Le problème de la composition parallèle : une approche supervisée

Andra-Ecaterina Hugo

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Le problème de la composition parallèle : une approche supervisée

A.-E Hugo

INRIA, LaBRI, Université de Bordeaux,
Talence, France

Résumé
L'utilisation simultanée de plusieurs bibliothèques de calcul parallèle au sein d'une application soulève bien souvent des problèmes d'efficacité. En compétition pour l'obtention des ressources, les routines parallèles, pourtant optimisées, se gênent et l'on voit alors apparaître des phénomènes de surcharge, de contention ou de défaut de cache.
Nous présentons une technique de cloisonnement de flux de calculs qui permet de limiter les effets de telles interférences. Le cloisonnement est réalisé à l'aide de contextes d'exécution qui partitionnement les unités de calculs voire en partagent certaines. La répartition des ressources entre les contextes peut être modifiée dynamiquement afin d'optimiser le rendement de la machine. À cette fin, nous proposons l'utilisation de métriques par un superviseur pour redistribuer automatiquement les ressources aux contextes. Nous décrivons l'intégration des contextes d'exécution au support d'exécution pour machines hétérogènes StarPU et présentons des résultats d'expériences démontrant la pertinence de notre approche.

1. Introduction

L'introduction des processeurs multicoeurs et des accélérateurs au sein des plateformes de calcul haute performance a suscité de nombreux travaux de recherche autour de la portabilité des performances. Ces travaux se sont focalisés autour de supports d'exécution capables de fournir aux programmeurs des techniques et des outils permettant d'exploiter des architectures matérielles toujours plus complexes. Désormais les programmeurs disposent de supports d'exécution suffisamment matures pour exploiter de telles architectures (tels Cilk [8], OpenMP ou Intel TBB [6] pour les multicoeurs, Anthill [15], DAGuE [4], Charm++ [11], Harmony [7], KAAPI [10], StarPU [2] ou StarSs [3] pour les configurations hétérogènes) et peuvent maintenant construire des bibliothèques de calcul performantes. Ainsi la bibliothèque d'algèbre linéaire MAGMA [16], reposant sur des algorithmes particulièrement optimisés, s'appuie sur le support d'exécution StarPU pour exploiter de façon efficace et portable des architectures complexes.
L'émergence de telles bibliothèques facilite la tâche du programmeur qui a tout intérêt à recycler celles-ci pour construire son application. Cependant on observe que ces bibliothèques se comportent mal, au sens des performances, lorsqu'elles sont utilisées simultanément. En fait leur utilisation en série garantie souvent de meilleures performances. Cette dégradation des performances a été identifiée et est appelée problème de la composition parallèle [13, 12]. C'est un problème important puisqu'il représente un frein aux respects des bons vieux principes de bases de la programmation (modularité, réutilisation).
Ce problème de composition survient lorsque plusieurs supports d'exécution entrent en compétition pour l'obtention des ressources et même parfois plus simplement lorsqu'on fait des appels simultanés à des routines d'une même bibliothèque. Les routines parallèles, pourtant optimisées, se gênent et l'on voit alors apparaître des phénomènes de surcharge (l'application utilise plus de threads qu'il n'y a de cœurs à sa disposition), de contention sur les bus ou défaut de cache. En effet, à des fins d'optimisation, les programmeurs de bibliothèques de calcul parallèle ont pris l'habitude d'avoir la maîtrise du nombre et du placement des threads, de l'ordonnancement des tâches de calcul et de l'utilisation des caches. Cependant ces optimisations locales peuvent s'avérer contre-productives dans un cadre plus général notamment en présence de plusieurs flux de calcul parallèles. Ce type de problème a amené la communauté à promouvoir des supports d'exécution capable d'exécuter de nombreux flots de calcul

sans pour autant surcharger de threads la machine [8, 6]. À l'image des machines virtuelles, il existe aussi des supports d'exécution, tel Lithe [13], qui permettent d'attribuer dynamiquement des ressources matérielles à des flux de calcul : en cloisonnant ces flux on espère ainsi limiter leurs interférences. Il reste cependant un problème majeur : celui de l'attribution (automatique) des ressources à chaque partition.

Dans cet article, nous présentons un support d'exécution où les différents flux de calcul parallèles s'exécutent dans des contextes d'ordonnancement séparés. Un contexte d'ordonnancement encapsule une instance d'un support d'exécution qui s'exécute sur un ensemble d'unités de calcul. La répartition des ressources entre les contextes peut être modifiée à la demande afin d'optimiser le rendement de la machine. À cette fin, nous proposons l'utilisation de métriques (utilisation des ressources, progression du calcul) par un superviseur pour redistribuer automatiquement les ressources aux contextes. Nous avons implémenté cette approche en étendant le support d'exécution StarPU [2]. Les codes et bibliothèques de calcul développées au-dessus de StarPU peuvent ainsi tirer parti de cette fonctionnalité sans modification de code. Ceci nous permet de montrer expérimentalement l'intérêt de notre approche en l'appliquant à des exemples concrets.

