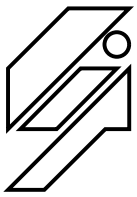
Un problème d'ordonnancement en milieu hétérogène appliqué aux réseaux de neurones

Richard Baron, Xavier-Francois Vigouroux



Laboratoire de l'Informatique du Parallélisme

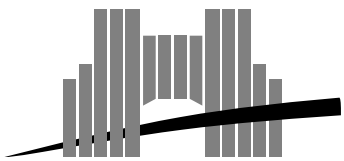
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Un problème d'ordonnancement en milieu hétérogène appliqué aux réseaux de neurones

R. Baron
X.-F. Vigouroux

4 juillet 1996

Research Report N° 96-15



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Un problème d'ordonnancement en milieu hétérogène appliqué aux réseaux de neurones

R. Baron
X.-F. Vigouroux

4 juillet 1996

Abstract

Neural networks are well known as universal approximators. But the performance of neural network depends on several parameters. Finding the best set of parameters has no theoretical answer. Neural network users are forced to try different sets of parameters, and then, choose the best among them. In this paper, we present a parallel solution, based on a master-slave software architecture. Tests are reported on two platforms (workstations network and Cray T3D), using the PVM environment. The load-balancing problem is also addressed. Results are given, for the approximation and the speed-up of the parallel execution.

Keywords: neural networks, mapping, scheduling, load balancing, master/slave,parallelism

Résumé

Les réseaux de neurones constituent des approximateurs universels. Mais leurs performances dépendent de plusieurs paramètres dont aucune méthode théorique ne permet de déterminer les valeurs. Il faut donc tester différentes configurations, ce qui représente un énorme coût en temps, afin d'extraire le choix d'un bon ensemble de paramètres. Nous proposons une solution parallèle s'appuyant sur le modèle maître-esclave. Une implémentation a été réalisée sur différentes machines cibles (réseau de stations et Cray T3D) utilisant le système PVM, et prenant en compte le problème de l'équilibrage de charge. Des résultats sont donnés, concernant la qualité de l'approximation, et l'accélération apportée par le calcul en parallèle.

Mots-clés: réseaux de neurones, placement, ordonnancement, équilibrage de charge, architecture maître-esclave

1 Introduction

Les réseaux de neurones peuvent réaliser n'importe quelle approximation de fonction avec une précision arbitraire [HSW89]. Mais afin de résoudre un problème donné, un réseau de neurones doit subir une phase d'apprentissage. Les performances obtenues en approximation dépendent alors de plusieurs paramètres. Certains définissent l'architecture du réseau de neurones (nombre de cellules cachées, nombre de cellules par couche, ...), d'autres concernent la phase d'apprentissage (taux d'apprentissage, critère d'arrêt, ...).

Trouver les paramètres optimaux pour un problème particulier n'a pas de réponse théorique. Aussi doit-on procéder par une méthode d'essai-erreur, et choisir parmi un ensemble de paramètres ceux donnant les meilleurs résultats. Cette phase de recherche est longue et toutes les évaluations sont indépendantes les unes des autres, d'où l'idée d'une implémentation aisée en parallèle. Dans [PM93], l'auteur présente cette méthode de parallélisation et les moyens d'affiner les critères. Ici, nous ne nous attachons pas à la méthode d'affinage des paramètres mais plutôt à la meilleure utilisation des ressources pour évaluer la qualité des réseaux.

Dans la suite, nous présentons les réseaux de neurones étudiés, leurs buts et propriétés, ainsi que deux heuristiques pour utiliser au mieux des machines de diverse puissance. Ces deux heuristiques ont été comparées sur deux ordinateurs différents et ont donné de bons résultats que nous présentons en fin d'article. Nous commençons par la présentation des réseaux d'ondelettes.

2 Le modèle de réseaux de neurones

2.1 Présentation

Un réseau de neurones artificiel est constitué d'un ensemble de cellules de calcul formant les nœuds d'un graphe de connexion. Lorsque ce graphe ne contient aucun cycle, le réseau est dit de type *feed-forward*. Un réseau multi-couche est un réseau *feed-forward*. Les neurones y sont répartis par couches de telle sorte que deux couches consécutives sont totalement connectées : tous les neurones d'une couche reçoivent en entrée les sorties de tous les neurones de la couche précédente. Un signal se propage de la couche d'entrée vers la couche de sortie à travers les couches cachées. Les cellules prennent en compte les signaux à leurs entrées, effectuent une somme pondérée, à laquelle s'applique une fonction de transfert. Le résultat de ce calcul constitue la sortie de la cellule, qui est transmise à la couche suivante. Le nombre de couches cachées, le nombre de cellules par couche et leurs connexions définissent l'architecture du réseau de neurones. La fonction de transfert servant au calcul de la sortie de chaque cellule est souvent une fonction de type sigmoïde.

