leq \frac{N}{P}$ processeurs ayant le même travail $O(N^2)$ (où N représente la taille du problème).*

Preuve : On regroupe sur chacun des P processeurs du modèle CGM les données contenues dans $\frac{N}{P}$ processeurs systoliques. Ainsi, le calcul local réalisé sur ces $\frac{N}{P}$ données correspond au travail des $\frac{N}{P}$ processeurs systoliques et sa complexité est donc de $\frac{N^2}{P^2}$. Le schéma de communication entre les processeurs du modèle CGM est bien évidemment lié à l'architecture systolique de laquelle est dérivée la solution CGM. Comme nous avons affaire à des architectures systoliques linéaires, le nombre de rondes de communication dans nos solutions CGM est de $O(P)$, ce qui conduit à un travail de $O(N^2)$. □

Il serait maintenant intéressant d'étudier d'autres types d'architectures systoliques, par exemple quadratiques, afin de leurs appliquer la même démarche.

Chapitre 6

Résultats expérimentaux

Ce chapitre présente les expérimentations que nous avons effectuées sur une plateforme de PC. Ces résultats expérimentaux illustrent bien la performance de nos algorithmes. Nous présenterons le cadre expérimental puis les résultats obtenus pour les quatres problèmes LIS, LCS, LRSE et DR. En fin de chapitre, nous donnerons les résultats pour notre méthode d'équilibrage de charge.

6.1 Cadre expérimental

Pour les quatres problèmes LIS, LCS, LRSE et DR, nous avons développé et implanté les programmes en langage C en utilisant la librairie de communication MPI. Nous avons testé ces programmes sur une plateforme multiprocesseur Celeron 466MHz fonctionnant sous LINUX. Les communications entre les processeurs ont été réalisées à travers un switch Ethernet. Les tests ont été lancés avec des valeurs de $N{=}2^k$, où k est un entier tel que $12 \leq k \leq 18$. Nous avons utlisé le même cadre expérimental pour le développement et l'implantation des programmes issus de notre méthode d'équilibrage de charges.
Les tables 6.1, 6.3, 6.5 et 6.7 présentent le temps total d'exécution (en secondes) pour chaque configuration de 1, 4, 8 et 16 processeurs.
Les figures 6.1, 6.3, 6.5 et 6.7 présentent le temps total d'exécution (en secondes) pour différents nombres de données. Les courbes représentent les configurations de 1, 4, 8 et 16 processeurs.
Les tables 6.2, 6.4, 6.6 et 6.8 présentent le temps de communication (en secondes) pour chaque configuration de 4, 8 et 16 processeurs.

Les figures 6.2, 6.4, 6.6 et 6.8 montrent le temps de communication (en secondes) pour différents nombres de données. Ces temps de communication correspondent à l'envoi et la réception de N données par un processeur aux autres.

6.2 Problèmes sur réseau linéaire unidirectionnel et bidirectionnel

6.2.1 Plus longue sous-suite croissante

N	P=1	P=4	P=8	P=16
4 096	0,33	0,38	0,22	0,14
8 192	1,66	1,48	0,85	0,50
16 384	7,12	5,89	3,34	1,91
32 768	35,40	23,95	13,47	7,51
65 536	153,30	111,26	54,08	30,44
131 072	624,16	508,53	248,92	122,33
262 144	2 509,00	2 104,24	1 123,26	551,10

TAB. 6.1 – Temps total d'exécution pour le problème LIS (en secondes).

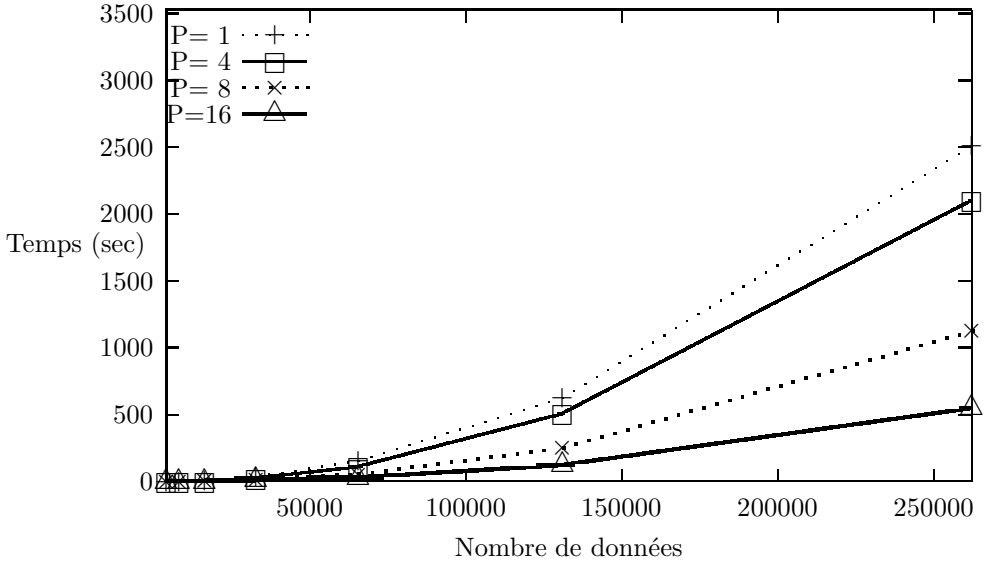


FIG. 6.1 – Temps total d'exécution pour le problème LIS (en secondes) pour différents nombres de données.

N	P=4	P=8	P=16
4 096	0,007	0,043	0,035
8 192	0,009	0,045	0,042
16 384	0,042	0,051	0,064
32 768	0,057	0,065	0,112
65 536	0,093	0,100	0,231
131 072	0,134	0,164	0,567
262 144	0,260	0,426	0,930

TAB. 6.2 – Temps total de communication pour le problème LIS (en secondes).

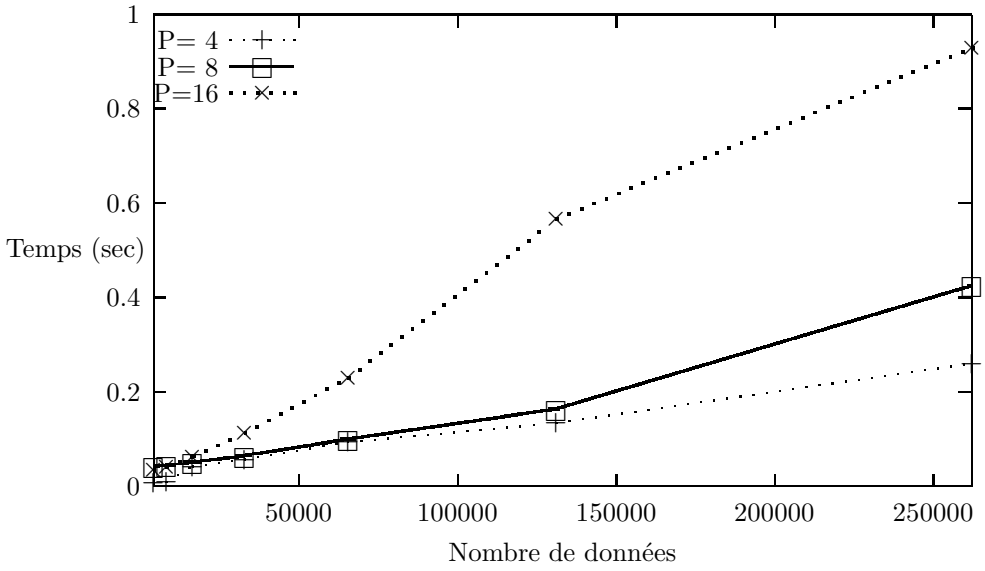


FIG. 6.2 – Temps de communication pour le problème LIS (en secondes) pour différents nombres de processeurs.

6.2.2 Plus longue sous-suite commune à deux mots

N	P=1	P=4	P=8	P=16
4 096	3,85	3,17	2,62	2,46
8 192	17,00	12,41	10,34	9,56
16 384	72,96	49,82	41,26	37,77
32 768	292,49	222,66	169,36	150,54
65 536	1 165,53	947,61	726,58	613,33
131 072	4 658,87	3 768,51	3 146,09	2 582,57
262 144	18 643,85	15 082,07	12 495,77	11 431,88

TAB. 6.3 – Temps total d'exécution pour le problème LCS (en secondes).

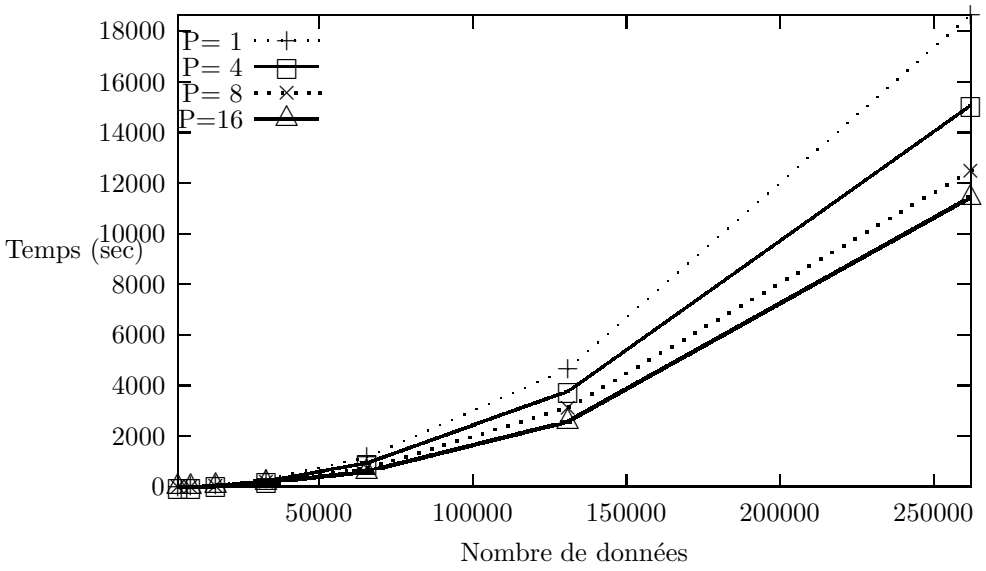


FIG. 6.3 – Temps total d'exécution pour le problème LCS (en secondes) pour différents nombres de données.

N	P=4	P=8	P=16
4 096	0,025	0,514	0,107
8 192	0,046	0,084	0,222
16 384	0,161	0,153	0,339
32 768	0,297	0,541	0,517
65 536	0,459	0,804	1,156
131 072	0,690	1,360	2,382
262 144	1,356	2,407	4,688

TAB. 6.4 – Temps total de communication pour le problème LCS (en secondes).

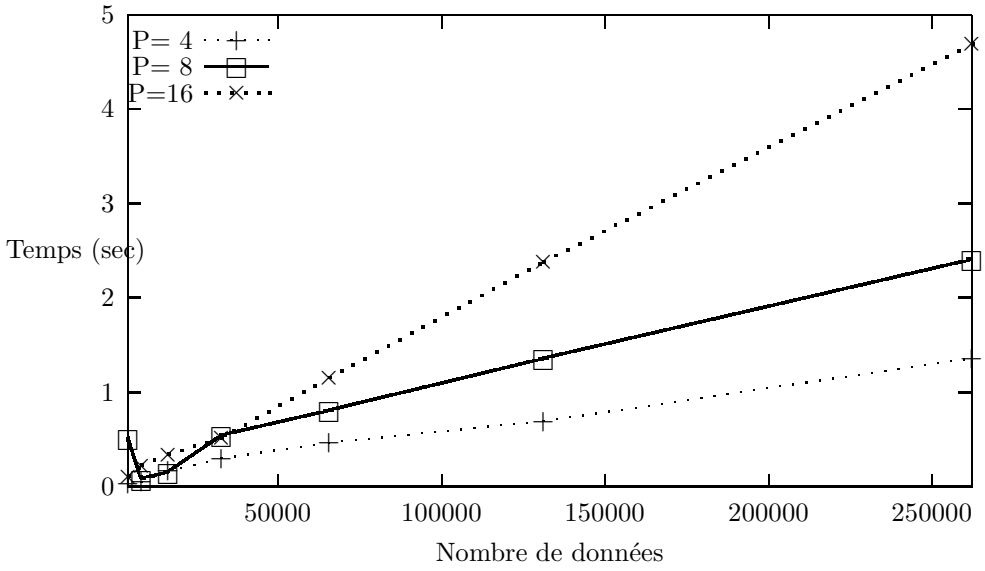


FIG. 6.4 – Temps de communication pour le problème LCS (en secondes) pour différents nombres de processeurs.

6.2.3 Plus long suffixe répété en chaque caractère d'un mot

N	P=1	P=4	P=8	P=16
4 096	0,67	0,99	0,63	0,41
8 192	2,91	3,98	2,47	1,54
16 384	15,70	15,89	9,84	6,09
32 768	77,86	70,95	39,21	24,08
65 536	338,38	301,64	168,24	96,26
131 072	1 387,90	1 341,77	796,86	402,70
262 144	5 559,11	5 408,22	3 398,74	1 797,66

TAB. 6.5 – Temps total d'exécution pour le problème LRSE (en secondes).

