Mise en œuvre de méthodes itératives asynchrones avec communication flexible II. Implémentation sur CRAY T3E, SMP et réseau de stations

Mohamed JARRAYA, Didier EL BAZ, Didier GAZEN

LAAS du CNRS
7 avenue du Colonel Roche
31077 Toulouse Cedex 4, France
E-mail : elbaz@laas.fr

RÉSUMÉ. Cet article constitue la seconde partie d'une publication parue dans le vol.8 n.4 de ce journal. Dans la première partie nous avons proposé une mise en œuvre originale des itérations asynchrones avec communication flexible sur la machine à mémoire distribuée TNode. Nous présentons ici comment nous avons mis en œuvre les algorithmes asynchrones avec communication flexible sur diverses architectures de machines : un supercalculateur à mémoire distribuée CRAY T3E, une machine SMP de type mémoire partagée et un réseau de stations. Ce travail complète aussi l'étude préliminaire de performance qui avait été commencée dans la première partie.

ABSTRACT. This paper is the second part of a paper which was published in the same journal vol.8 n.4. In the first part, we have proposed an original implementation of asynchronous iterations with flexible communication on a distributed memory multiprocessor TNode. Implementations are presented here for several parallel architectures: a distributed memory supercomputer CRAY T3E, a Symetric MultiProcessor and a network of workstations. Computational results are given and analyzed.

MOTS-CLÉS : méthodes parallèles, itérations asynchrones, communication flexible, problèmes de flot, CRAY T3E, SMP, réseau de stations, threads.

KEY WORDS : parallel methods, asynchronous iterations, flexible communication, network flow problems, CRAY T3E, SMP, network of workstations, threads.

1. Itérations asynchrones avec communication flexible

Cet article constitue la deuxième partie d'une publication parue dans le Vol. 8 N. 4 de ce journal (cf. [EGM 96]). Dans la première partie nous avons présenté une application d'un nouveau schéma de calcul parallèle, les itérations asynchrones avec communication flexible, à la résolution d'une classe de problèmes d'optimisation: les problèmes de flot dans les réseaux. Nous avons aussi proposé une mise en œuvre originale sur machine à mémoire distribuée qui est basée sur l'utilisation de requêtes. Une série d'expérimentations numériques sur un multiprocesseur TNode de Telmat a été présentée et analysée. Les résultats numériques ont montré que pour ce type de problème les schémas itératifs asynchrones avec communication flexible proposés sont très rapides et plus performants que tout autre schéma parallèle.

Dans cet article nous complétons l'étude entreprise en mettant en œuvre les schémas de calcul asynchrones avec communication flexible sur diverses architectures parallèles : un supercalculateur à mémoire distribuée CRAY T3E (pour lequel nous présentons deux types de mise en œuvre), un réseau de stations de travail et une machine à mémoire partagée SUN SMP [1].

Les schémas itératifs asynchrones avec communication flexible ont été introduits dans [MES 94] pour la résolution de problèmes de point fixe : $p^* = F(p^*)$ où F est une application de point fixe associée à une M-fonction continue, surjective de R^n dans R^n, ils ont été appliqués dans ce même article à la résolution d'équations aux dérivées partielles et peuvent être modélisés comme suit. Soit T_i, le sous-ensemble infini des itérations auxquelles la i-ème composante du vecteur p est réactualisée tel que $T_i \cap T_j = \emptyset$, pour tout $i, j, j \neq i$. Soit $\{\rho_i(k)\}$ la suite de numéros d'itération retardés satisfaisant pour tout i, $0 \leq \rho_i(k) \leq k - 1$ avec $\rho_i(k) = k - 1$ si $k \in T_i$ et $\lim_{k \to \infty} \rho_i(k) = +\infty$. Une itération asynchrone avec communication flexible associée à $\check{F}$ une sous-application m-continue et au point initial p^0 satisfaisant $p^0 \leq F(p^0)$ est une suite $\{p^k\}$ de vecteurs de R^n telle que pour tout i :