Récemment, la théorie de la *décomposition en ondelettes* a été mise en relation avec le domaine des réseaux de neurones [BZ92]. Ceci nous permet de proposer des approximations discrètes pour une fonction $f : \mathbb{R}^n \mapsto \mathbb{R}$ sous la forme :

$$s(x) = \sum_{i=1}^{N} w_i \cdot \Psi(D^{(i)}(x - t_i)) + \theta \tag{1}$$

où Ψ est une fonction *ondelette* en dimension n (c'est à dire une fonction présentant certaines propriétés mathématiques que nous ne détaillerons pas ici). Cette expression peut être écrite comme la sortie d'un réseau de neurones, appelé *réseau d'ondelettes*, à une couche cachée (voir figure 1).

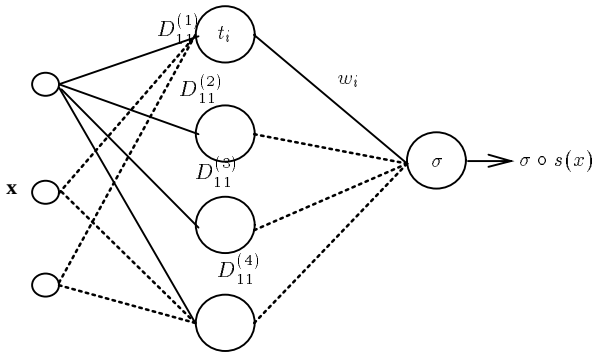


FIG. 1 - *Architecture d'un réseau d'ondelettes.*

Dans la formule (1), t_i et θ peuvent être considérés comme des poids du réseau de neurones, au même titre que w_i et $d_j^{(i)}$, ce qui est utilisé au cours de la phase d'apprentissage.

2.2 Algorithme d'apprentissage

A partir d'un vecteur d'entrée x, le réseau de neurones calcule un signal de sortie $s(x)$. Si f est la fonction que l'on souhaite approcher, on souhaite minimiser une certaine distance entre $s(x)$ et $f(x)$. Un intérêt de l'approche connexionniste est que le réseau de neurones peut subir un apprentissage, afin de donner une valeur de sortie plus proche de la valeur désirée. Un algorithme classique d'apprentissage est la descente de gradient en ligne (ou stochastique) : au cours de la *phase d'apprentissage*, à chaque itération, un exemple est choisi au hasard parmi l'ensemble d'apprentissage $\mathcal{S}$. Chaque élément de $\mathcal{S}$ est un couple entrée/sortie $(x_k, f(x_k))$, avec f la fonction à approcher. Les poids ($w_i, d_j^{(i)}, t_i$ et θ) sont alors modifiés selon une descente de gradient. La fonction de coût à minimiser à chaque étape est

$$c(x_k) = \frac{1}{2}(s(x_k) - f(x_k))^2$$

Après chaque présentation d'un couple $(x_k, f(x_k))$, les poids sont corrigés dans la direction inverse du gradient de la fonction c par rapport aux poids. A l'itération t correspondant à une présentation d'un exemple x_k, chaque poids X (X représentant $w_i, d_j^{(i)}, t_i$ou θ) du réseau est ainsi modifié selon la formule

$$X_{t+1} = X_t - \gamma \frac{\partial c(x_k)}{\partial X} \tag{2}$$

Finalement, une mesure de la performance du réseau après apprentissage est donnée par

$$c_T = \frac{1}{|\mathcal{S}|} \sum_{x_k \in \mathcal{S}} c(x_k) \tag{3}$$

$$= \frac{1}{2|\mathcal{S}|} \sum_{x_k \in \mathcal{S}} (s(x_k) - f(x_k))^2 \quad (4)$$

Le réseau donne une réponse d'autant plus satisfaisante que c_T est petit.

2.3 Espace des paramètres

Plusieurs paramètres interviennent dans la mise au point d'un réseau de neurones afin de résoudre un problème donné :

- Pour les réseaux multicouches à une couche cachée, l'architecture peut être définie par le **nombre de cellules** C de cette couche cachée. Ce nombre de cellules a une influence importante sur les capacités de ce réseau de neurones, c'est-à-dire sur son aptitude à fournir une approximation correcte d'une fonction f.
- La précision de l'approximation dépend également de la phase d'apprentissage, et donc du **nombre de présentations** de couples entrée/sortie, ou encore nombre d'itérations I.
- Enfin, un autre paramètre est le **coefficient d'apprentissage** γ, qui contrôle la descente de gradient dans la formule (2). Dans notre cas, nous considérerons γ constant, identique pour tous les poids X.