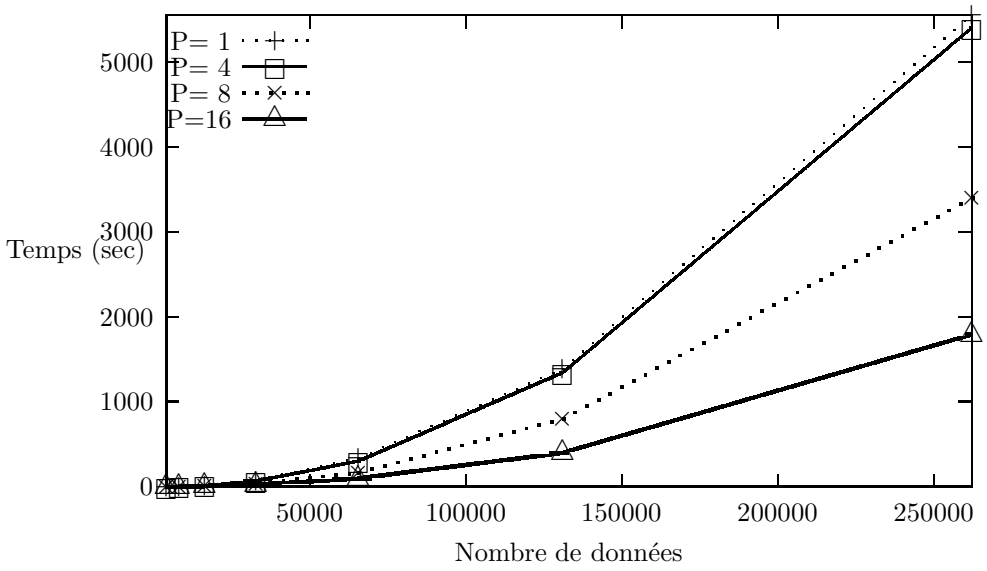


FIG. 6.5 – Temps total d'exécution pour le problème LRSE (en secondes) pour différents nombres de données.

N	P=4	P=8	P=16
4 096	0,008	0,044	0,035
8 192	0,011	0,045	0,042
16 384	0,046	0,055	0,050
32 768	0,068	0,071	0,073
65 536	0,070	0,111	0,135
131 072	0,154	0,276	0,279
262 144	0,302	0,720	1,025

TAB. 6.6 – Temps total de communication pour le problème LRSE (en secondes).

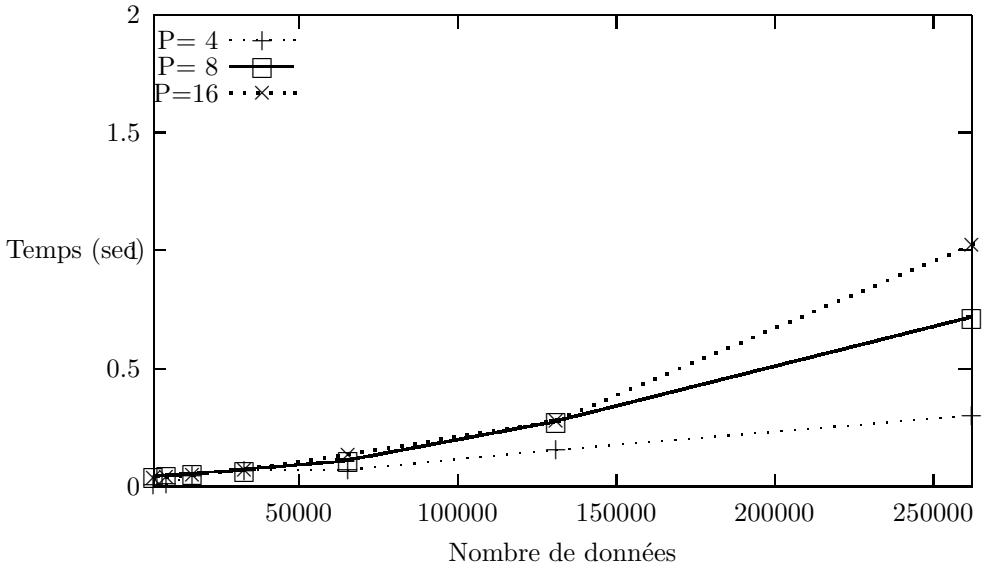


FIG. 6.6 – Temps de communication pour le problème LRSE (en secondes) pour différents nombres de processeurs.

6.2.4 Recherche de répétitions

N	P=1	P=4	P=8	P=16
4 096	4,17	2,33	1,48	0,81
8 192	16,82	9,22	5,86	3,14
16 384	67,62	36,95	23,89	12,46
32 768	272,58	172,85	93,63	49,45
65 536	1 103,57	693,67	457,53	198,08
131 072	4 420,59	3 250,28	1 845,57	1 012,15
262 144	17 694,27	12 022,14	9 747,33	4 089,96

TAB. 6.7 – Temps total d'exécution pour le problème DR (en secondes).

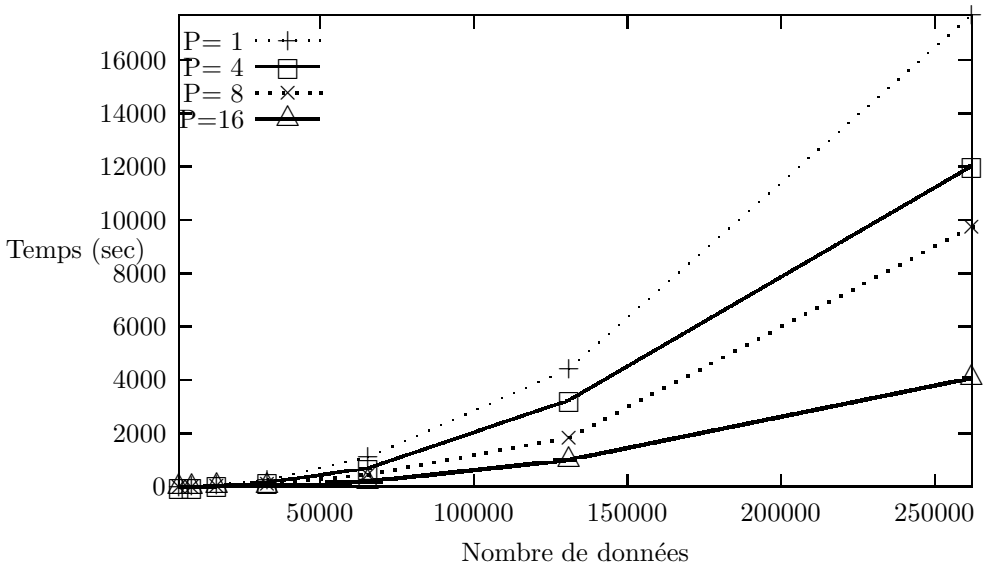


FIG. 6.7 – Temps total d'exécution pour le problème DR (en secondes) pour différents nombres de données.

N	P=4	P=8	P=16
4 096	0,009	0,041	0,035
8 192	0,015	0,038	0,038
16 384	0,060	0,036	0,051
32 768	0,085	0,091	0,081
65 536	0,148	0,176	0,197
131 072	0,256	0,447	0,611
262 144	0,510	0,847	1,540

TAB. 6.8 – Temps total de communication pour le problème DR (en secondes).

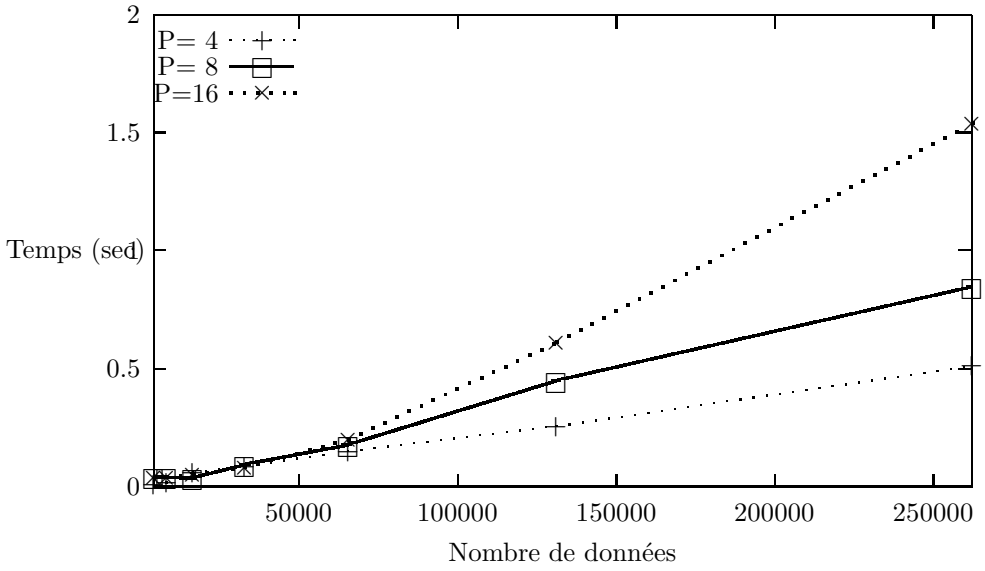


FIG. 6.8 – Temps de communication pour le problème DR (en secondes) pour différents nombres de processeurs.

6.2.5 Analyse des résultats expérimentaux

Les résultats expérimentaux ont été effectués sur des PC dédiés aux enseignements et par conséquent utilisés par des étudiants, le WE, la nuit ou bien durant les congés universitaires. Comme nous n'avions pas de réinitialisation des machines, nous ne pouvons pas garantir la stabilité de ces machines ni l'uniformité des résultats obtenus. En effet, certains ordinateurs n'étaient pas exempt de tâches de fond initiées par des étudiants. Mais, les résultats expérimentaux réalisés donne une bonne indication des résultats attendus et permettent une validation de nos algorithmes.

Comme il a été dit précédemment, les valeurs de N (le nombre d'entiers) varie linéairement de 4 096 à 262 144 soit 2^{12} à 2^{18} (N=2^k, $12 \leq k \leq 18$). Nous observons que les temps d'exécution obtenus expérimentalement augmentent de façon quadratique pour un nombre fixé P de processeurs ($4 \leq P \leq 16$) par rapport à N (voir figures 6.1, 6.3, 6.5 et 6.7). En effet, pour passer d'une ligne à la ligne suivante dans les tables 6.1, 6.3, 6.5 et 6.7, on multiplie la valeur de N par deux alors que le temps d'exécution pour un nombre fixé de processeurs est multiplié par environ quatre (à l'exception de $\frac{9747.33}{1845.57} = 5,28$ et de $\frac{1012.15}{198.08} = 5,10$ pour le problème DR). En fait pour un nombre fixé de processeurs P, si le temps d'exécution est de T_{ex} pour $N = 2^k$, nous aurons un temps d'exécution de $4^i \times T_{ex} = (2^i)^2 \times T_{ex}$ pour $N = 2^{k+i}$. Ceci est en parfait accord avec la complexité théorique de $O(N^2)$ des algorithmes utilisés pour effectuer les calculs locaux.

Pour les temps de communications, il est plus difficile de trouver une telle cohérence dans les résultats que nous pouvons observer tables 6.2, 6.4, 6.6 et 6.8. Notons toutefois que le temps de communication augmente en fonction du nombre de processeurs ainsi que du nombre de données. Ainsi, le temps de communication pour P=16 et N= 262 144 est plus important que pour P=16 et N= 131 072 ou que pour P=16 et N= 4 096. Ce résultat n'est évidemment pas surprenant compte tenu de ce qu'il mesure. Ce qui est plus intéressant c'est de noter que ces temps de communications sont très faibles par rapport au temps global d'exécution. En effet, ceux-ci représentent en moyenne moins de 1% du temps global.

6.3 Équilibrage de charge sur le modèle CGM

Les tables 6.9, 6.10, 6.11, 6.12 présentent le temps total d'exécution (en secondes) avec la méthode classique et la méthode avec répartition de charge, pour les problèmes LIS, LCS, LRSE, DR respectivement. Ainsi, pour chaque résultat obtenu par la méthode classique sur

une configuration de 4, 8 et 16 processeurs correspond un résultat obtenu sur une configuration de 2, 4 et 8 processeurs par la méthode d'équilibrage de charges.

6.3.1 Plus longue sous-suite croissante

N	lbP=2	lbP=4	lbP=8	P=4	P=8	P=16
4 096	0,49	0,29	0,19	0,38	0,22	0,14
8 192	1,87	1,14	0,64	1,48	0,85	0,50
16 384	7,47	4,49	2,51	5,89	3,34	1,91
32 768	30,45	17,97	9,76	23,95	13,47	7,51
65 536	143,33	76,74	38,92	111,26	54,08	30,44
131 072	872,37	343,80	158,63	508,53	248,92	122,33
262 144	3 584,23	1 997,78	731,97	2 104,24	1 123,26	551,10

TAB. 6.9 – Temps total d'exécution pour le problème LIS (en secondes) de la méthode classique (pour lbP=2, 4 et 8) et de la méthode avec équilibrage de charge (pour P=4, 8 et 16).

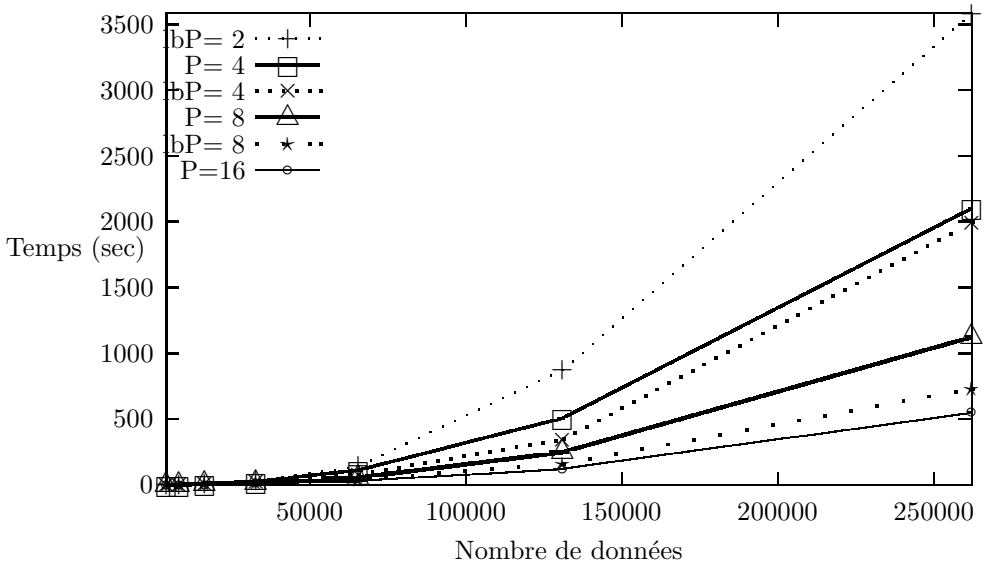


FIG. 6.9 – Temps d'exécution par nombre de processeurs pour le problème LIS (en secondes) pour différents nombres de données.

