Les modèles de programmation du NEC SX5 : discussion critique

Victor Alessandrini

1. Introduction

Cette note se propose de présenter une analyse simple et pédagogique des différents modèles de programmation de la configuration NEC SX5, afin de faciliter le choix d'une stratégie de développement des codes de la part des utilisateurs de l'IDRIS. La question fondamentale à laquelle un certain nombre de nos utilisateurs cherche une réponse est : quelle stratégie adopter pour la parallélisation des codes ? Car la disponibilité d'une mémoire partagée de grande taille ouvre la voie à l'utilisation des modèles de programmation autres que le standard MPI (Message Passing Interface) couramment utilisé pour les machines scalaires ou vectorielles à mémoire distribuée (Cray T3E, Fujitsu VPP …). Il est donc intéressant d'essayer de comprendre de manière simple quels sont les avantages et les inconvénients des différents choix. Clairement, la pertinence de chaque modèle de programmation sera déterminée « in fine » par la nature et les caractéristiques de l'application, mais un certain nombre de considérations préalables peuvent être développées, conduisant à des critères susceptibles de guider un choix.

Cette note est organisée de la manière suivante :

- La Section 2 rappelle les principes de base du calcul vectoriel, dans le but de montrer comment le « parallélisme d'instructions » qui le caractérise se traduit par une performance accrue pour les opérations arithmétiques exécutées par le processeur.

- La Section 3 présente les différences entre les Cray C98 et le NEC SX5, afin de montrer pourquoi cette dernière architecture nécessite des vecteurs « longs ». Cette analyse est importante pour comprendre les limitations des modèles de programmation de mémoire partagée (autotasking, OpenMP, threads Posix,…).

- La Section 4 rappelle les caractéristiques de MPI sur la grappe NEC SX5.

- La Section 5 examine les modèles de programmation de mémoire partagée, et elle fait le point sur leurs avantages et inconvénients.

2. Les atouts des vecteurs

Imaginons que l'on souhaite faire l'addition de deux chiffres **x** et **y** en virgule flottante, et affecter le résultat à **z** (**z** = **x**+**y**). En notation scientifique, les chiffres en virgule flottante sont représentés par une mantisse de la forme 0,126543… fois une puissance de 10 (l'exposant). Pour effectuer l'addition, au moins trois actions sont nécessaires :

- déplacer la virgule de **x** ou de **y** afin que les exposants soient les mêmes,

- ajouter les mantisses et affecter le résultat à **z**,

- déplacer la virgule de **z** afin que sa mantisse retrouve sa forme canonique 0,xxxxxx .

Imaginons que le processeur fonctionne à un certain rythme déterminé par sa fréquence d'horloge (cycle), et que chacune des opérations énoncées plus haut s'effectue en un cycle (ce n'est qu'un exemple). L'addition s'effectuerait alors en trois cycles. Si **x**, **y**, **z** étaient des vecteurs et si le processeur exécutait une boucle pour ajouter toutes leurs composantes, on aurait alors un résultat tous les 3 cycles.

Imaginons par la suite qu'on décompose l'unité de calcul en trois unités plus spécialisées appellées A, B, et C dans la Figure 1, qui travaillent comme dans une chaîne de montage, et qui effectuent leur tâche en un cycle :

– A égalise les exposants des flottants cibles et passe le résultat à B,

– B ajoute les mantisses et passe le résultat à C,

– C, enfin, normalise le résultat et l'éjecte dehors.

Cette organisation d'unités de calcul s'appelle dans le jargon un « pipeline ». Si un flot d'instructions vectorielles passe à travers le pipeline, comme indiqué dans la Figure 1, il en résulte que, au bout des trois cycles nécessaires pour remplir le pipeline, le processeur produit un résultat par cycle. Il va donc trois fois plus vite, parce que le « parallélisme d'instructions » - c'est-à-dire, le fait que les instructions du flot soient totalement indépendantes - permet de recouvrir leur exécution dans le temps et de gagner un facteur trois.