Ces trois paramètres influencent de manière déterminante la performance d'un réseau de neurones ainsi défini. Il faut donc tester un certain nombre d'ensembles de paramètres pour obtenir la meilleure approximation possible. Si N_C, N_I, N_G représentent respectivement le nombre de valeurs de cellules, d'itérations et de valeurs de γ qui doivent être testées, alors $N = N_C.N_I.N_G$ ensembles de paramètres doivent être testés. Ces tests peuvent être menés de manière indépendante les uns des autres. Ces opérations s'avèrent coûteuses en temps, et justifient l'utilisation d'une implémentation en parallèle.

3 Parallélisation

3.1 PVM

PVM (Parallel Virtual Machine)[GBD+93] est un environnement logiciel permettant d'attacher un ensemble hétérogène d'ordinateurs fonctionnant sous Unix, reliés en réseau, et de les utiliser comme un ordinateur parallèle. Ainsi, les problèmes calculatoires de grande taille peuvent être résolus en utilisant la somme des différentes ressources, c'est-à-dire, la somme de la mémoire, des CPU, mais aussi des disques locaux. De plus, de part la portabilité du système PVM, un programme peut être compilé sur un grand nombre de machines différentes.

En revanche, l'utilisation du réseau local, partagé par tous les utilisateurs, implique que les communication entre stations sont un point critique de l'application parallèle et doivent être étudiées de près afin de les minimiser.

Un gros avantage de PVM est la possibilité qu'a l'utilisateur de conserver l'environnement auquel il est habitué afin de développer un programme parallèle. Il n'est ainsi pas nécessaire qu'il apprenne de nouveaux outils, différents pour chaque plateforme. De plus, toute la phase de débogage et amélioration de l'algorithme peut se faire sur un petit nombre de stations de travail sans monopoliser une *vraie* machine

parallèle dont l'utilisation est plus critique. Ensuite, lorsque le programme est correct, des tests peuvent être fait sur cette machine cible en étant sûr d'obtenir rapidement des résultats.

Il est évident que le comportement du programme ne sera pas le même sur le réseau de stations de travail et sur une machine parallèle réelle. Il faut donc prévoir les caractéristiques de l'exécution sur les différentes plate-formes. Dans notre cas, nous avons mis en place une stratégie de distribution de travaux qui prend en compte les vitesses relatives des différents processeurs et, par conséquent, s'adapte dynamiquement aux conditions d'exécution.

3.2 Modèle maître-esclave

Comme nous l'avons vu, les tests sur chaque ensemble de paramètres sont indépendants les uns des autres. Plusieurs tests peuvent donc se dérouler simultanément, et les résultats doivent être collectés lorsque les tests se terminent. Une architecture de processus maître-esclave répond donc à notre problème. Un processus maître calcule les paramètres à distribuer, et les envoie à des processus esclaves. Ces paramètres définissent l'architecture d'un réseau de neurones et la phase d'apprentissage qu'il doit subir. Le maître collecte ensuite les résultats obtenus par les esclaves, c'est à dire la performance obtenue par le réseau de neurones après apprentissage (erreur moyenne).

Dans notre cas, le processus esclave va donc recevoir trois paramètres, un définissant le nombre de cellules de la couche cachée, et les deux autres concernant la phase d'apprentissage (nombre d'itérations d'apprentissage, et valeur du paramètre γ). Chaque processus esclave applique alors l'algorithme d'apprentissage au réseau de neurones, et calcule l'erreur moyenne. Il renvoie ce résultat au processus maître, et attend un autre ensemble de paramètres, jusqu'au signal de fin, délivré par le processus maître.

Afin que le processus maître calcule les paramètres à envoyer aux esclaves, l'utilisateur doit spécifier, pour chaque paramètre, l'intervalle qui doit être testé (valeur minimale et maximale du paramètre), ainsi que le nombre de valeurs à tester. Comme ce nombre N peut être grand, un processus par ensemble de paramètres n'est pas la meilleure stratégie. Le nombre de processus esclaves est restreint à P, chaque processus traitant successivement différents ensembles de paramètres. Ces processus peuvent être répartis sur différentes machines ou sur plusieurs processeurs d'une machine parallèle.

4 Première heuristique de placement

Pour un réseau de neurones à une couche cachée, la complexité de notre algorithme d'apprentissage est en $\mathcal{O}(I.C)$, avec I le nombre de présentations d'exemples de la base, et C le nombre de cellules de la couche cachée. Le temps d'exécution de la phase d'apprentissage effectuée par un processus esclave est donc proportionnel à $I.C$ (fig. 3). Cependant, dans un environnement hétérogène, le temps d'éxecution d'un processus peut varier du fait de la charge liée à d'autres processus. Il peut y avoir des fluctuations dans le temps pour un processeur donné, mais également des différences de puissance de calcul d'un processeur à l'autre. De ce fait, il est difficile d'effectuer un placement statique optimal [Dow95]. On choisit donc ici de mettre en œuvre une première heuristique de placement.