$$\begin{cases} p_i^k = \check{F}_i(\tilde{p}^k), & \text{si } k \in T_i, \\ p_i^k = p_i^{k-1}, & \text{si } k \notin T_i, \end{cases}$$

où $\tilde{p}^k$ est implicitement défini par $\tilde{p}^k \in \langle \max\{p^{\rho(k)}, \tilde{p}^q\}, p^{k-1} \rangle$, les sous-applications de R^n dans R^n m-continues $\check{F}$ définies en détail dans [EGM 96] sont des applications de type relaxation approchée monotones minorées par une application continue pour l'ordre. La notation $\langle , \rangle$ représente un segment d'ordre dans R^n, le vecteur $p^{\rho(k)}$ a pour composantes $p_i^{\rho_i(k)}$ et $q = \max\{l \in T_i | l < k\}$ (on pose $q = 0$ et $\tilde{p}^0 = p^0$, si $\{l \in T_i | l < k\} = \emptyset$).

Dans le modèle présenté ci-dessus, la valeur de $\tilde{p}_i^k$ utilisée lors de la réactualisation de p_i^k est incluse dans un segment d'ordre, elle n'est donc pas nécessairement issue d'une nouvelle réactualisation de p_i et peut correspondre à la valeur courante du prix p_i produite par quelques pas du processus de relaxation approchée construisant la

1. Symetric Multi-Processor

sous-application $\check{F}$. Ainsi, $\tilde{p}_i^k$ peut provenir d'une réactualisation ou d'une réactualisation partielle. La convergence de ces méthodes a été montrée dans [MES 94]. Dans le cas de problème de flot, la convergence à été étudiée dans [ESM 96] dans un contexte plus général que celui des M-fonctions ; une première mise en œuvre sur machine à mémoire distribuée étant présentée dans ce même article.

2. Mise en œuvre

2.1. *Sur SMP*

La machine cible est une SUN Sparcstation 20 quadriprocesseurs. Les mises en œuvre reposent toutes sur l'utilisation de processus multithreadés qui permettent d'exploiter efficacement le parallélisme de l'architecture. Rappelons qu'un processus multithreadé est composé de plusieurs flots d'instruction, ou threads, exécutant simultanément des portions de code sur différents processeurs. Les threads sont des sortes de processus légers qui consomment peu de ressource (ils possèdent leurs propres registres, compteur ordinal et petite pile d'exécution), ils peuvent donc coexister ou être crées en grand nombre. En supplément de la souplesse offerte à l'utilisateur, l'emploi de bibliothèque de threads accroît l'efficacité et présente l'avantage de la portabilité grâce à des standards comme POSIX[2] disponible sous Solaris 2.

De manière générale, nous avons adopté un modèle de programmation de type SPMD où l'on crée autant de threads qu'il y a de processeurs. Chaque thread met donc en œuvre la même fonction sur des données et des processeurs différents. Les échanges de données se font par accès direct à la mémoire partagée.

2.1.1. *Mise en œuvre parallèle synchrone*

Nous rappelons que dans le cas d'une mise en œuvre synchrone, la réactualisation d'une composante à l'itération k+1 ne peut commencer que si la réactualisation des composantes à l'itération k est terminée. Les données sont structurées de la manière suivante : on définit deux copies des valeurs des composantes du vecteur itéré qui sont accessibles par tous les threads. La première copie contient l'ancienne valeur des composantes du vecteur itéré et l'on n'y accède qu'en lecture afin de construire le nouvel itéré. Dans la seconde copie, accessible en écriture, les threads rangent le résultat de la phase de réactualisation. Cette structure est introduite afin d'assurer un fonctionnement synchrone et d'éviter les problèmes d'accès concurent à la mémoire. La synchronisation des threads est réalisée au moyen d'une barrière de synchronisation développée en utilisant les deux primitives de base offertes par les threads POSIX qui sont: le verrou d'exclusion mutuelle et la variable de condition. La barrière de synchronisation utilise un compteur d'appel ; initialement ce compteur est égal à zéro. Chaque appel à la barrière provient d'un thread différent. L'accès au compteur est protégé par le verrou d'exclusion mutuelle et le blocage des threads est réalisé par attente

