CHAPITRE 5

Parallélisation du classifieur sur une machine parallèle virtuelle

Introduction

Au début du chapitre précédent, nous avons remarqué que la parallélisation modulaire nécessite une architecture à forte granularité (section 4.1). Cette analyse a conduit à la parallélisation du classifieur sur une machine MIMD. Ce chapitre présente le portage de la parallélisation modulaire sur une autre architecture à forte granularité : un réseau de stations de travail.
La section 5.1 rappelle la topologie mise en œuvre sur la Volvox pour la comparer à celle imposée par un réseau local avec le protocole *Ethernet* sous PVM. Cette étude nous conduit à proposer une parallélisation de l'acquisition des données (section 5.2), ainsi qu'un nouvel algorithme adapté à PVM (section 5.3). Le risque d'un apprentissage déséquilibré dû au caractère asynchrone de l'algorithme est mis à jour et résolu (section 5.4). Nous verrons que le risque de défaillance d'un processeur de la machine virtuelle (c'est-à-dire d'une station de travail) est plus important que pour une machine parallèle. Aussi, nous proposons un algorithme tolérant aux pannes dans la section 5.5. L'étude du classifieur modulaire sous PVM est achevée par la présentation d'une implémentation pour valider l'algorithme (sections 5.6 à 5.8).

5.1 Parallélisation modulaire sur réseau de stations

Les processeurs d'un ordinateur parallèle peuvent être interconnectés de nombreuses façons. Les topologies les plus classiques sont l'anneau, la grille, la grille torique, et l'hypercube. L'algorithme d'apprentissage modulaire supervisé présenté au chapitre précédent n'a besoin que d'une configuration en anneau pour atteindre une accélération quasi-linéaire (accélération de 9.69 avec 10 processeurs), voire supra-linéaire (accélération de 5.1 avec 5 processeurs).

Le protocole Ethernet

Des ordinateurs connectés à un réseau local suivant le protocole standard *Ethernet*, ne communiquent pas avec autant de facilité que les processeurs d'une machine parallèle MIMD. L'énumération qui suit donne les trois principales raisons de cette infériorité.

- La topologie du réseau reliant les stations est limitée au bus (figure 5.1), l'envoi d'un message d'une machine à une autre mobilise la totalité du réseau durant la totalité de la communication.
- La gestion du trafic est décentralisée. Au niveau du protocole interne de communication du réseau, l'envoi d'un message impose potentiellement de multiples essais suivis d'attentes aléatoires.
  En effet, si une machine désire envoyer un message, elle doit « écouter » et attendre que le bus soit libre. Elle tente alors d'envoyer son message tout en l'écoutant. Le message est inaudible si une autre station a également commencé une communication. La gestion de cette « collision » est résolue en attendant un temps tiré aléatoirement avant de réitérer la tentative d'émission (pour peu que le bus soit resté libre, c'est-à-dire si le hasard a fait que la station a tiré un temps d'attente plus court que ses concurrentes).
- L'initialisation de la communication se fait à travers de nombreuses couches logicielles pour garantir l'indépendance d'*Ethernet* et du matériel (station de travail, ordinateur compatible IBM ou Apple, *etc.*) et du système d'exploitation (Unix, MS-Mindows, MacOS, *etc.*). La latence est donc bien plus élevée que sur le réseau d'interconnexion des processeurs d'une machine parallèle.

L'environnement PVM sur réseau Ethernet

En fait, un réseau « *Ethernet* » est avant tout un ensemble de cartes et de câbles respectant une norme. Cette infrastructure peut utiliser plusieurs protocoles, dont « IP » (*Internet Protocol*) retenu par des développeurs de PVM. De nombreux équipements cohabitent sur un réseau de type *Ethernet*, pour améliorer sa fluidité, sa sécurité, ou encore son ouverture (par exemple des « *hub* », des barrières (*firewalls*), des passerelles, *etc.*). Cette complexité rend délicate toute tentative d'optimisation des fonctions de communication de PVM.
À titre d'exemple, le protocole ne supporte pas de fonctions évoluées comme la diffusion d'un message d'une station vers toutes les autres alors que la topologie bus le permet en théorie. Une diffusion vers p correspondants passe par p messages indépendants.

Latence et taux de transfert

De même, la latence et la bande passante sont largement en-dessous de ce dont est capable le réseau d'interconnexion d'une machine parallèle moderne. La table 5.1 donne latence et bande passante pour un réseau *Ethernet* (avec IP) sur un réseau de stations de travail SUN (des « *Sparc 5* »).

Protocole	β (en μs)	τ (en $\mu s/o$)
IP sur *Ethernet*	695	0.84
PVM sur IP, IP sur *Ethernet*	1500	0.85

TAB. 5.1 – *Latence et taux de transfert sur Ethernet [PT96, PT97]*

On constate que l'augmentation importante de la latence induite par l'utilisation du couple *Ethernet*-IP est aggravée par PVM, la situation se dégrade encore davantage si la configuration des machines mises en œuvre est hétérogène. En effet, échanger un message entre deux machines d'architectures différentes, avec éventuellement un système d'exploitation différent, oblige à coder les données dans un format commun. En revanche, la bande passante est faiblement diminuée par PVM (1.17 contre 1.19 Mo/s). Notons qu'un nouveau standard peu coûteux, et d'une bande passante 10 fois supérieure, nommé « *Fast Ethernet* » remplace progressivement *Ethernet*.

Multidiffusion sous PVM

À de maintes reprises, nous avons mis en œuvre des multidiffusions sur un anneau. Avec p processeurs, cette opération ne nécessite que $(p \Leftrightarrow 1)$ communications. Une multidiffusion avec PVM et *Ethernet* (figure 5.1) nécessite $(p \Leftrightarrow 1) \times p$ communications. De plus, chaque machine tente de communiquer au même moment, le lourd protocole de gestion des collisions décrit à la section précédente intervient donc dans les pires conditions. En effet, chaque machine va détecter une collision, attendre un temps aléatoire et essayer de nouveau si le réseau est libre.

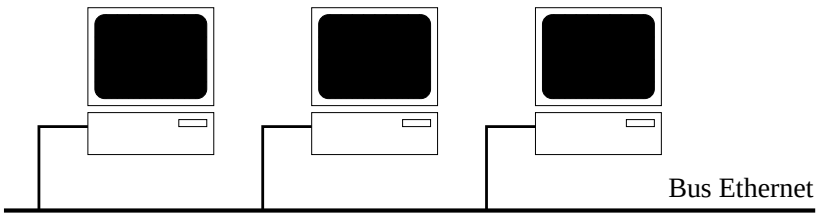


FIG. 5.1 – *Réseau local de stations de travail (bus)*

5.2 Disponibilité des exemples

Difficultés en apprentissage

Nous avons étudié la meilleure façon d'acheminer les exemples vers les processeurs d'une machine parallèle MIMD au chapitre précédent, en apprentissage (section 4.4) et en généralisation (section 4.2). La parallélisation étant toujours modulaire, les conclusions restent valides :

- la généralisation ne pose aucun problème ;
- l'apprentissage peut être ralenti si la machine parallèle, ou le réseau de stations, ont un rapport « calcul / communication » trop élevé.