2. Travaux connexes

Dans cette section nous passons rapidement en revue les supports d'exécution hybrides et présentons quelques travaux traitant du problème de la composition parallèle.

2.1. Support d'exécution pour architectures multicoeurs

Les modèles de programmation basés sur la notion de tâche, tels Cilk [8] ou Intel TBB [6], sont les mieux préparés au problème de la composition parallèle. Intel, au travers de sa suite Intel Composer, s'est attaqué depuis quelques années spécifiquement à ce problème [14, 12] en utilisant un socle commun pour exécuter ses différents environnements de programmation et bibliothèques de calcul (Intel Cilk plus, IPP, MKL, TBB). Ce socle, basé sur un ordonnanceur à vol de travail, permet de composer des applications parallèles sans créer de surcharge. De plus, pour améliorer l'efficacité de la composition parallèle[12], TBB isole automatiquement les threads maîtres de l'application (ceux créés directement par l'application) dans des arènes séparées, le pool des threads étant distribué aux arènes au prorata des demandes. Cette technique d'isolement permet d'attribuer des priorités aux arènes et d'affecter les threads aux arènes les plus prioritaires. On peut cependant noter qu'Intel TBB n'est pas composable avec Intel OpenMP, et dans les faits, la plupart des implémentations OpenMP ne sont pas composables avec elles-mêmes.

Lithe [13] est un support d'exécution qui permet l'interopérabilité entre différentes instances de supports d'exécution parallèles tels Intel TBB et Intel OpenMP. Lithe implémente une interface de partage de ressources qui définit comment des *harts* (abstraction de threads) sont transférés entre différents supports d'exécution au sein d'une même application. Toutefois, Lithe impose une organisation hiérarchique entre les instances de supports d'exécution ainsi qu'une implémentation spécifique du multitâches.

2.2. Support d'exécution pour plateformes hybrides

Parmi les supports d'exécution hybrides, on peut distinguer un ensemble de supports d'exécution orientés flot de données : DAGuE [4] et KAAPI [10] qui utilisent le vol de travail comme technique d'ordonnancement, l'extension de Anthill aux GPUs [15] qui distribue les tâches à la demande et, deux plateformes plus génériques OmpSs [3] et StarPU [2] où les ordonnanceurs sont interchangeables. Tous ces supports d'exécution hybrides sont basés sur des techniques d'ordonnancement en ligne et utilisent les techniques de recouvrement des transferts par le calcul. Cependant, même si ces supports ne provoquent pas de surcharge et prennent l'affinité mémoire en compte, ils ne proposent pas d'outils de cloisonnement de flots de données et ne permettent pas l'utilisation simultanée de différents ordonnanceurs au sein d'une même application.

OpenCL 1.2 [9] propose les notions de contexte et de *device fission* qui permettent de partitionner finement les unités de calcul de la machine. Cependant les unités de calcul au sein d'un contexte doivent être identiques (il n'est pas possible de créer un contexte mêlant CPU et GPU) et OpenCL ne propose pas d'ordonnanceur de haut niveau.

3. StarPU et le problème de la composabilité

3.1. Le support exécutif StarPU

Le support exécutif StarPU [2] est une bibliothèque qui propose aux programmeurs une interface portable pour ordonnancer des graphes de tâches dynamiques sur un ensemble hétérogène d'unités de calcul (CPUs et GPUs).

Toute tâche est associée à une voire plusieurs implémentations afin de pouvoir être indifféremment exécutée sur un CPU ou un GPU, par exemple. De plus, StarPU utilise une mémoire virtuellement partagée automatisant la disponibilité et la cohérence des données.
StarPU a été conçu comme une plate-forme de développement et d'expérimentation d'ordonnanceurs hétérogènes. Implémenter un ordonnanceur consiste essentiellement à créer des listes d'ordonnancement, à les associer à des unités de calcul et à définir les fonctions de dépôt et de retrait d'une tâche. Plusieurs ordonnanceurs ont été intégrés, ils sont basés sur divers algorithmes gloutons classiques ou encore sur la technique du vol de travail. StarPU dispose aussi de stratégies d'ordonnancement plus complexes qui implémentent des variations de l'heuristique *Minimum Completion Time* (MCT) [17]. Cette famille d'ordonnanceurs exploite des modèles de performances mis au point automatiquement afin d'estimer automatiquement la durée d'exécution et des transferts de données nécessaire à l'exécution d'une tâche donnée sur une unité de calcul donnée. L'usage de cette technique est limité aux tâches dont la durée est prédictible.

