CHAPITRE 4

Parallélisation modulaire du classifieur incrémental

Introduction

Ce chapitre traite d'une parallélisation « modulaire » du classifieur incrémental. Le programme parallèle est évalué sur le problème de reconnaissance de dessins au trait. Comme dans le chapitre précédent, nous commençons par analyser la granularité des algorithmes à écrire (section 4.1) afin de choisir une architecture cible. La section 4.2 étudie les performances du classifieur sur le problème d'OCR. Nous verrons que les bons résultats obtenus peuvent profiter à la parallélisation précédente (section 4.3). Le reste du chapitre est dédié au processus d'apprentissage. La section 4.4 donne et commente un algorithme d'apprentissage parallèle dont le comportement et la rapidité sont discutés sections 4.5 et 4.6. Une analyse de l'algorithme et de la manière de l'accélérer (section 4.7) conduit à une version très efficace, mais uniquement dédiée à un apprentissage supervisé (section 4.8). Les sections terminant le chapitre améliorent encore le programme parallèle par une gestion de l'équilibrage des charges (section 4.9).

4.1 Approche modulaire

La parallélisation du chapitre précédent souffrait d'une granularité trop faible, notamment pour le prétraitement : trop de petits messages nuisaient à l'accélération. De plus, une utilisation sur un réseau de stations était impossible, notamment à cause de la latence importante d'un réseau comme *Ethernet* (voir l'annexe B.2.3). Nous allons donc utiliser une granularité plus élevée en plaçant des réseaux connexionnistes complets et autonomes sur chaque processeur. Cette approche est bien loin du parallélisme massif intrinsèque aux réseaux de neurones (une cellule par processeur) mais reste « biologiquement plausible ». En effet, à un niveau fonctionnel (ou cognitif), le cerveau est modulaire. Nos processeurs vont donc accueillir des modules, et nous verrons qu'il est intéressant de diversifier et de spécialiser leurs fonctions.

Définition d'un module

D'un point de vue connexionniste, un module peut être défini comme un ensemble de cellules accomplissant une fonction cognitive précise, avec une connectivité « intra-module » largement supérieure à la connectivité « inter-module ». Le point de vue informatique est un processus implémentant un module sur chaque processeur, le nombre réduit de connexions inter-module limitant les communications entre les processeurs.

Architecture cible

La forte granularité de la parallélisation que nous allons étudier autorise l'utilisation d'un environnement aux communications moins performantes, notamment un réseau de stations de travail. Ce travail sera présenté au prochain chapitre. Le présent chapitre présente l'implémentation sur une architecture MIMD à mémoire distribuée : la machine parallèle Volvox.

Principe de la parallélisation

La parallélisation modulaire de l'apprentissage et de la généralisation suivent le même principe. Chaque processeur héberge tous les prototypes et une partie de l'ensemble des exemples à apprendre ou à reconnaître. L'algorithme séquentiel (prétraitement et classification) est implémenté sur chaque processeur.

Remarque

Les modules traitant des exemples différents simultanément, il est ambigu de considérer la présentation d'un exemple comme unité de temps. Aussi, nous parlerons de « cycles » pour désigner la présentation d'un exemple à chaque module.

Exemple

La figure 4.1 présente une parallélisation modulaire sur 4 processeurs configurée en anneau. Les N formes à apprendre ou à reconnaître sont réparties entre les processeurs. Le classifieur incrémental séquentiel est implémenté sous la forme d'un module sur chaque processeur. Le classifieur traite donc N exemples en $N/4$ cycles ; chaque module apprend ou reconnaît $N/4$ exemples en $N/4$ cycles.

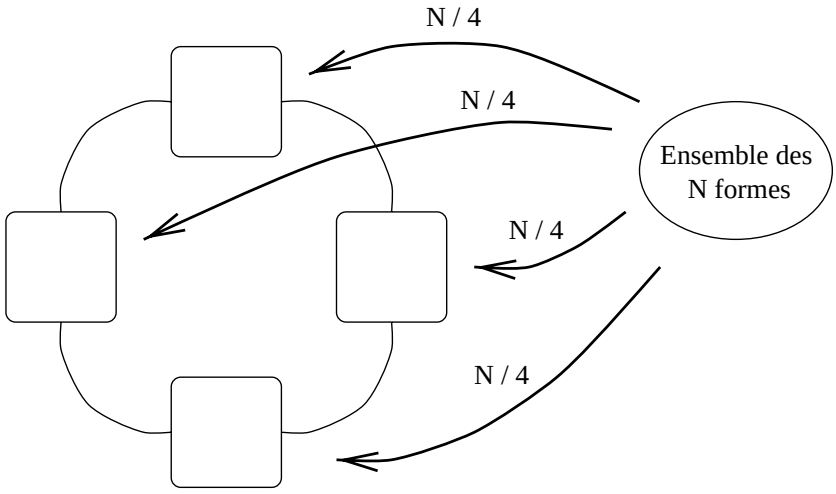

FIG. 4.1 – *Chaque processeur reçoit une partie de l'ensemble des N exemples*

4.2 Parallélisation de la généralisation

L'implémentation de cette stratégie de parallélisation n'a pas posé de problème particulier pour l'algorithme de généralisation. La table 4.1 présente les temps de généralisation du classifieur modulaire en fonction du nombre de processeurs. Les accélérations correspondantes sont calculées à partir du temps d'exécution du programme sur un seul processeur (voir les temps de référence donnés dans la section 1.2).

Processeurs	2	3	4	5	6	7	8	9	10
Temps (s)	54.38	36.22	27.08	21.69	18.23	15.83	13.87	12.41	11.09
Accélérations	1.89	2.84	3.80	4.74	5.64	6.49	7.41	8.28	9.27

TAB. 4.1 – *Temps et accélérations en généralisation pour 2 à 10 processeurs*

On constate que le programme parallèle va 9.27 fois plus vite avec 10 processeurs. L'accélération est donc quasi-linéaire. La régularité de l'accroissement de l'accélération confirme l'excellent comportement de l'algorithme de généralisation modulaire.

Temps réel

Une application nécessitant une tâche de classification en temps réel peut donc recourir au classifieur incrémental et à la stratégie modulaire. En effet, la bonne scalabilité constatée (voir l'annexe B.2.1) permet au système, c'est-à-dire aussi bien le prétraitement que la classification, de donner une réponse en un temps donné pour peu qu'un nombre suffisant de processeurs soient disponibles.

Disponibilité des exemples

Dans la version que nous présentons, la distribution des exemples est réalisée avant le début du processus de reconnaissance. Cette méthode permet d'évaluer le comportement du classifieur.
Dans un contexte industriel, les exemples peuvent être envoyés au fur et à mesure de leur disponibilité, on parle de traitement « en-ligne » (ou plus souvent « *on-line* »). Un pipeline d'exemples doit bien sûr être implémenté pour que le classifieur puisse simultanément traiter un exemple et recevoir le suivant[1]. Une fois la première forme reçue (chargement du pipeline), cette distribution en-ligne n'induit pas de ralentissement car le réseau d'interconnexion des processeurs est laissé totalement libre par la généralisation modulaire.

