Exploitation efficace du parallélisme dans les systèmes à base de processeurs généralistes pour les applications de traitement d'images et de vidéos

Eric Debes[1], Fulvio Moschetti[2]

[1]Microprocessor Research Labs, Intel Corporation
2200 Mission College Blvd., Santa Clara, CA 95052-8119

[2]Laboratoire de Traitement des Signaux
Ecole Polytechnique Federale de Lausanne, 1004 Lausanne, Suisse
Eric.Debes@intel.com, Fulvio.Moschetti@epfl.ch

Résumé – Le but de cet article est d'expliquer comment exploiter efficacement les différents niveaux de parallélisme disponibles dans les systèmes à base de processeurs généralistes pour l'implémentation d'applications multimédias. Les différents parallélismes sont classés en trois niveaux : intra-processeur, inter-processeur et au niveau du système. L'exploitation des différents niveaux de parallélisme est étudiée du point de vue théorique ainsi que par des exemples pratiques en soulignant le gain potentiel, l'effort requis par le programmeur, le management du niveau de parallélisme et la taille idéale des données à traiter en parallèle. Un modèle d'étude des différents niveaux de parallélisme est proposé pour aider le programmeur à choisir quelles fonctionnalités permettent d'obtenir le meilleur gain en vitesse d'exécution avec le minimum d'effort de programmation et de temps de développement. De plus le temps nécessaire au management du parallélisme ainsi que la taille idéale des données à traiter en parallèle sont expliqués pour aider le programmeur à distribuer son application de manière efficace.

Abstract – The aim of this paper is to explain how to exploit efficiently the different levels of parallelism available in general-purpose processor based systems for multimedia applications. The different parallelisms are classified in three levels: intra-processor, inter-processor and at the system level. The exploitation of the different levels of parallelism is studied from a theoretical point of view as well as with application examples to show the potential gain, the effort required by the programmer, the management of the given level of parallelism and the ideal size of data to be processed in parallel. A Model of study of the different levels of parallelism is proposed in order to help the programmer to choose which features can give the best speed up with the least programming effort and development time. In addition the time needed to manage the parallelism as well as the ideal size of data to processed are given to help the programmer to efficiently distribute its application.

1 Introduction

1.1 Applications multimédias sur PCs

Durant les dernières années, les PCs ont évolué afin de tenir compte du domaine grandissant des applications multimédias. L'intégration de nouvelles fonctionnalités (instructions multimédias, parallélisme explicite d'instructions, multiprocesseurs, cartes réseaux spécialisées pour l'interconnexion de grappes de PCs, processeurs des cartes graphiques dédiés au traitement d'images et de vidéos, etc.) a révolutionné l'exécution des applications multimédias sur les systèmes à base de processeurs généralistes.

1.2 Parallélisme dans les PCs

Dans le cadre de cette étude, les types de parallélisme sont classés en trois niveaux:

- Le parallélisme *intra-processeur* inclut les instructions multimédias qui exploitent le parallélisme de données ainsi que les processeurs superscalaires et les architectures VLIW qui bénéficient du parallélisme d'instructions.
- Le parallélisme *inter-processeur* à l'intérieur de systèmes multiprocesseurs symétriques à mémoire partagée et dans les grappes de stations de travail à mémoire distribuée.
- Le parallélisme au niveau du *système* avec en particulier les entrées/sorties et l'exploitation de processeurs externes.

1.3 Présentation du modèle d'étude

Un modèle d'étude de ces différents niveaux de parallélisme est proposé pour aider le programmeur à choisir quelles fonctionnalités permettent d'obtenir le meilleur gain en vitesse d'exécution avec le minimum d'effort et de temps de développement. Ce modèle est dénommé GEMT pour Gain, Effort, Management et Taille.

Le but premier de l'exploitation du parallélisme est d'améliorer la vitesse d'exécution de l'application en exploitant au maximum les fonctionalités des stations de travail PC. C'est pourquoi la première donnée intéressante à étudier pour chaque niveau de parallélisme est le *Gain* en vitesse obtenu. Ce gain est d'abord étudié de manière théorique puis vérifié de manière pratique pour des appli-

cations multimédias.