Les machines parallèles modernes sont souvent pourvues des mêmes processeurs que les stations de travail, or nous venons de voir que le réseau d'interconnexion des processeurs d'une machine parallèle est bien plus performant qu'un réseau local du type d'*Ethernet*. En conséquence, approvisionner les stations au fur et à mesure d'un apprentissage risque de ne pas être aisé. On peut toutefois approcher un approvisionnement en-ligne en envoyant aux stations des paquets d'exemples dès que la charge du réseau diminue.

Parallélisation de l'acquisition

Cependant, apporter une solution simple et efficace au problème d'un travail en-ligne sur un réseau de stations est possible. En effet, contrairement à un nœud d'une machine parallèle MIMD, une station de travail n'est pas un simple processeur accompagné de quelques méga-octets : elle dispose souvent de mémoires de masse (notamment un disque dur), et intègre des ports de communications (série, parallèle, SCSI, *etc.*). De plus, une station peut héberger de nombreuses cartes d'extension (notamment des cartes d'acquisition vidéo ou sonore), commander des relais, interagir avec un utilisateur.
Une station sur le réseau n'est donc pas obligée de recevoir les exemples à apprendre ou à reconnaître, elle peut en être la source. Cette ouverture permet de paralléliser de fait l'acquisition des données alors que le problème est complexe avec une machine parallèle. De plus, la station a la capacité d'utiliser directement le résultat de la classification en intervenant sur le monde extérieur.

Exemple des codes postaux

Reprenons l'exemple de la reconnaissance de codes postaux présenté dans la section 1. La numérisation et la segmentation des codes postaux peuvent être faites *via* une carte dédiée standard, de façon distribuée (figure 5.2).

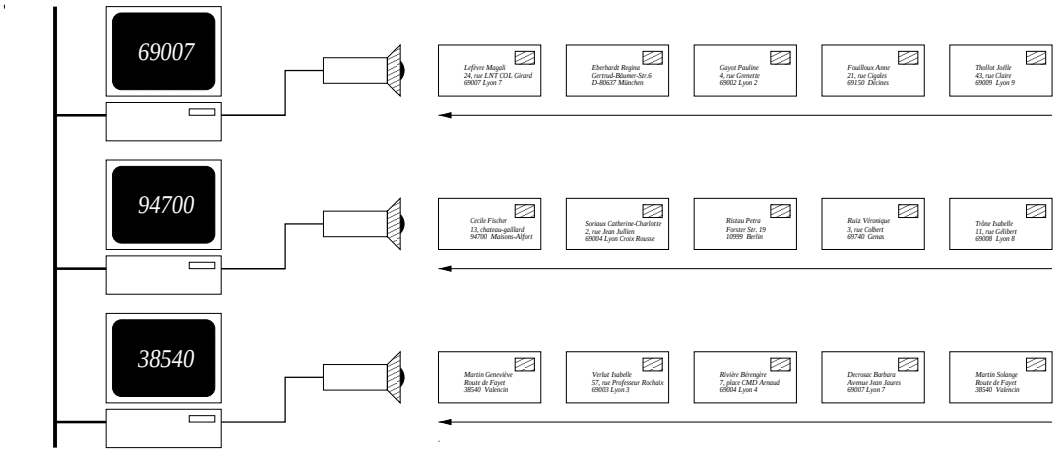


FIG. 5.2 – *Parallélisation de l'acquisition et libération du réseau local*

Exemple (suite)

L'utilisation d'un nombre suffisant d'ordinateurs permet d'atteindre la vitesse nécessaire, tout en utilisant du matériel standard peu onéreux. De plus, même le meilleur matériel ayant une limite, la solution distribuée a des capacités d'évolution (de « sca-

labilité ») supérieures. Enfin, la station a la possibilité d'imprimer directement un « code barre » sur l'enveloppe, ou d'éjecter une lettre dont le code postal est jugée illisible. Pour le même travail, une machine parallèle doit recourir à une caméra et à un système d'acquisition très performant, le cas échéant développés spécialement. Les exemples doivent être acheminés vers les processeurs comme nous l'avons déjà vu. Enfin, pour agir sur le monde extérieur, la machine parallèle doit à nouveau solliciter son réseau d'interconnexion et recourir à un matériel spécialisé.

Remarque

Notons que, l'algorithme d'apprentissage modulaire tire les mêmes bénéfices de la parallélisation de l'acquisition. De plus, l'algorithme spécialisé peut facilement être mis en œuvre en présentant à chaque station les formes d'une même classe.

5.3 Algorithme d'apprentissage parallèle asynchrone

Le goulot de la multidiffusion

Afin d'éviter le goulot d'étranglement de la multidiffusion, l'algorithme d'apprentissage sur réseau de stations de travail doit être asynchrone. Le but n'est plus d'instaurer l'ordre nécessaire à une communication bien organisée, mais de recouvrir le plus possible les calculs et les communications.

Date limite de réception des prototypes

Dans l'algorithme synchrone présenté section 4.8, les prototypes reçus des autres modules arrivent lors de la multidiffusion. Dans la nouvelle version, la multidiffusion devient une simple diffusion. On a donc la possibilité de désolidariser l'envoi et la réception des prototypes. Les prototypes n'étant pas utilisés avant la recherche d'un éventuel prototype gagnant (étape *4* de l'algorithme suivant), on peut recouvrir la phase de réception des prototypes avec la phase de présentation et de prétraitement de l'exemple (étapes *1* et *2* de l'algorithme ci-après). Une nouvelle étape fait donc son apparition entre le prétraitement et la diffusion : les modules traitent tous les prototypes (éventuellement plusieurs par module) en arrivée (étape *3*).

Pour tous les modules :

1 Présenter un exemple.

2 Prétraiter l'exemple.

3 Si des prototypes sont arrivés,
créer ou remplacer des prototypes locaux concernés.

4 Rechercher un éventuel prototype gagnant $P_{gagnant}$,
créer un nouveau prototype $P_{nouveau}$ ou modifier $P_{gagnant}$.

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

5.4 Dangers d'un apprentissage spécialisé asynchrone

Dans un souci de lisibilité, les algorithmes présentés dans ce document restent de haut niveau (au sens algorithmique), en effet une implémentation parallèle reste complexe. Outre une bonne maîtrise du langage de programmation utilisé et des outils du système, la programmation parallèle impose la maîtrise de techniques spécifiques qu'il serait fastidieux de décrire (et de lire) ici. Nous allons faire une petite exception dans la présente section afin de mettre en lumière un danger dû à l'asynchronisme de l'algorithme présenté dans la section précédente.

Une réception non-bloquante

Le danger annoncé vient de l'utilisation de ce que les bibliothèques de communication classiques (dont celles de la Volvox et de PVM) nomment une réception « non-bloquante ».
On s'en doute, ce terme s'oppose à une réception « bloquante ». Le programmeur fait appel à une fonction de réception bloquante dans son algorithme s'il désire que le programme ne passe à l'instruction suivante qu'une fois un message attendu reçu. Le message est stocké dans une zone mémoire précisée par le programmeur. Par exemple, les multidiffusions présentées dans les chapitres précédents utilisent des réceptions bloquantes[1] afin de conserver un algorithme synchrone et ainsi orchestrer la communication au mieux de la topologie utilisée.
En revanche, l'appel d'une fonction de communication non-bloquante passe directement à l'instruction suivante du programme, à moins que le message ne soit déjà arrivé. Si un message est effectivement présent, son contenu est recopié d'une zone mémoire tampon vers la zone choisie par le programmeur.