3.2. Problèmes liés à la composabilité

Pour optimiser les performances obtenues par leur bibliothèque de noyaux de calcul, les programmeurs ont pris l'habitude de forcer les décisions de l'ordonnanceur sous jacent. Par exemple, ils étudient le comportement de l'ordonnanceur pour adapter l'ordre de soumission des tâches ou pour créer de fausses dépendances entre les tâches, ils n'hésitent pas à placer les données nécessaires à l'exécution d'une tâche sur une unité pour favoriser/imposer son exécution sur celui-ci. L'entrelacement des tâches issues de différents flux de calcul vient rompre de telles mécaniques, le programmeur ne contrôle plus le déroulement de son code. Pour résoudre ce problème, on peut modifier des paramètres de l'ordonnanceur pour, par exemple, augmenter la localité du calcul sur la machine (au détriment du taux d'occupation de l'unité de calcul). Une autre technique est de donner aux programmeurs d'application le moyen de cloisonner les flux de calcul à l'instar de ce que fait TBB.
L'introduction de mécanismes de cloisonnement peut donc être utile à la composition de bibliothèques de calcul, ils s'avèrent en fait nécessaires lorsque deux bibliothèques reposent sur des ordonnanceurs incompatibles voire des supports d'exécution différents. Dans cette optique, il nous parait intéressant d'étudier le problème de la composition à l'aide de StarPU car il est orienté tâche et dispose d'une interface de programmation d'ordonnanceurs.

4. Notre approche pour co-ordonnancer plusieurs codes parallèles

Pour étudier le problème de la composition, nous avons introduit dans StarPU la notion de *contexte d'ordonnancement*. À l'image des machine virtuelles légères, les contextes d'ordonnancement permettent une partition flexible de la machine et la cohabitation de flux de calcul parallèles et ce sans nécessiter de modification de code. La notion de contexte permet donc aux programmeurs non seulement de contrôler la distribution d'unités de calculs (CPUs, GPUs) à des flux parallèles concurrents mais aussi d'adopter des stratégies d'ordonnancement adaptées à chaque flux.

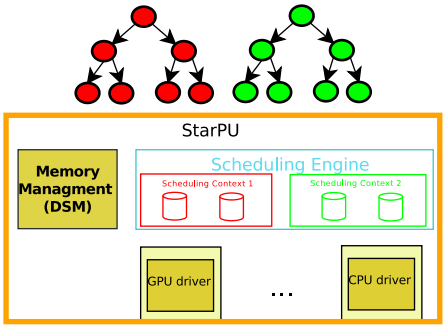
4.1. Des contextes d'ordonnancement au sein de StarPU

Au niveau de StarPU, il s'agit de cloisonner les contextes en faisant en sorte que chaque contexte détienne (ou partage) des unités de calcul, que tout thread de l'application puisse s'inscrire à un contexte pour soumettre ses tâches qui seront prises en charge par un ordonnanceur associé au contexte (voir Figure 2). Au niveau technique, l'intégration des contextes au sein de StarPU a été mise en œuvre à l'aide d'un dispositif permettant d'ajouter ou de supprimer dynamiquement une file d'ordonnancement à une instance d'ordonnanceur et ce de façon asynchrone. Ainsi alertés, les ordonnanceurs réagissent à toute nouvelle distribution des ressources permettant ainsi le redimensionnement dynamique des contextes. Cette capacité est importante puisque les besoins des différents flux de calcul évoluent au cours du déroulement de l'application.
De plus, expérimentalement, on observe que certaines unités de calcul (par exemple les GPUs) ne sont pas toujours exploités au niveau de leur potentiel par un seul flux de calcul. Aussi il parait intéressant de pouvoir allouer une même unité de calcul à plusieurs contextes. Pour réaliser ce partage nous avons modifié les pilotes des unités de calcul de StarPU pour qu'ils consultent (selon l'algorithme du tourniquet) l'ensemble des listes d'ordonnancement qui leur sont associés. Cependant partager une unité de calcul peut compromettre les propriétés d'ordonnancement visées, les contextes partageant une unité doivent donc avoir des stratégies d'ordonnancement compatibles voire coopératives entres elles. Aussi nous avons été amené à mettre au point un ordonnanceur de la famille MCT dont les instances coopèrent pour estimer la date à laquelle une unité partagée sera inoccupée.

```
int resources1[3] = {CPU_1, CPU_2, GPU_1};
int resources2[4] = {CPU_3, CPU_4, CPU_5, CPU_6};

/* define the scheduling policy and the table
  of resource ids */

sched_ctx1 = starpu_create_sched_ctx("heft",resources1,3);

sched_ctx2 = starpu_create_sched_ctx("greedy",resources2,4);
```

```
/* define the context associated
   to kernel 1 */
starpu_set_sched_ctx(sched_ctx1);

/* submit the set of tasks of the parallel
   kernel 1*/
for( i = 0; i < ntasks1; i++)
 starpu_task_submit(tasks1[i]);
```