6.3.2 Plus longue sous-suite commune à deux mots

N	lbP=2	lbP=4	lbP=8	P=4	P=8	P=16
4 096	3,36	2,98	2,98	3,17	2,62	2,46
8 192	13,41	11,78	10,95	12,41	10,34	9,56
16 384	54,05	47,55	43,05	49,82	41,26	37,77
32 768	237,58	191,30	171,49	222,66	169,36	150,54
65 536	1 025,78	839,00	694,88	947,61	726,58	613,33
131 072	4 633,99	3 583,12	3 056,11	3 768,51	3 146,09	2 582,57
262 144	18 515,89	16 820,35	13 135,14	15 082,07	12 495,77	11 431,88

TAB. 6.10 – Temps total d'exécution pour le problème LCS (en secondes) de la méthode classique (pour lbP=2, 4 et 8) et de la méthode avec équilibrage de charge (pour P=4, 8 et 16).

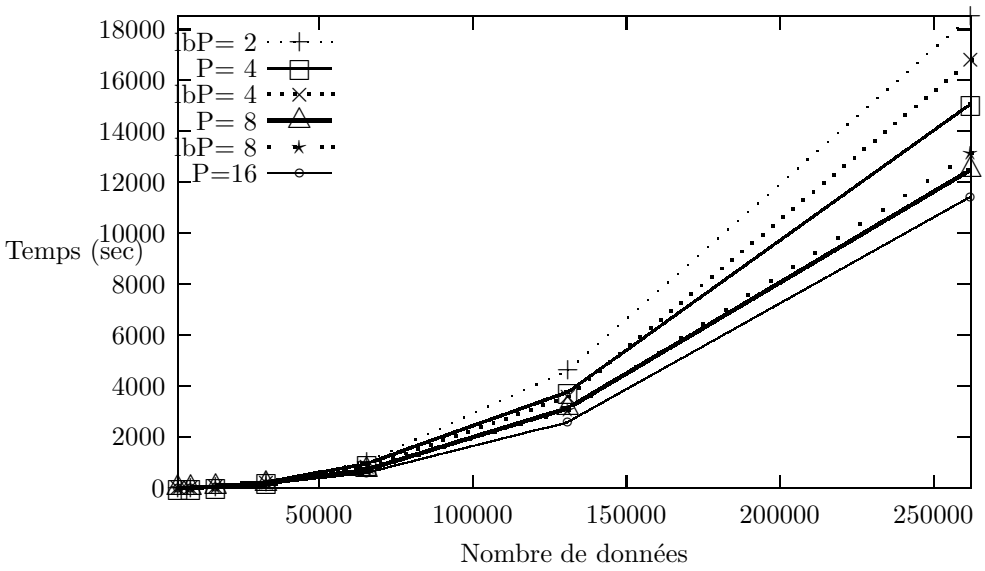


FIG. 6.10 – Temps d'exécution par nombre de processeurs pour le problème LCS (en secondes) pour différents nombres de données.

6.3.3 Plus long suffixe répété en chaque caractère d'un mot

N	lbP=2	lbP=4	lbP=8	P=4	P=8	P=16
4 096	0,99	0,63	0,36	0,99	0,63	0,41
8 192	3,97	2,48	1,39	3,98	2,47	1,54
16 384	15,95	9,82	5,40	15,89	9,84	6,09
32 768	68,73	39,23	21,52	70,95	39,21	24,08
65 536	302,48	166,76	85,87	301,64	168,24	96,26
131 072	1 474,76	748,90	371,06	1 341,77	796,86	402,70
262 144	5 988,63	3 997,75	1 640,80	5 408,22	3 398,74	1 797,66

TAB. 6.11 – Temps total d'exécution pour le problème LRSE (en secondes) de la méthode classique (pour lbP=2, 4 et 8) et de la méthode avec équilibrage de charge (pour P=4, 8 et 16).

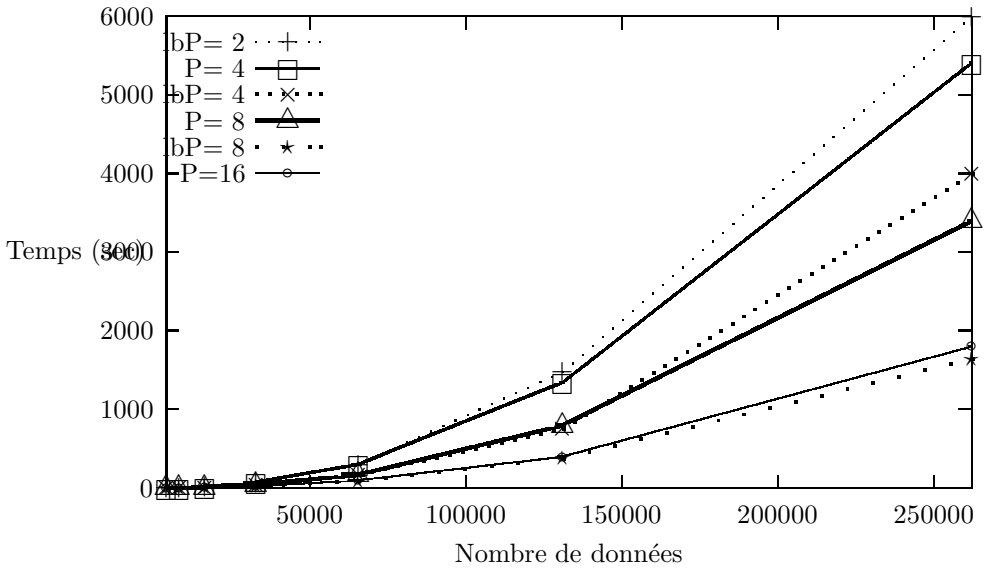


FIG. 6.11 – Temps d'exécution par nombre de processeurs pour le problème LRSE (en secondes) pour différents nombres de données.

6.3.4 Recherche de répétitions

N	lbP=2	lbP=4	lbP=8	P=4	P=8	P=16
4 096	2,31	1,36	0,75	2,33	1,48	0,81
8 192	9,25	5,39	2,91	9,22	5,86	3,14
16 384	36,98	21,53	11,56	36,95	23,89	12,46
32 768	164,71	86,18	46,15	172,85	93,63	49,45
65 536	663,10	444,67	184,57	693,67	457,53	198,08
131 072	2 975,06	1 785,86	1 029,63	3 250,28	1 845,57	1 012,15
262 144	11 983,66	9 191,08	4 155,06	12 022,14	9 747,33	4 089,96

TAB. 6.12 – Temps total d'exécution pour le problème DR (en secondes) de la méthode classique (pour lbP=2, 4 et 8) et de la méthode avec équilibrage de charge (pour P=4, 8 et 16).

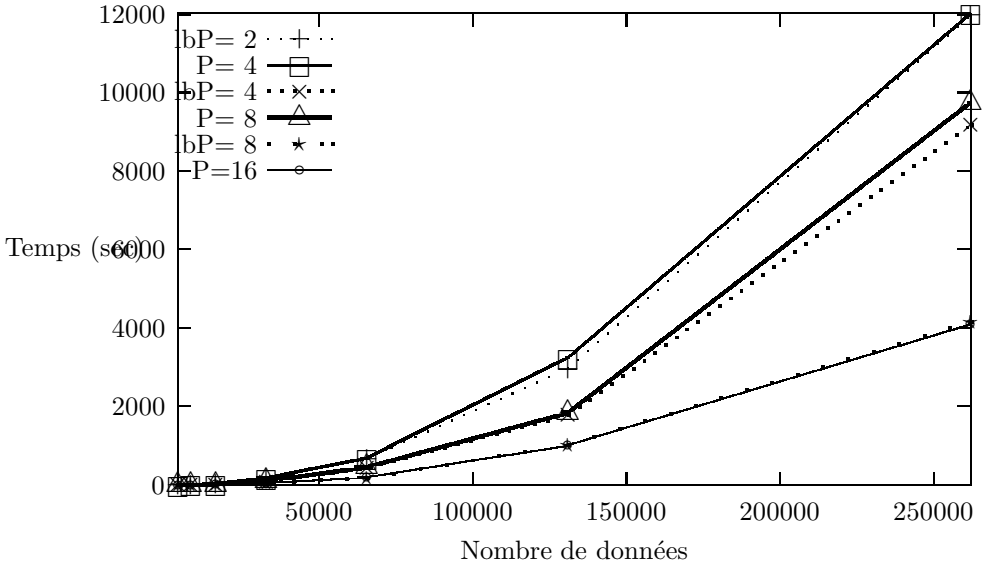


FIG. 6.12 – Temps d'exécution par nombre de processeurs pour le problème DR (en secondes) pour différents nombres de données.

6.3.5 Analyse des résultats expérimentaux

Les figures 6.9 à 6.12 nous offrent un comparatif entre la méthode dite classique et celle avec équilibrage de charge. Les analyses des résultats expérimentaux effectués valident les résultats théoriques de la section 5.2.1. En effet, pour les problèmes LRSE et DR, nous remarquons que les courbes pour différents nombres de processeurs sont presques identiques. Cela provient du fait qu'il n'y a pas de surcharge des processeurs utilisés. Pour les problèmes LIS et LCS, nous remarquons que les courbes diffèrent. Cela provient de la surcharge des processeurs utilisés. Nous pouvons même constater que la surcharge est en moyenne de 33% pour la solution avec équilibrage de charge pour le problème LIS ce qui est un peu éloigné des prédictions théoriques qui annoncaient une surcharge de 25%. Pour la solution avec équilibrage de charge du problème LCS, nous constatons une surcharge de 15% ce qui est également un peu éloigné de ce qui avait été annoncé précédemment avec une surcharge théorique de 25%.

Chapitre 7

Conclusions

Dans cette thèse nous nous sommes intéressés à des problèmes d'algorithmique du texte. Ainsi, nous avons proposé des solutions CGM aux problèmes de recherche de la plus longue sous-suite croissante, de la plus longue sous-suite commune à deux mots, du plus long suffixe répété en chaque caractère d'un mot et de répétitions. Pour cela, nous sommes partis de solutions systoliques existantes que nous avons adaptées au modèle CGM. Le but de ce travail était en fait double. D'une part, nous voulions proposer pour la première fois des solutions CGM à ces quatre problèmes. D'autre part, nous souhaitions montrer comment des solutions systoliques pouvait être dérivées en algorithmes CGM. En effet, de nombreux problèmes ont été étudiés sur des architectures systoliques, c'est-à-dire des machines dédiées, non réutilisables pour d'autres problèmes. Le modèle CGM quant à lui permet de travailler avec des machines peu coûteuses et réutilisables à souhaits.

De plus, l'expérience acquise au cours de ces travaux nous permet d'avoir une bonne idée des solutions systoliques adaptables au modèle CGM. Ceci pourrait permettre de consolider le pont existant entre modèles à grains fin et modèles à gros grain. D'autres résultats ont montré comment passer du modèle PRAM aux modèles à gros grain tels que BSP ou CGM (voir [Vie96, FGL98]). Dans le chapitre 5 nous avons proposé un théorème qui s'inscrit dans cette démarche. Ainsi, il dit que toute architecture systolique linéaire de N processeurs ayant une complexité en temps de N peut être dérivée en solution CGM utilisant $P \leq \frac{N}{P}$ processeurs ayant le même travail $O(N^2)$ (où N représente la taille du problème). Ce résultat est particulièrement intéressant puisque nous pouvons réutiliser les nombreux résultats sur réseaux systoliques linéaires.

Notre étude s'est portée uniquement sur des architectures systolique linéaires solutions de quatre problèmes d'algorithmique du texte. Il serait également intéressant de se pencher sur des problèmes dont les solutions systoliques reposent sur des architectures bidimensionnelles. Nous pourrions alors essayer de suivre la même démarche afin de compléter notre étude et d'établir de nouveaux liens entre des modèles de parallélisme.

Un autre champ d'exploration concerne le traitement de problèmes avec des données de très grande taille. En effet, le modèle CGM ne décrit absolument pas ce qui se passe lorsque la taille des données est telle qu'il est nécessaire de stocker celles-ci sur disque. Les problèmes de gestion mémoire et de gestion de disque deviennent alors cruciaux et peuvent très rapidement prendre le pas sur les temps de calculs locaux. Il serait alors très important d'inclure ce genre de paramètre dans le modèle CGM et d'en proposer une extension. D'autant plus que des travaux récents permettent à présent de lire des données sur disque par l'intermédiaire de cartes réseaux sans passer par la mémoire (voir [CRU03] et [CRU02]). Ceci permet donc aux processeurs de communiquer non plus par l'envoi de messages mais par mise en service d'un partage à distance des données. On pourrait donc imaginer qu'un processeur consomme ses données locales afin d'y effectuer un traitement puis va chercher une partie des données d'un processeur voisin sans déranger le calcul de celui-ci pour y effectuer un traitement et dépose le résultat sans même que le processeur voisin ne s'en rende compte.

Il serait également intéressant d'exploiter les implémentations des quatre problèmes traités dans cette thèse sur d'autres plate-formes afin d'en étudier la portabilité. En effet, les solutions envisagées ne prennent pas en compte l'architecture de la machine parallèle ou de la grappe de machines utilisées. Ainsi, on peut penser que les codes développés sont tout à fait prêts pour une utilisation sur d'autres plate-formes. Une tentative a d'ailleurs été réalisée au cours de cette thèse sur une grappe de machines à processeurs alpha AXP. Cette tentative fut concluante puisque les programmes fonctionnaient en l'état. Malheureusement, les tests n'ont pas pu durer très longtemps car cette plate-forme n'était pas stable puisqu'elle était utilisée pour des travaux « hardware ». Toutefois, cette expérience laisse à penser qu'il serait tout à fait possible d'utiliser plusieurs plate-formes connectées entre-elle. Rien ne nous empêche de croire, en effet, que nos programmes ne puissent fonctionner sur des plate-formes hétérogènes. Ils utilisent une barrière de synchronisation naturelle puisque

les envois et les réceptions sont bloquants. Il n'est donc pas necéssaire que les processeurs aient la même vitesse d'horloge. La synchronisation se fait donc au niveau des échanges de données entre deux phases de calculs locaux.