Priorité au calcul

Dans notre nouvel algorithme, la réception des messages contenant les prototypes (étape *3*, section 5.3) est non-bloquante. Si aucun message n'est en attente à la fin du prétraitement, le module passe directement au traitement de la forme (étape *4*). Cette stratégie donne la priorité au calcul pour atteindre la meilleure accélération possible : on ne laisse pas le processeur désœuvré.

Un environnement imprévisible

Cependant, cette approche présente un risque. En effet, sous un réseau *Ethernet*, dans un environnement multiutilisateur, un message peut facilement prendre du retard. Les modules en avance apprennent avec un déficit en prototypes. Les modules étant spécialisés, le module émetteur dont les messages sont en retard prive l'apprentissage de toute information sur la classe dont il a la charge. Les prototypes créés seront donc de mauvaise qualité et leurs performances en généralisation risquent de s'en ressentir.

1. Notons que le fait de recourir à une communication bloquante ne signifie en aucun cas que la machine soit « bloquée » durant la communication. Seul le processus appelant la fonction est stoppé, et uniquement si le message n'est pas déjà arrivé (ou pas complètement) lorsque la fonction de réception bloquante est appelée.

Gel des modules en avance

Une solution consiste à « geler » les modules en avance, c'est-à-dire ceux ayant appris plus d'exemples qu'au moins un autre module. Les modules en avance restent à l'étape *3* et sont disponibles pour traiter tout message de prototypes qui arriverait.

Tolérance au retard

Cette solution ne va pas dans le sens d'une bonne accélération puisqu'on laisse volontairement des processeurs désœuvrés au moindre retard d'un message ou d'une station.
Aussi, introduisons-nous un compromis entre vitesse et qualité d'apprentissage, en ajoutant un peu de souplesse. Nous allons tolérer une certaine avance, ou de façon duale, autoriser un retard noté λ (en nombre d'exemples). Un faible λ impose une avancée de front des modules, alors qu'un λ plus grand autorise un retard. La mise en œuvre d'une telle tolérance au retard, dans un environnement asynchrone, impose une grande prudence pour ne pas provoquer « d'interblocage ».

Le risque d'interblocage en séquentiel

Précisons la notion d'interblocage (aussi appelé « *deadlock* » ou encore « étreinte fatale ») avant de décrire l'algorithme autorisant un retard. L'avènement des systèmes d'exploitation multitâches donne la possibilité de faire s'exécuter plusieurs programmes simultanément.

- Ces programmes en exécution (ou processus) peuvent simplement cohabiter. Le partage potentiel d'une ressource par plusieurs processus entraîne le risque d'un interblocage, c'est-à-dire une situation dans laquelle plusieurs processus s'attendent mutuellement (par exemple lors d'un accès à une ressource non partageable : un fichier en écriture, un périphérique, *etc.*). Le risque augmente dans un environnement multiutilisateur, car les ressources peuvent être mises à la disposition de tout utilisateur connecté au réseau.

- Les processus d'un système multitâches peuvent également collaborer. On peut citer le cas d'un environnement « client-serveur », ou celui de programmes créant des processus fils (programmation dite *multithread*). Dans ce dernier cas, l'interblocage n'est pas imputable à un défaut du système d'exploitation mais aux programmes écrits par l'utilisateur. Le risque d'interblocage est difficile à gérer car il est difficile de déterminer ou de tester tous les comportements possibles, notamment avec des algorithmes asynchrones.

Programmes parallèles

Un programme parallèle connaît les mêmes risques que les processus d'un environnement multitâche. Cependant, la communication par message imposée par le modèle MIMD à mémoire distribuée implique de nouveaux cas d'interblocage. Une situation classique est un processeur attendant un message d'un autre processeur, lui-même en attente d'un message du premier processeur.

Interblocage du classifieur modulaire

La condition de gel des modules en avance présentée ci-dessous (condition 5.1) ne provoque pas d'interblocage durant l'apprentissage. Cependant, dans le cas où les ensembles d'apprentissage n'ont pas la même taille, le module terminant le premier va bloquer le classifieur parallèle. Il est donc nécessaire d'ajouter une condition sur « l'activité » du module le plus en retard (c'est-à-dire M_j). Pour ne pas générer de communication inutile, un module terminant son processus d'apprentissage prévient les autres modules en envoyant un « signal de fin d'apprentissage » en queue du message contenant les derniers prototypes créés, c'est-à-dire à l'étape *5* de l'algorithme section 5.3. Notre stratégie est guidée par le refus de communiquer inutilement : cette implémentation d'une tolérance au retard utilise exclusivement une information locale (figure 5.3).

$$\forall i \qquad \exists j \quad tel\ que \quad \begin{cases} (s_j \Leftrightarrow r_j) > \lambda \\ M_j \quad n'a\ pas\ annoncé\ sa\ fin \end{cases} \qquad \Rightarrow M_i\ est\ gelé. \qquad (5.1)$$

Avec :

- r_j : nombre de messages reçus par le module M_i depuis le module $M_{j(j\neq i)}$;
- s_j : nombre de messages envoyés par le module M_i vers le module $M_{j(j\neq i)}$.

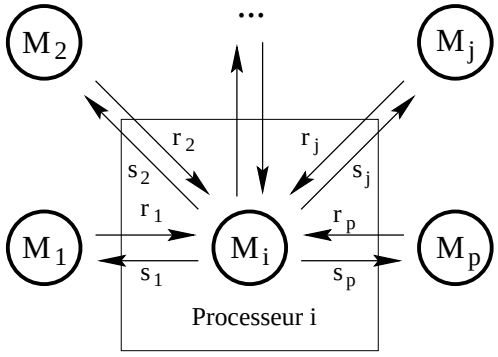


FIG. 5.3 – *Utilisation d'une information locale pour contrôler l'avance des modules*

De l'asynchrone au synchrone ($\lambda = 1$)

Comme annoncé en conclusion du chapitre précédent, le classifieur modulaire asynchrone peut être utiliser avec succès sur une machine parallèle MIMD à mémoire distribuée.

Les processeurs et le réseau de communication de la machine parallèle étant à l'entière disposition de l'application, on peut refuser tout retard. Aussi, il suffit de prendre λ égale à 1 pour obliger tous les modules à avancer de front, on est alors proche du fonctionnement synchrone de la parallélisation modulaire. Cependant, une série de diffusions asynchrones étant moins efficace qu'une multidiffusion synchrone, cette solution n'est viable que si l'on communique peu (grand δ).

5.5 Tolérance aux pannes

Travail autonome du classifieur

Le classifieur peut être amené à travailler longtemps de façon autonome. En pratique, ces situations sont fréquentes car un réseau d'ordinateurs de bureau peut être utilisé comme machine virtuelle, en-dehors des heures ouvrables, pour exécuter tout travail non-interactif (traitement « par lot » ou « *batch processing* »). Même dans le cas d'une application pour laquelle un ensemble de machines est réservé, un travail autonome est utile : étude de l'influence d'un paramètre en le faisant varier ; moyennage d'un résultat sur plusieurs exécutions ; mise en œuvre du classifieur sur un problème lourd ; *etc.*

Le risque de défaillance d'une station

Or, si le traitement dure longtemps, ou si le nombre de machines impliquées est important, la probabilité qu'un des modules devienne indisponible avant la fin du traitement compromet l'intérêt de la parallélisation. En effet la « mise hors service logicielle » d'une machine, c'est-à-dire un « plantage », bloque complètement le fonctionnement du classifieur dès le retard λ consommé.