2. Portable Operating System Interface eXchange

sur variable de condition. La condition est satisfaite lorsque le dernier thread fait appel à la barrière. Le dernier thread qui n'est donc pas bloqué lance alors une fonction qui permute le nom des copies anciennes et nouvelles du vecteur itéré. Il en résulte que l'ancienne copie contient alors le résultat de la dernière réactualisation. Puis le dernier thread débloque les autres threads en signalant la variable condition, ce signal a pour effet d'annuler l'attente des threads sur la variable de condition.

2.1.2. *Mise en œuvre asynchrone avec communication flexible*

Les algorithmes parallèles avec communication flexible sont les seuls schémas de calcul asynchrones a avoir été mis en œuvre sur SMP étant donné qu'ils s'adaptent particulièrement bien et tout à fait naturellement à ce type d'architecture. Dans ce cas, l'introduction de variables locales nécessaire à la mise en œuvre d'algorithmes itératifs asynchrones standards serait en effet artificielle.

Une seule copie du vecteur itéré est suffisante pour mettre en œuvre les itérations asynchrones avec communication flexible. Cette copie est accessible en lecture et en écriture par tous les threads. La bibliothèque de threads POSIX n'est alors utilisée que pour la création des différents threads qui peuvent être ainsi affectés aux différents processeurs. Notons cependant qu'il est difficile de savoir si un thread donné a été ordonnancé sur le même processeur tout au long d'une application.

2.2. *Sur CRAY T3E*

2.2.1. *Mise en œuvre parallèle synchrone*

Chaque processeur réactualise un sous-ensemble de composantes, envoie aux autres processeurs les valeurs des composantes nécessaires à leurs calculs à l'itération suivante, et se met en attente des valeurs des composantes qui sont réactualisées lors de la même itération par les autres processeurs et qui seront utilisées au cours de ses calculs à l'itération suivante. L'envoi et la reception des données se font respectivement à l'aide des fonctions MPI_Isend() et MPI_Recv() de la bibliothèque MPI[3]. Les fonctions MPI_Isend() et MPI_Recv() établissent un protocole de communication entre le processeur source et le processeur destination. La fonction non bloquante MPI_Isend() permet un retour immédiat et évite l'interbloquage entre processeurs. La synchronisation est garantie par la fonction bloquante MPI_Recv().

2.2.2. *Mise en œuvre parallèle asynchrone*

Dans le cas d'une mise en œuvre asynchrone, chaque processeur itère à son propre rythme sans attente de réception de données provenant des processeurs voisins. La politique suivie pour échanger les données entre processeurs consiste à diffuser après chaque réactualisation les nouvelles valeurs des composantes à tous les processeurs qui doivent les utiliser lors de leurs calculs. Cette politique utilise la fonction put()

3. Message Passing Interface

de la bibliothèque SHMEM de CRAY. Plus précisément, cette fonction effectue une copie de la mémoire du processeur source vers la mémoire du processeur cible.