Le travail ainsi présenté dans cette thèse n'est que le début d'un travail plus conséquent sur le modèle CGM car, comme nous venons de la voir, de nombreuses voies restent encore à explorer.

Annexe A

Programme LIS_CGM

```
/*--------------------------------------------------------------*/
/* Program     : CGM Solution for the LIS problem               */
/*                                                              */
/* Authors     : Thierry Garcia, Jean-Frederic Myoupo, David Seme */
/* Programming : Thierry Garcia                                 */
/* Created     : 15/02/2001                                     */
/* Modified    : 11/04/2001                                     */
/*--------------------------------------------------------------*/

/* Parameters of the program :             */
/* N (data dim)                            */

/*--------------------------------------------------------------*/
/* Function    : Include files                                  */
/* Programming : Thierry Garcia                                 */
/* Created     : 15/02/2001                                     */
/* Modified    : 17/03/2001                                     */
/*--------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

/*--------------------------------------------------------------*/
/* Function    : Define                                         */
/* Programming : Thierry Garcia                                 */
/* Created     : 15/02/2001                                     */
/* Modified    : 11/04/2001                                     */
/*--------------------------------------------------------------*/
#define LIMIT 100        /* up limit for random number        */
#define control 0
#define verbose 0

/*--------------------------------------------------------------*/
/* Function    : Generate random number                         */
/* Programming : Thierry Garcia                                 */
/* Created     : 21/02/2001                                     */
/* Modified    : 21/03/2001                                     */
/*--------------------------------------------------------------*/
void generate_sub(long *sub, long dim, int numproc)
{
    long i;
    srand(numproc+1);
```

```
    i=0;
    while (i<dim)
    {
       sub[i++]=rand() % LIMIT;
    }
}

/*-------------------------------------------------------------*/
/* Function    : Main of the program                           */
/* Programming : Thierry Garcia                                */
/* Created     : 15/02/2001                                    */
/* Modified    : 11/04/2001                                    */
/*-------------------------------------------------------------*/
main (int argc, char **argv)
{
    int myid;                       /* Processor id in MPI     */
    int namelen;                    /* Length of Processor name*/
    int size;                       /* Number of Processor     */
    long P;                         /* P                       */
    long N;                         /* N                       */
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Status status;              /* Variable for MPI        */
    MPI_Request req;
    MPI_Datatype vector;

    long num,nump;      /* Processor Name          */
    long *A,*AP;                    /* Subsequences            */
    long i,ii,j,p,s;               /* Counters                */
    long *Major,*MajorP;            /* Major List              */
    long *RLIS;                     /* Result List = LIS       */
    long Max;             /* Maximum of Major List   */
    long Maxp;                      /* Maximum of the processor*/
    long *Max_proc;                 /* Maximum for a processor */
    float begintime,endtime;        /* Begin and end timer     */
    float begintimec,endtimec;      /* Begin and end timer comm*/
    char filename[14];              /* Name of file            */
    char filename2[14];             /* Name of file            */
    char convertstr[5];             /* Convert number to string*/
    FILE *fp;                       /* File pointer            */

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* How Many Processor ? */
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    P=size;          /* number of processor */

    N=atol(argv[1]); /* data length */

    if (((A=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((AP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Major=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((MajorP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Max_proc=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((RLIS=(long*) malloc(N * sizeof(long))) == NULL))
    {
        printf("Memory allocation error");
        exit(1);
    }

    /* Where am i ?     */
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    num = myid;      /* processor number    */
```

```
/* Vector datatype for communication    */
MPI_Type_vector(N/P,1,1,MPI_LONG,&vector);
MPI_Type_commit(&vector);

/* Get processor name */
MPI_Get_processor_name(processor_name,&namelen);

/* Generate random subsequence A */
generate_sub(A,N/P,num);

if (control==1)
{
    filename[0]='\0';
    strcat(filename,"RLISP");
    gcvt((double) size,5,convertstr);
    strcat(filename,convertstr);
    strcat(filename,"N");
    strcat(filename,argv[1]);
    strcat(filename,"p");
    gcvt((double) num,5,convertstr);
    strcat(filename,convertstr);
    fp=fopen(filename,"a+");
}
if (verbose==1)
{
    printf("Execute on Proc : %d [%s]\n",num,processor_name);
    for (i=0;i<N/P;i++) printf("A[%i]=%i\n",i,A[i]);
}
if (control==1)
{
    fprintf(fp,"Execute on Proc : %d [%s]\n",num,processor_name);
    for (i=0;i<N/P;i++) fprintf(fp,"A[%i]=%i\n",i,A[i]);
}

/* Begin counter on last processor */
if(num==P-1) begintime=MPI_Wtime();

/* Initialize Major List to 1 */
i=0;
while(i<=N/P-1)
{
    Major[i++]=1;
}

for (ii=0;ii<=P-1;ii++)
{
    if (ii==num)
    {
        for (i=1;i<=N/P-1;i++)
            for (j=0;j<i;j++)
                if ((A[j]<A[i]) && (Major[i]<Major[j]+1)) Major[i]=Major[j]+1;

for (s=num+1;s<=P-1;s++)
        {
   if (verbose==1) printf("[%i] broadcast A and Major to %i\n",num,s);
   if (control==1) fprintf(fp,"[%i] broadcast A and Major to %i\n",num,s);
          MPI_Isend(A,1,vector,s,s,MPI_COMM_WORLD,&req);
          MPI_Isend(Major,1,vector,s,s+P,MPI_COMM_WORLD,&req);
}
    }
    else
    {
        if (ii<num)
```

```
         {
             MPI_Recv(AP,1,vector,ii,num,MPI_COMM_WORLD,&status);
     MPI_Recv(MajorP,1,vector,ii,num+P,MPI_COMM_WORLD,&status);
             if (verbose==1) printf("[%i] receive AP=%i and Major from %i\n",num,AP[0],ii);
             if (control==1) fprintf(fp,"[%i] receive AP=%i and Major from %i\n",num,AP[0],ii);

          for (i=0;i<=N/P-1;i++)
                for (j=0;j<=N/P-1;j++)
                   if ((AP[j]<A[i]) && (Major[i]<MajorP[j]+1)) Major[i]=MajorP[j]+1;
         }
    }
 }

 Max=0;
 for (i=0;i<=N/P-1;i++)
     if (Major[i] >= Max) Max=Major[i];

 for (s=0;s<=P-1;s++)
 {
MPI_Isend(&Max,1,MPI_LONG,s,s,MPI_COMM_WORLD,&req);
     if (verbose==1) printf("[%i] broadcast Max=%i to %i\n",num,Max,s);
     if (control==1) fprintf(fp,"[%i] broadcast Max=%i to %i\n",num,Max,s);
 }

 for (i=0;i<=P-1;i++)
 {
     MPI_Recv(&Maxp,1,MPI_LONG,i,num,MPI_COMM_WORLD,&status);
     if (verbose==1) printf("[%i] receive Maxp=%i from %i\n",num,Maxp,i);
     if (control==1) fprintf(fp,"[%i] receive Maxp=%i from %i\n",num,Maxp,i);
     Max_proc[i]=Maxp;
 }

 Maxp=Max_proc[0];

 for (i=1;i<=P-1;i++)
     if (Maxp<Max_proc[i]) Maxp=Max_proc[i];

 if (verbose==1)
 {
     printf("Recap :");
     for (i=0;i<=N/P-1;i++)
     {
         printf("%i:%i-%i\n",num,A[i],Major[i]);
     }
 }

 if (control==1)
 {
     fprintf(fp,"Recap :");
     for (i=0;i<=N/P-1;i++)
     {
         fprintf(fp,"%i:%i-%i\n",num,A[i],Major[i]);
     }
 }


 if (num==P-1)   /* on the last processor */
 {
     j=0;
     i=N/P-1;
     while (i>=0)
     {
          if (Major[i]==Maxp)
          {
```

```
                RLIS[j]=A[i];
                if (verbose==1) printf("RLIS-->%i",RLIS[j]);
                if (control==1) fprintf(fp,"RLIS-->%i",RLIS[j]);
                Maxp=Maxp-1;
                j++;
             }
             i=i-1;
        }
        if (verbose==1) printf("[%i] send Maxp=%i to %i\n",num,Maxp,num-1);
        if (control==1) fprintf(fp,"[%i] send Maxp=%i to %i\n",num,Maxp,num-1);
        MPI_Isend(&Maxp,1,MPI_LONG,num-1,0,MPI_COMM_WORLD,&req);
    }
    else
    {
        MPI_Recv(&Maxp,1,MPI_LONG,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
        if (verbose==1) printf("[%i] receive Maxp=%i from %i\n",num,Maxp,MPI_ANY_SOURCE);
        if (control==1) fprintf(fp,"[%i] receive Maxp=%i from %i\n",num,Maxp,MPI_ANY_SOURCE);
        j=0;
        i=N/P-1;
        while (i>=0)
        {
            if (Major[i]==Maxp)
            {
                RLIS[j]=A[i];
                if (verbose==1) printf("RLIS-->%i",RLIS[j]);
                if (control==1) fprintf(fp,"RLIS-->%i",RLIS[j]);
                Maxp=Maxp-1;
                j++;
            }
            i=i-1;
        }
        if (verbose==1) printf("[%i] send Maxp=%i to %i\n",num,Maxp,num-1);
        if (control==1) fprintf(fp,"[%i] send Maxp=%i to %i\n",num,Maxp,num-1);
        if (num!=0) MPI_Isend(&Maxp,1,MPI_LONG,num-1,0,MPI_COMM_WORLD,&req);
    }

    if (control==1) fclose(fp);

    if(num==P-1)
    {
        endtime=MPI_Wtime();  /* Stop counter */
        filename2[0]='\0';
        strcat(filename2,"LISP");
        gcvt((double)size,5,convertstr);
        strcat(filename2,convertstr);
        strcat(filename2,"N");
        strcat(filename2,argv[1]);
        fp=fopen(filename2,"a+");
        fprintf(fp,"P=%d => %f\n",P,endtime-begintime);
        fclose(fp);
        printf("P=%d => %f\n",P,endtime-begintime);
    }
    MPI_Type_free(&vector);
    MPI_Finalize();
}
```