Préemption et contrôle des accès

L'environnement PVM a principalement été porté vers des systèmes d'exploitation modernes qui ne plantent théoriquement pas.
En effet, les systèmes tels que Unix et MS-Windows NT sont multitâches « préemptifs » : ils ont la possibilité matérielle de reprendre le contrôle du processeur après l'avoir confié à un des processus en exécution. Ainsi, le système apporte la garantie qu'un programme mal écrit ne puisse pas bloquer la machine. Un système non préemptif (par exemple Ms-Windows 3.x et tout MacOS) attend que le processus rende lui-même le processeur, en espérant qu'il soit en état de le faire.
Par ailleurs, les systèmes modernes vérifient tous les accès à la mémoire et au bus. Ce contrôle ralentit les processus mais assure qu'un programme mal écrit ne puisse sortir de la zone mémoire qui lui a été allouée, débordant sur un autre programme (ou sur le système). Ainsi, le système apporte la garantie qu'un processus ne puisse pas en faire planter un autre (le processus fautif est simplement « tué »).

Danger subsistant

Malgré les progrès des systèmes d'exploitation, il reste le risque qu'un traitement intensif lancé par un autre utilisateur ralentisse significativement le classifieur. Dans le cas extrême, les ressources (processeur, mémoire, disque, *etc.*) n'étant pas infinies, le système peut « s'écrouler ». Enfin, il reste le risque qu'un utilisateur éteigne ou réinitialise sa machine, ce comportement est fréquent avec des ordinateurs de bureau, ne serait-ce que pour installer un logiciel.

La fragilité du classifieur face au plantage d'une machine peut être résolu en modifiant l'étape *3* de l'algorithme présenté. Les modules mesurent le temps écoulé depuis le début du gel de leur activité dans le processus d'apprentissage. Si cette

inactivité dépasse une durée T_{max} fixée par l'utilisateur, la machine la plus en retard (M_j) est remplacée « par » (c'est-à-dire « sur l'initiative ») de la machine la plus en avance (M_i). L'environnement PVM ne permettant pas de déplacer un processus d'une machine à une autre, il est nécessaire de gérer la migration. Dans l'algorithme suivant, le « temps système » est une variable du système constamment croissante.

Proposition d'un algorithme tolérant aux pannes pour le classifieur

Pour tout module M_i **:**

1 Présenter un exemple.

2 Prétraiter l'exemple.

3 *Traitement des prototypes envoyés par les autres modules*

Pour tous les prototypes arrivés,
créer ou remplacer des prototypes locaux concernés.

Si $\exists j \;\; tq \;\; \begin{cases} (s_j \Leftrightarrow r_j) > \lambda \\ M_j \; n'a \; pas \; annoncé \; sa \; fin \end{cases}$ **Alors** $t_{début} = temps\ système$

Tant que $t_{début} + T_{max} < temps\ système$
Si un message arrive, aller au début de l'étape 3.

Remplacer la machine hébergeant M_j.

Lancer un nouveau module M_j sur la nouvelle machine.

Envoyer tous les prototypes existants (sur M_i) à M_j.

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

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

Anticipation de la migration

Notons qu'une machine attendue peut être simplement ralentie par une surcharge. Le classifieur étant gelé, le réseau est disponible. On peut donc commencer la migration sans gêner l'apprentissage, et sans attendre que T_{max} soit atteint.
Le transfert est annulé si la situation de la machine en retard se normalise. Mettre un terme à la migration n'induit aucun temps supplémentaire.
En revanche, si la durée de la migration est inférieure à T_{max}, le remplacement de la station n'aura rien coûté (car masqué par le délai qui est toujours accordé).

Remarque

Enfin, si une même machine est souvent impliquée dans les ralentissements, le classifieur peut décider de continuer le remplacement malgré la reprise, en utilisant les prototypes de M_j (non plus M_i) tant qu'ils sont disponibles.

5.6 Apprentissage asynchrone des formes d'ondes

La validation du classifieur parallèle asynchrone est réalisée sur le problème de classification des formes d'ondes déjà présenté section 1.3. Le choix de cette application vient de la bonne connaissance que nous avons du comportement du classifieur sur ce problème.

La machine virtuelle

La configuration matérielle utilisée comprend des stations de travail de type *Sparc 5* de marque SUN sous le système d'exploitation Solaris. Le réseau suit le protocole *Ethernet*, cependant, la configuration n'utilise pas le réseau local très encombré du laboratoire. En effet, chaque station est équipée d'une deuxième carte réseau de type « ATM non-bloquant » (matériel ASX-200 de la société *Fore Systems* [Man95]).

Nombre de stations de travail utilisées

Le nombre maximal de stations SUN utilisées pour l'apprentissage supervisé modulaire est imposé par le problème. En effet, nous utilisons 3 stations pour apprendre les 3 classes du problème des formes d'ondes. Aussi, nous n'avons pas jugé utile d'implémenter l'algorithme de tolérance aux pannes. L'algorithme mis en œuvre pour l'apprentissage modulaire supervisé est la version donnée section 5.3, complétée par la gestion d'un retard (paramètre λ, équation 5.1).
La phase de généralisation ne connaît pas la limite d'un processeur par classe. Les 6 machines disponibles sur le réseau ATM au moment des tests sont utilisées.

Le protocole ATM

Les initiales ATM signifient *Asynchronous Transfer Mode* [Vet95]. Ce « mode de transmission asynchrone » définit une architecture et un protocole capables de couvrir les besoins d'un réseau local (type *Ethernet*), comme d'un réseau global (type *internet*). Un réseau ATM peut faire aussi bien de la commutation de circuits que de paquets [KW95], l'objectif à terme est de regrouper tous les services de communications (donnée, téléphonie, vidéo, *etc.*). Nous n'en dirons pas plus sur ce nouveau standard car l'implémentation actuelle de l'environnement PVM nous impose une utilisation très pauvre d'ATM[2]. En effet, dans l'état actuel des bibliothèques de fonctions de PVM, le protocole ATM n'est pas directement mis en œuvre. Nous utilisons donc l'environnement PVM avec son protocole IP, lui-même implémenté sur le matériel ATM. Aussi, les gains apportés par le nouveau protocole sont limités à :

- un meilleur taux de transfert (voir la table 5.2) ;
- moins de nuisances induites par l'environnement multiutilisateur ; la classification peut toujours être perturbée par des processus s'exécutant sur les stations, mais le trafic du laboratoire n'est pas une gêne pour les communications de l'algorithme.

2. A titre indicatif, l'utilisation d'ATM sans PVM, sous IP donne une bande passante de 7.81 Mo/s et une latence de 690 μs ; avec le protocole « AAL5 » la bande passante atteint 8.77 Mo/s et la latence est de 510 μs.

Protocole	β (en μs)	τ (en $\mu s/o$)
PVM sur IP sur *Ethernet*	1500	0.850
PVM sur IP sur ATM	1890	0.181

TAB. 5.2 – *Latence et taux de transfert sous PVM [PT96, PT97]*

Remarque Notons que si l'utilisation d'ATM améliore le taux de transfert (on passe de 1.17 Mo/s à 5.52 Mo/s), les performances vont être dégradées par une latence supérieure. Il faudra donc essayer de communiquer moins, éventuellement de plus gros messages.