4.3 Indépendance des prototypes

Au chapitre précédent, nous avons parallélisé l'apprentissage, supervisé et non supervisé, en distribuant l'espace d'entrée entre les processeurs. Nous avons vu que

1. Dans l'éventualité peu probable d'un temps de communication supérieur au temps de reconnaissance de l'exemple (prétraitement et classification), notamment à cause d'une importante latence du réseau de communication, une zone tampon (un « *buffer* ») peut être géré afin de regrouper les temps d'initialisation de la communication.

les solutions mises en œuvre peuvent être utilisées en généralisation, l'unique différence étant la suppression de la dernière étape de l'algorithme parallèle (c'est-à-dire l'étape 7 de l'algorithme donné section 3.4).

Généralisation à moyen grain

Cependant, toutes les communications nécessaires durant l'apprentissage le sont encore durant la généralisation. Même dans le cas le plus favorable, c'est-à-dire avec un réseau rapide par rapport à la puissance des processeurs, et avec des exemples d'une grande dimension, certaines communications ne peuvent être recouvertes par les calculs. La généralisation à moyen grain est donc toujours moins rapide que la version à forte granularité. La différence s'accentue dans le cas d'un apprentissage en-ligne car le réseau de communication est déjà fortement sollicité par l'algorithme de généralisation.

Passage des prototypes

Cependant, la réponse donnée par le classifieur en généralisation ne dépend que des prototypes du classifieur et de la forme en entrée. Aussi, notre programme d'apprentissage parallèle à moyenne granularité peut il envoyer ses prototypes au programme de généralisation modulaire.

Fichiers de prototypes

Notons qu'il est également possible de sauvegarder les prototypes créés en apprentissage dans un fichier. Selon l'application, on peut donc se passer de la phase d'apprentissage. Par exemple, dans une application de reconnaissance de caractères dactylographiés l'apprentissage peut être fait une fois pour toute. Il suffit alors d'utiliser le programme de reconnaissance avec le « fichier des polices » ou si la police est (re)connue, le fichier d'une police précise.

Validation des programmes

Enfin, notons que le recours aux fichiers de sauvegarde des prototypes a facilité le long travail de validation des différents programmes en confrontant les résultats des différentes versions du classifieur. Ces expériences de validation sont capitales, notamment pour un réseau de neurones artificiels car un programme peut comporter un léger bogue et ne pas travailler réellement sur les exemples, tout en donnant de très bons résultats. Or, un programme parallèle présente des procédures complexes et donc propices aux erreurs.

Les tests suivants donnent deux exemples afin de préciser cette notion de validation :

- un fichier de prototypes donné doit conduire aux mêmes taux d'erreur avec chaque programme de généralisation (aussi bien les versions parallèles que la séquentielle) ;

- pour un ordre de présentation des exemples donné, l'algorithme d'apprentissage parallèle par distribution des exemples doit donner le même fichier de prototypes que l'algorithme d'apprentissage séquentiel.

4.4 Parallélisation de l'apprentissage

Cohérence des copies de l'ensemble des prototypes

Nous voulons donc apprendre une forme à chaque cycle, chaque module apprenant simultanément une forme différente. Aussi, chaque module va modifier son propre prototype gagnant, ou le cas échéant créer un nouveau prototype. Si l'on veut garder le même réseau de neurones sur chaque processeur, il est nécessaire de garder identiques toutes les « copies » de l'ensemble des prototypes.
Une étape de communication est donc nécessaire à l'issue de chaque cycle : chaque processeur envoie, à tous les autres, le prototype qu'il vient de créer ou de modifier.

Algorithme parallèle modulaire

Nous pouvons maintenant écrire l'algorithme d'apprentissage parallèle modulaire. Cette algorithme se distingue de la version séquentielle par l'étape de communication finale (étape *4*). Cette multidiffusion maintient la cohérence des copies de l'ensemble des prototypes en utilisant au mieux la configuration en anneau[2]. Comme pour la parallélisation par distribution de l'espace d'entrée, un schéma résume tout le processus (en fin de chapitre, figure 4.10, page 80).

Pour tous les processeurs :

1 Présenter un exemple.

2 Activer les cellules à champ récepteur (prétraitement).

3 Activer les prototypes,
rechercher un éventuel prototype gagnant $P_{gagnant}$,

3' créer un nouveau prototype $P_{nouveau}$ ou modifier $P_{gagnant}$.

4 *Maintenir la cohérence des copies de l'ensemble des prototypes* :

- Diffuser le prototype $P_{nouveau}$ ou $P_{gagnant}$.
- Recevoir les prototypes des autres modules,
intégrer localement les prototypes reçus.

Disponibilité des exemples

De même qu'en généralisation, la distribution des exemples est réalisée avant le début de l'apprentissage. La nécessité d'envoyer les exemples au fur et à mesure se fait moins sentir qu'en généralisation. Toutefois, cette seconde approche peut être motivée par une mémoire locale insuffisante pour stocker tous les exemples (chaque processeur héberge déjà un module). Bien qu'il soit moins disponible qu'en généralisation, le réseau d'interconnexion des processeurs peut être utilisé pour acheminer les exemples durant les étapes *1*, *2*, et *3* de l'algorithme donné ci-dessus.

2. Nous avons déjà utilisé cette technique dans le chapitre précédent, seules $p-1$ communications synchrones sont nécessaires avec p processeurs.

Gestion des arrivées

Lorsqu'un module reçoit un prototype (étape *4*), il doit être en mesure de déterminer si ce prototype est nouveau et, dans l'affirmative, le créer localement. Cette vérification impose l'utilisation d'une identification unique de chaque prototype. Nous identifions donc chaque prototype par un numéro affecté par le module où il a été créé, associé à un numéro identifiant le module en question.
Si un prototype reçu existe déjà localement, le module doit prendre en compte les modifications faites par le module émetteur. Une solution est le moyennage des deux prototypes, avec une éventuelle pondération par « l'âge » des prototypes (par exemple le nombre d'exemples présentés depuis sa création), ou par son « expérience » (par exemple le nombre de modifications intervenues sur le prototype considéré).
Cependant, les différentes politiques testées n'ont pas donné de différences significatives sur le problème d'OCR. Aussi, les résultats présentés on été obtenus en accordant la même importance aux prototypes locaux et reçus. Nous pouvons résumer la gestion des prototypes en arrivée par le petit algorithme suivant :

- si le prototype reçu, noté P_{recu}, est nouveau au sens de ses identifieurs, alors : ce prototype doit être créé localement : $P_{nouveau} \leftarrow P_{recu}$;
- sinon (une version P_i du prototype P_{recu} existe localement) : moyenner le prototype existant et le prototype reçu : $P_i \leftarrow \frac{1}{2}(P_i + P_{recu})$.

4.5 Comportement du classifieur durant l'apprentissage

Fluctuation du nombre de formes reconnues

La courbe 4.2 présente le pourcentage d'exemples reconnus en fonction du nombre de modules. Les variations du nombre d'exemples reconnus, bien que faibles, impliquent que l'ensemble des prototypes créés en apprentissage dépend du nombre de modules. On constate que l'utilisation d'un nombre important de modules dégrade la reconnaissance : les prototypes créés en apprentissage sont de moins bonne qualité.