Si les paramètres sont distribués dans un ordre quelconque, il risque d'y avoir des problèmes d'équilibrage de la charge (fig. 4 et fig. 5). On constate figure 4 que le retard à la terminaison de la machine *alcools* pénalise l'ensemble de l'éxecution. Dans l'éxecution illustrée par la figure 5, les temps de travail de tous les processeurs sont

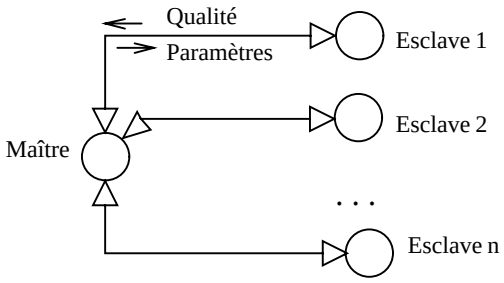


FIG. 2 - *Échange d'information pour la première heuristique.*

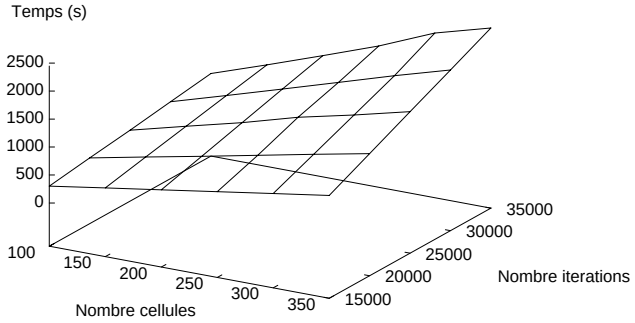


FIG. 3 - *Evolution du temps d'apprentissage.*

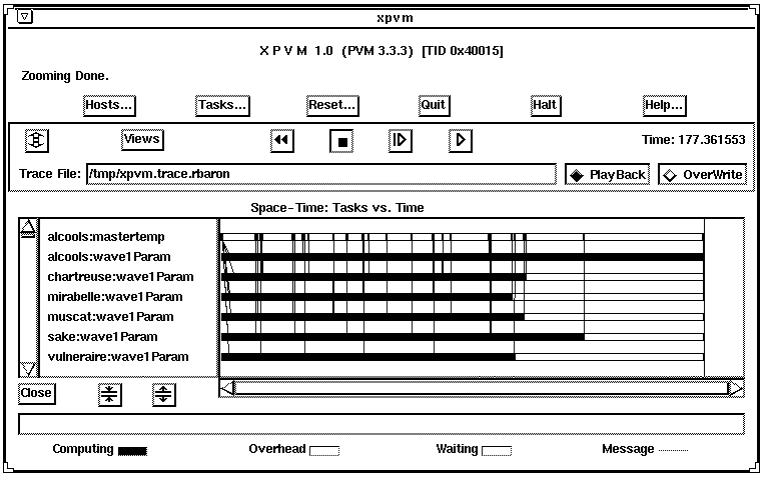


FIG. 4 - *Exécution sans distribution particulière des paramètres.*

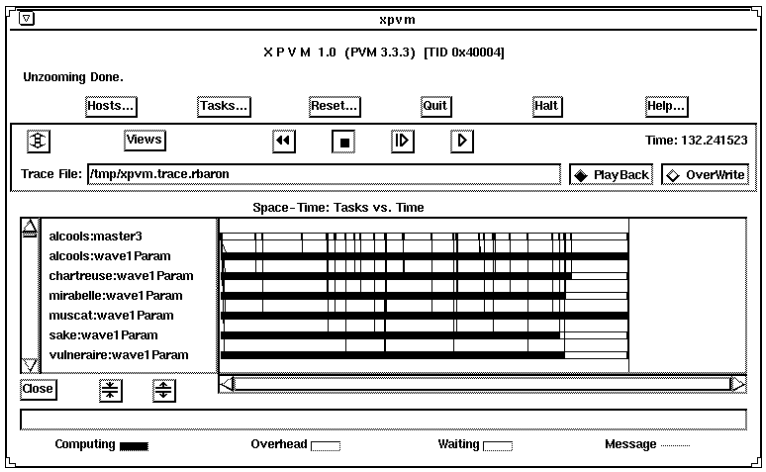


FIG. 5 - *Exécution avec l'heuristique de placement.*

plus homogènes. Ce résultat a été obtenu avec la première heuristique de placement. Suivant cette heuristique, les paramètres sont distribués selon deux listes. L'une des listes explore l'espace des couples (C, I) dans un ordre ascendant, et en suivant des diagonales $C + I = Constante$, en commençant par les valeurs de I et de C les plus petites. Ceci permet de distribuer des travaux dont la complexité croît de manière régulière. En effet, dans notre étude, l'intervalle de variation sur C est plus réduit que celui sur I. La variation sur C étant donc petite, la fonction $T(I, C)$ représentant la complexité des tâches à effectuer peut être approchée par son plan tangent $T(I, C) \simeq \alpha I + \beta C + Cte$. En décrivant des diagonales sur ce plan, on garantit ainsi une distribution des travaux à peu près régulière. L'autre liste distribue les couples en partant des valeurs élevées de I et C, toujours en suivant les diagonales $C + I = Constante$ (fig. 6).