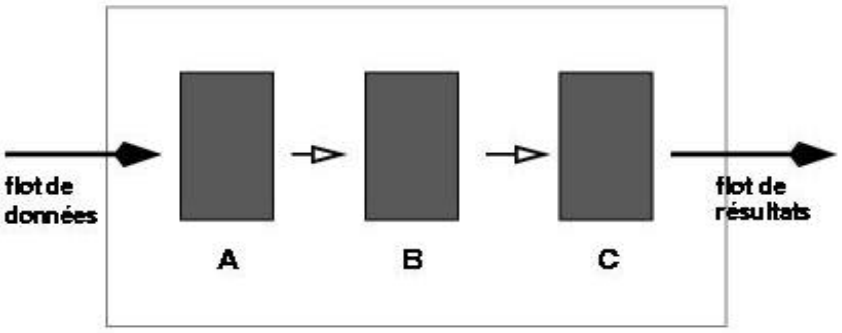

Figure 1 : pipeline

Les processeurs vectoriels tirent en grande partie leur performance de la possibilité d'exécuter de cette manière des flots d'instructions vectorielles suffisamment longs pour amortir le coût entraîné par le démarrage et le remplissage du pipeline. Si les vecteurs sont trop courts, l'activation de cette modalité de traitement est en revanche pénalisante. C'est pourquoi, lorsque l'on optimise un code vectoriel, les boucles très courtes ne sont pas vectorisées.

Ceci n'est qu'un exemple parce que, dans la réalité, les choses sont légèrement différentes : les opérations qui s'effectuent en un cycle ne sont pas celles que nous avons signalées. L'addition demande plus de trois cycles, et les temps de remplissage des pipelines sont plus longs. Mais, les principes de fonctionnement et les idées de base sont correctes.

Signalons enfin que les processeurs scalaires utilisent aussi les pipelines : ceux-ci sont à la base de toutes les architectures modernes des microprocesseurs. Mais il est bien évident que, dans un code scalaire, les flots d'instructions indépendantes sont en général nettement plus courts et que les pipelines se bloquent plus souvent. Car, si une instruction dépend du résultat d'une instruction précédente pour son exécution (branchement conditionnel, dépendance des données, …) alors elle doit attendre à l'entrée du pipeline que celle-ci soit sortie par l'autre bout : le pipeline se vide.

Mais alors, peut-on arguer, puisque les opérations dans les boucles vectorielles sont toutes indépendantes, elles devraient s'exécuter sur un processeur scalaire aussi efficacement que sur un processeur vectoriel ! Cela n'est pas le cas parce qu'il y a d'autres facteurs qui interviennent dans la performance, et en particulier l'efficacité avec laquelle les données sont rapatriées de la mé-

moire, qui n'est pas la même dans les deux cas. C'est le sujet de la section suivante : continuez à lire !

3. Différences entre le Cray C98 et le NEC SX5

La Figure 2 montre l'articulation entre la mémoire et le processeur dans une machine vectorielle. Le point fondamental est que le « tuyau » qui transporte les données entre la mémoire et le processeur est suffisamment performant pour que les données arrivent sur le processeur avec un débit très élevé. L'idée est que les processeurs ne soient jamais affamés, et que les données arrivent à la même cadence qu'à laquelle elles sont utilisées. La vitesse de transfert des données s'appelle la « bande passante » B. Les bandes passantes mémoire processeur dans les machines vectorielles sont nettement plus élevées que celles des machines scalaires. Ceci est indispensable pour que le processeur vectoriel puisse délivrer toute sa performance.

Prenons l'exemple du NEC SX5. La puissance de calcul maximale du processeur est de 8 gigaflops, soit 8 milliards d'opérations par seconde. Chaque opération flottante nécessite au moins une donnée : il faut donc injecter dans le processeur au moins 8 milliards de flottants par seconde. Comme chaque flottant en double précision est codé sur 8 octets, il nous faut donc une bande passante de 64 milliards d'octets par seconde, soit 64 gigaoctets/s, pour que la vitesse à laquelle le processeur reçoit les données de la mémoire soit en harmonie avec sa performance. Telle est précisément la bande passante entre la mémoire et le processeur du NEC SX5. Les bandes passantes sur les microprocesseurs scalaires sont aujourd'hui de quelques gigaoctets/s.