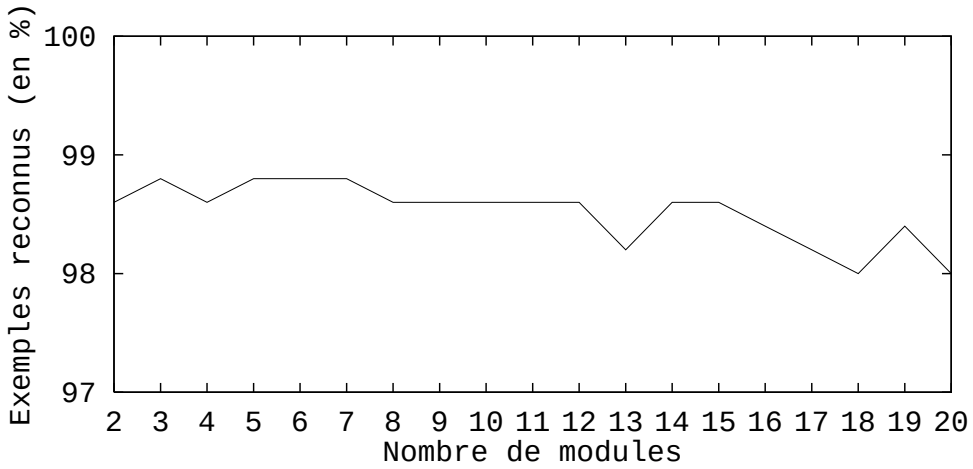

FIG. 4.2 – *Pourcentage d'exemples reconnus en fonction du nombre de modules*

Non respect de l'algorithme séquentiel

Analysons les causes des fluctuations du nombre de formes reconnues. Contrairement à la parallélisation par partage de l'espace d'entrée présentée au chapitre précédent, la parallélisation modulaire de l'apprentissage ne suit pas le même comportement que la version séquentielle. En effet, pendant qu'un module apprend une forme, il ne peut pas prendre en compte les formes apprises simultanément par les autres modules. Si davantage de modules apprennent simultanément, davantage d'exemples ne sont pas pris en compte par un module donné.

Apprentissage cloisonné

Aussi, lors de l'implémentation parallèle, l'augmentation du nombre de processeurs conduit à une augmentation de l'écart de comportement entre les versions parallèle et séquentielle du classifieur et conduit à un apprentissage indépendant que l'on peut qualifié de « cloisonné ».

Prototypes inutiles

Sur le problème d'OCR, ce problème de cloisonnement conduit à une augmentation du nombre de prototypes (figure 4.3), alors que le taux de formes reconnues diminue légèrement (figure 4.2). Le classifieur produit donc des prototypes inutiles, voir gênants.

Exemple

Prenons un exemple pour illustrer ce phénomène. Considérons que le classifieur n'a encore appris que des chiffres « 7 » dépourvus de barre horizontale. On présente au classifieur deux « 7 » avec une barre. Etudions la réaction des classifieurs séquentiel et parallèle. La version séquentielle crée un nouveau prototype à partir du premier « 7 », puis l'affine grâce au deuxième. Pour peu que les deux nouveaux « 7 » soient appris en même temps, la version parallèle va créer deux prototypes. Outre l'inutilité de la démarche, on remarque que les doublons vont se faire concurrence. La proximité des deux prototypes va empêcher l'émergence d'un gagnant si un nouveau « 7 barré » est présenté, ce qui va une fois de plus conduire à la création d'un prototype inutile.

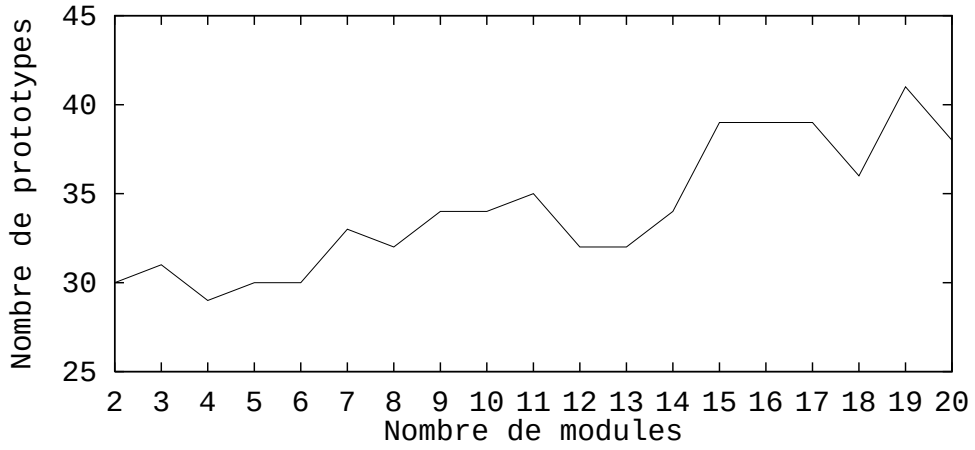

FIG. 4.3 – *Nombre de prototypes créés en fonction du nombre de modules*

4.6 Accélération atteinte

Les temps d'exécution en apprentissage, ainsi que les accélérations correspondantes, sont présentés dans la table 4.2 pour 2 à 10 processeurs. La figure 4.4 donne l'évolution de l'accélération en utilisant jusqu'à 20 processeurs, en apprentissage et en généralisation.

Processeurs	2	3	4	5	6	7	8	9	10
Temps (s)	74.04	63.77	62.09	55.00	57.00	52.73	51.25	48.86	49.38
Accélération	1.47	1.71	1.76	1.98	1.92	2.07	2.13	2.24	2.21

TAB. 4.2 – *Temps d'apprentissage et accélérations pour 2 à 10 processeurs*

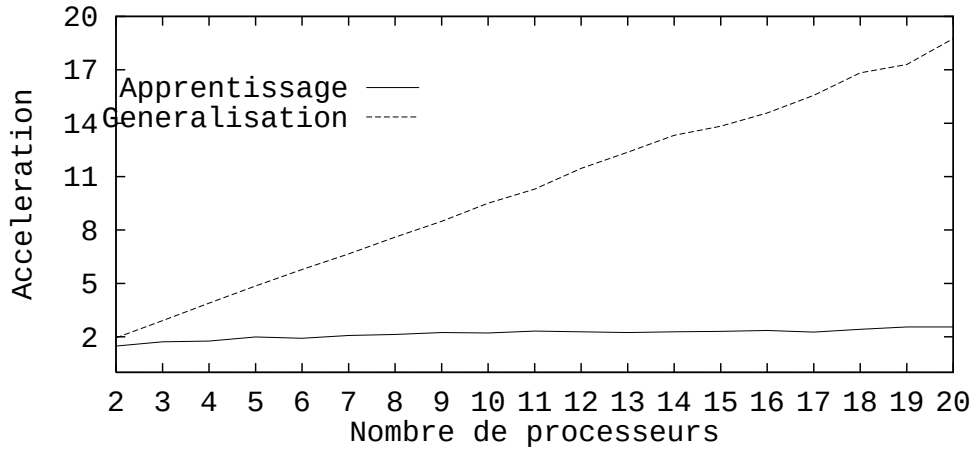

FIG. 4.4 – *Accélération en apprentissage et en généralisation*

Comparaison avec la distribution de l'espace d'entrée

La table 4.3 reprend les accélérations atteintes par la parallélisation modulaire (table 4.2) et par la distribution de l'espace d'entrée (table 3.1 du chapitre précédent). On constate qu'il est nécessaire d'utiliser au moins 9 processeurs pour que l'approche modulaire dépasse l'accélération obtenue avec la parallélisation par distribution de l'espace d'entrée avec 4 processeurs.