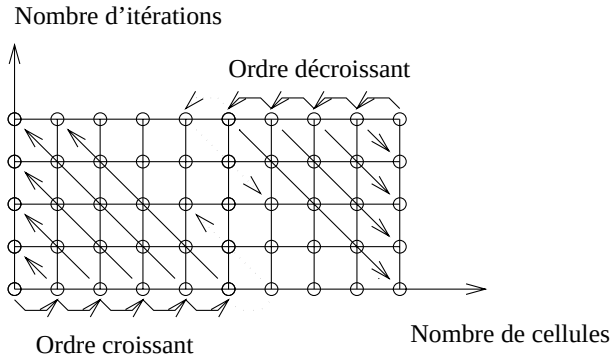


FIG. 6 - *Exploration de l'espace des paramètres.*

Les deux premiers ensembles de paramètres sont choisis l'un dans la première liste l'autre dans la seconde. Ensuite, lorsqu'un esclave envoie au maître un résultat, il spécifie également si ce résultat a été obtenu avec un ensemble de paramètres

venant de la première liste ou de la seconde. Le maître renvoie alors un ensemble de paramètres venant de l'autre liste. Sur un processus esclave vont donc se succéder des ensembles de paramètres venant de la première liste (ayant des valeurs de I et de C plus petite), entraînant des calculs plus courts en temps d'exécution ; et des ensembles de paramètres venant de la seconde liste, impliquant des phases d'apprentissage plus longue.

5 Heuristique en milieu hétérogène

Cette seconde heuristique est spécialement dédiée aux réseaux de stations, néanmoins, comme nous le verrons lors de l'analyse des résultats sur le Cray T3D, elle est utilisable pour n'importe quel type de machines parallèles supportant PVM.

Dans un premier temps, nous avons remarqué la difficulté de lancer notre système de tâches sur le réseau de stations de travail et d'obtenir des performances convenables sans prendre en compte la puissance en calcul des machines utilisées. De plus, le parc de processeurs variant d'une exécution à l'autre (pas le même choix des machines utilisées, variation de l'occupation des différents sites, utilisation variable du réseau de communication, ...), et durant une même exécution, le système de distribution des tâches doit prendre en compte ces paramètres ([Fol92]).

Dans un second temps, comme le réseau de stations est partagé par un ensemble d'utilisateurs, il nous paraissait important de libérer une machine dès la fin de son utilisation par notre programme.

Comme il est écrit dans [Ber95], nous poursuivons « l'amélioration des performances globales du système », et ceci par deux moyens : la prise en compte des **vitesses effectives de chacune des machines** utilisées avec une mise à jour le plus souvent possible, et la **libération des machines inutiles**. Ce dernier point vise à un principe d'équité dans le partage des ressources entre différents procesus

5.1 Définition de la vitesse d'une machine

Nous définissons la vitesse comme étant :

$$\frac{\text{Nombre d'opérations}}{\text{temps}} = \frac{complexité(Param)}{durée(f(Param))}$$

Cette vitesse est donc fonction de la machine utilisée, de sa charge, et de la complexité du calcul que l'on fait. Pour définir cette valeur, il faut connaître la complexité du calcul ainsi que la valeur des paramètres. Nous avons bien ces deux données, nous pouvons donc évaluer la vitesse d'une machine. Ajoutons que la fonction $complexité$ correspond au calcul f et donc le rapport ci-dessus ne dépend que de $Param$. Cette vitesse permet donc d'évaluer le temps que mettra un calcul sur un processeur donné.

Comme nous l'avons dit plus haut, la charge de la machine change pendant le temps d'une exécution, par conséquent la vitesse varie. Ceci fait que lors du retour des informations des esclaves vers le maître, l'esclave communique le temps qu'il a mis à faire le calcul dont il retourne le résultat. La vitesse de cet esclave est alors mise à jour.

5.2 Stratégie du maître

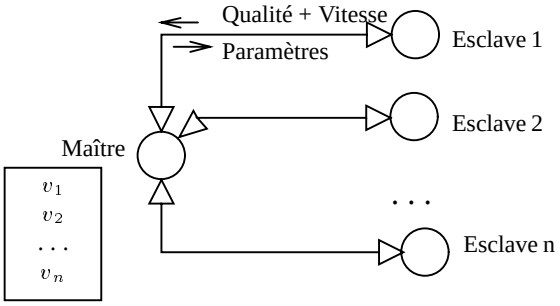


FIG. 7 - *Echange d'information pour la seconde heuristique (avec prise en compte des puissances de calcul).*

Contrairement à la stratégie précédente, le maître possède les vitesses de chacun des processeurs (voir Fig. 7). De plus, il connaît l'ensemble des calculs à effectuer. Il lui est alors facile d'évaluer le temps minimal que mettra l'exécution totale.