Annexe B

Programme LCS_CGM

```
/*-------------------------------------------------------------*/
/* Program     : CGM Solution for the LCS problem              */
/*                                                             */
/* Authors     : Thierry Garcia, Jean-Frederic Myoupo, David Seme */
/* Programming : Thierry Garcia, David Seme                    */
/* Created     : 16/07/2001                                    */
/* Modified    : 18/11/2001                                    */
/*-------------------------------------------------------------*/

/* Parameters of the program :             */
/* N (data dim)                            */

/*-------------------------------------------------------------*/
/* Function    : Include files                                 */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 16/07/2001                                    */
/*-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

/*-------------------------------------------------------------*/
/* Function    : Define                                        */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 18/11/2001                                    */
/*-------------------------------------------------------------*/
#define LIMIT    4       /* up limit for random characters    */
#define control  0
#define verbose  0

/*-------------------------------------------------------------*/
/* Function    : Generate random number                        */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 16/10/2001                                    */
/*-------------------------------------------------------------*/
void generate_sub(long *sub, long *subp, long dim, int numproc)
{
    long i;
```

```
    srand(numproc+1);
    for (i=0;i<dim;i++)
    {
     sub[i] = (rand() % LIMIT) + 65 /* 65=A in ascii */;
        subp[i] = sub[i];
    }
}

/*------------------------------------------------------------*/
/* Function    : Main of the program                          */
/* Programming : Thierry Garcia                               */
/* Created     : 16/07/2001                                   */
/* Modified    : 18/11/2001                                   */
/*------------------------------------------------------------*/
main (int argc, char **argv)
{
    int myid;                       /* Processor id in MPI     */
    int namelen;                    /* Length of Processor name*/
    int size;                       /* Number of Processor     */
    long P;                         /* P                       */
    long N;                         /* N                       */
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Status status;              /* Variable for MPI        */
    MPI_Request req;
    MPI_Datatype vector;

    long num,nump;      /* Processor Name          */
    long *A,*B,*AP;                 /* Subsequences            */
    long *LL,*LLP,*LLPP,*LLS,*PRED; /* LCS longest             */
    long *X,*Y,*XP,*YP;
    long TEMP,AA;
    long i,ii,j,p;               /* Counters                */
    long *RLCS;                     /* Result List = LCS       */
    long Max;             /* Maximum of LL           */
    long Maxp;                      /* Maximum of all LL       */
    long *Max_proc;                 /* Maximum for a processor */
    float begintime,endtime;        /* Begin and end timer     */
    float begintimec,endtimec;      /* Begin and end timer comm*/
    char filename[14];              /* Name of file            */
    char filename2[14];             /* Name of file            */
    char convertstr[5];             /* Convert number to string*/
    FILE *fp;                       /* File pointer            */

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* How Many Processor ? */
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    P=size;          /* number of processor */

    N=atol(argv[1]); /* data length */

    if (((A=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((AP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((B=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((X=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((XP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Y=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((YP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((LL=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((LLP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((LLPP=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((LLS=(long *) malloc(N/P * sizeof(long))) == NULL)
```

```
   || ((PRED=(long *) malloc(N/P * sizeof(long))) == NULL)
   || ((Max_proc=(long *) malloc(N/P * sizeof(long))) == NULL)
   || ((RLCS=(long *) malloc(N * sizeof(long))) == NULL))
   {
       printf("Memory allocation error");
       exit(1);
   }

   /* Where am i ?     */
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   num = myid;      /* processor number    */

   /* Vector datatype for communication    */
   MPI_Type_vector(N/P,1,1,MPI_LONG,&vector);
   MPI_Type_commit(&vector);

   /* Get processor name */
   MPI_Get_processor_name(processor_name,&namelen);

   if (argc==2)
   {
   /* Generate random Ascii subsequence A */
   generate_sub(A,AP,N/P,num);

   /* Generate random Ascii subsequence B */
   generate_sub(B,B,N/P,num+P);
   }
   else
   {
      filename[0]='\0';
      strcat(filename,"DLCS");
      fp=fopen(filename,"r");
      for (i=1;i<=num*8;i++)
      {
AA=fgetc(fp);
      }
      for (i=0;i<N/P;i++)
      {
        A[i]=fgetc(fp);
        AP[i]=A[i];
        B[i]=fgetc(fp);
      }
      fclose(fp);
   }

   if (control==1)
   {
       filename[0]='\0';
       strcat(filename,"RLCSP");
       gcvt((double) size,5,convertstr);
       strcat(filename,convertstr);
       strcat(filename,"N");
       strcat(filename,argv[1]);
       strcat(filename,"p");
       gcvt((double) num,5,convertstr);
       strcat(filename,convertstr);
       fp=fopen(filename,"a+");
   }
   if (verbose==1)
   {
       printf("Execute on Proc : %d [%s]\n",num,processor_name);
       for (i=0;i<N/P;i++) printf("A[%i]=%c --AP[%i]=%c-- B[%i]=%c\n",i,A[i],i,AP[i],i,B[i]);
   }
   if (control==1)
```

```
{
    fprintf(fp,"Execute on Proc : %d [%s]\n",num,processor_name);
    for (i=0;i<N/P;i++) fprintf(fp,"A[%i]=%c -- AP[%i]=%c -- B[%i]=%c\n",i,A[i],i,AP[i],i,B[i]);
}

/* Begin counter on last processor */
if(num==P-1) begintime=MPI_Wtime();

/* Initialize LL,X,Y,LLP,XP,YP to 0 */
for (i=0;i<N/P;i++)
{
    LL[i]=0;
    X[i]=0;
    Y[i]=0;
    LLP[i]=-1;
    LLPP[i]=-1;
    XP[i]=0;
    YP[i]=0;
    LLS[i]=0;
    PRED[i]=-1;
    RLCS[i]=0;
}

for (ii=0;ii<=P-1;ii++)
{
    /* on processor num and not on the first processor */
    /* send A, XP, YP to the first processor           */
    if ((num==ii) && (ii!=0))
    {
if (control==1)
        {
          fprintf(fp,"P%i send A, XP, YP to P0\n",num);
          for (i=0;i<N/P;i++) fprintf(fp,"A[%i]=%c -- XP[%i]=%i -- YP[%i]=%i\n",i,A[i],i,XP[i],i,YP[i]);
        }

        MPI_Isend(A,1,vector,0,1,MPI_COMM_WORLD,&req);
        MPI_Isend(XP,1,vector,0,2,MPI_COMM_WORLD,&req);
        MPI_Isend(YP,1,vector,0,3,MPI_COMM_WORLD,&req);
        MPI_Isend(LLPP,1,vector,0,4,MPI_COMM_WORLD,&req);
    }
    /* not on the first processor                 */
    /* receive AP, X, Y from processor before num */
    if (num!=0)
    {
        MPI_Recv(AP,1,vector,num-1,1,MPI_COMM_WORLD,&status);
        MPI_Recv(X,1,vector,num-1,2,MPI_COMM_WORLD,&status);
        MPI_Recv(Y,1,vector,num-1,3,MPI_COMM_WORLD,&status);
        MPI_Recv(LLP,1,vector,num-1,4,MPI_COMM_WORLD,&status);
        if (control==1)
        {
         fprintf(fp,"P%i receive Ap, X, Y from %i\n",num,num-1);
         for (i=0;i<N/P;i++) fprintf(fp,"AP[%i]=%c -- X[%i]=%i -- Y[%i]=%i -- LLS[%i]=%i -- PRED[%i]=%i\n",i,AP[i],i,X[i],i,Y[i],i,LL[i],i,PRED[i]);
        }
    }

    /* local computation */
    for (i=0;i<N/P;i++)
    {
        for (j=0;j<N/P;j++)
        {
            TEMP=LL[j];
            if (Y[i]>LL[j])
            {
                if((B[j]==AP[i]) && (LL[j]<1+X[i]))
```

```
                {
                    LLS[j]=1+X[i];
                    PRED[j]=LLP[i];
                }
                LL[j]=Y[i];
                LLP[i]=num*N/P+j;
            }
            else
            {
                if ((B[j]==AP[i]) && (LL[j]<1+X[i]))
                {
                    LL[j]=1+X[i];
                    LLS[j]=LL[j];
                    PRED[j]=LLP[i];
                }
                else
                {
                    if(LLS[j]==LL[j])
                    {
                        LLP[i]=num*N/P+j;
                    }
                }
            }
            Y[i]=LL[j];
            X[i]=TEMP;
         }
    }
    if (control==1)
    {
      fprintf(fp,"Traitement local\n");
      for (i=0;i<N/P;i++) fprintf(fp,"LLS[%i]=%i -- X[%i]=%i -- Y[%i]=%i\n",i,LL[i],i,X[i],i,Y[i]);
    }

    if (num!=P-1)
    {
       if (control==1)
       {
         fprintf(fp,"P%i send AP, X, Y to %i\n",num,num+1);
         for (i=0;i<N/P;i++) fprintf(fp,"AP[%i]=%c -- X[%i]=%i -- Y[%i]=%i -- LLS[%i]=%i -- PRED[%i]=%i\n",i,AP[i],i,X[i],i,Y[i],i,LLS[i],i,PRED[i]);
       }
       MPI_Isend(AP,1,vector,num+1,1,MPI_COMM_WORLD,&req);
       MPI_Isend(X,1,vector,num+1,2,MPI_COMM_WORLD,&req);
       MPI_Isend(Y,1,vector,num+1,3,MPI_COMM_WORLD,&req);
       MPI_Isend(LLP,1,vector,num+1,4,MPI_COMM_WORLD,&req);
    }
    if ((num==0) && (ii<P-1))
    {
       MPI_Recv(AP,1,vector,ii+1,1,MPI_COMM_WORLD,&status);
       MPI_Recv(X,1,vector,ii+1,2,MPI_COMM_WORLD,&status);
       MPI_Recv(Y,1,vector,ii+1,3,MPI_COMM_WORLD,&status);
       MPI_Recv(LLP,1,vector,ii+1,4,MPI_COMM_WORLD,&status);
       if (control==1)
       {
        fprintf(fp,"P%i receive AP, X, Y from %i\n",num,ii+1);
        for (i=0;i<N/P;i++) fprintf(fp,"AP[%i]=%c -- X[%i]=%i -- Y[%i]=%i -- LLS[%i]=%i -- PRED[%i]=%i\n",i,AP[i],i,X[i],i,Y[i],i,LLS[i],i,PRED[i]);
       }
    }
}

Max=0;
for (i=0;i<N/P;i++)
    if (LLS[i] >= Max) Max=LLS[i];

Max_proc[num]=Max;
```

```
for(j=0;j<=P-1;j++)
{
    if(j==num)
    {
        MPI_Bcast(&Max,1,MPI_LONG,num,MPI_COMM_WORLD);
        if (verbose==1) printf("[%i] broadcast Max=%i to all\n",num,Max);
        if (control==1) fprintf(fp,"[%i] broadcast Max=%i to all\n",num,Max);
    }
    else
    {
        MPI_Bcast(&Maxp,1,MPI_LONG,j,MPI_COMM_WORLD);
        if (verbose==1) printf("[%i] receive Maxp=%i from %i\n",num,Maxp,j);
        if (control==1) fprintf(fp,"[%i] receive Maxp=%i from %i\n",num,Maxp,j);
        Max_proc[j]=Maxp;
    }
}

Maxp=Max_proc[0];

for (i=1;i<=P-1;i++)
{
    if (Maxp<Max_proc[i])
    {
        Maxp=Max_proc[i];
        ii=i;
    }
}

if (verbose==1) printf("Maxp=%i venant de [%i]\n",Maxp,ii);
if (control==1) fprintf(fp,"Maxp=%i venant de [%i]\n",Maxp,ii);


if(ii==num)
{
    i=0;
    while(LLS[i]!=Maxp)  i++;
    if (verbose==1) printf("MAXP=%i\n",Maxp);
    if (control==1) fprintf(fp,"MAXP=%i\n",Maxp);

    do
    {
        Maxp=PRED[i];
        if (Maxp!=-1)
        {
           RLCS[i]=B[i];
           if (control==1) fprintf(fp,"LCS[%i]=%c\n",i,RLCS[i]);
           if (verbose==1) printf("MAXP=%i\n",Maxp);
           if (control==1) fprintf(fp,"MAXP=%i\n",Maxp);
           i=PRED[i]%(N/P);
        }
    } while((num==Maxp/(N/P)) && (Maxp!=-1));
    if (num!=0)
    {
        MPI_Send(&Maxp,1,MPI_LONG,Maxp/(N/P),1,MPI_COMM_WORLD);
        MPI_Send(&num,1,MPI_LONG,Maxp/(N/P),2,MPI_COMM_WORLD);
    }

}
else
{
    if(num<ii)
    {
        MPI_Recv(&Maxp,1,MPI_LONG,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&status);
        MPI_Recv(&i,1,MPI_LONG,MPI_ANY_SOURCE,2,MPI_COMM_WORLD,&status);
```

```
          if (i!=-1)
          {
            /* Stop receive for others processors between num+1 and i *//*
            for(j=num+1;j<i;j++)
            {
              MPI_Send(&Maxp,1,MPI_LONG,j,1,MPI_COMM_WORLD);
              TEMP=-1;
              MPI_Send(&TEMP,1,MPI_LONG,j,2,MPI_COMM_WORLD);
            }

            if (Maxp!=-1) /* We have an LCS *//*
            {

               if(Maxp/(N/P)==num)
               {
                 i=Maxp%(N/P);
                 if (verbose==1) printf("MAXP=%i\n",Maxp);
                 if (control==1) fprintf(fp,"MAXP=%i\n",Maxp);

                 do
                 {
                   Maxp=PRED[i];
                   RLCS[i]=B[i];
                   if (control==1) fprintf(fp,"LCS[%i]=%c\n",i,RLCS[i]);
                   if (verbose==1) printf("MAXP=%i\n",Maxp);
                   if (control==1) fprintf(fp,"MAXP=%i\n",Maxp);
                   i=PRED[i]%(N/P);
                 } while((num==Maxp/(N/P)) && (Maxp!=-1));

                 if (num!=0)
                 {
                   MPI_Send(&Maxp,1,MPI_LONG,Maxp/(N/P),1,MPI_COMM_WORLD);
                   MPI_Send(&num,1,MPI_LONG,Maxp/(N/P),2,MPI_COMM_WORLD);
                 }
               }
             }
          }
      }
   }

   if (control==1) fclose(fp);

   if(num==P-1)
   {
       endtime=MPI_Wtime();  /* Stop counter */
       filename2[0]='\0';
       strcat(filename2,"LCSP");
       gcvt((double)size,5,convertstr);
       strcat(filename2,convertstr);
       strcat(filename2,"N");
       strcat(filename2,argv[1]);
       fp=fopen(filename2,"a+");
       fprintf(fp,"P=%d => %f\n",P,endtime-begintime);
       fclose(fp);
       printf("P=%d => %f\n",P,endtime-begintime);
   }
   MPI_Type_free(&vector);
   MPI_Finalize();
}
```