FIGURE 1 – Programmer avec les Contextes d'ordonnancement



(a) Contextes d'ordonnancement dans StarPU.

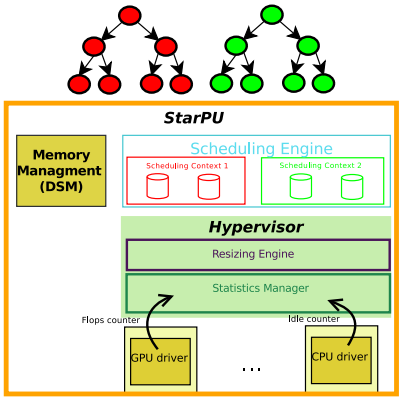
$$\max\left(\frac{1}{t_{max}}\right) \text{ tel que } \left(\begin{array}{l}\left(\forall c \in C, n_{\alpha,c}v_\alpha + n_{\beta,c}v_\beta \geq \frac{W_c}{t_{max}}\right)\\ \wedge\left(\sum_{c\in C} n_{\alpha,c} = n_\alpha\right)\\ \wedge\left(\sum_{c\in C} n_{\beta,c} = n_\beta\right)\end{array}\right) \quad (1)$$

FIGURE 2 – Contextes d'ordonnancement : Structure et dimensionnement.

Souvent les programmeurs d'application haute performance ont une bonne connaissance des caractéristiques (complexité, scalabilité) et des besoins (consommation mémoire) de noyaux de calcul qu'ils utilisent. Ils ont les compétences pour analyser et comprendre le comportement et les performances de leur application. Avec cette vision de haut niveau, ils peuvent vouloir indiquer comment les ressources devraient être distribuées aux contextes. Pour profiter de ces connaissances, on procure aux programmeurs le moyen de spécifier une distribution des ressources. Ainsi les programmeurs ont un outil pour piloter le déroulement de leur application à des fins d'optimisation ou de compréhension des performances. La Figure 1 décrit comment les programmeurs peuvent facilement créer des contextes et leur associer des ressources ainsi qu'une stratégie d'ordonnancement. Pour accompagner le déroulement de l'application (création / terminaison d'un flux de calcul), on fournit la possibilité de créer/détruire dynamiquement les contextes d'ordonnancement.

De plus, pour faciliter l'écriture des programmes dont l'efficacité est portable, nous avons intégré à StarPU une routine pour calculer automatiquement une distribution qui minimise la date de terminaison des flux en cours d'exécution. Cette estimation n'est possible que si l'on connaît approximativement la quantité de travail (le nombre d'opérations flottantes) que doit réaliser chaque contexte. L'algorithme de distribution est celui de la résolution du programme linéaire représenté par l'équation présentée dans la Figure (2), où l'on calcule le nombre de CPUs et GPUs nécessaire à chaque contexte de façon à minimiser la date de terminaison globale. Il s'agit ici d'une approximation grossière puisqu'on ne prend pas en compte ni les dépendances de données (le système serait alors trop complexe à résoudre) ni les spécificités des tâches. Nous disposons aussi d'un algorithme de distribution plus précis car basé sur le système d'estimation de StarPU et le volume de calcul par type de tâche - par manque de place nous ne présenterons pas ce système.

Dans ce programme linéaire, C représente la liste de contextes, $n_{\alpha,c}$ et $n_{\beta,c}$ représentent les inconnues du système, c'est-à-dire le nombre de CPUs et GPUs associés au contexte c, W_c est la quantité totale de travail associée au contexte c, t_{max} représente le temps maximal passé par le contexte à exécuter sa quantité de travail, v_α et v_β représente la vitesse (i.e. le nombre d'opérations flottantes par seconde) d'un CPU, respectivement d'un GPU sur la plateforme, n_α et n_β sont le nombre de CPUs, respectivement GPUs disponibles sur la machine. L'équation de la Figure (2) exprime le fait que chaque contexte doit avoir un nombre suffisant de CPUs et GPUs tel qu'il puisse exécuter la quantité de travail assignée avant le délai t_{max}. Évidemment le programme linéaire peut être généralisé facilement aux plateformes avec plus de deux types d'unités de calcul.