Processeurs	2	3	4	5	6	7	8	9	10
Chapitre 4	1.5	1.7	1.8	2.0	1.9	2.1	2.1	2.2	2.2
Chapitre 3	1.5	-	2.2	-	-	-	-	-	-

TAB. 4.3 – *Accélérations obtenues avec la parallélisation modulaire (chapitre 4) et par la parallélisation par distribution de l'espace d'entrée (chapitre 3)*

L'accélération de l'algorithme d'apprentissage modulaire reste stable. En effet, le temps d'apprentissage est divisé par deux quel que soit le nombre de processeurs.

Afin d'affiner l'analyse, les temps passés en prétraitement, en classification, et en communication ont été mesurés. Ces temps correspondent respectivement aux étapes *2*, *3* et *4* de l'algorithme d'apprentissage modulaire donné dans la section 4.4. La figure 4.5 décompose ces trois temps en fonction du nombre de processeurs. Les valeurs présentées sont calculées en cumulant les temps obtenus pour chacune des 500 formes apprises. On constate que ce qui est gagné en calcul est perdu en communication, d'où la faible pente de l'accélération en apprentissage.

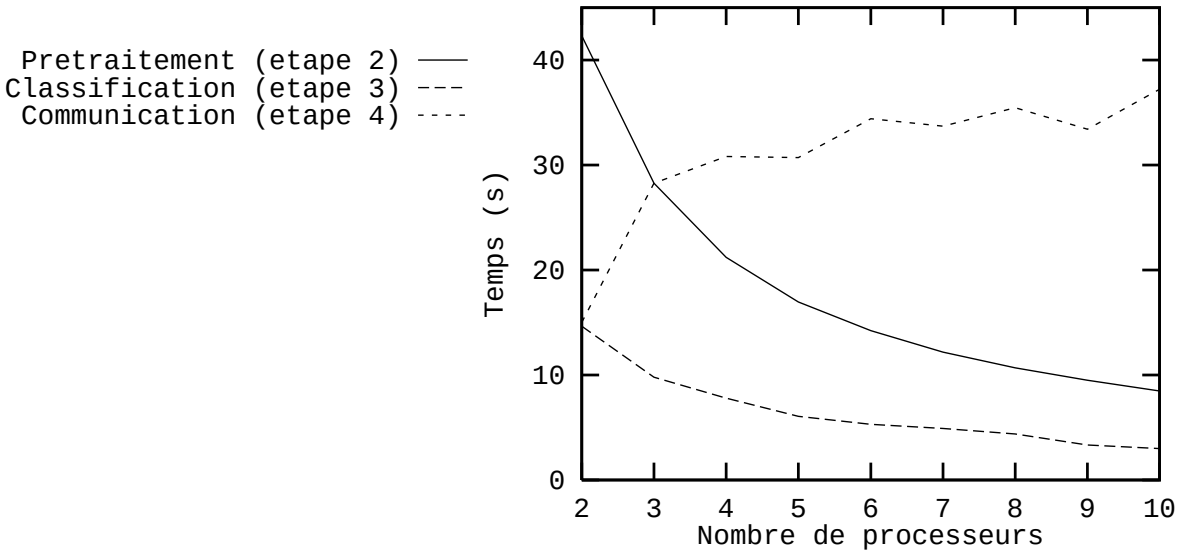

FIG. 4.5 – *Temps cumulés sur l'ensemble des exemples appris, pour le prétraitement, la classification, et les communications, en fonction du nombre de processeurs*

4.7 Amélioration de l'accélération en apprentissage

Dans la suite de ce chapitre, nous nous efforçons d'améliorer les performances du classifieur parallèle modulaire en apprentissage.

Réduction des communications

Nous avons constaté qu'une grande partie du temps d'apprentissage est perdue en communications. Aussi, une solution évidente pour améliorer l'accélération consiste en une diminution de la fréquence des communications (étape *4* de l'algorithme, section 4.4).
Cependant, il faut communiquer d'autant plus de données que les communications sont moins fréquentes. En effet, si une communication est effectuée tous les δ cycles, chaque module peut avoir créé ou modifié δ prototypes, et δ prototypes doivent être transmis par chaque processeur vers tous les autres.

Petits avantages

L'espacement des communications conduit tout de même à un gain de temps car le temps total passé à initialiser chaque communication est réduit. Le gain est d'autant plus important que le réseau a une latence élevée.
Par ailleurs, les prototypes comprenant un nombre variable de composantes nulles non transmises dans notre implémentation, la quantité de données effective à envoyer varie légèrement d'un prototype à l'autre. Aussi, regrouper plusieurs prototypes moyenne les tailles et diminue la perte de temps induite par la multidiffusion synchrone de messages de tailles différentes.

Gros inconvénients

Si le gain en vitesse induit par la diminution de la fréquence des communications est faible, il n'en va pas de même pour l'écart de comportement entre les versions séquentielle et parallèle. En effet, du point de vue d'un module, communiquer deux fois moins revient à apprendre deux fois plus d'exemples « simultanément ». Aussi, plus on augmente le nombre de processeurs, plus l'apprentissage est cloisonné. Pour le problème d'OCR, ce comportement du classifieur conduit à une diminution du taux de reconnaissance.

4.8 Solution dans le cas d'un apprentissage supervisé

Modification d'un même prototype

La proposition « Si une communication est effectuée tous les δ cycles, chaque processeur a δ prototypes à transmettre » énoncée dans la section précédente n'est pas toujours vraie. En effet, si un même prototype est modifié plusieurs fois entre deux communications, il n'a pas pour autant à être transmis plusieurs fois : seule la dernière version intéresse les autres processeurs. La solution consiste donc à augmenter la probabilité pour qu'un processeur modifie souvent les mêmes prototypes.

Regroupement des exemples par classe

En apprentissage supervisé, nous pouvons augmenter aisément la probabilité pour qu'un processeur modifie souvent les mêmes prototypes en regroupant l'apprentissage des exemples à apprendre d'une même classe sur un même processeur.

Vitesse de l'apprentissage

Nous avons résolu notre problème de vitesse puisque nous pouvons maintenant réduire le nombre de communications sans en augmenter la taille dans les mêmes proportions. De plus, il vient de l'algorithme d'apprentissage que seuls les prototypes de la même classe que l'exemple à apprendre peuvent être créés ou modifiés (*winner takes all*). Aussi, même si l'écart entre deux communications est très grand, le nombre de prototypes à envoyer ne peut excéder le nombre de prototypes représentant la classe dont le module émetteur à la charge.

Qualité des prototypes

Mais nous avons vu que, plus nombreux sont les modules, plus cloisonné est l'apprentissage, et plus bas sont les taux de reconnaissance en généralisation. Or, à quoi bon apprendre plus vite si c'est pour apprendre mal? En fait, les modules étant spécialisés, chaque processeur n'a à apprendre qu'une seule classe. Du point de vue

des prototypes, chaque processeur ne doit s'occuper que des prototypes d'une classe donnée. Les autres prototypes sont indispensables pour guider l'apprentissage, mais posséder la dernière version de ces prototypes n'est pas capital. Avec notre stratégie, la multidiffusion ne sert qu'à mettre à jour tous les prototypes modifiés et à diffuser les nouveaux prototypes, deux de ces mises à jour peuvent donc être séparées par l'apprentissage de nombreux exemples.