2.2.3. *Mise en œuvre asynchrone avec communication flexible*

Nous proposons deux manières de mettre en œuvre les communications flexibles : soit en envoyant aux autres processeurs un itéré partiel après un certain nombre de pas de calcul en utilisant la fonction put() (dans ce cas nous noterons par S le sous-ensemble des numéros de pas intermédiaires auxquels les itérés partiels sont communiqués) ; soit en prenant, s'il y a besoin, la valeur courante des composantes réactualisées par les autres processeurs en utilisant la fonction get() ; cette valeur n'étant pas forcément labellée par un numero d'itération. Les algorithmes suivants correspondent aux deux manières de réactualiser une composante p_i à l'itération k, NProc étant le nombre de processeurs cibles et P_j le numéro du processeur considéré.

```
q ← 0
p_i^q ← p_i^{k−1}
TANTQUE résidu > ε'
  q ← q + 1
  calcul de p_i^q(p̃^k)
  SI q ∈ S ALORS
    FAIRE j = 1 POUR NProc
      put(p_i^q,P_j)
    FINFAIRE
  FINSI
FINTANTQUE
p_i^k ← p_i^q
FAIRE j = 1 POUR NProc
  put(p_i^k,P_j)
FINFAIRE
```

mise en œuvre avec put

```
q ← 0
p_i^q ← p_i^{k−1}
FAIRE j = 1 POUR NProc
  get(P_j)
FINFAIRE
TANTQUE résidu > ε'
  q ← q + 1
  calcul de p_i^q(p̃^k)
FINTANTQUE
p_i^k ← p_i^q
```

mise en œuvre avec get

2.3. *Sur réseau de stations*

Le réseau de stations est vu comme une machine parallèle virtuelle où chaque instance du programme, nommée tâche de l'application, est exécutée sur une station de travail suivant le modèle SPMD. La communication entre les tâches de l'application est gérée par LANDA[4] (cf. [LAN 96]). LANDA ne supporte pas de primitives d'échange de données entre machines permettant la copie de mémoire à mémoire. Cet échange se fait à l'aide d'un mécanisme de boite aux lettres géré par un serveur de message qui crée au moment de l'envoi de messages, une boite aux lettres au niveau

4. LANDA (Local Area Network for Distributed Applications) est un système multi-serveur développé au LAAS qui permet à des applications réparties de communiquer tout en surveillant le bon déroulement de celles-ci.

de la machine cible pour stocker le message soit avant le message qui l'a précédé (Lsend() : gestion par pile), soit après (Lsend() : gestion par file), soit en le recouvrant (Lsende() : gestion par écrasement et mise à jour). Les stations utilisées sont des Sparcstations ayant une fréquence d'horloge de 110 MHz et une taille mémoire de 32 ou 64 Mo.

2.3.1. *Mise en œuvre synchrone de l'algorithme.*

La mise en œuvre synchrone de l'algorithme sur un réseau de stations gérée par LANDA se fait de la même manière que celle gérée par MPI sur le CRAY T3E. Chaque tâche effectue une réactualisation sur une partie des composantes du vecteur itéré, envoie les composantes aux autres tâches par le biais de la fonction non bloquante Lsend() et effectue une lecture bloquante des composantes envoyées par les autres processeurs ceci à l'aide de la fonction Lreadb() avant de commencer l'itération suivante.

2.3.2. *Mise en œuvre asynchrone de l'algorithme.*

Dans le cas de la mise en œuvre asynchrone nous utilisons le mécanisme d'envoi avec recouvrement. Cette stratégie permet d'avoir une boite aux lettres contenant un seul prototype de chaque message qui correspond au dernier envoi. Dans le cas d'une convergence monotone, l'utilisation de cette stratégie permet d'avoir toujours la valeur la plus proche de la solution et offre une plus grande souplesse au niveau de la mise en œuvre. La composante réactualisée est alors envoyée vers les autres machines par le biais de Lsende() qui écrase la valeur précédente de la même composante. Chaque machine lit le contenu de sa boite aux lettres avant de faire une nouvelle réactualisation en appelant la fonction de lecture non bloquante Lread().

2.3.3. *Mise en œuvre asynchrone avec communication flexible*

Dans le cas de la mise en œuvre asynchrone avec communication flexible, l'envoi et la réception des itérés partiels se font d'une manière similaire à celle utilisée pour la mise en œuvre asynchrone standard et ceci en appelant les fonctions Lsende() et Lread() de la bibliothèque LANDA. De plus, en fixant une politique de communication flexible, chaque processeur envoie les itérés partiels après avoir fait un certain nombre de pas intermédiaires de calcul.