(a) L'Hyperviseur dans StarPU.

```
/* select an existing resizing policy */
struct hypervisor_policy policy;
policy.custom = 0;
policy.name = "idle_policy";

/* initialize the hypervisor and set its resizing
   policy */
sched_ctx_hypervisor_init(policy);

/* register context 1 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx1);

/* register context 2 to the hypervisor */
sched_ctx_hypervisor_register_ctx(sched_ctx2);

/* define the constraints for the resizing */
sched_ctx_hypervisor_ctl(sched_ctx1,
   HYPERVISOR_MIN_CPU_WORKERS, 3,
   HYPERVISOR_MAX_CPU_WORKERS, 7,
   NULL);
```

FIGURE 3 – L'hyperviseur : Architecture et interface de programmation.

4.2. Superviser pour redistribuer

Afin d'automatiser la répartition des ressources nous avons mis au point un système de supervision de l'activité des unités de calcul et de progression des contextes. Ce dispositif est mis en œuvre par un *hyperviseur* exécuté de façon distribuée par les CPUs. Le rôle de l'hyperviseur est de redimensionner les contextes d'ordonnancement lorsque des dégradations de performance sont observées. L'hyperviseur (voire Figure 3) peut être invoqué depuis l'application (directement ou indirectement lors de création / destruction / modification de contexte) ou peut être déclenché automatiquement.

Lors de l'exécution, la progression du déroulement de l'application est évaluée au travers d'indications fournies soit directement par l'application, soit par les compteurs de performance ou encore par la routine d'estimation de la date de terminaison. Au besoin, une redistribution des ressources est appliquée. De plus cette redistribution peut être contrainte par le programmeur, les contraintes étant explicitées sous forme d'intervalle d'unités de calcul à respecter. Dans la Figure 3 on peut voir comment l'intervalle de redimensionnement peut être spécifiée sur les contextes enregistrés auprès de l'hyperviseur.

Dans ce cadre, nous avons étudié deux métriques pour piloter le redimensionnement des contextes. La première est basée sur un compteur de bas niveau mesurant l'inactivité des ressources et la seconde est basée sur la vitesse instantanée d'un contexte et sur le nombre de flops qu'il reste à exécuter. Le choix entre les deux est déterminé par les informations fourni par le programmeur. Redimensionner lors de l'inactivité des ressources peut être vue comme la solution qui automatisent le plus l'intervention du programmeur.

Dans la stratégie *Idleness-base resizing*, les contextes sont redimensionnés lorsqu'une des ressources est inactive pendant une période plus longue que celle spécifiée par le programmeur. Expérimentalement on s'est aperçu que ce seuil peut être dépendent du parallélisme l'application.

Dans la stratégie *Speed-based resizing* l'application procure une estimation de la quantité totale de travail (le nombre de flops) correspondante à chaque noyau parallèle et à chaque tâche. Avec ces indications, l'hyperviseur calcule la vitesse instantanée de chaque noyau parallèle ce qui permet d'estimer la date de terminaison de chaque noyau. Lorsque la différence de dates de terminaison entre les contextes est suffisamment grande, l'hyperviseur redimensionne les contextes de façon à minimiser le temps de terminaison des flux en cours. Pour ce faire on utilise le programme linéaire décrit par l'équation de la Figure (2). Cette stratégie est un bon exemple de stratégie collaborative entre le programmeur, l'application et le support exécutif.

5. Évaluation

Nous étudions expérimentalement l'impact de l'utilisation de contextes au sein d'applications mettant en jeu plusieurs flux de calculs parallèles.

	Total execution time	Total data transfered
1 context : 9 CPUs / 3 GPUs	52.0 s	113 GB
3 contexts : 3 x (3 CPUs / 1 GPU)	34.8 s	37 GB
9 contexts : 9 x (1 CPUs / 0.3 GPU)	34.4 s	41 GB
serial execution	44.3 s	87 GB

(a) Exécution de 9 factorisations de Cholesky de matrices d'ordre 20000.

	Execution time
Best empirical detected distribution	18.6 s
Arbitrary distribution	24.8 s
Speed-based resized distribution	23.8 s
Idleness-based resized distribution	24.3 s

(b) redistribution de ressources.

TABLE 1 – Résultats expérimentaux.