5.7 Accélération sur le problème des formes d'ondes

Comme nous l'avons vu dans la section 2.7, il est délicat de mesurer une accélération significative sous PVM. D'autant que le nombre d'exemples de nos problèmes d'évaluation est faible face à l'inertie de PVM. Aussi, contrairement à ce qui a été fait pour les implémentations sur machines parallèles, nous ne donnons pas de résultats quantitatifs pour l'accélération. La présente section analyse les algorithmes de généralisation et d'apprentissage afin d'estimer les gains en vitesse.

En généralisation Comme dans le cas de l'implémentation sur machines parallèles, aucune communication ne vient perturber la généralisation. On peut donc estimer sans risque que l'accélération est quasi-linéaire. Toutefois, dans le cas d'un travail en ligne, l'acheminement des exemples à reconnaître *via* le réseau local peut ralentir le classifieur et borner l'accélération. Nous avons vu qu'une solution consiste à acquérir les données à partir des stations. Enfin, pour une application temps réel sous PVM du classifieur modulaire, il faut dédier les stations et le réseau à l'application. En effet, même si les délais de réponse demandés par l'application sont grands au regard de la puissance des stations et des caractéristiques du réseau, aucun outil PVM ne permet de garantir un pourcentage du temps d'utilisation des processeurs, ni de la bande passante du réseau.

En apprentissage Prévoir l'accélération durant l'apprentissage est plus délicat. Le paramètre δ est déterminant car la diffusion, située au cœur de la boucle de l'algorithme d'apprentissage, est très gourmande en temps (étape *5* de l'algorithme présenté section 5.3). De plus, λ détermine le temps que les modules vont perdre à s'attendre mutuellement. Une valeur contraignante (un λ faible) ralentit l'apprentissage (étape *3*, section 5.3; conformément à la condition 5.1).

Analyse de l'influence de δ et λ L'importance de δ et λ motive une étude plus poussée de leur rôle dans l'apprentissage parallèle. Les figures 5.4 à 5.11 donnent le nombre de prototypes créés par chacun des trois modules au cours d'un apprentissage supervis é du classifieur mo-

dulaire, en fonction du nombre d'exemples déjà présentés. Les figures sont présentées deux par deux, cote à cote :

- les figures de gauche montrent les prototypes créés localement, par l'algorithme d'apprentissage en lui-même ;
- les figures de droite suivent la totalité des prototypes créés, en incluant les prototypes nouveaux reçus depuis d'autres modules.

Déséquilibre des bases d'exemples

Toutes les courbes ne prennent pas fin à la même position car la base d'apprentissage du problème des formes d'ondes n'est pas parfaitement équilibrée (les effectifs précis sont donnés section 1.3.3, page 24).
Ces différences vont nous permettre de vérifier le critère 5.1 pour éviter les interblocages dus à un module terminant son travail avant les autres.
De plus, ce déséquilibre facilite la lecture des figures. En effet, toutes les courbes (créations locales ou totales) se terminant à la même position sur l'axe des abscisses correspondent à une même classe, et donc à un même module. Par exemple, la courbe la plus haute de la figure 5.4 s'arrête après l'apprentissage d'environ 100 exemples, elle correspond donc à la deuxième courbe en partant du haut sur la figure 5.5.

Apprentissage avec $\delta = 1$ et $\lambda = 1$

Dans une première expérience, résumée par les courbes des figures 5.4 et 5.5 : une communication intervient à la suite de chaque exemple appris ($\delta = 1$) ; les modules sont gardés aussi synchrones que possible ($\lambda = 1$).
On constate que le nombre total de prototypes créés par chaque module (courbes de la figure 5.4), ainsi que le nombre de prototypes créés localement (courbes de la figure 5.5) croissent régulièrement. Ce cas de figure est proche du comportement de la parallélisation modulaire sur machine parallèle, seule la stratégie de diffusion diffère. Cependant, cette solution sollicite beaucoup le réseau ($\delta = 1$) et en attend beaucoup ($\lambda = 1$). Aussi, l'accélération est-elle médiocre : à moins que le réseau de stations ne soit réellement excellent, le classifieur parallèle sera probablement moins rapide que le programme séquentiel.

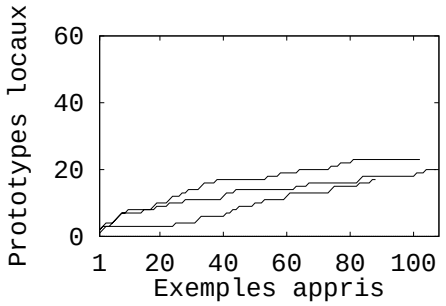


FIG. 5.4 – *Local, $\delta = 1$ et $\lambda = 1$*

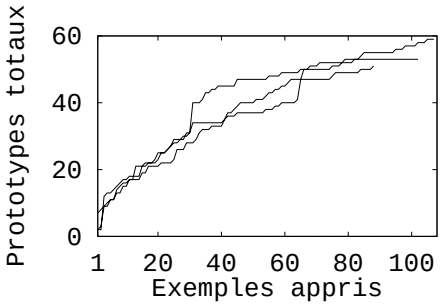


FIG. 5.5 – *Total, $\delta = 1$ et $\lambda = 1$*

Dans les expériences suivantes, nous étudions l'influence de δ et λ en considérant la présente expérience comme référence.

Apprentissage avec $\delta = 20$ et $\lambda = 1$

Les courbes des figures 5.6 et 5.7 résument une nouvelle expérience pour laquelle : on augmente le nombre d'exemples appris entre deux communications ($\delta = 20$) ; les modules sont gardés aussi synchrones que possible ($\lambda = 1$).

La figure des créations locales (5.6) a le même aspect qu'avec un délai unitaire ($\delta = 1$, figure 5.4). On constate toutefois que moins de prototypes sont créés par l'algorithme d'apprentissage et que les courbes sont plus proches les unes des autres. Les trois classes à apprendre étant de même nature, obtenir une évolution semblable du nombre de prototypes créés sur chaque module, c'est-à-dire pour chaque classe, est signe de bonnes conditions d'apprentissage. Cette analyse est confirmée par l'homogénéité des résultats obtenus en séquentiel (section 1.3.3).

La diminution des créations locales et le rapprochement des courbes que nous venons de constater se répercutent bien sûr directement sur le nombre total de créations. De plus, l'accroissement du délai entre les communications ($\delta = 20$) donne aux courbes du nombre total de créations une forme d'escalier (figure 5.7). Le nombre de prototypes d'un module donné observe des paliers séparés d'accroissements rapides après chaque communication, c'est-à-dire tous les 20 exemples. Enfin, on note que le retard autorisé étant minimal ($\lambda = 1$), le nombre total de prototypes évolue de manière synchrone, en cascade.

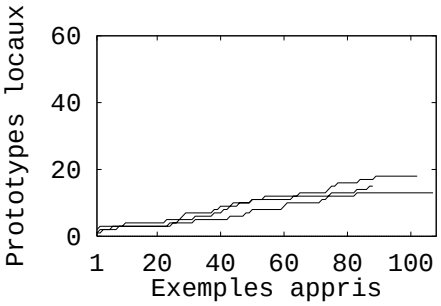


FIG. 5.6 – *Local*, $\delta = 20$ *et* $\lambda = 1$

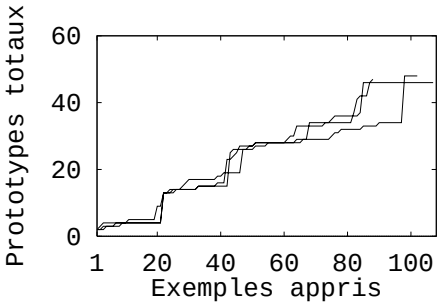


FIG. 5.7 – *Total*, $\delta = 20$ *et* $\lambda = 1$

Apprentissage avec $\delta = 1$ et $\lambda = 20$

Dans une nouvelle expérience (courbes des figures 5.8 et 5.9) on choisit d'accroître la tolérance au retard (λ), sans espacer la diffusion des prototypes créés localement : une communication intervient à la suite de chaque exemple appris ($\delta = 1$) ; les messages de prototypes peuvent avoir un retard important ($\lambda = 20$).