Annexe C

Programme LRSE_CGM

```
/*-------------------------------------------------------------------*/
/* Program     : CGM Solution for the LRSE problem                   */
/*                                                                   */
/* Authors     : Thierry Garcia, Jean-Frederic Myoupo, David Seme    */
/* Programming : Thierry Garcia, David Seme                          */
/* Created     : 04/06/2002                                          */
/* Modified    : 10/07/2002                                          */
/*-------------------------------------------------------------------*/

/* Parameters of the program :              */
/* N (data dim)                             */

/*-------------------------------------------------------------*/
/* Function    : Include files                                 */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 26/06/2002                                    */
/*-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>

/*-------------------------------------------------------------*/
/* Function    : Define                                        */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 18/11/2001                                    */
/*-------------------------------------------------------------*/
#define LIMIT    4       /* up limit for random characters    */
#define control  0
#define verbose  0

/*-------------------------------------------------------------*/
/* Function    : Generate random number                        */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 26/06/2002                                    */
/*-------------------------------------------------------------*/
void generate_sub(long *sub, long dim, int numproc)
{
    long i;
```

```
    srand(((unsigned) time(NULL))+numproc);
    for (i=1;i<=dim;i++)
    {
     sub[i] = (rand() % LIMIT) + 65 /* 65=A in ascii */;
    }
}

/*-------------------------------------------------------------*/
/* Function    : Main of the program                           */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 18/11/2001                                    */
/*-------------------------------------------------------------*/
main (int argc, char **argv)
{
    int myid;                       /* Processor id in MPI     */
    int namelen;                    /* Length of Processor name*/
    int size;                       /* Number of Processor     */
    long P;                         /* P                       */
    long N;                         /* N                       */
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Status status;              /* Variable for MPI        */
    MPI_Request req;
    MPI_Datatype vector;

    long num;                      /* Processor Name          */
    long *A,*AR;                    /* Subsequences            */
    long *S;                        /* LRSE longest            */
    long *C,*L;                     /* Column and line         */
    long *TC,*TL;                   /* Temp Column and line    */
    long i,ii,j,p;                /* Counters                */
    long max,maxp;                  /* Max length              */
    long *max_proc;                 /* Max on all processor    */
    long *RLRSE;                    /* Result List = LRSE      */
    float begintime,endtime;        /* Begin and end timer     */
    float begintimec,endtimec;      /* Begin and end timer comm*/
    char filename[14];              /* Name of file            */
    char filename2[14];             /* Name of file            */
    char convertstr[5];             /* Convert number to string*/
    FILE *fp;                       /* File pointer            */

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* How Many Processor ? */
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    P=size;          /* number of processor */

    N=atol(argv[1]); /* data length */

    if (((A=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((AR=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((TC=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((TL=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((S=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((L=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((C=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((max_proc=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((RLRSE=(long *) malloc(N * sizeof(long))) == NULL))
    {
        printf("Memory allocation error");
        exit(1);
```

```
}

/* Where am i ?     */
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
num = myid;      /* processor number    */

/* Vector datatype for communication    */
MPI_Type_vector(N/P+1,1,1,MPI_LONG,&vector);
MPI_Type_commit(&vector);

/* Get processor name */
MPI_Get_processor_name(processor_name,&namelen);

if (argc==2)
{
/* Generate random Ascii subsequence A */
generate_sub(A,N/P,num);

}
else
{
   filename[0]='\0';
   strcat(filename,"DLRSE");
   fp=fopen(filename,"r");
   for (i=1;i<=N/P;i++)
   {
     A[i]=fgetc(fp);
   }
   fclose(fp);
}

if (control==1)
{
    filename[0]='\0';
    strcat(filename,"RLRSEP");
    gcvt((double) size,5,convertstr);
    strcat(filename,convertstr);
    strcat(filename,"N");
    strcat(filename,argv[1]);
    strcat(filename,"p");
    gcvt((double) num,5,convertstr);
    strcat(filename,convertstr);
    fp=fopen(filename,"a+");
}
if (verbose==1)
{
    printf("Execute on Proc : %d [%s]\n",num,processor_name);
}
if (control==1)
{
    fprintf(fp,"Execute on Proc : %d [%s]\n",num,processor_name);
}

/* Begin counter on first processor */
if(num==P-1) begintime=MPI_Wtime();

/* Initialize S,L,C to 0 */
for (i=0;i<=N/P;i++)
{
    S[i]=0;
    L[i]=0;
    C[i]=0;
}
```

```
    for (ii=0;ii<=num;ii++)
    {
        if (num==ii)
        {
          if (verbose==1)
             for (i=0;i<=N/P;i++)
             {
                  if (i==0)
                           printf("A[%i]=  --A[%i]=  --S[%i]=  --L[%i]=%i -- C[%i]=\n",i,i,i,i,L[i],i);
                  else
                           printf("A[%i]=%c --A[%i]=%c --S[%i]=%i --L[%i]=%i -- C[%i]=%i\n",i,A[i],i,A[i],i,S[i],i,L[i],i,C[i]);
             }
          if (verbose==1) printf("Local Computation T pour %i\n",num);
          /* local computation */
          for (i=1;i<=N/P;i++) C[i]=0;

          for (i=2;i<=N/P;i++)
          {
            for (j=1;j<=i-1;j++)
            {
              TC[j]=0;
              if (A[i]==A[j])
              {
                if (j==1)
                         TC[j]=L[i-1]+1;
                else
                         TC[j]=C[j-1]+1;

                if (TC[j]>S[i]) S[i]=TC[j];
              }
             }
             for (j=1;j<=i-1;j++) C[j]=TC[j];
          }

          if (verbose==1)
             for (i=0;i<=N/P;i++)
             {
                  if (i==0)
                           printf("A[%i]=  --A[%i]=  --S[%i]=  --L[%i]=%i -- C[%i]=\n",i,i,i,i,L[i],i);
                  else
                           printf("A[%i]=%c --AR[%i]=%c --S[%i]=%i --L[%i]=%i -- C[%i]=%i\n",i,A[i],i,AR[i],i,S[i],i,L[i],i,C[i]);
             }

          if (verbose==1) for (i=1;i<=N/P;i++) printf("S sur processeur %i : S[%i]=%i\n",num,i,S[i]);
          if (num<P-1)
          {
             if (verbose==1) printf("%i send A and C to %i -- tag %i and %i\n",num,num+1,1+2*ii,2+2*ii);
             MPI_Isend(A,1,vector,num+1,1+2*ii,MPI_COMM_WORLD,&req);
             MPI_Isend(C,1,vector,num+1,2+2*ii,MPI_COMM_WORLD,&req);
          }
        }
else
if (ii<num)
        {
          MPI_Recv(AR,1,vector,num-1,1+2*ii,MPI_COMM_WORLD,&status);
          MPI_Recv(C,1,vector,num-1,2+2*ii,MPI_COMM_WORLD,&status);
          if (verbose==1) printf("%i receive AR and C from %i -- tag %i and %i\n",num,num-1,1+2*ii,2+2*ii);
          if (verbose==1)
             for (i=0;i<=N/P;i++)
             {
                  if (i==0)
                           printf("A[%i]=  --A[%i]=  --S[%i]=  --L[%i]=%i -- C[%i]=\n",i,i,i,i,L[i],i);
                  else
                           printf("A[%i]=%c --AR[%i]=%c --S[%i]=%i --L[%i]=%i -- C[%i]=%i\n",i,A[i],i,AR[i],i,S[i],i,L[i],i,C[i]);
```

```
        }

      if (verbose==1) printf("Local computation S on %i\n",num);
      /* local computation */

      TL=L;
      L[0]=C[N/P];

      for (i=1;i<=N/P;i++)
      {
        for (j=1;j<=N/P;j++)
        {
          TC[j]=0;
          if (A[i]==AR[j])
          {
            if (j==1)
                   TC[j]=TL[i-1]+1;
            else
                   TC[j]=C[j-1]+1;

            if (TC[j]>S[i]) S[i]=TC[j];
          }
        }
        L[i]=TC[N/P];
        for (j=0;j<=N/P;j++) C[j]=TC[j];
      }
      if (verbose==1)
         for (i=0;i<=N/P;i++)
         {
              if (i==0)
                       printf("A[%i]=  --A[%i]=  --S[%i]=  --L[%i]=%i -- C[%i]=\n",i,i,i,i,L[i],i);
              else
                       printf("A[%i]=%c --AR[%i]=%c --S[%i]=%i --L[%i]=%i -- C[%i]=%i\n",i,A[i],i,AR[i],i,S[i],i,L[i],i,C[i]);
         }


      if (num<P-1)
      {
        if (verbose==1) printf("%i send AR and C to %i -- tag %i and %i\n",num,num+1,1+2*ii,2+2*ii);
        MPI_Isend(AR,1,vector,num+1,1+2*ii,MPI_COMM_WORLD,&req);
        MPI_Isend(C,1,vector,num+1,2+2*ii,MPI_COMM_WORLD,&req);
      }
  }
}

/*max=0;
for (i=1;i<=N/P;i++)
   if (S[i]>=max) max=S[i];
max_proc[num]=max;

if (verbose==1) printf("%i send max=%i to all others\n",num,max);
for (j=0;j<=P-1;j++) if (j!=num) MPI_Send(&max,1,MPI_LONG,j,0,MPI_COMM_WORLD);

for (j=0;j<=P-1;j++)
{
   if (j!=num)
   {
      MPI_Recv(&maxp,1,MPI_LONG,j,0,MPI_COMM_WORLD,&status);
      if (verbose==1) printf("%i receive maxp=%i from %i\n",num,maxp,j);
      max_proc[j]=maxp;
   }
}

maxp=max_proc[0];
```

```
    for (i=1;i<=P-1;i++)
    {
      if (maxp<max_proc[i])
      {
        maxp=max_proc[i];
        ii=i;
      }
    }

    if (verbose==1) printf("maxp=%i sur %i\n",maxp,ii);

    if (ii==num)
    {
      i=1;
      while (S[i]!=maxp) i++;
      do
      {
        RLRSE[maxp]=A[i];
        if (verbose==1) printf("RLRSE[%i]=%c\n",maxp,RLRSE[maxp]);
        maxp=maxp-1;
        i=i-1;
      } while ((maxp!=0) && (i!=0));
      if (num!=0) MPI_Send(&maxp,1,MPI_LONG,num-1,0,MPI_COMM_WORLD);
    }
    else
    {
      if (num<ii)
      {
        MPI_Recv(&maxp,1,MPI_LONG,num+1,0,MPI_COMM_WORLD,&status);
        if (maxp!=0)
        {
          i=N/P;
          do
          {
            RLRSE[maxp]=A[i];
            if (verbose==1) printf("RLRSE[%i]=%c\n",maxp,RLRSE[maxp]);
            maxp=maxp-1;
            i=i-1;
          } while ((maxp!=0) && (i!=0));
        }
        if (num!=0) MPI_Send(&maxp,1,MPI_LONG,num-1,0,MPI_COMM_WORLD);
      }
    }*/

    if(control==1) fclose(fp);

    if(num==P-1)
    {
        endtime=MPI_Wtime();  /* Stop counter */
        filename2[0]='\0';
        strcat(filename2,"LRSEP");
        gcvt((double)size,5,convertstr);
        strcat(filename2,convertstr);
        strcat(filename2,"N");
        strcat(filename2,argv[1]);
        fp=fopen(filename2,"a+");
        fprintf(fp,"P=%d => %f\n",P,endtime-begintime);
        fclose(fp);
        printf("P=%d => %f\n",P,endtime-begintime);
    }
    MPI_Type_free(&vector);
    MPI_Finalize();
}
```