5.1. Plateforme expérimentale et codes utilisés

La plateforme *mirage* est composée de deux processeurs Intel hexa-core X5650 cadencés à 2.67 GHz et dotés de 12 Mo de cache L3 et de 36 GB de mémoire principale. À cela s'ajoute trois cartes NVIDIA Tesla M2070 dotées chacune de 6 Go de mémoire. Notons que trois des douze cœurs de la machine servent à piloter les cartes NVIDIA. Cette machine correspond à un serveur de calcul performant et le coté multi-gpus est particulièrement intéressant pour nos expériences.

Nous avons utilisé la bibliothèque hybride d'algèbre linéaire MAGMA [16], et plus précisément son implémentation basée sur StarPU [1] et la stratégie MCT. Nous avons sélectionné le noyau de factorisation suivant la méthode de Cholesky (`potrf`) pour sa simplicité et sa régularité. Le nombre de tâches générées par ce noyau dépend de la taille des matrices et de celle du facteur de blocage utilisées par la bibliothèque pour décomposer les opérations algébriques. Dans ces expériences nous avons utilisé deux facteurs de blocage : 960 x 960 éléments (expérimentalement très performantes sur GPUs) et 192 x 192 éléments (très performantes sur CPUs).

Nous avons aussi utilisé un solveur issu de la dynamique des fluides (CFD) disponible dans la suite Rodinia [5]. Ce code est représentatif des calculs sur des grilles non structurés. Le nombre de tâches engendrées par ce noyau dépende du nombre de domaines utilisés pour décomposer la grille et du nombre d'itérations réalisées. Les tâches d'une même itération sont indépendantes entre elles. Là encore un ordonnanceur de la famille MCT est utilisé.

5.2. Composition de noyaux de calculs différents

Nous étudions le comportement de l'exécution simultanée d'une factorisation de Cholesky d'une matrice de 15 000 x 15 000 éléments et du solveur CFD pour 2957K éléments sur 200 itérations. Le domaine de CFD à simuler est coupé en deux. Notons qu'exécuté seul, StarPU associe un GPU par sous domaine (2 GPUs en tout), aucun CPU n'entrant en jeu. Expérimentalement, on observe que l'exécution entremêlée des deux flux de calcul prend 19.83s contre 14.26s lorsqu'on utilise 2 contextes (2 GPUs sont dédiés à la CFD, le reste de la machine étant laissée à la factorisation). L'analyse des traces montre que, sans cloisonnement, des tâches issues de la factorisation viennent perturber celles issues de la CFD provoquant des transferts mémoires intempestifs.

5.3. Renforcer la localité des calculs grâce aux contextes

Nous comparons l'exécution de trois factorisations de 20 000 x 20 000 éléments indépendants avec 1 contexte puis 3 contextes (où 1 GPU et 3 cœurs sont associés à chaque contexte). Les résultats expérimentaux montrent qu'utiliser des contextes permet d'augmenter de 10% le nombre de requêtes ne provoquant pas de transferts mémoires : le taux de succès passe ainsi de 82% à 92%, le nombre de giga octets transférés passant lui de 27,3 à 12,7.

Nous nous intéressons ensuite au temps d'exécution où l'on effectue en parallèle 9 factorisations (20 000 x 20 000 éléments) et aussi en série (les unes après les autres). Les résultats présentés dans la Table 5.3 (1(a)) montre que sans cloisonnement on observe significativement plus de transferts mémoire ce qui provoque, d'après l'analyse des traces, des périodes d'inactivité sur les GPUs. En fait ici les 9 flux se battent agressivement pour l'obtention des 3 GPUs à la fois, provoquant des transferts mémoire et de la contention sur le bus. En cantonnant l'utilisation de chaque flux à un seul GPU on observe une nette amélioration due à une meilleure localité des calculs.

5.4. Améliorer l'efficacité de la composition de codes grâce à la supervision

On présente dans la section suivante des résultats qui montrent que l'hyperviseur est un outil important pour améliorer la performance des applications composées de plusieurs noyaux parallèles. D'une part, il est utile au programmeur pour retomber sur une bonne distribution de ressources sur les contextes lorsque la configuration initiale n'est pas suffisamment précise. D'autre part, il est nécessaire lorsqu'une distribution statique ne peut pas satisfaire les besoins des noyaux à cause d'un parallélisme irrégulier.

Ajuster la distribution de unités de calcul sur les contextes

Dans l'expérience suivante, on montre que l'hyperviseur peut détecter une dégradation de performance liée à la composition des contextes, déterminer une meilleure distribution des ressources et dynamiquement redimensionner les contextes en conséquence.