Une bande passante adaptée à une performance de crête est donc la caractéristique de toutes les machines vectorielles. Quelles sont alors les différences essentielles entre le Cray C98 et le NEC SX5 ? Elles se situent à deux niveaux :

- **D'une part, la multiplication des pipelines**. Sur le Cray C98, chaque pipeline associé à une opération vectorielle donnée est doublé. Le vecteur est coupé en deux morceaux et chaque morceau est envoyé à un pipeline différent (un pipeline reçoit les composants pairs, un autre les composants impairs). Sur le NEC SX5, chaque pipeline associé à une opération vectorielle existe en 16 exemplaires, un vecteur est donc découpé en 16 morceaux, chaque morceau étant envoyé à un pipeline différent. Comme chaque pipeline produit en principe en régime stationnaire un résultat par cycle, et le cycle étant le même pour les deux machines (4 nanosecondes), le rapport des pipelines explique le rapport des performances de crête : 1 Gigaflop contre 8 Gigaflops.

- **D'autre part, la latence de la mémoire**. On appellera «latence » L le temps minimum demandé pour un accès mémoire. La mémoire du C98 est une mémoire rapide, dont la latence est de l'ordre de 15 nanosecondes (4 cycles du processeur tout de même)… mais la taille est de 4 Gigaoctets. La mémoire du NEC SX5 est une mémoire plus lente, dont la latence est de l'ordre de 60 nanosecondes (15 cycles du processeur !) … mais la taille de la mémoire sur un nœud va jusqu'à 128 Gigaoctets.

Nous verrons par la suite que ces deux facteurs entraînent la nécessité, pour le NEC SX5, de vecteurs nettement plus longs que pour le Cray C98. Que la prolifération de pipelines demande des vecteurs plus longs, c'est évident, puisque chaque vecteur est cassé en davantage de morceaux, et chaque morceau doit alimenter un pipeline différent avec un flot de données suffisamment long pour amortir le coût du démarrage . Sur le NEC SX5, les vecteurs doivent être plus longs pour aboutir à la même fraction de la performance de crête que sur le C98. Plus longs, mais de combien ? Cela dépendra bien évidemment de chaque code particulier.

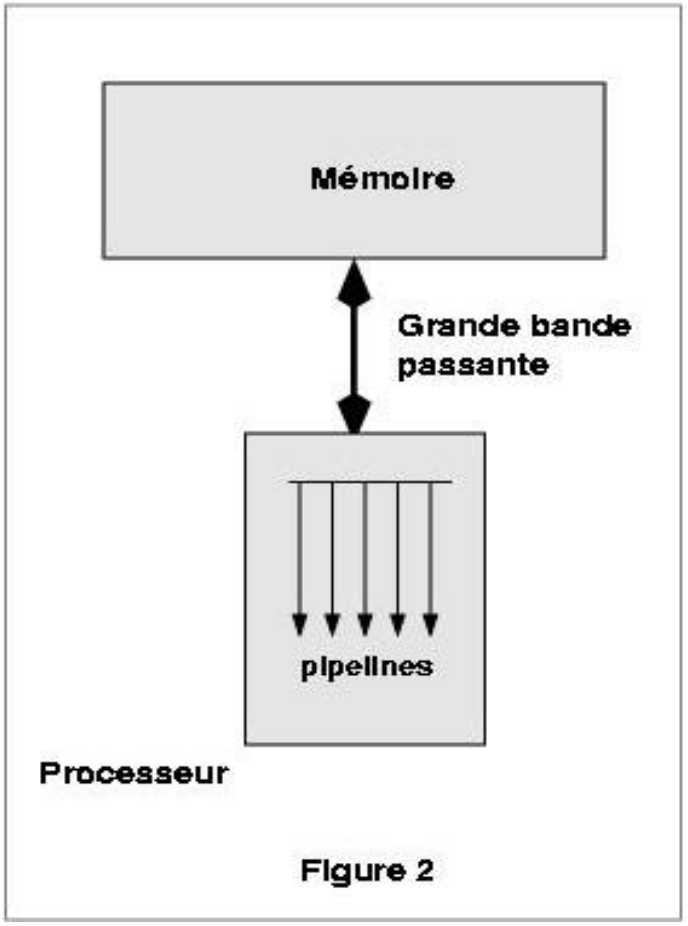

Figure 2

Mais la question de la latence est aussi très pertinente. En effet, si chaque processeur devait attendre 15 cycles pour chaque accès mémoire, les performances seraient nulles. Le charme du calcul vectoriel est que la latence des accès mémoire est affectée à un vecteur et non à un élément particulier. Une fois que le transfert a commencé, il se poursuit très vite parce que la bande passante est grande. Le temps T nécessaire pour transférer un vecteur de N octets de la mémoire au processeur est donné par :