2.4. *Conclusion*

Dans ce paragraphe nous avons présenté différentes façons de mettre en œuvre les itérations asynchrones avec communication flexible sur des types d'architectures très divers. Nous constatons que la mise en œuvre sur une machine de type mémoire partagée est très naturelle et se fait assez simplement. En ce qui concerne les architectures de type “message passing”, si la mise en œuvre sur des supercalculateurs tels que le CRAY T3E est relativement simplifiée grâce à des outils performants comme la librairie SHMEM de CRAY, il n'en reste pas moins que la mise en œuvre est beaucoup plus complexe; ce qui se manifeste particulièrement dans le cas des réseaux de stations.

3. Résultats expérimentaux

3.1. *Problèmes traités*

Afin de rester dans le même cadre applicatif que dans [EGM 96], les itérations asynchrones avec communication flexible ont été testées pour la résolution d'une classe de problèmes d'optimisation non linéaire de type flot dans les réseaux. Pour d'autres applications et notamment la résolution d'équations aux dérivées partielles, le lecteur pourra se reporter à [MES 94].

Soit $G = (N, A)$, un graphe connexe orienté. N est l'ensemble des sommets, $A \subset N \times N$ est l'ensemble des arcs. Soit $c_{ij} : R \to (-\infty, +\infty]$, la fonction de coût associée à l'arc $(i, j) \in A, c_{ij}$ est une fonction du flux f_{ij} dans l'arc (i, j). Soit b_i l'entrée ou la sortie au sommet $i \in N$ (on a $\sum_{i \in N} b_i = 0$). Le problème de flot à coût minimum consiste à minimiser le coût total sous des contraintes de conservation de flux en chaque sommet du graphe :

$$min \sum_{(i,j) \in A} c_{ij}(f_{ij}), \text{ sous } \sum_{(i,j) \in A} f_{ij} - \sum_{(m,i) \in A} f_{mi} = b_i, \forall i \in N. \qquad [1]$$

Nous supposons que le problème [1] a une solution admissible et nous considérons les hypothèses suivantes sur les fonctions de coût : c_{ij} est strictement convexe, semi-continue inférieurement et sa convexe conjuguée est à valeur réelle. Nous avons considéré plus précisémment le dual du problème [1] qui est continûment différentiable, sans contrainte et se prête bien à une résolution parallèle. Comme dans [EGM 96], nous avons traité des problèmes de distribution avec des fonctions de coût du type : $c_{ij}(f_{ij}) = |f_{ij}|^{\frac{1+b}{b}}$, $b = 1.85$. Les topologies correspondent à des réseaux maillés. La dimension des problèmes traités sur la SUN et le CRAY T3E est de l'ordre d'une centaine de sommets et d'un millier d'arcs ; le degré maximum étant égal à 32. Pour les réseaux de stations, il a été nécessaire d'augmenter la dimension à un millier de sommets, car une granularité trop faible ne permettait pas d'obtenir des résultats significatifs. Les méthodes parallèles de relaxation approchée mises en œuvre sont de type gradient synchrones, asynchrones et asynchrones avec communication flexible, elles sont respectivement notées TGS, TGA et TGAF comme dans [EGM 96]. La précision ϵ' pour tous les algorithmes et notamment ceux présentés au sous-paragraphe 2.2 est égale à 10^{-4}.