Pour illustrer l'apport de l'hyperviseur, on simule une application dont l'exécution arrive à un point où la distribution des ressources n'est plus adaptée. Pour ce faire, on considère une application constituée de deux factorisations de Cholesky, la première sur une matrice d'ordre 15 000 (le facteur de blocage utilisé est de 192) et la seconde sur une matrice d'ordre 30 000 (le facteur de blocage utilisé est de 960). De plus, on se place dans une situation dans laquelle la distribution initiale des ressources est mauvaise. Plus précisément, la distribution choisie consiste à attribuer 3 CPUs au petit noyau et donner le reste, à savoir 6 CPUs et 3 GPUs, au second noyau (la meilleure configuration obtenue empiriquement consiste à donner les 9 CPUs au petit noyau et à associer les 3 GPUs à l'autre noyau).

Dans le tableau 1(b) on peut observer que la stratégie "Speed-based" corrige le comportement de l'application mais la distribution initial de ressources impacte sensiblement les performances. La détection du manque de performance de la distribution initiale est réalisée trop tardivement et ne permet pas son amortissement. La stratégie "Idleness-based" a encore plus de mal à trouver une configuration stable parce que la stratégie n'a ni une vision globale ni une vision à long terme du système.

Augmenter l'efficacité grâce à l'intuition du programmeur

Dans cette section on montre que l'hyperviseur peut utiliser la connaissance du programmeur pour réagir plus rapidement à l'irrégularité du parallélisme des applications.

On utilise la même application proposée dans les expériences précédentes, sauf que cette fois-ci on démarre deux flux, chacun exécute trois factorisations Cholesky consécutives. Le premier flux s'exécute linéairement et le deuxième est déclenché de temps en temps. Le premier flux est constitué de factorisations de matrices d'ordre 30 000 (avec une taille de blocage de 960). Le second, quant à lui, est constitué de noyaux manipulant des matrices d'ordre 10 000 (avec une taille de blocage de 192). On compare le cas dans lequel les deux flux soumettent leurs tâches respectives à un même contexte au cas où chaque flux est associé à un contexte différent. Ce dernier scénario est mis au point en indiquant à l'hyperviseur, lorsque chaque "petit" noyau du second flux doit démarrer son exécution, de lui attribuer (resp. libérer) des ressources (4 CPUs) dans ce cas, au début (resp. fin) du noyau.

On peut observer une amélioration de 2 secondes (ce qui équivaut à un gain de l'ordre de 10%) lorsqu'on redimensionne dynamiquement les contextes quand le petit flux est déclenché (17.2 s). La localité est affectée si on les laisse partager les ressources (19.7 s). En attribuant périodiquement des ressources au petit flux la gestion du cache du grand flux est affectée seulement quand le petit démarre un noyau.

Dimensionnement automatique de noyaux ne passant pas à l'échelle

L'hyperviseur a une mission importante lorsqu'on compose des noyaux qui ne passent pas à l'échelle. Nous reprenons notre toute première expérience en plaçant dans un contexte le solveur CFD (2957k cellules découpés en 2 sous domaines, durée de 200 itérations) et dans un deuxième contexte une factorisation de Cholesky d'une matrice d'ordre 15 000. Ici, on attribue 3 GPUs à CFD et 9 CPUs à la factorisation de Cholesky. Évidement cette configuration est inefficace (53.08 s) parce que la factorisation de Cholesky aurait besoin du GPU que CFD n'arrive pas à utiliser. Avec l'intervention de l'hyperviseur le GPU inactif est déplacé dans le contexte correspondant à la factorisation et on observe une amélioration sensible des performances (14,25 s). Ainsi, le gain obtenu dans cette configuration est de l'ordre de 75%.

6. Conclusion et perspectives

Nous avons étudié le problème de la composition parallèle à l'aide de StarPU. Pour cela, nous avons introduit la notion de contexte d'ordonnancement pour donner au programmeur la maîtrise des ressources attribuées à différents flux de calcul : les contextes peuvent être redimensionnés à volonté et les unités de calcul peuvent être partagées ou non. De plus, afin d'assurer la portabilité des performances, nous proposons d'utiliser un programme linéaire dont la solution nous permet d'obtenir une bonne distribution correspondant aux indications de charge transmisses par l'application. Dans la même optique, nous proposons l'utilisation de métriques par un hyperviseur pour redistribuer automatiquement les ressources aux contextes. Enfin, nous avons montré expérimentalement l'intérêt de cette approche et, en particulier, l'apport d'un système de supervision pour redistribuer automatiquement les ressources aux contextes.