Exemple

À titre d'exemple, dans une application de reconnaissance de chiffres manuscrits, le module en charge de la classe des « 8 » crée et affine les prototypes correspondants. Ce processeur a besoin des prototypes des autres chiffres, notamment les chiffres proches comme le « 3 ». Les représentants de la classe « 3 » sont par exemple indispensables pour qu'un prototype de la classe « 8 » s'interpose lorsqu'un exemple de « 8 » à apprendre active injustement un prototype de la classe « 3 ». Cependant, il n'est pas indispensable que les prototypes de la classe « 3 » soient les plus précis actuellement disponibles, notamment si cela économise des communications.

Affinage de la gestion des prototypes arrivant d'un module spécialisé

Nous avons vu que, chaque module étant spécialisé dans une classe, un prototype reçu provient toujours d'un module en charge de la classe correspondante. Nous pouvons donc affiner l'intégration des prototypes reçus. En effet si le prototype reçu existe localement, il ne peut s'agir que d'une ancienne version. La version locale n'a jamais été affinée localement et doit être remplacée par le prototype reçu :

- si le prototype reçu, noté P_{recu}, est nouveau au sens de ses deux identifieurs, alors : ce prototype doit être créé localement : $P_{nouveau} \leftarrow P_{recu}$;
- sinon (une version P_i du prototype P_{recu} existe localement) :
  remplacer le prototype existant par le prototype reçu : $P_i \leftarrow P_{recu}$.

Algorithme

Nous pouvons maintenant donner l'algorithme d'apprentissage modulaire spécialisé. Cet algorithme, nécessairement supervisé, présente une étape *4* modifiée : δ exemples sont appris entre deux multidiffusions successives.

Pour tous les modules :

1 Présenter un exemple (d'une classe donnée).

2 Activer les cellules à champ récepteur (prétraitement).

3 Activer les prototypes, rechercher un éventuel prototype gagnant $P_{gagnant}$,
3' créer un nouveau prototype $P_{nouveau}$ ou modifier $P_{gagnant}$.

4 **Si** δ exemples ont été présentés depuis la dernière communication,
alors diffuser les nouveaux prototypes et les prototypes modifiés.

Cas du problème d'OCR

Nous avons implémenté cette variante de l'algorithme d'apprentissage modulaire. Nous utilisons 10 processeurs, chaque processeur héberge un module spécialisé sur une classe. Le réseau apprend les 500 formes, chaque processeur doit donc apprendre 50 exemples en 50 cycles. La table 4.4, ainsi que les courbes 4.6 et 4.7, présentent l'évolution du temps d'apprentissage et de l'accélération en fonction du délai entre 2 communications : le délai varie de $\delta = 1$ (une communication par cycle) à $\delta = 30$ (une communication au bout de 30 cycles) ; de plus, une communication est systématiquement effectuée après le dernier cycle.
On constate que l'accélération est de 9.39 pour un δ de 30 : le classifieur apprend presque 10 fois plus vite avec 10 processeurs. Nous pouvons donc apprendre avec une accélération quasi-linéaire.

Délai (δ)	1	2	3	4	5	10	15	20	25	30
Temps (s)	58.43	34.56	26.19	21.70	18.30	14.18	13.29	12.50	11.39	11.29
Accélér.	1.87	3.16	4.17	5.04	5.97	7.71	8.23	8.75	9.60	9.69

TAB. 4.4 – *Temps d'apprentissage et accélérations de l'algorithme modulaire spécialisé en fonction du délai (δ), pour 10 processeurs*

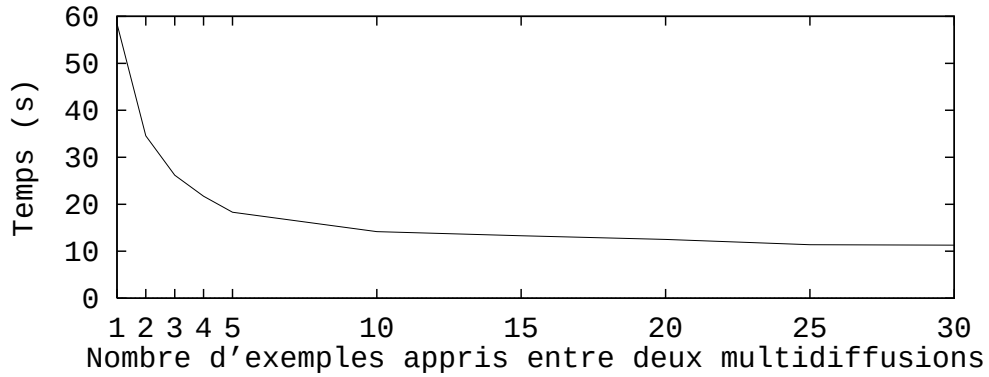

FIG. 4.6 – *Temps d'apprentissage en fonction du délai, pour 10 processeurs*

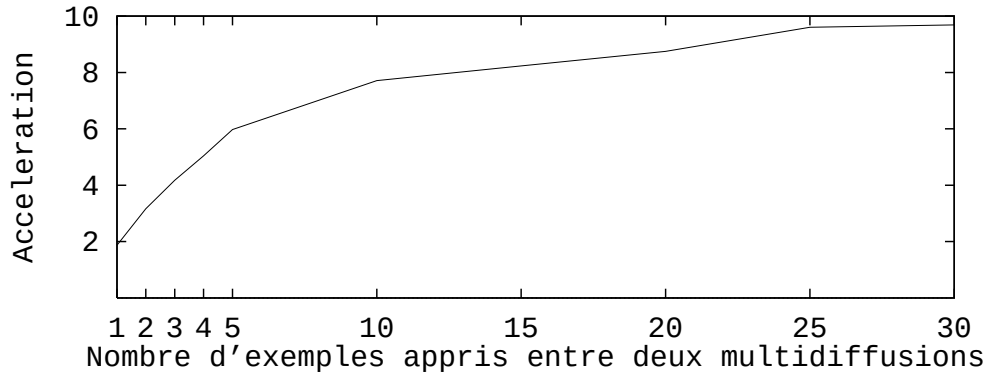

FIG. 4.7 – *Accélérations en apprentissage en fonction du délai, pour 10 processeurs*

Évolution du pourcentage de formes reconnues

Afin de vérifier la qualité de l'apprentissage, la table 4.5 et la courbe 4.8 donnent l'évolution du pourcentage moyen de formes reconnues en généralisation (moyenne calculée à partir de plusieurs ordres de présentation des exemples). Les résultats sont toujours donnés en fonction du délai (δ) entre les communications en apprentissage. On constate que le nombre d'erreurs n'augmente pas, malgré l'augmentation du délai, et donc la diminution du nombre de communications. Nous avons atteint notre objectif: communiquer moins durant l'apprentissage et obtenir une accélération linéaire, sans diminuer les performances du classifieur en généralisation.