Ensuite, l'*Effort* requis de la part du programmeur est estimé en tenant compte du temps de développement et du temps nécessaire à l'apprentissage des techniques requises pour bénéficier du parallélisme.

Le temps inutilisable pour le traitement des données, mais nécessaire au *Management* du parallélisme durant l'exécution de l'application est aussi estimé en fonction des techniques de parallélisation utilisées. Le but d'une parallélisation efficace est de trouver un bon compromis qui permette une bonne repartition des tâches tout en gardant un temps de management faible.

Finalement, la *Taille* idéale des données à traiter en parallèle ou la taille idéale des fonctions qui peuvent être exécutées en parallèle sont indiquées. Cette taille est très utile au programmeur pour partitionner son application.

1.4 Plan de l'article

Dans le reste de l'article le modele GEMT est utilisé pour étudier les niveaux de parallélisme présentés au paragraphe 1.2. Dans la section 2 le parallélisme disponible à l'intérieur du processeur est presenté. Le parallélisme entre plusieurs processeurs dans le meme PC et entre différents systèmes est étudié dans la section 3. Finalement, quelques exemples de parallélisme qui peuvent être utilisés au niveau du système sont expliqués dans la section 4 et la section 5 conclue l'article.

2 Parallélisme intra-processeur

2.1 Parallélisme d'instructions

La plupart des processeurs généralistes actuels intègrent plusieurs unités arithmétiques et logiques afin d'exécuter plusieurs instructions indépendantes en parallèle. Deux techniques principales existent pour exploiter le parallélisme d'instructions:

- les processeurs superscalaires tel le Pentium 4 d'Intel. Dans ce cas, les instructions sont réordonnées durant l'exécution pour etre exécutées dans les différentes unités.
- les architectures VLIW avec plusieurs instructions par mot d'instruction tel l'Itanium d'Intel[1]. Dans cette technique, chaque mot d'instruction contient plusieurs champs qui correspondent à différentes instructions qui vont s'exécuter en parallèle dans les unités d'exécution respectives.

Dans les deux cas, une bonne approximation du gain potentiel est donnée par :

$$G = \frac{T}{\frac{TF}{NF} + \frac{TE}{NE} + \frac{TS}{NS} + \frac{TES}{NES} + M} \quad (1)$$

oú T est le temps d'exécution linéaire. TF, NF, TE, NE, TS, NS, TES, NES et TES sont respectivement les temps passés dans les opérations flottantes, entières, sauts et entrée/sortie et le nombre N d'unités d'exécutions correspondantes. M est le temps de management perdu par rapport au gain idéal.

Pour les processeurs superscalaires, l'effort du programmeur est limité, alors que pour les architectures VLIW, le compilateur ou le programmeur doivent faire l'effort de paralléliser les instructions dans un même mot d'instruction.

2.2 Techniques SIMD et Instructions multimédias

Les instructions multimédias ont été rajoutées aux architectures pour exploiter le parallélisme de données principalement dans les applications de traitement du signal. Elles permettent, par exemple, d'effectuer une même opération en parallèle sur plusieurs pixels à l'intérieur d'un même registre. Le gain qu'il est possible d'obtenir avec cette technique est:

$$G = \frac{1}{1 - F + F \cdot \frac{TD}{TR} + M} \quad (2)$$

où F est la fraction temporelle de code qui exploite les instructions multimédias, TD la taille des données, TR la taille des registres et M le management. Pour une application pour laquelle TD=8 (comme par exemple pour les pixels d'une image) et TR=64 (ce qui est le cas dans les registres MMX d'Intel), le gain potentiel est de 8, car les mêmes opérations peuvent être exécutées en parallèle sur 8 données. L'implémentation de certaines fonctions d'un encodeur MPEG2 en assembleur avec les instructions MMX [2] a permis d'atteindre le gain intéressant de 3.9 pour un maximum de 8. Cette différence s'explique par le fait que seule une partie du code est optimisée et par les instructions nécessaires pour transférer les données des registres entiers vers les registres MMX.