$$T = L + (N / B)$$

où L est la latence et B la bande passante. Si la taille N du vecteur transféré est suffisamment grande, la latence passe inaperçue : les vecteurs longs sont efficaces pour « cacher » la latence de la mémoire.

Remarquez que, dans l'expression précédente, non seulement la latence du SX5 est 4 fois plus grande, mais la taille du vecteur est divisée par une bande passante qui est, elle aussi, environ 10 fois plus grande. Pour que la latence passe inaperçue - c'est-à-dire, que $L<<(N/B)$ - la longueur des vecteurs sur le NEC SX5 doit être sensiblement plus grande que sur le C98.

Conclusion : *des vecteurs longs sont nécessaires pour qu'un processeur SX5 délivre une fraction importante de sa puissance de crête.*

L'exposé que nous avons donné ici est très schématique, destiné simplement à faire comprendre les idées de base. La relation entre la performance des accès mémoire et la performance des codes vectoriels est davantage développée dans le cours « Optimisation SX5 » de l'IDRIS.

4. Codes parallèles : cas général (MPI)

Dans les machines à mémoire distribuée type SPMD (Single Program Multiple Data) comme le T3E ou le VPP, le code et les structures des données sur chaque processeur sont les mêmes : ils ont simplement été répliqués N fois, N étant le nombre de processeurs alloués. Chaque processeur exécute donc un processus Unix distinct (rappelons qu'un processus Unix est une unité d'allocation des ressources : mémoire, fichiers, CPU …). Les données sont, naturellement, accessibles seulement dans le contexte du processus auquel elles appartiennent. Pour accéder aux données appartenant à un autre processus, il faut gérer explicitement un protocole d'envoi de messages, fourni par la bibliothèque MPI.

Ce modèle de programmation est incontournable pour les machines (comme le T3E, VPP) à mémoire distribuée, constituées de nœuds où chaque processeur est physiquement associé à une mémoire qui n'est adressable que par lui. Les échanges de données entre nœuds activent un réseau d'interconnexion. La nécessité de programmer les machines parallèles à mémoire distribuée est donc à la base de la diffusion du standard de programmation parallèle MPI.

La gestion explicite d'une bibliothèque de communication entre processus demande un effort d'apprentissage non négligeable mais parfaitement abordable. Cet effort est souvent amplement justifié par les atouts majeurs qu'offre MPI, à savoir :

- **L'universalité** : MPI permet la programmation parallèle de n'importe quel problème scientifique, et en particulier de ceux dont la parallélisation n'est pas évidente (maillages non-structurés, par exemple).

- **La portabilité** : MPI n'est pas un langage mais une bibliothèque dont les interfaces sont standardisées. Un programme écrit en MPI doit s'exécuter sur n'importe quelle plate-forme.

- **L'extensibilité** : un programme bien écrit en MPI doit pouvoir s 'exécuter sur un nombre arbitraire de processeurs, et aucune barrière matérielle sous-jacente ne limite ce fait.

La configuration NEC SX5 est constituée de trois gros nœuds comportant chacun une grosse mémoire partagée, alors que MPI a été conçu pour les machines à mémoire distribuée. Mais qui peut le plus peut le moins : MPI est très bien implémenté sur cette architecture. Comme le montre la Figure 3, un programme MPI parallèle peut s'exécuter à l'intérieur d'un nœud ou à cheval sur deux nœuds (ou plus). S'il est entièrement contenu dans un nœud, les « communications » sont simulées par la copie directe des données – par le système – entre les espaces d'adressage alloués à des processus différents. Cette modalité de « communication » est bien plus performante que l'activation d'un réseau ! Si le programme parallèle est à cheval sur deux nœuds, par exemple, les communications entre les nœuds se font à travers un réseau à très haute performance – IXS, « Internode Crossbar Network » - dont la bande passante est de 8 Go par seconde bidirectionnelle. C'est de loin la plus grosse bande passante disponible aujourd'hui pour interconnecter les nœuds d'une grappe de supercalculateurs.