Délai (δ)	1	2	3	4	5	10	15	20	25	30
Ex. reconnus (%)	98.6	98.6	98.6	98.6	98.8	99.2	99.4	99.0	99.0	98.8

TAB. 4.5 – *Pourcentage d'exemples reconnus en fonction du délai (δ)*

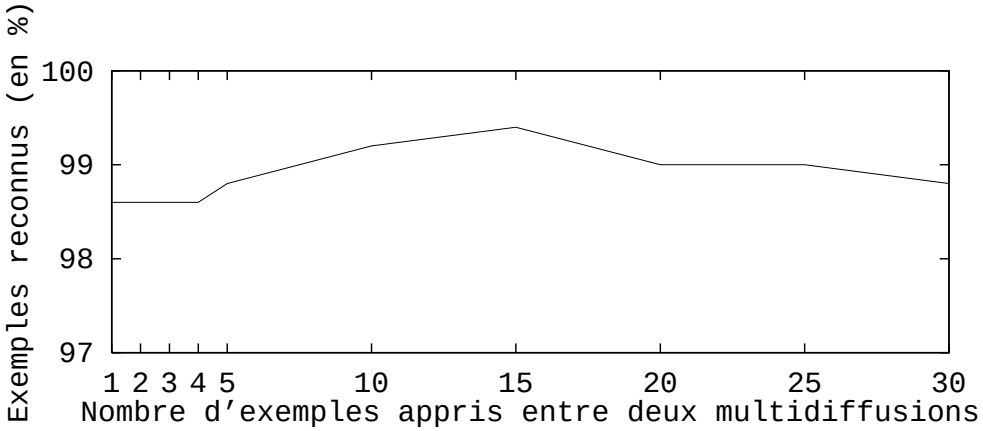

FIG. 4.8 – *Pourcentage d'exemples reconnus en fonction du délai (δ)*

4.9 Équilibrage des calculs

Etude du réseau construit

L'étude des connexions d'un réseau de neurones ayant recours à des prototypes pour discriminer l'espace d'entrée peut être riche d'enseignements, notamment sur l'apprentissage. De plus, contrairement à un réseau MLP, il est relativement facile d'étudier le réseau car la « connaissance » n'est pas distribuée sur l'ensemble des connexions. Dans cette section, nous étudions le nombre de prototypes créés par chaque module (c'est-à-dire sur chaque processeur).

Prototypes créés localement

Le nombre de prototypes créés sur chaque processeur pour le problème d'OCR est donné par la table 4.6. Seules les créations locales sont comptées, cela exclut les prototypes reçus. Il s'agit donc des créations de prototypes représentant la classe dont le module a la charge. Les valeurs présentées sont obtenues avec une multidiffusion après chaque exemple ($\delta = 1$). Les modules sont numérotés en fonction de la classe dont ils ont la charge (le module i à la charge des chiffres « i »).

Module	0	1	2	3	4	5	6	7	8	9
Créations	3	2	5	3	4	1	2	3	3	5

TAB. 4.6 – *Nombre de prototypes créés par chaque module (sans délai, $\delta = 1$)*

On constate que l'amplitude du nombre de créations d'un module à un autre est élevée. En effet, les classes sont plus ou moins difficiles à discriminer les unes par rapport aux autres.

Notons que ces variations n'induisent pas de déséquilibre de la charge d'un processeur à l'autre car pour chaque exemple à apprendre : (i) chaque module teste les 31 prototypes (c'est-à-dire les reçus comme les locaux) ; (ii) chaque module modifie ou crée 1 et seulement 1 prototype. La charge de travail est donc la même.

Diminution des temps de classification

Nous allons voir que la diminution du nombre de communications n'influe pas seulement sur le temps perdu à communiquer, mais aussi sur le temps de classification. En effet, deux phénomènes favorisent l'accélération.

1. Du point de vue d'un module donné, une réduction du nombre de multidiffusions ($\delta > 1$) est un gage de tranquillité : les prototypes créés par les autres modules arrivent moins souvent. Du point de vue du classifieur, moins de communications diminue le nombre moyen de prototypes présents sur chaque processeur à chaque instant (c'est-à-dire pour chaque exemple). En d'autres termes, communiquer moins diminue le nombre total de prototypes à activer sur l'ensemble de tous les exemples à apprendre et diminue d'autant le temps de classification.

2. De plus, la baisse du nombre de communications diminue le nombre total de prototypes créés du fait du cloisonnement de l'apprentissage. La table 4.7 présente le nombre de prototypes créés sur chaque module pour plusieurs délais dans le cas de l'application d'OCR. On constate que le nombre de prototypes est inchangé pour certaines classes et diminue pour d'autres. Le nombre de prototypes total passe de 31 à 23 sans incidence sur les taux d'erreur. Une fois de plus, ce phénomène diminue le temps de classification et participe à l'excellente accélération de l'apprentissage.

L'interprétation précise de ce deuxième phénomène est difficile. Avec une communication après chaque exemple, une forme atypique présentée à un des processeurs va générer un prototype qui va être reçu par tous les modules au cycle suivant. Ce prototype peut alors entrer en concurrence avec un prototype local et induire la création de nouveaux prototypes parfois inutiles. Si les communications sont plus rares, le prototype atypique a le temps d'évoluer pour représenter plusieurs exemples et s'éloigner d'un prototype d'une autre classe (en somme devenir moins atypique).

Délais (δ)	1	2	3	4	5	10	15	20	25	30
Module 0	3	3	3	3	3	3	3	3	3	3
Module 1	2	2	2	2	2	2	2	2	2	2
Module 2	5	4	4	4	4	4	4	4	4	5
Module 3	3	3	3	3	2	2	2	2	2	1
Module 4	4	4	4	4	4	2	2	2	1	1
Module 5	1	1	1	1	1	1	1	1	1	1
Module 6	2	2	2	2	2	2	2	1	1	1
Module 7	3	3	3	3	3	2	2	2	2	2
Module 8	3	3	3	3	3	3	3	3	3	3
Module 9	5	5	5	4	4	6	5	5	3	4
Total	31	30	30	29	28	27	26	25	22	23

TAB. 4.7 – *Nombre de prototypes créés par chaque module, pour différents délais (δ)*

Exemple

À titre d'exemple, le premier « 8 » non complètement fermé appris va devenir un prototype et être diffusé aux autres processeurs où il peut générer de nouveaux prototypes, notamment pour les modules en charge des classes « 6 », « 9 » et « 3 ». Si peu de communications interviennent, le prototype issu du « 8 » mal fermé peut être affiné par d'autres exemples et s'éloigner des autres chiffres (tout en restant une manière différente de faire un « 8 »). Notons qu'un prototype issu d'un exemple atypique peut également rester suffisamment proche d'un prototype d'une autre classe et provoquer une création. Prenons par exemple le cas du « 7 » barré ou non barré. Si le classifieur n'a encore appris que des « 7 » barrés, un « 7 » non barré va devenir un prototype. Si l'on communique souvent, ce prototype va rapidement arriver au module en charge des « 1 » et provoquer une création. Si l'on communique rarement, les éventuels autres « 7 » non barrés vont affiner leur prototype mais n'ont aucune raison de l'éloigner d'un « 1 ». Le module en charge des « 1 » va donc toujours créer un prototype pour s'interposer entre les exemples de « 1 » proches d'un « 7 » non barré et le prototype reçu.

Mesure des temps de prétraitement, de classification et de communication

L'influence du délai entre les communications sur le temps de classification est intéressante. Aussi avons-nous mesuré individuellement les temps de prétraitement, de classification et de communication. Toutes ces mesures ont été réalisées pour plusieurs valeurs du délai entre deux communications, sur l'application d'OCR. La figure 4.9 présente ces temps cumulés sur l'ensemble des exemples à apprendre.