> **Remarques :** deux versions de cette heuristique ont été réalisées, l'une avec prédiction du premier travail se terminant afin d'anticiper le choix du meilleur calcul, l'autre utilisant la vitesse du processeur, que celui-ci communique en même temps que les résultats. Le désavantage de la prédiction est que l'on ne connaît pas la dernière valeur de la vitesse du processeur ce qui fait que ce que l'on gagne en recouvrement calcul-communication, se perd en précision.

Considérons qu'un esclave vient de retourner un résultat, le maître connait les travaux restants et les différentes vitesses. Le temps du calcul à donner à la tâche libre est, alors, au maximum égal au temps jusqu'à la fin de l'algorithme :

$$\frac{\sum_{t=calculs\ restants} complexité(t)}{\sum_{p=proc} vitesse(p)}$$

Si aucun calcul n'a une durée d'exécution (sur l'esclave libre) inférieure à cette valeur, il est inutile de conserver cette machine dans le parc car elle ralentira à coup sûr l'exécution globale (ceci si l'on s'intéresse à un principe d'équité par rapport aux autres processus, sans quoi on pourrait lancer tout de même le calcul). Nous souhaitons donc donner le plus grand calcul restant inférieur à la précédente.

Cependant, l'invalidation d'une tâche entraîne une baisse de la puissance de calcul. Par conséquent, le temps du calcul à donner à la tâche libre est au maximum égal au temps jusqu'à la fin de l'algorithme **sans elle** (disons i) :

$$MaxTime0(i) = \frac{\sum_{t=calculs\ restants} complexité(t)}{\sum_{p=proc-\{i\}} vitesse(p)}$$

Dans les précédentes améliorations nous considérons que tout le temps jusqu'à la fin de l'algorithme est dédié aux calculs qu'il reste à faire. Or, il y a aussi des calculs à finir. La borne supérieure des temps d'exécution devient alors :

$$MaxTime(i) = MaxTime0(i) + \tau$$

avec

$$\tau = \frac{\sum_{p=proc} (terminaison(p) - horloge(p))}{Nb\ proc}$$

Finalement, le maître recherchera dans les calculs restant à faire, celui qui est le plus long mais inférieur à $MaxTime(i)$. S'il y en a plusieurs, il peut prendre indifféremment n'importe lequel. Ceci fait que cette méthode est tout à fait déterministe.

> **Remarque sur le démarrage :** nous venons de voir la sélection du nouveau calcul que le maître doit envoyer à un esclave libre. Au démarrage, toutes les machines sont libres. Une petite boucle est éxecutée sur chacunes d'elles, afin d'avoir une idée de leurs puissances respectives. Nous commençons ensuite la ditribution des calculs par les machines plus rapides, en leur donnant les plus gros travaux et ensuite descendons dans les puissances. Cela permet de réduire la perte de temps sur les grosses machines qui doivent être exploitées au maximum.

6 Résultats

6.1 Efficacité des heuristiques

Nous avons ici comparé les résultats en accélération obtenus en utilisant un calculateur parallèle Cray T3D, et en utilisant un réseau de stations de travail hétérogène (LAN). Dans le tableau 1, nous avons mis en concurrence la première heuristique de répartition des paramètres (Heuristique 1) avec celle prenant en compte le temps de terminaison des travaux (Heuristique 2). La machine T3D ne permet qu'un nombre de processeurs puissance de 2, et ces processeurs ne peuvent exécuter qu'un seul processus. Dans le cas d'une architecture maître-esclave, il faut un processus maître, ce qui nous laisse $2^n - 1$ processus esclaves effectuant la tâche d'apprentissage. Les résultats sont donnés en temps d'exécution (en secondes), en accélération $acc = \frac{T_{seq}}{T_{//}}$ et en efficacité $eff. = \frac{acc.}{Nb\ Esclaves}$.

> **Remarque** : cette mesure n'a pas vraiment de signification pour un réseau de stations, l'efficacité devant prendre en compte la vitesse de chacune des machines. On pourrait penser aux pourcentages d'occupation, chacun pondéré par la vitesse du processeur. En effet, il est plus grave de perdre 10 % du temps sur une machine rapide que sur une machine lente. Cette pondération permet de se rapprocher d'une mesure du nombre d'« opérations » perdues. En effet, par définition, la vitesse est le rapport d'une complexité en nombre d'opérations et d'un temps. En retournant cette équation, le produit d'un temps par une vitesse est un nombre d'opérations.