Ceci confirme donc la remarque à propos de l'extensibilité faite plus haut. Un programme MPI bénéficie de l'image unique de la grappe SX5, et il peut donc s'exécuter sur un nombre arbitraire de processeurs du système global. Non seulement cela permet de dépasser le nombre maximum de processeurs d'un nœud, mais de surcroît on bénéficie d'une souplesse accrue pour l'exécution du programme de travail. En effet, le système expert qui pilotera Uqbar pourra profiter d'une disponibilité limitée de ressources sur deux nœuds pour lancer un job parallèle important. Cette souplesse dans l'allocation des ressources de la machine fait partie de la panoplie de moyens qui seront mis en œuvre par l'IDRIS pour optimiser son fonctionnement.

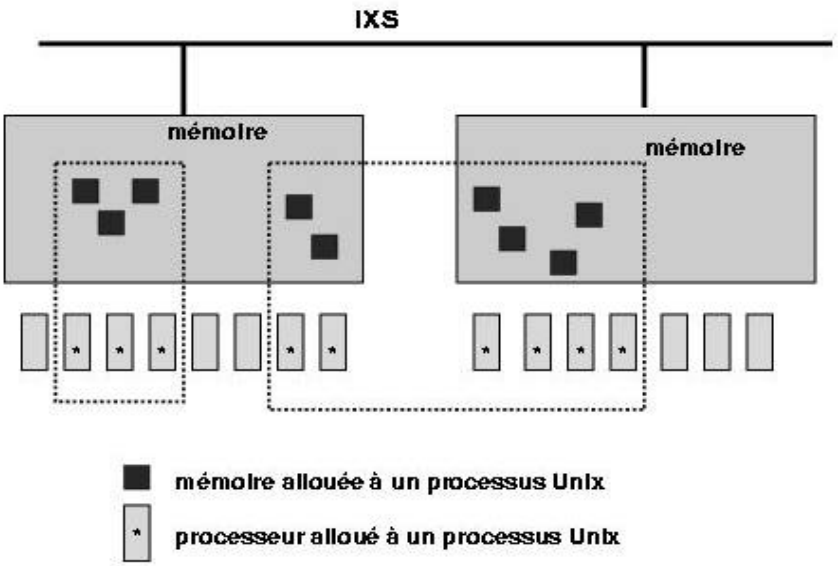

Figure 3

Il va de soi que la parallélisation sur Uqbar doit être une parallélisation à gros grain : chaque processeur doit avoir beaucoup de travail à faire, et disposer de vecteurs longs, pour optimiser son rendement.

5. Codes parallèles : mémoire partagée

La disponibilité des nœuds à mémoire partagée ouvre la voie à d'autres modèles de programmation parallèle (*autotasking, OpenMP, Pthreads*) bien adaptés à ce cas particulier. La caractéristique commune de tous ces modèles de programmation est que tous, sans exception, travaillent avec un seul processus Unix. Il y a donc un seul espace d'adressage, un seul ensemble de fichiers, etc.

Le processus Unix est certes l'unité d'allocation des ressources, mais il n'est pas nécessairement l'unité de « dispatching » : dans un processus Unix peuvent coexister plusieurs flots d'exécution ou « threads » Ces threads constituent des routines dont l'exécution avance en même temps mais, comme ils appartiennent au même processus, ils partagent toutes les ressources (mémoire, fichiers). En particulier, la totalité des données du processus est accessible à tous les threads. Dans le cas d'un système monoprocesseur, les threads s'exécutent à tour de rôle par « time sharing », mais un système multiprocesseurs à mémoire partagée offre la possibilité de les faire exécuter de manière simultanée par des processeurs différents.