Malheureusement actuellement aucun compilateur n'est capable de produire du code optimisé exploitant efficacement le parallélisme de données à l'aide des instructions multimédias. Le seul moyen consiste à utiliser des librairies, des macros ou à écrire directement le code en assembleur. Même si cela représente une tâche fastidieuse, les gains possibles en vitesse d'exécution devraient être suffisants pour motiver les programmeurs intéressés.

Le temps perdu en management du parallélisme à ce niveau est du à l'exécution d'instructions multimédias qui sont souvent plus lentes à exécuter que les instructions entières et aux changements de résolution souvent nécessaires lorsque les données sont transférées des registres entiers vers les registres SIMD (et reciproquement).

La taille des données qui peuvent s'exécuter en parallèle dépend souvent des instructions disponibles et, bien sur, des tailles des données que l'on souhaite traiter.

3 Parallélisme inter-processeur

3.1 Systèmes multiprocesseurs à mémoire partagée

La plupart des stations de travail actuelles intègrent deux processeurs ou plus alors que la plupart des applications qui tournent sur ces systèmes n'utilisent qu'un seul

thread. Si un seul thread et une seule application tourne sur une station de travail double-processeur, la charge maximale de travail est de 50% et le reste de la capacité de calcul de la station est perdu. Cependant, compte tenu de la charge de travail du au management des threads, seuls les threads nécessaires à une bonne répartition de la charge de travail doivent être créés. Le but de l'optimisation à ce niveau consiste donc a créér différents threads pour les tâches les plus importantes afin de distribuer la tâche sur tous les processeurs disponibles dans la station de travail.

Le gain qu'il est possible d'obtenir en créant plusieurs threads est donné par une adaptation de la loi d'Amdahl[3]:

$$G = \frac{1}{(1-F) + \frac{F}{N} + \frac{M}{T}} \quad (3)$$

où F est la fraction temporelle de code parallélisée, N est le nombre de processeur, T est le temps d'exécution linéaire et M est le temps de management.

L'effort nécessaire pour utiliser ce genre de technique n'est pas tellement élevé car de nombreuses interfaces de programmation existent. Cependant il est difficile de déboguer ce genre de code et un soin particulier doit être attribué à la synchronisation des différents threads.

Le management des threads dépend en grande partie de la technique utilisée. Par exemple, sous Windows 2000, il existe de nombreuses techniques pour multithreader une application[4]. Une première technique classique consiste à créer et à détruire les threads chaque fois que la fonction doit être exécutée. Dans ce cas le management des threads prend environ 50,000 cycles d'horloge [5]. Cependant, dans les applications multimédias, les mêmes fonctions sont exécutées plusieurs fois de manière consécutive avec des paramètres différents. Dans ce cas, il est possible de ne créer les threads qu'au début de l'application et d'utiliser des évènements pour démarrer et arrêter l'éxecution des threads. Le management des threads avec cette technique est réduit à 5000 cycles d'horloge.

Si l'on considère que cela vaut la peine de paralléliser l'application à l'aide de threads si le gain est supérieur à 1.5 sur une machine à double processeurs on obtient, en utilisant l'eq. 3, que T doit être plus grand que 6M. Ceci signifie que la taille minimale d'une fonction à paralléliser est de 300 000 cycles pour la méthode classique et de 30 000 cycles pour la méthode utilisant les évènements.

3.2 Grappes de stations de travail à mémoire distribuée

Certaines applications multimédias (telles que les encodeurs vidéos HDTV) ne peuvent pas être exécutées en temps réel sur de simples stations de travail, même multiprocesseurs. Elles peuvent alors être distribuées sur plusieurs machines ou sur des machines massivement parallèles. Ces dernières restent très onéreuses et rarement accessibles pour les applications multimédias. De plus avec le développement de réseaux rapides à basse latence (comme Myrinet [6]), une grappe de PCs devient une solution très intéressante d'un point de vue performance/prix pour un grand nombre d'applications [7].