- Comme prévu, communiquer moins souvent diminue drastiquement le temps passé à communiquer du fait de la spécialisation des modules. Cette baisse est la principale cause de l'accélération de l'apprentissage.
- On vérifie également la baisse du temps de classification, bien que l'incidence sur l'accélération soit faible, ce résultat est des plus intéressant.

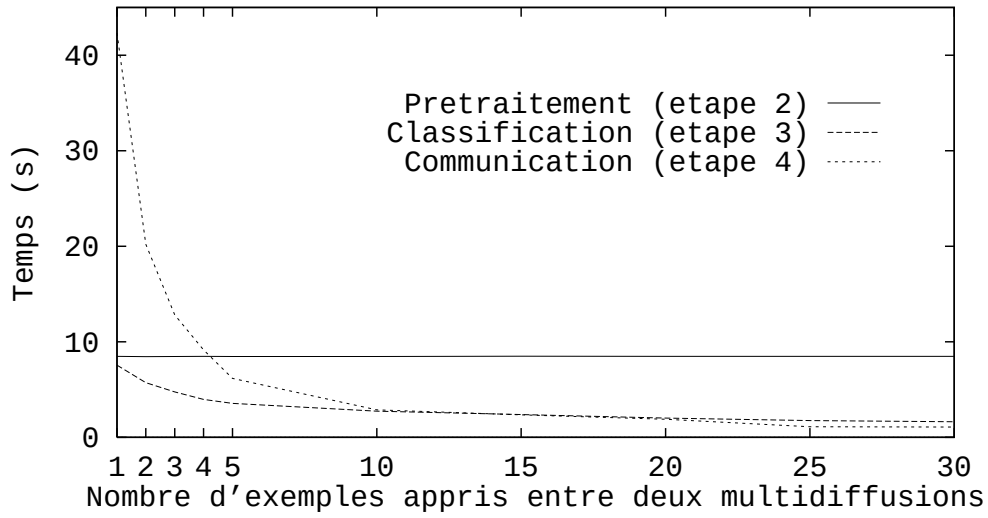

FIG. 4.9 – *Temps cumulés sur l'ensemble des 500 exemples appris, pour le prétraitement, la classification, et les communications, en fonction du délai (δ)*

Déséquilibre avec $\delta > 1$

Mais l'augmentation du délai entre deux communications a aussi une influence négative sur le temps de classification. En effet, un nouveau prototype créé par un des modules ne va pas arriver immédiatement sur les autres processeurs. Aussi, moins on communique, plus les modules créant peu de prototypes ont longtemps moins de prototypes à activer. Ce déséquilibre de la charge des processeurs est nuisible à l'accélération. En effet, les modules étant synchrones, les processeurs les moins chargés en prototypes seront inactifs jusqu'au début de la multidiffusion suivante.

Solution

Le déséquilibre peut être réduit si l'on utilise moins de processeurs ou si l'application comprend plus de classes qu'on ne dispose de processeurs. En effet, on peut dès lors placer plusieurs modules sur un même processeur. Un placement judicieux permet de minimiser le déséquilibre en minimisant l'amplitude entre le nombre total de prototypes créés sur chaque processeur. Ce placement peut être effectué par le classifieur statiquement à l'aide de la table 4.7, c'est-à-dire avant de commencer l'apprentissage. Cependant, cette optimisation impose d'avoir déjà réalisé un apprentissage (pour connaître la table).

Accélération supra-linéaire

Cet équilibrage n'a pas une influence considérable sur la vitesse d'apprentissage. Cependant, cette ultime optimisation permet à l'accélération de dépasser le nombre de processeurs. Nous avons observé un tel comportement du classifieur en apprentissage avec 5 processeurs et un délai de 50 exemples entre deux communications ($\delta = 50$). Comme proposé ci-dessus, les 10 classes ont été regroupées par 2 en minimisant le nombre de prototypes créés donnés par la table 4.7. L'apprentissage ne nécessite que 21.5 secondes, ce qui donne une accélération supra-linéaire de 5.1 avec 5 processeurs (98 % des exemples sont reconnus).

Un apprentissage facilité

Ce résultat mérite une explication car il vient de la définition théorique de l'accélération (formule B.1) qu'une accélération ne peut être supérieure au nombre de processeurs. Mais cette limite n'est vraie que pour une parallélisation « classique », pour laquelle le programme parallèle doit se comporter de la même manière que le programme séquentiel. C'est par exemple le cas pour la parallélisation présentée au chapitre précédent.
Notre stratégie modulaire revient à paralléliser la manière d'apprendre, aussi elle n'impose nullement au programme parallèle le cheminement séquentiel. La seule contrainte est d'obtenir de bons prototypes, donnant de bons résultats en généralisation. On constate que sur le problème d'OCR, notre distribution du travail « facilite » l'apprentissage et conduit à une accélération supra-linéaire.

Cela dit, nous pouvons dans une certaine mesure modifier l'algorithme séquentiel pour mettre à profit nos constatations et faciliter l'apprentissage.

Définition MIMD de l'accélération

En effet, nous calculons l'accélération grâce à la formule adaptée à une architecture MIMD (voir si nécessaire la formule B.2 donnée dans l'annexe). On compare donc les temps d'exécution obtenus sur un processeur et sur p processeurs de la machine parallèle. Avec cette définition, nous trouvons :

$$\frac{temps\ du\ programme\ sur\ 1\ processeur}{temps\ du\ programme\ avec\ p\ processeurs} > p$$

Définition théorique de l'accélération

Mais la définition théorique de l'accélération fait référence au « meilleur programme séquentiel » et implique que :

$$p \geq \frac{temps\ du\ meilleur\ algorithme\ séquentiel}{temps\ de\ l'algorithme\ parallèle\ évalué}$$

Donc :

$$temps\ du\ meilleur\ algorithme\ séquentiel < temps\ du\ programme\ sur\ 1\ processeur$$

L'algorithme mis en œuvre sur l'un des processeurs, c'est-à-dire l'algorithme séquentiel, peut donc être amélioré. Par exemple, lors d'un apprentissage supervisé, l'algorithme séquentiel pourrait ne pas toujours vérifier l'ensemble de tous les prototypes, mais seulement ceux de la classe à tester. Cette remarque sera mise à profit dans un prochain chapitre (section 6.6).

Conclusion

Nous avons parallélisé le classifieur en implémentant un ou plusieurs modules sur chacun des processeurs, et en répartissant le travail (c'est-à-dire les exemples à apprendre ou à reconnaître) entre les modules. Cette stratégie est originale car un algorithme parallèle suit généralement à la lettre l'algorithme séquentiel. En ce sens, la parallélisation modulaire diverge donc radicalement de la parallélisation par partage de l'espace d'entrée présentée au chapitre précédent. En fait, on ne cherche

pas à paralléliser le réseau de neurones mais la stratégie d'apprentissage, en partant du postulat qu'un apprentissage en parallèle n'a pas de raison d'être moins efficace qu'un apprentissage séquentiel. Les processeurs collaborent en travaillant sur un sous-problème cohérent (l'apprentissage d'une classe donnée) et s'échangent régulièrement leurs résultats.