Les modèles de programmation des systèmes de mémoire partagée s'appuient sur le mécanisme de base schématisé dans la Figure 4. Au démarrage, le programme (processus Unix) tourne sur un processeur et le processus ne comporte qu'un thread. Au début d'une « section parallèle », plusieurs threads supplémentaires sont créés par le système et affectés à des processeurs différents. A la fin d'une « section parallèle », les threads supplémentaires ainsi créés sont désactivés et le processus redevient monoprocesseur. La création et la désactivation des threads supplémentaires entraînent toujours un certain coût, car le système doit intervenir et allouer des proces-

seurs supplémentaires. Pendant la totalité de leur durée de vie, les threads accèdent à la totalité des ressources (données en mémoire, fichiers) du code initial.

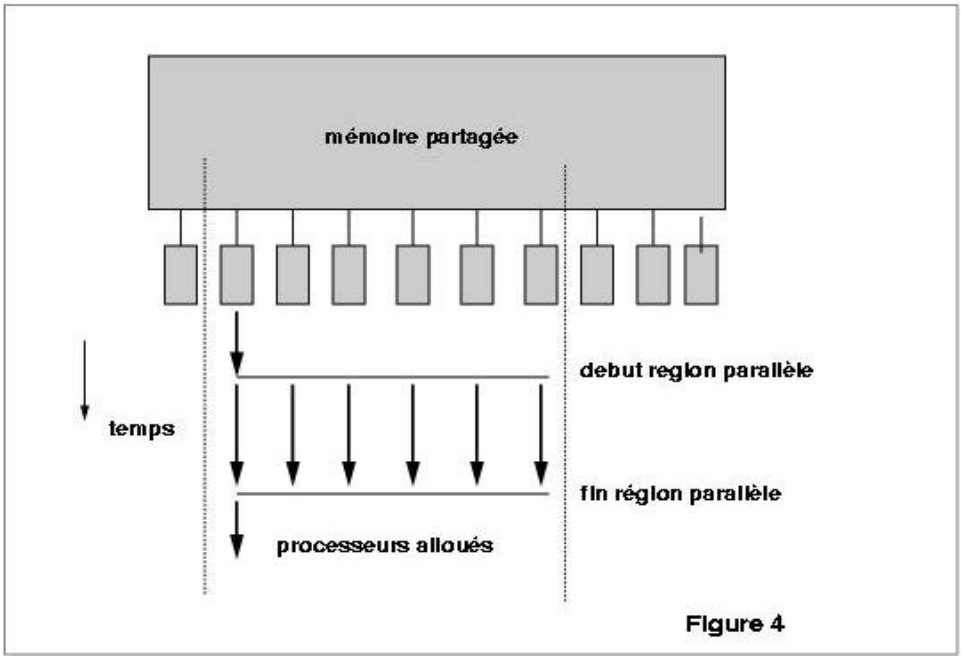

Figure 4

La gestion des threads se fait :

- Par des directives intercalées dans le code, dans les modèles de programmation proches de la « parallélisation automatique » comme l'autotasking ou bien OpenMP.

- Par l'utilisation d'une bibliothèque de fonctions (écrite en C) dans le cas de la programmation par Pthreads (threads conformes au standard Posix). Ceci permet, bien entendu, une programmation plus précise et raffinée que dans le cas de OpenMP, lorsque le code est écrit en C (ou C++).

Une manière évidente d'utiliser ce mécanisme de programmation parallèle serait de l'utiliser pour éclater une longue boucle sur plusieurs processeurs. La région parallèle est alors très courte : elle se limite à la seule boucle, et en principe on devrait obtenir une accélération de l'exécution de la boucle un peu inférieure à N (le nombre des processeurs) en raison du coût d'activation et de désactivation des threads. Cependant, ce type de programmation parallèle fonctionnera en général moins bien sur le SX5 que sur le Cray, en raison du besoin de vecteurs longs pour optimiser les performances monoprocesseur. Eclater une boucle sur 4 processeurs revient à la couper en 8 morceaux sur le Cray (en raison des deux pipelines par processeur) et en 64 morceaux sur le SX5 (en raison des 16 pipelines par processeur). Il faut vraiment que le vecteur d'origine soit long ! Sur le SX5, ce type d'optimisation par traitement parallèle d'une boucle peut entrer en conflit avec l'optimisation monoprocesseur, car il y a déjà énormément de parallélisme dans le processeur lui même.