Une fois encore, les courbes des créations locales (figure 5.8) ont le même aspect qu'avec un délai unitaire ($\delta = 1$, figure 5.4). Cependant, le nombre de prototypes créés localement se situe entre ce qui a été observé dans les deux premières expériences. On constate par ailleurs, que l'augmentation du retard maximum autorisé (λ) « éloigne » les courbes de la figure 5.8 les unes des autres. Comme nous l'avons remarqué dans l'expérience précédente, il est souhaitable que ces courbes soient proches. Aussi, il apparaît qu'un retard a une mauvaise influence sur les prototypes créés et ainsi sur la qualité de l'apprentissage.

Outre l'influence directe du nombre de créations locales sur le nombre de créations totales, la figure 5.9 se singularise par le comportement d'une des courbes. En effet, on constate que malgré l'utilisation d'un réseau dédié, un module a pris un retard important qui entraîne une stagnation du nombre de ses prototypes suivie d'un quasi-doublement à l'occasion d'une seule communication. La cause la plus probable de ce retard est la présence d'un processus étranger au classifieur sur une des machines (par exemple un service du système, ou les processus d'un autre utilisateur).

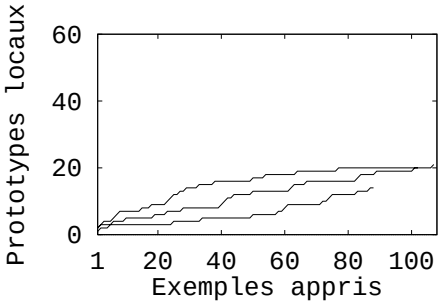


FIG. 5.8 – *Local,* $\delta = 1$ *et* $\lambda = 20$

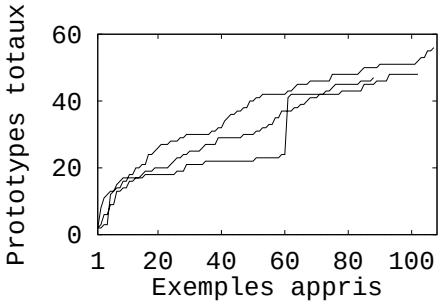


FIG. 5.9 – *Total,* $\delta = 1$ *et* $\lambda = 20$

Apprentissage avec $\delta = 20$ *et* $\lambda = 20$

Une dernière expérience est présentée par les courbes des figures 5.10 et 5.11. Cette situation est la moins lourde pour le réseau, et donc la plus rapide parmi celles que nous avons étudiées : une communication après 20 exemples ($\delta = 20$) ; les messages de prototypes peuvent avoir un retard important ($\lambda = 20$).

Le nombre de prototypes créés localement est très faible (figure 5.10). Le classifieur n'a manifestement plus d'informations de la part des autres modules, en conséquence l'algorithme d'apprentissage ne crée des prototypes que pour les seuls exemples qu'il apprend. L'apprentissage étant spécialisé, très peu de prototypes sont nécessaires. Durant l'expérience dont rend compte la figure 5.10, le phénomène s'est fait particulièrement sentir sur l'une des trois classes qui ne crée plus de prototypes dès le dixième exemple.

La figure 5.11 confirme notre diagnostic, l'envoi tardif des prototypes et la grande tolérance sur leur réception font qu'un des modules ne reçoit jamais de prototypes.

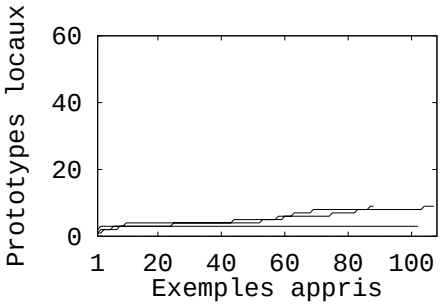


FIG. 5.10 – *Local,* $\delta = 20$ *et* $\lambda = 20$

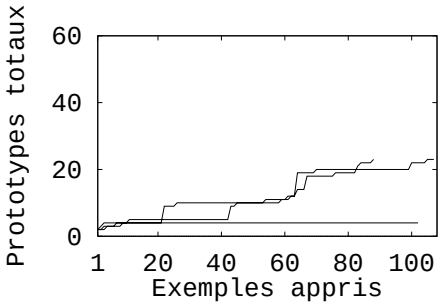


FIG. 5.11 – *Total,* $\delta = 20$ *et* $\lambda = 20$

Erreur en fonction de δ et λ

Nous allons étudier l'influence de δ et λ sur la qualité de l'apprentissage parallèle, en testant en généralisation les prototypes créés en apprentissage. La figure 5.12 présente les taux d'erreur obtenus. On constate que les taux diminuent avec l'augmentation du délai entre les communications. Afin de mieux visualiser le phénomène, la figure 5.13 présente la moyenne des taux d'erreur obtenus pour chaque valeur de λ testée (en fonction de δ). Cette courbe confirme la baisse des taux d'erreur, et montre qu'au-delà d'un délai de 95 exemples entre deux communications, les taux remontent. Enfin, la figure 5.14 présente les taux d'erreur en fonction de λ, pour $\delta = 95$. On constate que pour la tâche confiée au classifieur, le retard autorisé a peu d'influence sur la qualité de l'apprentissage.

Une erreur minimale de 19.35 % est atteinte pour un délai de 95 exemples entre deux communications, et en tolérant un retard de 85 exemples. Cette plage de valeurs élevées pour δ et λ conduit bien sûr à d'excellentes accélérations.