Annexe D

Programme DR_CGM

```
/*-------------------------------------------------------------------*/
/* Program     : CGM Solution for the repetition problem             */
/*                                                                   */
/* Authors     : Thierry Garcia, David Seme                          */
/* Programming : Thierry Garcia, David Seme                          */
/* Created     : 05/03/2003                                          */
/* Modified    : 04/04/2003                                          */
/*-------------------------------------------------------------------*/

/* Parameters of the program :              */
/* N (data dim)                             */

/*-------------------------------------------------------------------*/
/* Function    : Include files                                       */
/* Programming : Thierry Garcia                                      */
/* Created     : 16/07/2001                                          */
/* Modified    : 26/06/2002                                          */
/*-------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <mpi.h>

/*-------------------------------------------------------------------*/
/* Function    : Define                                              */
/* Programming : Thierry Garcia                                      */
/* Created     : 16/07/2001                                          */
/* Modified    : 04/04/2003                                          */
/*-------------------------------------------------------------------*/
#define LIMIT    4       /* up limit for random characters    */
#define control  0
#define verbose  0
#define result   0

/*-------------------------------------------------------------------*/
/* Function    : Generate random number                              */
/* Programming : Thierry Garcia                                      */
/* Created     : 16/07/2001                                          */
/* Modified    : 05/03/2003                                          */
/*-------------------------------------------------------------------*/
void generate_sub(long *sub,long *sub2, long dim, int numproc)
{
```

```
    long i;

    srand(((unsigned) time(NULL))+numproc);
    for (i=1;i<=dim;i++)
    {
     sub[i] = (rand() % LIMIT) + 65 /* 65=A in ascii */;
sub2[i] = sub[i];
    }
}

/*-------------------------------------------------------------*/
/* Function    : Main of the program                           */
/* Programming : Thierry Garcia                                */
/* Created     : 16/07/2001                                    */
/* Modified    : 04/04/2003                                    */
/*-------------------------------------------------------------*/
main (int argc, char **argv)
{
    int myid;                       /* Processor id in MPI     */
    int namelen;                    /* Length of Processor name*/
    int size;                       /* Number of Processor     */
    long P;                         /* P                       */
    long N;                         /* N                       */
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Status status;              /* Variable for MPI        */
    MPI_Request req;
    MPI_Datatype vector,vectorN;

    long num;                      /* Processor Name          */
    long *X,*Y;                     /* Subsequences            */
    long *Ic,*Lc;                   /* Icounter and Lcounter   */
    long i,ii,p;              /* Counters                */
    char XX;
    float begintime,endtime;        /* Begin and end timer     */
    float begintimec,endtimec;      /* Begin and end timer comm*/
    char filename[14];              /* Name of file            */
    char filename2[14];             /* Name of file            */
    char convertstr[5];             /* Convert number to string*/
    FILE *fp,*fpr;                  /* File pointer            */

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* How Many Processor ? */
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    P=size;          /* number of processor */

    N=atol(argv[1]); /* data length */

    if (((X=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Y=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Ic=(long *) malloc(N/P * sizeof(long))) == NULL)
    || ((Lc=(long *) malloc(N/P * sizeof(long))) == NULL))
    {
        printf("Memory allocation error");
        exit(1);
    }

    /* Where am i ?     */
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    num = myid;      /* processor number    */

    /* Vector datatype for communication    */
```

```
    MPI_Type_vector(N/P+1,1,1,MPI_LONG,&vector);
    MPI_Type_commit(&vector);

    /* Get processor name */
    MPI_Get_processor_name(processor_name,&namelen);

    if (argc==2)
    {
    /* Generate random Ascii subsequence A */
    generate_sub(X,Y,N/P,num);

    }
    else
    {
       filename[0]='\0';
       strcat(filename,"DREPET");
       fp=fopen(filename,"r");
       for (i=1;i<=num*N/P;i++)
       {
 XX=fgetc(fp);
       }
       for (i=1;i<=N/P;i++)
       {
         X[i]=fgetc(fp);
 Y[i]=X[i];
       }
       fclose(fp);
    }

    if (control==1)
    {
        filename[0]='\0';
        strcat(filename,"RREPETP");
        gcvt((double) size,5,convertstr);
        strcat(filename,convertstr);
        strcat(filename,"N");
        strcat(filename,argv[1]);
        strcat(filename,"p");
        gcvt((double) num,5,convertstr);
        strcat(filename,convertstr);
        fp=fopen(filename,"a+");
    }
    if (verbose==1)
    {
printf("Execute on Proc : %d [%s]\n",num,processor_name);
for (i=1;i<=N/P;i++) printf("X[%i]=%c ",i,X[i]);
printf("\n");
    }

    if (control==1)
    {
fprintf(fp,"Execute on Proc : %d [%s]\n",num,processor_name);
for (i=1;i<=N/P;i++) fprintf(fp,"X[%i]=%c ",i,X[i]);
fprintf(fp,"\n");
    }

    if (result==1)
    {
       filename[0]='\0';
       strcat(filename,"LREPETP");
       gcvt((double) size,5,convertstr);
       strcat(filename,convertstr);
       strcat(filename,"N");
       strcat(filename,argv[1]);
```

```
       strcat(filename,"p");
       gcvt((double) num,5,convertstr);
       strcat(filename,convertstr);
       fpr=fopen(filename,"w+");
    }

    /* Begin counter on last processor */
    if(num==P-1) begintime=MPI_Wtime();

    for (ii=0;ii<=(P-1)-num;ii++)
    {
        if ((ii==0) && (num!=0))
{
   if (verbose==1) printf("Send X to 0\n");
   if (control==1) fprintf(fp,"Send X to 0\n");
   MPI_Isend(X,1,vector,0,num,MPI_COMM_WORLD,&req);
}

if ((ii!=0) && (num==0))
{
  if (verbose==1) printf("Receive Y from %i\n",ii);
  if (control==1) fprintf(fp,"Receive Y from %i\n",ii);
  MPI_Recv(Y,1,vector,ii,ii,MPI_COMM_WORLD,&status);
}

if (num!=0)
{
  MPI_Recv(Y,1,vector,num-1,num-1,MPI_COMM_WORLD,&status);
  if (verbose==1)
         {
      printf("Receive Y from %i\n",num-1);
      for (i=1;i<=N/P;i++) printf("Y[%i]=%c\n",i,Y[i]);
  }
  if (control==1) fprintf(fp,"Receive Y from %i\n",num-1);
}

if (ii!=0)
{
           if (verbose==1) printf("Local computation reverse on %i\n",num);
   if (control==1) fprintf(fp,"Local computation reverse on %i\n",num);

   if (ii*N/P<=N/2)
          {
     for (p=(ii-1)*N/P+1;p<=ii*N/P;p++)
     {
               for (i=ii*N/P-p+1;i<=N/P;i++)
               {
                   if (X[i]!=Y[i+p-(ii*N/P)])
                   {
                     Ic[p-(ii-1)*N/P]=Ic[p-(ii-1)*N/P]+Lc[p-(ii-1)*N/P]-p+1;
                     Lc[p-(ii-1)*N/P]=p;
                   }
                   else
                   {
                     Lc[p-(ii-1)*N/P]=Lc[p-(ii-1)*N/P]+1;
                     if (Lc[p-(ii-1)*N/P]>=2*p)
     {
 if (verbose==1) printf("R(%i,%i,%i)\n",Ic[p-(ii-1)*N/P],p,Lc[p-(ii-1)*N/P]);
 if (result==1) fprintf(fpr,"R(%i,%i,%i)\n",Ic[p-(ii-1)*N/P],p,Lc[p-(ii-1)*N/P]);
     }
                   }
 }
       }
     }
```

```
    if (verbose==1)
            {
printf("Send Y,Lc,Ic to %i\n",num+1);
for (i=1;i<=N/P;i++) printf("Y[%i]=%c - Lc[%i]=%i - Ic[%i]=%i\n",i,Y[i],i-1,Lc[i-1],i-1,Ic[i-1]);
    }
    if (control==1) fprintf(fp,"Send Y,Lc,Ic to %i\n",num+1);
    MPI_Isend(Y,1,vector,num+1,num,MPI_COMM_WORLD,&req);
    MPI_Isend(Lc,1,vector,num+1,num+P,MPI_COMM_WORLD,&req);
    MPI_Isend(Ic,1,vector,num+1,num+P+1,MPI_COMM_WORLD,&req);
        }

if (num!=0)
{
   if (control==1) fprintf(fp,"Receive Lc,Ic from %i\n",num-1);
   MPI_Recv(Lc,1,vector,num-1,num-1+P,MPI_COMM_WORLD,&status);
   MPI_Recv(Ic,1,vector,num-1,num-1+P+1,MPI_COMM_WORLD,&status);
   if (verbose==1)
   {
        printf("Receive Lc,Ic from %i\n",num-1);
      for (i=1;i<=N/P;i++) printf("Lc[%i]=%i - Ic[%i]=%i\n",i-1,Lc[i-1],i-1,Ic[i-1]);
   }
         }

 if (verbose==1) printf("Local computation on %i\n",num);
 if (control==1) fprintf(fp,"Local computation on %i\n",num);

 if ((ii+1)*N/P-1<=N/2)
         {
    for (p=ii*N/P+1;p<=(ii+1)*N/P-1;p++)
    {
       if (num==0)
       {
Lc[p-ii*N/P]=p;
Ic[p-ii*N/P]=1;
       }
       for (i=1;i<=(1+ii)*N/P-p;i++)
       {
  if (X[i]!=Y[i+p-(ii*N/P)])
  {
Ic[p-ii*N/P]=Ic[p-ii*N/P]+Lc[p-ii*N/P]-p+1;
Lc[p-ii*N/P]=p;
  }
  else
  {
Lc[p-ii*N/P]=Lc[p-ii*N/P]+1;
if (Lc[p-ii*N/P]>=2*p)
{
            if (verbose==1) printf("R(%i,%i,%i)\n",Ic[p-ii*N/P],p,Lc[p-ii*N/P]);
    if (result==1) fprintf(fpr,"R(%i,%i,%i)\n",Ic[p-ii*N/P],p,Lc[p-ii*N/P]);
}
  }
       }
    }
  }
    }

    if (control==1) fclose(fp);

    if (result==1) fclose(fpr);

    if(num==P-1)
    {
        endtime=MPI_Wtime();  /* Stop counter */
```

```
        filename2[0]='\0';
        strcat(filename2,"REPETP");
        gcvt((double)size,5,convertstr);
        strcat(filename2,convertstr);
        strcat(filename2,"N");
        strcat(filename2,argv[1]);
        fp=fopen(filename2,"a+");
        fprintf(fp,"P=%d => %f\n",P,endtime-begintime);
        fclose(fp);
        printf("P=%d => %f\n",P,endtime-begintime);
    }
    MPI_Type_free(&vector);
    MPI_Finalize();
}
```

Bibliographie

[A. 76] A. Aho and D. Hirschberg and J.D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem. *J. ACM*, 23(1) :1–12, 1976.

[A. 95] A. Alexandrov and M.F. Ionescu and K.E. Schauser and C. Scheiman. LogGP : Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. *7-th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA'95)*, pages 95–105, 1995.

[AALM90] A. Apostolico, M. Atallah, L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5) :968–988, 1990.

[ABG92] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92 :3–17, 1992.

[ABOR90] R. Airiau, J.M. Bergé, V. Olive, and J. Rouillard. *VHDL : du Langage à la Modélisation.* Presses Polytechniques et Universitaires Romandes, 1990.

[ACDS02] C.E.R. Alves, E.N. Cáceres, F. Dehne, and S.W. Song. Parallel Dynamic Programming For Solving The String Editing Problem On A CGM/BSP. *Fourteenth ACM Symposium on Parallel Algorithms and Architectures - SPAA 2002*, pages 275–281, 2002.

[ACDS03] C.E.R. Alves, E.N. Cáceres, F. Dehne, and S.W. Song. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. *International Conference of Computational Science and Its Applications (ICCSA'03)*, 2 :249–258, 2003.

[ACS03] C.E.R. Alves, E.N. Cáceres, and S.W. Song. A BSP/CGM Algorithm for the All-Substrings Longest Common Subsequence Problem. *17th IEEE Annual*

International Parallel and Distributed Processing Symposium (IPDPS 2003). Nice, France, page 57, 2003.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F.P. Preparata. Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones. *SIAM Journal on Computing*, 16(5) :808–835, 1987.

[Aho90] A.V. Aho. *Algorithms for Finding Patterns in Strings*, chapter 5, pages 255–300. Handbook of Theoretical Computer Science. J. Van Leeuwen Editor, Elsevier, 1990.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[Akl85] S.G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Akl90] S.G. Akl. Parallel Synergy or Can a Parallel Computer Be More Efficient Than the Sum of Its Parts ? Technical report, 90-285, Queen's University, Kingston Ontario, 1990.

[AN82] A. Apostolico and A. Negro. A Structure for the Statistics of All Substrings of a Textstring With or Without Overlap. In *World Conf. on Math. at Science of Man*, 1982.

[AN84] A. Apostolico and A. Negro. Systolic Algorithm for String Manipulations. *IEEE Transactions on computer*, c-33(4) :361–364, 1984.

[AP83] A. Apostolico and F.P. Preparata. Optimal Off-line Detection of Repetitions in a String. *Theoretical Computer Science*, 22 :297–315, 1983.

[BCDL99] P. Bose, A. Chan, F. Dehne, and M. Latzel. Coarse grained parallel maximum matching in convex bipartite graph. *Proc. 13th International Parallel Processing Symposium (IPPS'99)*, pages 125–129, 1999.

[Ber79] J. Berstel. *Transductions and Context-free Languages*. Teubner, Stuttgart, 1979.

[Ber81] J. Berstel. Mots de Fibonacci. *Séminaire d' Informatique Théorique, LITP, Paris*, pages 57–78, 1981.

[Ber92] J. Berstel. Axel Thue's Work on Repetitions in Words. In *Conf. on Formal Power Series and Algebric Combinatorics*, 1992.

[BHP+96] G. Bilardi, K. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. *8-th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 25–32, 1996.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10) :762–772, 1977.

[Bra80] C.H. Braunholtz. Solution to Problem 5030. *Ann. of Math.*, 70 :558–567, 1980.

[BS93] J. Berstel and P. Séébold. A Characterization of Overlap-free Morphisms. *Discrete Applied Mathematics*, 46 :275–281, 1993.

[CD99] A. Chan and F. Dehne. A note on coarse grained parallel integer sorting. *Parallel Processing Letters*, 9(4) :533–538, 1999.

[CDF+02] E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. Song. Efficient Parallel Graph Algorithms For Coarse Grained Multicomputers and BSP. *Algorithmica*, 33 :183–200, 2002.

[CDM92] C. Cérin, C. Dufourd, and J.F. Myoupo. A Linear systolic Architecture for the Longest Increasing Subsequence Problem. In *ICCS'92*, 1992.

[CDM93] C. Cérin, C. Dufourd, and J.F. Myoupo. An Efficient Parallel Solution for the Longest Increasing Subsequence Problem. In *Fifth International Conference on Computing and Information (ICCI'93)Sudbury, Ontario, IEEE Press*, pages 200–224, 1993.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. Logp : Towards a realistic model of computation. *4-th ACM SIGPLAN Symp. on Principles and Practises of Parallel Programming*, pages 1–12, 1993.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, chapter 16, pages 314–320. MIT Press, 1990.

[CMS99a] J. Carle, J.F. Myoupo, and D. Semé. All-to-all broadcasting algorithms on honeycomb networks and applications. *Parallel Processing Letters*, 9 :539–550, 1999.

[CMS99b] J. Carle, J.F. Myoupo, and D. Semé. Star-honeycomb meshes and tori : Topological properties, communication algorithms, and ring embedding. *OPODIS'99*, 1999.

[Cro81] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5) :244–250, 1981.

[Cro83] M. Crochemore. Recherche linéaire d' un carré dans un mot. *C. R. Acad. Sci. (série I)*, 296 :781–784, 1983.

[Cro84] M. Crochemore. Optimal Factor Transducers. In *NATO Advanced Research Workshop On Combi. Algo. on Words*, 1984.

[Cro86] M. Crochemore. Transducers and Repetitions. *Theoretical Computer Science*, 45 :63–86, 1986.

[CRU02] O. Cozette, C. Randriamaro, and G. Utard. Improving cluster io performance with remote efficient access to distant devices. In *HSLN'02, Workshop on High Speed Local Network, LCN'02*, pages 629–638, 2002.

[CRU03] O. Cozette, C. Randriamaro, and G. Utard. READ2 : Put disks at network level. In *Workshop on Parallel IO, CCGRID'03*, pages 698–704, 2003.

[Dan94] V. Dančík. *Expected Length of Longest Common Subsequences.* PhD thesis, Univ. Warwick, 1994.

[DDD+95] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Khokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 27–33, 1995.

[DFRC93] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputer. *9-th ACM Symp. on Computational Geometry*, 1993.

[DFRC96] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3) :379–400, 1996.

[DFRCU99] M. Diallo, A. Ferreira, A. Rau-Chaplin, and S. Ubeda. Scalable 2d convex hull and triangulation algorithms for coarse grained multicomputers. *Journal of Parallel and Distributed Computing*, 56(1) :47–70, 1999.

[DH84] M. Du and W. Hsu. New Algorithms for the Longest Common Subsequence Problem. *Journal of Computer and System Sciences*, 29 :133–152, 1984.

[DM96] J.-L. Dekeyser and P. Marquet. *The Data Parallel Programming Model*. chapter Supporting irregular and dynamic computations in data-parallel languages. Lecture Notes on computer Science Tutorial Series (LNCS-TS), Springer Verlag, 1996.

[DP94] V. Dančík and M. S. Paterson. Longest Common Subsequences. In *Proc.* 11^{th} *Annual Symp. Theor. Aspects Comp. Sci.*, LNCS 775, pages 127–142, 1994.

[Dun90] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Comput.*, pages 5–16, 1990.

[Duv78] J.P. Duval. *Sur la périodicité des mots*. PhD thesis, Faculté des Sciences, Université de Rouen, France, 1978.

[Eis85] M.J. Eisler. A Research Report on the Design of an Approximate String Matching Device using the LCS Device Approach. Master's thesis, Univ. Central Florida, 1985.

[Epp89] D. Eppstein. *Efficient Algorithms for Sequence Analysis with Concave and Convexe Gap Costs*. PhD thesis, Columbia Univ., 1989.

[Fer96] A. Ferreira. *Parallel and communication algorithms for hypercube multiprocessors*. Chapter : Handbook of parallel and Distributed Computing. (A. Zomaya, ed.), McGraw-Hill, 1996.

[FGL98] A. Ferreira and I. Guerin-Lassous. *Algorithmique discrète parallèle : le chaînon manquant*. Parallélisme et Répartition - Ed. Hermes, 1998.