Le modèle de programmation OpenMP s'appuie sur des directives, mais l'édition des liens doit se faire avec une bibliothèque explicite. OpenMP est assez complet et flexible, et il permet une modalité de programmation où chacun des threads dans une région parallèle exécute beaucoup de calcul et de traitement de l'information. En fait, OpenMP permet de mettre en œuvre des algorithmes de décomposition des domaines, où chaque thread activé par le code principal se comporte

comme un processus MPI et prend en charge la simulation d'une partie du système physique global, avec l'avantage qu'aucun envoi de messages n'est nécessaire, car tous les threads accèdent à la totalité des données. Pour certains problèmes physiques, ce type de programmation parallèle est plus aisée : nous pensons en particulier à des systèmes de particules en interaction (comme la dynamique moléculaire) où la dérive des particules dans le domaine physique de simulation fait que les particules affectées à un domaine donné changent en fonction du temps. Le fait que tous les threads connaissent toutes les particules facilite bien les choses !

Cependant, rien n'est entièrement gratuit dans la vie, et le fait que les données sont partagées par tous les threads introduit des problèmes nouveaux dans le cas où plusieurs threads peuvent accéder (surtout écrire) la même donnée. Dans certains cas, si ces accès se font de manière totalement asynchrone et si l'ordre d'accès des données de chaque thread est aléatoire, le code peut donner des résultats non reproductibles (et faux, par conséquent). Cela oblige donc à introduire des directives de synchronisation à des endroits bien précis (sections critiques).

Pour ce qui concerne OpenMP, une formation sera dispensée à l'IDRIS assez fréquemment dans les mois à venir, car ce modèle de programmation parallèle est pertinent non seulement pour le NEC SX5 « Uqbar » mais aussi pour l'IBM NightHawk « Tlon » en voie d'installation, où 8 processeurs se partagent 16 gigaoctets de mémoire. Une formation sur Pthreads pour les programmeurs C/C++ est en préparation.

Conclusions :

- L'avantage de OpenMP est une programmation un peu plus aisée.
- Le désavantage est que le code ne s'exécutera pas sur n'importe quelle machine, mais seulement sur une machine à mémoire partagée. Il sera moins extensible, car le code ne s'exécutera pas non plus sur un nombre arbitraire de processeurs. Sur Uqbar, l'allocation des ressources à cheval sur deux nœuds est impossible pour ce type de code.
- Il ne faut pas s'attendre à obtenir des performances formidables en jetant simplement quelques directives sur un code séquentiel (voir l'exposé précédent sur l'éclatement d'une boucle).
- OpenMP peut être très efficace, lorsqu'il est utilisé de manière intelligente, ce qui en général demande une part importante de réflexion et d'analyse du code.
- Nous conseillons donc prudence et discernement dans l'utilisation des modèles de programmation à mémoire partagée.

Remarque sur les modèles de programmation « mixtes » :

Les architectures des supercalculateurs s'orientent de plus en plus vers l'interconnexion, par l'intermédiaire de réseaux performants, de « gros » nœuds où un certain nombre de processeurs (4, 8, 16 …) accèdent à une mémoire partagée relativement importante. Le cas d'Uqbar est un cas extrême où chaque nœud est en lui même un supercalculateur d'avant-garde, mais il est tout à fait courant aujourd'hui de trouver des machines scalaires parallèles où les nœuds sont des systèmes multiprocesseurs (quatre pour Compaq, huit pour IBM, …), ou bien des grappes de PC multiprocesseurs.

Dans ce contexte, il est important de réaliser que l'on peut très bien avoir le meilleur des deux mondes en utilisant un modèle de programmation mixte, où MPI est utilisé pour exécuter un pro-

cessus Unix sur chaque nœud et pour gérer les communications entre eux. Chacun de ces processus est à son tour parallélisé à l'intérieur d'un nœud par OpenMP, Pthreads, etc.

L'IDRIS a l'intention d'expérimenter cette modalité de programmation dès que la totalité de la grappe SX5 sera opérationnelle (fin avril 2000).