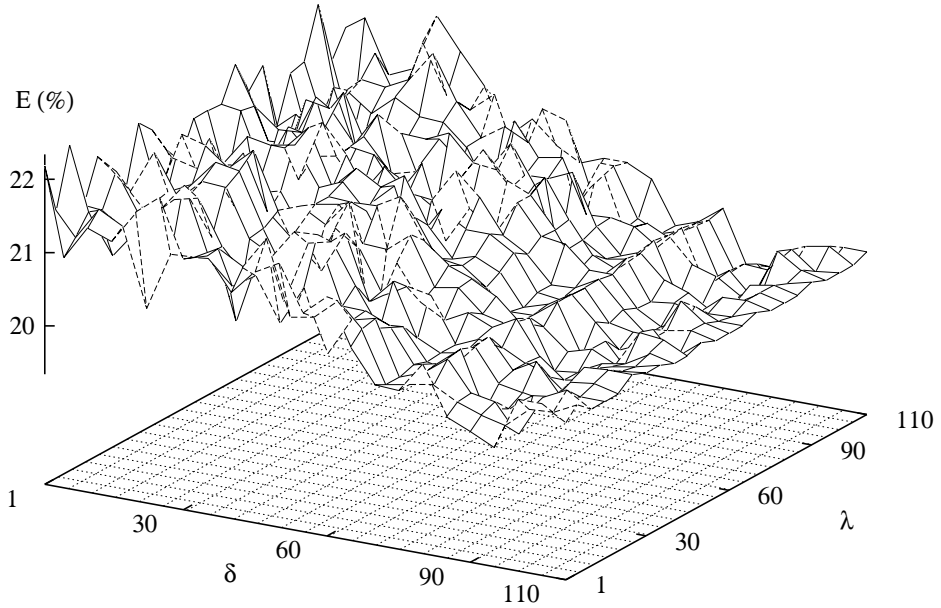


FIG. 5.12 – *Pourcentage d'erreurs en généralisation, en fonction de δ et de λ*

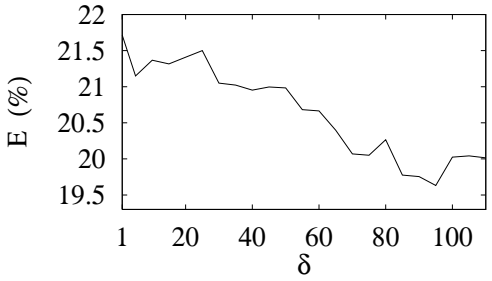


FIG. 5.13 – *Erreur, en fonction de δ*

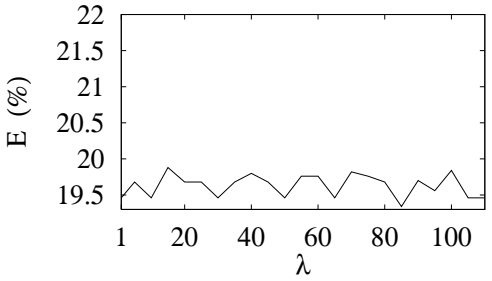


FIG. 5.14 – *Erreur, en fonction de λ*

5.8 Analyse des ensembles d'apprentissage

Les résultats obtenus dans les sections précédentes sont surprenant au regard de ce que nous avons constaté sur le problème de reconnaissance de formes manuscrites. En effet, sur l'application d'OCR, communiquer peu diminue la qualité des prototypes et augmente les taux d'erreur en généralisation. Or, nous venons de constater que pour la reconnaissance des formes d'ondes de Breiman, les communications sont inutiles : la spécialisation des modules suffit à donner de bons taux d'erreur. De plus, on note que les meilleures performances sont obtenues en limitant drastiquement la fréquence des communications.

Bien que l'on puisse se réjouir du comportement du classifieur modulaire spécialisé sur le *benchmark* de Breiman, il est capital de comprendre en quoi cette tâche diffère du problème d'OCR, afin de pouvoir généraliser l'efficacité de la parallélisation modulaire asynchrone à d'autres applications.

Un problème élémentaire

Aussi, cette section traite un problème de classification élémentaire comprenant deux classes (□ et ○). La dimension de l'espace d'entrée est limitée à deux réels afin de permettre une représentation des bases d'exemples dans le plan. Les figures 5.15 à 5.17 schématisent trois situations caractéristiques de l'ensemble d'apprentissage. Les larges cercles présentent la limite de la zone d'influence des prototypes (la position du centre du prototype n'est pas représentée pour ne pas alourdir les schémas).

Situation indifférente aux communications

Pour commencer, nous allons étudier une situation pour laquelle communiquer n'est pas utile. Considérons deux classes dont les exemples sont nettement éloignés sur au moins une des dimensions de l'espace de leurs caractéristiques. Le classifieur crée d'excellents prototypes $P_{\square}$ and $P_{\bigcirc}$ sans qu'aucune communication ne soit nécessaire. Cette situation est reproduite figure 5.15-**A**.

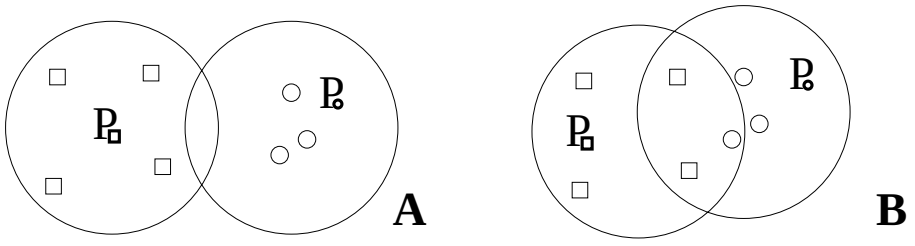


FIG. 5.15 – *Deux situations caractéristiques ne nécessitant pas de communications*

Situation perturée par les communications

Si les classes sont proches et que les zones d'influence des prototypes se recouvrent, des prototypes de classes différentes peuvent se recouvrir. Cette situation est proche du problème de Breiman *et al.* La figure 5.15-**B** schématise une telle situation. Le classifieur résout le problème en déplaçant les prototypes au centre du nuage d'exemples de chaque classe. Le classifieur fait des erreurs en généralisation, mais les performances sont correctes au regard de ce qui est possible. Communiquer ne peut que gêner l'apprentissage, en introduisant de nouveaux prototypes inutiles.

Nous rencontrerons de nouveau ce cas de figure au chapitre 6, ce qui motivera un nouvel algorithme séquentiel (section 6.6).

Situation nécessitant des communications

Cependant, comme nous l'avons vu pour l'application de reconnaissance de chiffres manuscrits, communiquer peut être vital à un bon apprentissage. Considérons les deux ensembles d'apprentissage présentés figure 5.16-**C**. Sur ce schéma, les ronds dessinés en pointillés (◌) sont de nouveaux exemples à apprendre, alors que le rond plein (●) est le **prochain** exemple à apprendre. Portons notre attention sur le module en charge de la classe des ronds (exemples ○).

Apprentissage sans communications

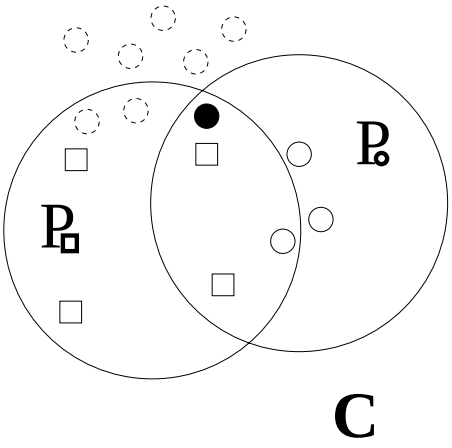
En l'**absence de communication**, l'apprentissage de l'exemple ● rapproche le prototype gagnant $P_{\bigcirc}$ de ●. Ce processus se poursuit avec les autres exemples à apprendre. La figure 5.16-**D** présente la nouvelle situation, alors qu'il ne reste qu'un seul exemple à apprendre. Le prototype $P_{\square}$ n'est pas connu du processeur en charge de la classe des ronds, il ne peut donc pas créer un prototype pour « s'interposer » entre les exemples de ronds dangereusement proches d'un carré et $P_{\square}$. Aussi, les exemples de ronds trop proches de $P_{\square}$ seront mal classifiés lors de la phase de généralisation. Notons que l'on ne fait même pas l'économie d'un prototype. En effet, l'apprentissage d'un exemple de plus (le dernier exemple ◌ de la figure 5.16-**D**) conduit à la création d'un nouveau prototype $P'_{\bigcirc}$ (figure 5.16-**E**) qui ne va « récupérer » que quelques exemples de la classe des ronds.