Nous avons une accélération maximale obtenue en divisant le temps d'exécution du plus long des travaux par la somme des temps d'exécution. Dans le meilleur des cas, ce travail sera effectué par un processeur. Nous avons ici $N_C = 3$ et $N_I = 7$, donc 21 travaux à distribuer. Pour 15 esclaves, le nombre de travaux total étant proche du nombre de processeurs, il y a peu de redistribution de travaux à faire (9 processeurs au minimum faisant un travail, et 6 processeurs au maximum en faisant 2). La première heuristique ne prend pas en compte la vitesse des processeurs, un travail coûteux est

donné à un processeur lent qui est le dernier à terminer. Alors que la seconde atteint une plus grande efficacité en donnant ce plus gros travail à la plus rapide des stations.

Cette limitation de l'accélération explique le mauvais résultat en efficacité pour 15 processeurs. Il est lié au découpage de l'espace des paramètres, notamment au fait que l'on ait peu de tâches à redistribuer. En effet, pour les deux heuristiques les gros travaux sont les premiers exécutés et donc les erreurs d'attributions peuvent être corrigées par la suite lors de l'exécution de travaux plus petits. Ceci est bien sûr d'autant plus vrai pour la seconde stratégie qui termine par les plus petits des travaux ; la première, elle, termine par des travaux de taille moyenne. D'une manière générale, le fait de terminer par les travaux les plus petits est l'une des premières caractéristiques que doit avoir une stratégie d'ordonnancement sans quoi elle est dans l'impossibilité de rattraper ces erreurs d'approximation.

Nous l'avons dit plus haut, la machine T3D est constituée de processeurs de même type et un ensemble de processeurs alloués est dédié à un seul utilisateur ; par conséquent, tous les nœuds vont calculer à la même vitesse. Ceci n'est plus vrai dans le cas du réseau de stations, qui rassemble des processeurs de puissances différentes (Sun SPARC Station 5, SPARC classic, Sun4). Ceci explique la dégradation des résultats lorsque le nombre de processeurs augmente : on ajoute alors des machines ayant des processeurs moins performants. Dans le cas de la première heuristique, la distribution des paramètres ne prenant pas en compte cette vitesse, c'est un travail long exécuté sur un processeur peu puissant qui ralentit toute l'exécution. Dans le cas de la seconde heuristique, ce travail étant exécuté par un processeur rapide (SPARC Station 5), on obtient une meilleure accélération.

		Heuristique 1			Heuristique 2		
	nb. esclaves	tps exec.(s)	acc.	eff.	tps exec. (s)	acc.	eff.
Cray T3D	15	841	9.35	0.62	837	9.35	0.62
	7	1313	5.99	0.85	1249	6.27	0.90
	3	2720	2.89	0.96	2637	2.97	0.99
	1	7867			7826		
LAN	15	1510	4.73	0.32	816	7.35	0.49
	7	1959	3.65	0.52	1450	4.14	0.59
	3	2647	2.70	0.96	2727	2.20	0.73
	1	7147			6000		

TAB. 1 - *Accélération pour différentes configurations.*

6.2 Performances en approximation

Dans notre cas, un réseau d'ondelettes est utilisé pour l'approximation d'une fonction de $\mathbb{R}^2$ dans $\mathbb{R}$:

$$f(x) = (x_1^2 - x_2^2) sin(\frac{1}{2} x_1)$$

L'ensemble d'apprentissage contient des valeurs de la fonction f sur le domaine $\mathcal{D} = [-10, 10] \times [-10, 10]$. L'ondelette scalaire de base est la fonction $\psi(x) = -xe^{-\frac{x^2}{2}}$. La fonction ondelette vectorielle est donc donnée par $\psi(x) = x_1 x_2 e^{-\frac{1}{2}(x_1^2 + x_2^2)}$. γ est fixé à la valeur 0.1.

Le résultat de l'approximation peut être illustré par les figures 8 et 9.

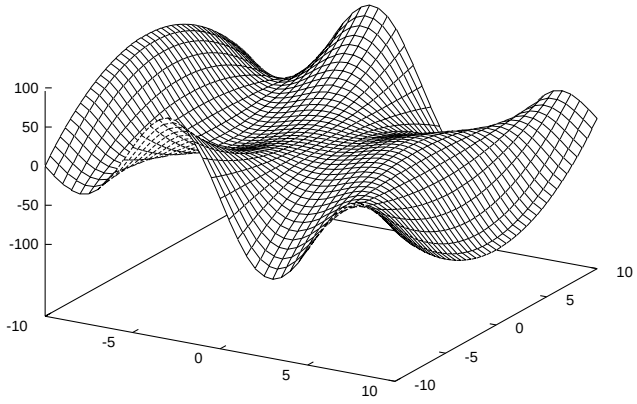


FIG. 8 - *Fonction d'origine.*

La figure 8 donne la valeur de la fonction originale f sur le domaine $\mathcal{D}$. La figure 9 donne le résultat de l'approximation fournie par le réseau de neurones, superposée à la fonction originale en pointillés.