Le gain qu'il est possible d'obtenir sur ce genre de système est à nouveau donné par la loi d'Amdahl [3]:

$$G = \frac{1}{(1-F) + \frac{F}{N} + \frac{M}{T}} \quad (4)$$

Les applications multimédias peuvent être parallélisées sur des grappes de PCs à l'aide de la librairie MPI (Message Passing Interface[8]). MPI permet d'envoyer et de recevoir des données d'un ordinateur à l'autre et d'exécuter des fonctions sur les differents noeuds de la grappe de PC. Il est n'est pas difficile d'utiliser cette librairie, mais le programmeur doit d'abord se familiariser avec le principe SPMD (Single Program Multiple Data) qui consiste à exécuter le même programme sur les différentes stations de travail.

Le temps perdu dans le management de ce niveau de parallélisme vient principalement du traffic sur le réseau entre les différentes machines. Non seulement le débit du réseau peut s'avérer insuffisant, mais la latence peut être le goulet d'étranglement, car le processeur devra souvent attendre les données provenant d'un autre ordinateur. Des réseaux à haut débit et à faible latence [6] ainsi que des librairies de parallélisation optimisées (comme MPI-BIP-SMP [9]) ont été développés pour atteindre des gains semblables à ceux des systèmes multiprocesseurs.

La taille idéale des données à traiter en parallèle dépend en grande partie de l'infrastructure réseau et du protocole de communication. Dans chaque cas, le débit de transfert et la latence doivent être estimés de manière expérimentale pour différents types de données. De tels graphes sont déjà disponible pour de nombreux protocoles et types de réseaux.

4 Parallélisme au niveau du système

4.1 Entrées - Sorties

La vitesse des processeurs a évolué très rapidement ces dernières années, mais les transferts d'entrées/sorties n'ont pas évolués au même rythme. De ce fait, le plus grand problème dans les applications de traitement d'images et de vidéos est de fournir suffisamment rapidement les données à traiter au processeur.

Dans le cas de l'encodage vidéo, par exemple, une solution consiste à lire en parallèle les images en avance afin qu'elles soient en mémoire dès que le processeur en a besoin. Le gain théorique de cette parallélisation est :

$$G = \frac{TTM + \frac{TI}{VL}}{TTM} \quad (5)$$

où TTM est le temps de traitement moyen par image, TI est la taille d'une image et VL est la vitesse de lecture du disque. Par exemple, pour une séquence en format YUV 4:2:0 CIF ($TI = 352{\cdot}288{\cdot}1.5{\cdot}8$), un disque dur avec une vitesse de lecture $VL = 40Mbits/s$ et $TTM = 0.1s$, le gain en vitesse est de 30% en considérant que l'utilisation du processeur pour la lecture peut être négligée par rapport à celle nécessaire au traitement des données. Cette optimisation a été utilisée pour un encodeur MPEG4. Pour une vitesse d'encodage de 7 images par seconde en format

CIF le gain obtenu varie entre 20% et 50% en fonction du processeur et de la vitesse de lecture du disque dur. Des résultats semblables peuvent être obtenus pour l'affichage à l'écran ou l'écriture sur un disque dur d'images issues d'un décodeur vidéo.

L'effort requis par un programmeur pour exploiter cette technique ainsi que le management sont les mêmes que pour la parallélisation sur plusieurs processeurs à l'aide de threads. Pour ce qui concerne la taille des données, l'idéal est de charger autant de données que possible en avance, tout en prenant soin d'avoir suffisamment de mémoire pour éviter de devoir écrire des données sur la mémoire virtuelle du disque. Cependant si cette technique est étendue à un système dans lequel l'acquisition et l'encodage de la séquence vidéo se font en temps réel, le délai entre l'acquisition et l'encodage est proportionnel au nombre d'images chargées en avance en mémoire. De ce fait, un bon compromis consiste à charger les données image par image juste avant qu'elles ne soient utilisées.

4.2 Exploitation de Ressources Externes

Les cartes graphiques et les cartes d'acquisition vidéo intègrent des processeurs de plus en plus puissants qui sont très utiles pour effectuer des traitements classiques en image et en vidéo. Des fonctions simples qui nécessitent une grande puissance de calcul peuvent être effectuées dans le processeur graphique en parallèle avec des traitements plus complexes effectués par le processeur généraliste.