[FJSD95] L.D. Fosdick, E.R. Jessup, C.J. Schauble, and G. Domik. *An Introduction to High-Performance Scientific Computing*. MIT Press, 1995.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. on Comp.*, C-21 :948–960, 1972.

[Fra95] I. Fraser. *Subsequences and Supersequences of Strings*. PhD thesis, Univ. Glasgow, 1995.

[Fre75] M.L. Fredman. On Computing the Length of Longest Increasing Subsequences. *Discrete Mathematics*, pages 29–35, 1975.

[Fru92] M.A. Frumkin. *Systolic Computations*. Kluwer Academic Publishers, 1992.

[FS99] A. Ferreira and N. Schabanel. A randomized bsp/cgm algorithm for the maximal independant set problem. *Parallel Processing Letters*, 9(3) :411–422, 1999.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *10-th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GGLGT00] Assefaw Hadish Gebremedhin, Isabelle Guérin-Lassous, Jens Gustedt, and Jan Arne Telle. Exploration Rapide et Exhaustive des Banques d'ADNGraph Coloring on a Coarse Grained Multiprocessor. Technical Report N3906, "INRIA", 2000.

[GKLST96] M. Goudreau, S. Rao K. Lang, T. Suel, and T. Tsantilas. Towards efficiency and portability : Programming with the bsp model. *8th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12, 1996.

[GL99] I. Guérin-Lassous. *Algorithmiques parallèles de traitement de graphes : une approche basée sur l'analyse expérimentale*. PhD thesis, Université Denis Diderot, Paris 7, France, 1999.

[GMS01] T. Garcia, J.F. Myoupo, and D. Semé. A work-optimal cgm algorithm for the longest increasing subsequence problem. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*, 2 :563–569, 2001.

[GMS03] T. Garcia, J.F. Myoupo, and D. Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. *11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP'03)*, pages 349–356, 2003.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, England, 1988.

[Gra88] A. Gram. *Raisonner pour programmer*. Dunod, Paris, 1988.

[Gri89] D. Gries. *The Science of Programming*. Springler Verlag, 1981 (fifth printing 1989).

[GSa] T. Garcia and D. Semé. A coarse-grained multicomputer algorithm for the detection of repetitions. *submitted*.

[GSb] T. Garcia and D. Semé. A load balancing technique for some coarse-grained multicomputer algorithms. *submitted*.

[GS03] T. Garcia and D. Semé. A coarse-grained multicomputer algorithm for the longest repeated suffix ending at each point in a word. *International Conference on Computational Science and its Applications (ICCSA'03)*, 2 :239–248, 2003.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology.* Cambridge University Press, 1997.

[Har78] M.A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.

[Hed67] G.A. Hedlund. Remarks on the work of Axel Thue on sequences. *Nord. Mat. Tidskr.*, 15 :148–150, 1967.

[Hir77] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4) :664–675, 1977.

[HL97] C. Hancart and T. Lecroq. Stringology. *http ://www-igm.univ-mlv.fr/l̃ecroq/lec_en.html*, 1997.

[HS77] J.W. Hunt and T.G. Szymanski. A Fast Algorithm for Computing Longest Common Subsequences. *CACM*, 20 :350–353, 1977.

[Hwa93] K. Hwang. *Advanced Computer Architecture : Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[JV92] G. Jacobson and K.P. Vo. Heaviest Increasing/Common Subsequence Problems. In *Proc. 3^{rd} Annual Symp. CPM*, LNCS 644, pages 52–66, 1992.

[KAP95] K.K. Keeton, T.E. Anderson, and D.A. Patterson. LogP Quantified : The Case for Low-overhead Local Area Networks. *Hot Interconnects III : A Symposium on High Performance Interconnects*, 1995.

[Kec91] J. Kececioglu. *Exact and Approximate Algorithms for DNA Sequence Reconstruction.* PhD thesis, Univ. Arizona, 1991.

[KK00] R. Kolpakov and G. Kucherov. On maximal repetitions in words. *J. Discret. Algorithms*, 1(1) :159–186, 2000.

[KL78] H.T. Kung and C.E. Leiserson. Algorithms for VLSI Processor Arrays. In *Proc. SIAM Sparse Matrix Proc.*, pages 256–282, 1978.

[KLJH87] S.Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang. Wavefront Array Processors : From Concept to Implementation. *IEEE Comp.*, pages 18–33, 1987.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6 :323–350, 1977.

[KNY82] Y. Kayambayashi, N. Nakatsu, and S. Yajima. A Longest Algorithm Suitable for Similar Text String. *Acta Informatica*, 18 :171–179, 1982.

[KP00] S.R. Kim and K. Park. Fully scalable fault-tolerant simulations for bsp and cgm. *Journal of Parallel and Distributed Computing*, 60 :1531–1560, 2000.

[KR90] R. Karp and V. Ramachandran. *Parallel algorithms for shared-memory machines*, volume A. Chapter : Handbook of Theorical Computer Science, Algorithms and Complexity, Elsevier, 1990.

[Kun80a] H.T. Kung. The Structure of Parallel Algorithms. *Advances in Comp.*, 19 :65–112, 1980.

[Kun80b] H.T. Kung. Why Systolic Architectures. *IEEE Comput.*, 15(1) :37–46, 1980.

[Kun84] S.Y. Kung. Wavefront Array Processor : Langage, Architecture and Applications. *IEEE Trans. Comp.*, C-31 :1054–1066, 1984.

[Lav89] D. Lavenier. *MicMacs : un Réseau Systolique pour le Traitement de Chaînes de Caractères*. PhD thesis, Univ. Rennes 1, 1989.

[Lav93] D. Lavenier. An Integrated 2D Systolic Array for Spelling Correction. *Integration : The VLSI J.*, 15, 1993.

[Lee93] J.V. Leeuwen. *Handbook of Theoretical Computer Science*. Elsevier, 1993.

[Lei91] F.T. Leighton. *Parallel Algorithms and Architectures : Arrays - Trees- Hypercube*. Morgan Kaufmann, 1991.

[LG99] Isabelle Guérin Lassous and Jens Gustedt. List Ranking on a Coarse Grained Multiprocessor. Technical Report N3640, "INRIA", 1999.

[Lin94] Y.C. Lin. New Systolic Arrays for the Longest Common Subsequence Problem. *Parall. Comp.*, 20 :1323–1334, 1994.

[LL82] F.T. Leighton and C.E. Leiserson. Wafer-Scale Integration of Systolic Arrays. In *Proc.* 23^{rd} *Symp. Fond. Comp. Sci.*, pages 297–311, 1982.

[LL94] M. Lu and H. Lin. Parallel Algorithms for the Longest Common Subsequence Problem. *IEEE Transactions on Computers*, 5(8) :835–848, 1994.

[LL00] A. Lefebvre and T. Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australia, 2000.

[LLM97] T. Lecroq, G. Luce, and J.F. Myoupo. A Faster Linear Systolic Algorithm for Recovering a Longest Common Subsequence. *Information Processing Letters*, 61(3) :129–136, 1997.

[LM97] G. Luce and J.F. Myoupo. Efficient Parallel Computing of a Longest Common Subsequence of Three Strings. In *8th SIAM Conf. Parall. Proc. Sci. Comp.*, 1997.

[LMS02] T. Lecroq, J.-F. Myoupo, and D. Semé. Exact computations of the longest repeated suffix ending at each point in a word. In R. Gantenbein and S. Shin, editors, *Proceedings of the 17th International Conference on Computers and Their Applications*, pages 18–21, 2002.

[Lop87] D.P. Lopresti. *Discounts for Dynamic Programming with Applications in VLSI Processor Arrays.* PhD thesis, Princeton Univ., 1987.

[Lot97] M. Lothaire. *Combinatorics on Words.* Cambridge University Press, 2nd edition, 1997.

[LT00] Isabelle Guérin Lassous and Eric Thierry. Generating Random Permutations in the Framework of Parallel Coarse Grained Models. Technical Report N3896, "INRIA", 2000.

[Luc97] G. Luce. *Algorithmique systolique pour le problème du plus long sous-mot commun et implémentation sur un réseau de Transputeurs.* PhD thesis, Faculté de Mathématiques et d' Informatique, Université de Picardie Jules Verne, Amiens, France, 1997.

[LZE97] W. Löwe, W. Zimmermann, and J. Eisenbiegler. On Linear Schedules of Tasks Graphs on Generalized LogP-Machines. *Europar'97 (LNCS, ed.)*, pages 895–904, 1997.

[Man89] U. Manber. *Introduction to Algorithms, a creative approach.* Adisson-Wesley, 1989.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Weysley, 1980.

[Mis78] J. Misra. A Technique of Algorithm Construction on Sequence. *IEEE Trans. Software Engineering*, SE-4(1) :65–69, 1978.

[ML79] M. Main and R. Lorentz. An O(n log n) Algorithm for Finding Repetition in a String. Technical report, CS-79-056, Computer Science Departement, Washington State University, 1979.

[ML84] M. Main and R. Lorentz. Linear Time Recognition of Square-free Strings. In *NATO Advanced Research Workshop on Combi. Algo. on Words*, 1984.

[Mol93] D.I. Moldovan. *Parallel Processing : From Applications to Systems*. Morgan Kaufmann, 1993.

[MP70] J.H. Morris and V.R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[MP80] W.J. Masek and M.S. Paterson. A Faster Algorithm for Computing String Edit Distances. *J. Comp. Syst. Sci.*, 20 :18–31, 1980.

[MS99] J.F. Myoupo and D. Semé. Topological properties and broadcasting algorithms on X-mesh and X-torus with applications. *International Journal on Computers and their Applications*, 7(2) :104–110, 1999.

[MW97] J-F. Myoupo and A. Wabbi. Improved Linear Systolic Algorithms for Substrings Statistics. *Information Processing Letters*, 61 :253–258, 1997.

[New66] J. Von Newmann. *Theory of Self Reproducing Automata*. Univ. Ill. Press, 1966.

[P. 35] P. Erdos and A. Szekers. A combinatorial problem in geometry. *Compositio Mathematica*, 2 :463–470, 1935.

[QFR87] P. Quinton, F. Fogelman, and Y. Robert. *Automata Networks in Computer Science*, chapter 'The systematic design of systolic arrays'. Manchester Univ. Press, 1987.

[QR89] P. Quinton and Y. Robert. *Algorithmes et Architectures Systoliques*. Études et Recherche en Informatique. Masson, 1989.

[Ran87] A.G. Ranade. How to Emulate Shared Memory. In *Proceedings of the 28th Annual Symposium on Fondations of Computer Science*, pages 185–194, 1987.

[RT85] Y. Robert and M. Tchuente. A Systolic Array for the Longest Common Subsequence Problem. *IPL*, 21 :191–198, 1985.

[Ryt80] W. Rytter. A correct preprocessing algorithm for boyer-moore string searching. *SIAM J. Comput.*, 9(3) :509–512, 1980.

[Sch96] R. Scheihing. Combinatorial Problems in Strings. Master's thesis, Univ. Chile, 1996.

[Séé91] P. Séébold. Fibonacci Morphisms and Sturmian Words. *Theor. Comp. Sci.*, 88 :365–384, 1991.

[Sem99a] D. Semé. *Algorithmique parallèle pour les problèmes de reconnaissance de formes et de motifs sur les modèles systolique et BSR.* PhD thesis, Université de Picardie Jules Verne, Faculté de Mathématiques et d'Informatique, Amiens, France, 1999.

[Sem99b] D. Semé. An Efficient Algorithm on the BSR-Based Parallel Architecture for the k-LCS Problem. In *Proc. of the Intern. Conf. on Paral. and Distr. Process. Techn. and Appli., PDPTA-99*, 1999.

[SK83] D. Sankoff and J.B. Kruskal. *Time Warps, String Edits, and Macromolecules : The Theory and Practice of Sequence Comparison.* Reading, MA. Addison-Wesley, 2nd edition, 1983.

[Smy00] W. F. Smyth. Repetitive perhaps, but not boring. *Theor. Comput. Sci.*, 249(5) :343–355, 2000.

[Sny86] L. Snyder. Type Architectures, Shared Memory and the Corollary of Modest Potential. *Annual Review of Computer Science*, 1 :289–317, 1986.

[Sto88] J. Storer. *Data Compression : Methods and Theory.* Comp. Sci. Press, 1988.

[Sto97] I. Stojmenovic. Honeycomb networks : Topological properties and communication algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 8(10) :1036–1042, 1997.

[Stu91] S.S. Sturrock. Biological Sequence Comparisons on a Transputer Network. Master's thesis, Univ. Kent, 1991.

[SW89] T.F. Smith and M.S. Waterman. *Mathematical Methods for DNA Sequences.* CRC Press, 1989.

[Szy75] T.G. Szymanski. A Special Case of the Max Common Subsequences Problem. Technical report, Dep. Elec. Eng. Princeton University, Princeton N.J. Tech. Rep., 1975.

[Tho95] E. Thomas. Détection de répétitions dans un mot en parallèle. Master's thesis, Faculté de Mathématiques et d'Informatique, Université de Picardie Jules Verne, Amiens, France, 1995.

[Thu12] A. Thue. Über die gegenseitige Lage gleicher Teile Gewisser Zeichenreichen. *Skr. Vid.-Kristiana I. Mat. Naturv. Klasse*, 1 :1–67, 1912.

[Thu77] A. Thue. *Selected Mathematical Papers.* T. Nagell, A. Selberg, S. Selberg, K. Thalberg (Eds), Universitetsforlaget, Oslo, 1977.

[Ull84] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press Inc., 1984.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, 1990.

[Vie96] L. Viennot. *Quelques algorithmes parallèles et séquentiels de traitement de graphes et applications.* PhD thesis, Université Denis Diderot Paris 7, 1996.

[Wab98] A. Wabbi. *Architectures Parallèles systoliques pour la compression de données et la manipulation des sous-chaînes.* PhD thesis, Faculté de Mathématiques et d'Informatique, Université de Picardie Jules Verne, Amiens, France, 1998.

[Wat85] M.S. Waterman. Sequence alignments. *Mathematical Methods for DNA Sequence*, Ch 3 :53–92, 1985.