Dans un futur proche, nous allons étudier de nouvelles métriques afin de mieux piloter la redistribution automa-

tique, en particulier nous nous intéresserons aux indications que pourrait fournir l'application au support d'exécution. D'un autre point de vue, nous pensons aussi étendre notre plateforme aux systèmes embarqués multicœurs et hétérogènes (tels les terminaux numériques portables). Dans ce cadre, il peut être intéressant d'utiliser conjointement une modélisation des performances et de la consommation énergétique des tâches pour ordonnancer celles-ci en fonction de contraintes temps-réels ou de consommation. On peut imaginer devoir faire s'exécuter simultanément des applications parallèles ayant des contraintes différentes et incompatibles entre elles.
Enfin nous pensons que notre approche pourraient être étendue à d'autres supports d'exécution. A l'image de Lithe nous devrions pouvoir exécuter en parallèle des bibliothèques basées sur StarPU, OpenMP ou Intel TBB.

Remerciements

Nous remercions Olivier Beaumont pour ses conseils quant à la mise au point des programmes linéaires que nous avons réalisés. Ce travail a été co-financé par la Région Aquitaine et la C.E.E. via le projet FP7 PEPPHER (contrat 248481).

Bibliographie

1. Agullo (E.), Augonnet (C.), Dongarra (J.), Ltaief (H.), Namyst (R.), Thibault (S.) et Tomov (S.). – A hybridization methodology for high-performance linear algebra software for GPUs. *in GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484.
2. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, vol. 23, février 2011, pp. 187–198.
3. Ayguadé (E.), Badia (R.), Igual (F.), Labarta (J.), Mayo (R.) et Quintana-Ortí (E.). – An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. *In : Euro-Par*. pp. 851–862. – Berlin, Heidelberg, 2009.
4. Bosilca (G.), Bouteiller (A.), Danalis (A.), Herault (T.), Lemarinier (P.) et Dongarra (J.). – Dague : A generic distributed dag engine for high performance computing. *Parallel Computing*, vol. 38, nIssues 1–2, 2012, pp. 37 – 51.
5. Che (S.), Boyer (M.), Meng (J.), Tarjan (D.), Sheaffer (J. W.), Lee (S.-H.) et Skadron (K.). – Rodinia : A benchmark suite for heterogeneous computing. *In : IISWC*. pp. 44–54. – IEEE.
6. Corporation (I.). – TBB reference manual. – http ://threadingbuildingblocks.org.
7. Diamos (G. F.) et Yalamanchili (S.). – Harmony : an execution model and runtime for heterogeneous many core systems. *In : HPDC '08 : Proceedings of the 17th international symposium on High performance distributed computing*. pp. 197–200. – New York, NY, USA, 2008.
8. Frigo (M.), Leiserson (C.) et Randall (K.). – The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, vol. 33, n5, 1998, pp. 212–223.
9. Group (T. K.). – OpenCL - the open standard for parallel programming of heterogeneous systems, 2011. http ://khronos.org/opencl/.
10. Hermann (E.), Raffin (B.), Faure (F.), Gautier (T.) et Allard (J.). – Multi-gpu and multi-cpu parallelization for interactive physics simulations. *In : Euro-Par 2010 - Parallel Processing*, éd. par D'Ambra (P.), Guarracino (M.) et Talia (D.), pp. 235–246. – Springer Berlin / Heidelberg, 2010.
11. Jetley (P.), Wesolowski (L.), Gioachin (F.), Kalé (L. V.) et Quinn (T. R.). – Scaling hierarchical n-body simulations on gpu clusters. *In : SC*. pp. 1–11. – IEEE.
12. Marochko (A.). – Tbb 3.0 task scheduler improves composability of tbb based solutions., 2012. http ://software.intel.com/en-us/blogs/2010/05/13/tbb-30-task-scheduler-improves-composability-of-tbb-based-solutions-part-1/.
13. Pan (H.), Hindman (B.) et Asanović (K.). – Composing parallel software efficiently with lithe. *SIGPLAN Not.*, vol. 45, June 2010, pp. 376–387.
14. Sabahi (M.). – Getting code ready for parallel execution with intel® parallel composer, 2012. http ://software.intel.com/en-us/articles/getting-code-ready-for-parallel-execution-with-intel-parallel-composer.
15. Teodoro (G.), Sachetto (R.), Sertel (O.), Gurcan (M.), Meira (W.), Catalyurek (U.) et Ferreira (R.). – Coordinating the use of gpu and cpu for improving performance of compute intensive applications. *In : Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pp. 1 –10.
16. Tomov (S.), Nath (R.), Ltaief (H.) et Dongarra (J.). – Dense linear algebra solvers for multicore with gpu accelerators. *In : Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1 –8.
17. Topcuoglu (H.), Hariri (S.) et Wu (M.-Y.). – Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, n3, Mar 2002, pp. 260–274.