3.2. *Résultats numériques*

Les figures 1, 2 et 3 montrent l'efficacité de TGS, TGA et TGAF suivant le nombre de processeurs sur le CRAY T3E, SMP et réseau de stations. Pour TGAF les résul-

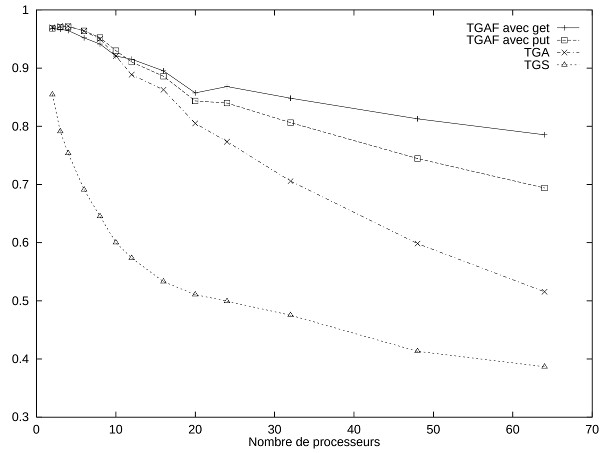

Figure 1. *Efficacité sur le CRAY T3E*

tats numériques correspondent à la meilleure politique d'envoi ou de reception des itérés partiels. Sur le CRAY T3E, par exemple, les itérés partiels sont envoyés tous les 10 pas pour TGAF avec put. Les résultats montrent que l'efficacité des mises en œuvres asynchrones avec communication flexible est plus grande que celle de toute autre mise en œuvre lorsque l'accès aux données est très rapide, c'est notamment le cas des machines de type mémoire partagée (cf. FIG. 2) et des supercalculateurs de type mémoire distribuée pour lesquels les temps de latence sont très faibles et les taux de transfert élevés (cf. FIG. 1). Dans les cas des réseaux de stations pour lesquels le temps de latence est de l'ordre de la milliseconde, TGAF présente un intérêt moindre (cf. FIG. 3). On notera que dans tous les cas, les mises en œuvre asynchrones sont plus performantes que les mises en œuvre synchrones pour le problème traité. Ceci est dû en particulier au déséquilibre indéterministe engendré par la méthode de relaxation approchée qui induit des temps d'attente importants dans le cas synchrone.

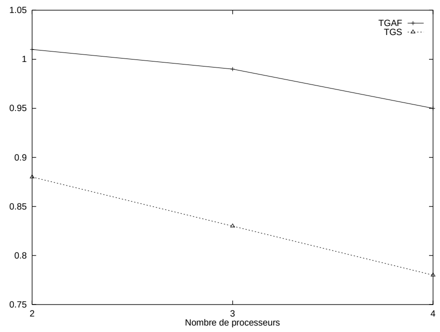

Figure 2. *Efficacité sur SMP*

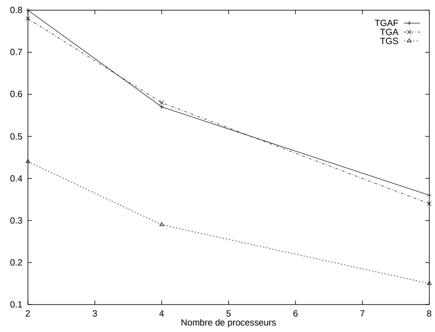

Figure 3. *Efficacité sur réseau*

Les auteurs tiennent à remercier l'IDRIS pour les heures de calcul attribuées sur le CRAY T3E.

Références

[EGM 96] D. El Baz, D. Gazen, J.C. Miellou, P. Spiteri, *Mise en œuvre de méthodes itératives asynchrones avec communication flexible. Application à la résolution d'une classe de problèmes d'optimisation*, Calculateurs Parallèles Vol. 8 N. 4, (1996), 393-410.

[ESM 96] D. El Baz, P. Spiteri, J.C. Miellou, et D. Gazen, *Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems*, Journal of Parallel and Distributed Computing, 38, (1996), 1-15.

[LAN 96] LANDA version 2.5.0, Manuel d'utilisation, Delta Partners S.A. 1996, Licence LAAS-CNRS.

[MES 94] J.C. Miellou, D. El Baz, P. Spiteri, *A new class of asynchronous iterative algorithms with order intervals*, Mathematics Of Computation, Vol. 67, No 221 p. 237-255, 1998.