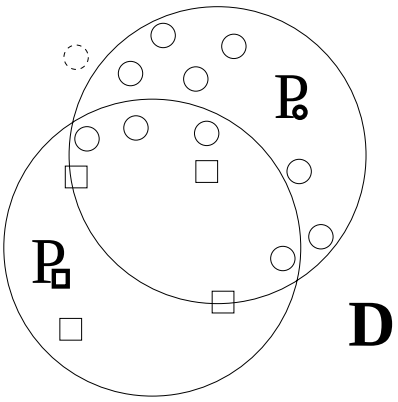
Apprentissage avec communications

À présent, reconsidérons la situation décrite par la figure 5.16-**C** (reprise figure 5.17-**C**) **avec des communications**. Le nouvel exemple ● ne remplit pas les conditions 1.13 (section 1.1.2). En effet, le nouvel exemple est trop proche du prototype $P_{\square}$. Même dans l'éventualité où le prototype $P_{\square}$ n'est pas plus proche de l'exemple que ne l'est le prototype $P_{\bigcirc}$ (le cas est limite sur le schéma), l'exemple ● ne remplit toujours pas les conditions car il se trouve dans la zone de confusion : $d(\bullet, P_{\square}) \Leftrightarrow d(\bullet, P_{\bigcirc}) \leq \Theta_{confusion}$. En conséquence, le classifieur incrémental crée un nouveau prototype $P"_{\bigcirc}$ centré sur l'exemple ● (figure 5.17-**F**). L'apprentissage des exemples restants (les ◌) affine le prototype $P"_{\bigcirc}$ en le déplaçant (figure 5.17-**G**) afin de mieux l'interposer entre $P_{\square}$ et les ronds dangereusement proches des carrés représentés par $P_{\square}$.

Apprentissage avec quelques communications

Le phénomène que nous venons de mettre en lumière est d'autant plus fréquent que l'on communique peu. En effet, le prototype $P_{\square}$ n'est invisible qu'avant la communication qui fait suite à sa création. Cependant, notons que l'utilisation d'un grands nombre de processeurs amplifie le phénomène car un plus grand nombre de prototypes peuvent être créés localement entre deux communications (et donc être invisibles aux autres processeurs). Il découle de cette remarque que même en communiquant à chaque cycle, on ne peut pas retrouver le comportement séquentiel car des prototypes peuvent être créés sur les autres processeurs. Heureusement, nous avons vu que la spécialisation des modules fait qu'il n'est pas nécessaire de suivre le cheminement de l'apprentissage séquentiel pour atteindre de bonnes performances.





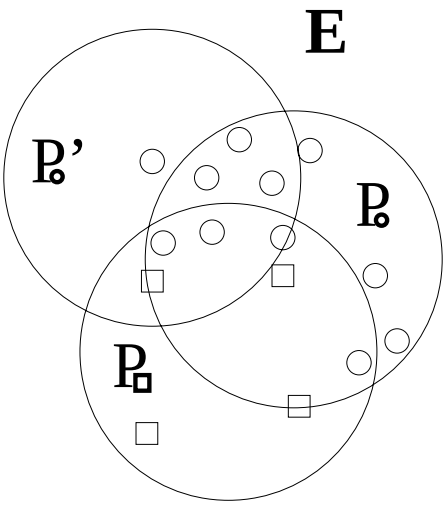
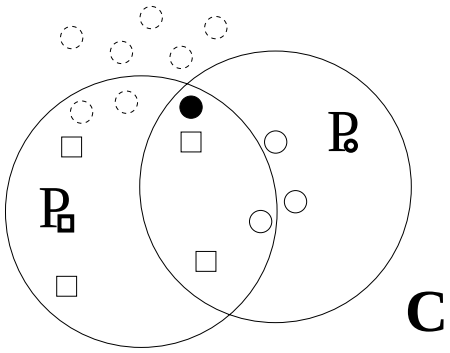
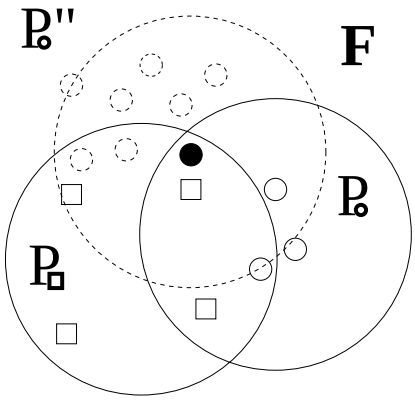


FIG. 5.16 – *Sans communications*





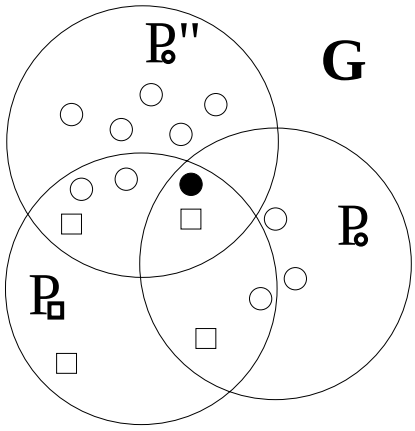


FIG. 5.17 – *Avec communications*

Conclusion

L'approche asynchrone et la résolution des problèmes qu'elle induit permet d'envisager d'excellentes accélérations sur un réseau de stations en choisissant un délai entre les communications (δ) et un retard autorisé pour les messages (λ) adaptés à l'application. Notre but est donc atteint. De plus, une étude des paramètres δ et λ a mis en lumière que trop peu de diffusions des prototypes et surtout une trop faible tolérance sur la régularité de la réception de ces prototypes, conduisent à un apprentissage « cloisonné ».

Par ailleurs, la diminution du nombre de communications due à la spécialisation de l'apprentissage a rendu viable une approche asynchrone sur machine parallèle. En effet, une multidiffusion utilise d'une manière plus efficace le réseau mais son synchronisme complique l'équilibrage de la charge des processeurs. Le comportement du classifieur sur machine parallèle est proche de la version synchrone du chapitre précédent en interdisant le retard des messages ($\lambda = 1$) et en fixant la fréquence des communications en fonction du problème.

Un unique classifieur pour toutes les plate-formes

Nous disposons donc à présent d'un unique classifieur parallèle modulaire capable de fonctionner sur une grande variété de réseaux d'ordinateurs, ainsi que sur toutes les machines parallèles MIMD à mémoire distribuée supportant PVM. Ces résultats sont obtenus aussi bien en généralisation qu'en apprentissage supervisé.

La généralisation dépend de la capacité de l'ensemble des prototypes créés durant l'apprentissage à discriminer les exemples à reconnaître. Notre étude montre que, sur le problème des formes d'ondes, une diminution de la fréquence de diffusion des prototypes créés améliore les taux d'erreur, quelle que soit la contrainte sur le retard en réception. En revanche, nous avons vu que si les communications sont trop rares, le classifieur a du mal à reconnaître correctement les chiffres manuscrits. Une étude du phénomène a permis de caractériser 3 configurations de distributions des exemples à apprendre et a permis de déterminer quels paramètres utiliser pour chacune.
Ce travail apporte une explication au phénomène de cloisonnement mis en lumière dans les chapitres précédents. Enfin, au-delà de la parallélisation, les résultats obtenus sur l'influence de la distribution des exemples à apprendre sur le comportement du classifieur nous sera utile au chapitre 6 dans un tout autre contexte.