Résultats en généralisation

Les résultats obtenus sont parfaits en généralisation : l'accélération est quasi-linéaire et la vitesse maximale du système complet (prétraitement et classification) ne dépend que du nombre de processeurs disponibles. De plus, le prétraitement propre à chaque application n'a pas à être modifié, une simple recompilation sur le processeur de la machine est nécessaire. La parallélisation modulaire est donc une très bonne solution, notamment pour une application temps réel. Enfin, la grande disponibilité du réseau d'interconnexion des processeurs autorise une reconnaissance en-ligne.

Apprentissage supervisé

De même, l'algorithme d'apprentissage ne nécessite aucune modification du prétraitement. Dans le cas d'un apprentissage supervisé, une spécialisation des processeurs par classe a conduit à une accélération linéaire sur la tâche de reconnaissance de dessins au trait. Bien que le comportement du classifieur diverge fortement du comportement séquentiel, l'accélération linéaire n'est pas obtenue au détriment de la qualité de l'apprentissage. Enfin, l'accélération peut être supérieure au nombre de processeurs. Ce fait impossible avec une parallélisation classique signifie que notre stratégie d'un apprentissage en parallèle facilite l'apprentissage du problème d'OCR.

Apprentissage non supervisé

En apprentissage non supervisé, il n'est *a priori* pas possible de décomposer le travail en sous-tâches homogènes. Les processeurs doivent donc communiquer constamment pour ne pas faire un travail à la fois redondant et incomplet. Le compromis entre vitesse et qualité de l'apprentissage étant difficilement acceptable, il est préférable d'utiliser la stratégie présentée au chapitre précédent lorsqu'un apprentissage non supervisé est nécessaire. Les prototypes résultant de cet apprentissage peuvent ensuite être utilisés par l'algorithme de généralisation modulaire.

Perspectives

Le travail réalisé se veut complet mais pas exhaustif. L'objectif est toujours d'arriver à une solution parallèle efficace, tout en étudiant les aspects les plus originaux des solutions proposées. Cela dit, de nombreuses études intéressantes peuvent encore être menées sur les programmes présentés dans ce chapitre.
Il serait par exemple intéressant de communiquer davantage au début de l'apprentissage, puis de diminuer progressivement la fréquence des communications.
Une autre voie intéressante est l'utilisation de plus de processeurs qu'il n'existe de classes à apprendre. On peut par exemple spécialiser plusieurs processeurs sur chaque classe, et communiquer davantage entre processeurs en charge d'une même classe.

Approche asynchrone

La parallélisation modulaire que nous venons d'étudier est issue d'une approche résolument synchrone qui est parfaitement adaptée à une machine parallèle. En effet, ayant la maîtrise totale des processeurs et du réseau d'interconnexion, on tire avantage à tout orchestrer pour optimiser une topologie donnée. Le chapitre suivant présente une approche asynchrone imposée par l'imprévisibilité de l'architecture mise en œuvre : un réseau local de stations de travail. De plus, nous avons vu que l'apprentissage supervisé modulaire peut être efficace et de qualité malgré un délai élevé entre deux communications. Dans ces conditions, l'intérêt d'optimiser l'utilisation du réseau pendant une phase de communication intense (et donc synchrone) diminue. Au contraire, il devient plus intéressant de profiter des longues périodes d'inutilisation du réseau d'interconnexion des processeurs pour communiquer. Aussi, si l'on communique peu, l'approche asynchrone présentée au prochain chapitre est utilisable aussi bien sûr un réseau de stations que sur une machine parallèle MIMD supportant PVM.

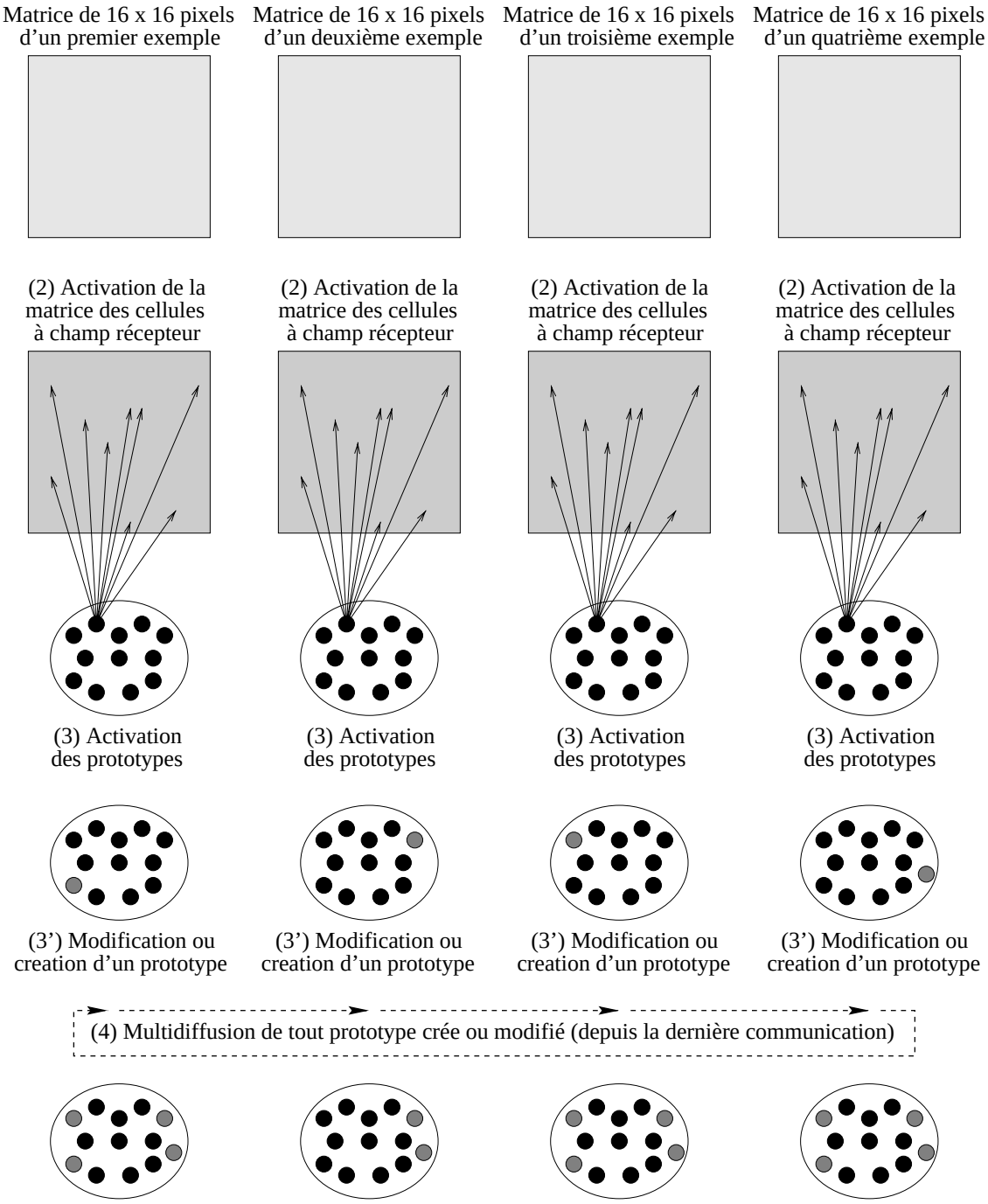

FIG. 4.10 – *Schéma complet (les numéros de l'algorithme donné section 4.4)*