L'espace des paramètres choisi pour nos expérimentations couvre un nombre de cellules variant de 100 à 300 (le pas étant de 100) pour un nombre d'itérations allant de 15000 à 45000 (le pas étant de 5000). la meilleure approximation est ici donnée par un réseau de 300 cellules après 30000 itérations.

Une mesure de la qualité de l'approximation peut être donnée par la distance euclidienne moyenne entre la sortie fournie par le réseau et la valeur de la fonction originale. On obtient dans notre cas des réseaux donnant une erreur moyenne de 0.013 alors que la fonction varie environ entre -100 et 100, ce qui constitue une bonne approximation de la fonction d'origine. Ces résultats sont bien sur du même ordre que ceux obtenus sans l'utilisation de la parallélisation.

7 Conclusion et Perspectives

L'implémentation en parallèle grâce à PVM d'un algorithme de recherche de paramètres optimaux permet de diminuer sensiblement le temps de recherche. Nous avons vu que la mise en œuvre sur un réseau de stations puissantes (Sun SPARC Station 5) peut être compétitive par rapport à l'implémentation sur machine parallèle. Mais dans le cas d'un réseau hétérogène incluant des stations de travail ayant des capacités de calcul plus faible, il faut prendre en compte la vitesse effective des processeurs pour obtenir des résultats similaires. C'est ce que fait la seconde heuristique mise en œuvre.

Nous envisageons désormais de compléter ce travail en permettant de continuer la recherche de paramètres à partir des meilleurs résultats obtenus dans une phase de recherche précédente[PM93]. On pourrait ainsi obtenir une diminution progressive de l'espace de recherche des paramètres, permettant de dimensionner de manière précise un réseau de neurones répondant à une application particulière.

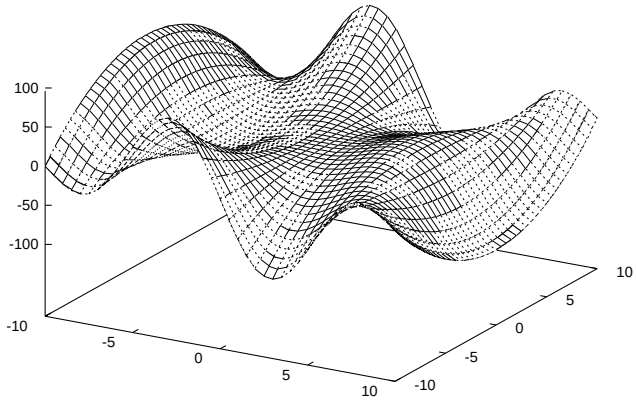


FIG. 9 - *Superposition de la fonction d'origine et de l'approximation par un réseau de neurones.*

D'un point de vue plus général, de nombreuses améliorations peuvent être apportées à l'algorithme de placement des tâches de calcul. On pourra par exemple essayer de renvoyer la vitesse du processeur avant la fin du calcul, afin que le maître en ait une estimation plus précise, et surtout, puisse anticiper l'envoi des paramètres à un processeur. Ceci permettrait d'effectuer un recouvrement calculs-communications. Enfin, on pourra également essayer de conserver l'ensemble des machines, lancer des tâches de calcul sur les plus lentes, en récupérant éventuellement le travail déjà effectué sur ces machines, pour le terminer sur des machines plus rapides. Ce mécanisme s'apparente à de la migration de tâches, et permettrait de conserver la quasi-totalité du parc de machines pendant toute l'éxecution (en sacrifiant donc en partie au principe d'équité).

Remerciements

Nous remercions Thibault Duboux pour les remarques qu'il a formulées concernant ce travail.

Références

[Ber95] Guy Bernard. Les problèmes de placement en environnement réparti : connaissances actuelles et questions ouvertes. In Bertil Folliot, Jacques Chassin de Kergommeaux, and Catherine Roucairol, editors, *Journées de Recherche sur le placement Dynamique et la Répartition de Charges : Application aux systèmes Répartis et Parallèles*, pages 3–4. IBP-MASI, Paris 6 et 7, May 1995.

[BZ92] A. Benveniste and Q. Zhang. Wavelet networks. *IEEE Transactions on Neural Networks*, 3(6):889–898, November 1992.

[Dow95] Salah Dowaji. *Contribution à l'étude des problèmes d'équilibrage de charge dans des environnements distibués*. PhD thesis, Université de Paris VI, July 1995.

[Fol92] Bertil Folliot. *Méthode et outils de partage de charge pour la conception et la mise en oeuvre d'applications dans les systèmes répartis hétérogènes*. PhD thesis, Université de Paris VI, December 1992.

[GBD+93] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennesse, May 1993.

[HSW89] K. Hornik, S. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[PM93] H. Paugam-Moisy. Parallel neural network computing based on network duplicating. In I. Pitas, editor, *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*, chapter 9, pages 305–340. John-Wiley, 1993.