Si les fonctions exécutées dans le processeur graphique et dans le processeur généraliste sont indépendantes, le gain potentiel peut être approximé par :

$$G = \frac{1}{1 - F} \tag{6}$$

où F est la fraction temporelle du code exécuté dans le processeur graphique. Sous Windows 2000, les processeurs des cartes graphiques peuvent être utilisés à l'aide de DirectX [10]. L'implémentation de la conversion YUV 4:2:0 en RGB à l'aide de DirectX a permis d'accélérer l'exécution d'un décodeur MPEG2 de 60% en utilisant une carte graphique ATI Rage Pro.

L'utilisation de ce type de parallélisme nécessite cependant un effort important de la part du programmeur, car il doit maitriser les techniques DirectX et COM [11].

Le temps perdu dans le management de ce niveau de parallélisme est faible si le temps d'exécution de la partie logicielle sur le processeur principal est plus élevé que le temps d'exécution dans le processeur graphique. Dans le cas contraire, la communication entre la mémoire principale et la mémoire vidéo doit être prise en compte.

La taille des données à transférer entre la mémoire principale et la mémoire externe doit être aussi grande que possible tout en tenant compte de la taille de la mémoire externe. Dans le cas du décodage vidéo, la solution idéale est d'envoyer les données image par image vers la mémoire vidéo et d'utiliser une technique de mémoire tampon double, c'est-à-dire d'avoir toujours une image déjà décodée en mémoire vidéo et un second tampon pour y écrire l'image suivante.

5 Conclusion

Cet article a donné quelques éléments pour exploiter efficacement les différents niveaux de parallélisme disponible dans les systèmes à base de processeurs généralistes. Davantage de détails sur le sujet peuvent être trouvés dans [12]. Un programmeur peut utiliser ces informations pour décider quel niveau de parallélisme donnera le meilleur gain avec le minimum d'effort. De plus, les informations sur les tailles idéales des données à traiter en parallèle ou des fonctions à paralléliser peuvent aider à partitionner, paralléliser et distribuer idéalement l'application. Ces informations peuvent être, bien sur, utilisés pour optimiser une application existante ou pour créer une nouvelle application qui exploitera de manière optimale les nouvelles fonctionnalités des processeurs généralistes et des systèmes associés.

Références

[1] Intel Corporation *IA-64 Application Developer's Architecture Guide*, Mai 1999

[2] Intel Corporation *Intel Architecture MMX*, Mai 1999

[3] G.T. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings AFIPS Conf., vol. 30, pp. 483, 1967.

[4] Johnson M. Hart *Win32 System Programming*, Chapitre 10, Addison Wesley, 1997

[5] Eric Debes, *Multithreading for Video Processing Applications running on PC Workstations*, Proceedings of the European Signal Processing Conference 2000, Tampere, Finland, September 2000.

[6] Myrinet Web Site, *http://www.myri.com*

[7] Rajkumar Buyya, *High Performance Cluster Computing, Vol.1, Architecture and Systems, Vol.2, Programming and Applications*, Prentice Hall, Upper Saddle River, New Jersey, 1999.

[8] Marc Snir, Steve W. Otto, Steven Huss-Lederman, Savid W. Walker and Jack Dongara, *MPI-the Complete Reference*, Scientific and Engineering Computation, MIT Press, Cambridge, MA, 1998.

[9] P. Geoffray, C. Pham and B. Tourancheau, *BIP-SMP : High performance message passing over a cluster of commodity SMPs*, Proceedings of Supercomputing, Novembre 1999.

[10] Bradley Bargen and Peter Donnelly, *Inside DirectX*, Microsoft Programming Series, Microsoft Press, Redmond, Washington, 1998.

[11] Chris Corry, Vincent W. Mayfield, John Cadman and Randy C. Morin, *COM/DCOM Primer Plus*, Sams Publishing, 1999.

[12] Eric Debes *Exploitation of Parallelism in General Purpose Processor Based Systems for Multimedia Applications*, Thèse N2319, Ecole Polytechnique Fédérale de Lausanne, Décembre 2000. http://www.eric.debes.net/phd.html