Migration d'applications pour les systèmes multi-cœurs et 64 bit

Les processeurs à cœurs multiples et hybrides arrivent à la tête du classement établi par Gartner pour les technologies de rupture de 2008 à 2012. Ces technologies sont depuis entrées dans le mainstream. Ainsi, le défi de supporter les processeurs de nouvelles générations est devenu une nécessité. Ceci nous amène à nous interroger sur les possibilités de migration pour les applications existantes. La migration vers les processeurs modernes peut poser plusieurs problèmes aux développeurs. Nous avons donc choisi de mettre l'accent sur les erreurs communes en développement 64 bit et les techniques les moins intrusives pour paralléliser son application.

Processeurs 64 bit

Est-il suffisant de recompiler le code avec un compilateur 64 bits ? La recompilation est suffisante pour plusieurs types de programmes. On vérifiera que l'application utilise seulement des librairies compilées en 64 bit, car un processus 64 bit peut exécuter uniquement du code 64 bit. Dans des applications triviales, cette étape suffit à elle seule. Cependant, en réalité, rien ne garantit que les librairies utilisées aient une version 64 bit, la conversion est d'autant plus compliquée si leurs codes source sont indisponibles.
Dans ce cas, on pourra envisager des scénarios de communication interprocessus entre l'application et la bibliothèque en question. Outre cela, la compilation ne garantit pas le bon fonctionnement de l'application. De nouveaux bugs et anomalies peuvent avoir lieu, allant des erreurs d'affichage jusqu'aux failles de sécurité les plus sévères dues à un dépassement de tampon « buffer overflow » par exemple.
La mauvaise manipulation des pointeurs, la lecture des fichiers binaires ainsi que l'usage du code assembleur sont des sources potentielles de bugs. Prenons un exemple simple de programme qui a un comportement différent selon l'architecture matérielle.

```
int main(int argc, char* argv[])
{
 ///Le type size_t est 64bit pour le
 ///compilateur 64bit.
 size_t size = sizeof(myObject);
 ///L'affectation peut engendrer une
 ///perte de données.
 long a = size;

 ///%x assume que le type est 32 bit
 printf("%x\n",a);
 printf("%x\n",size);

 return 0;
}
```

La procédure main ne compte que 5 lignes, pourtant on peut identifier déjà deux bugs. Le type size_t a une valeur 64 bit pour un compilateur 64 bit. Le type long est un type 32 bit, la conversion : long a = size; Cela peut donc engendrer la perte de données, car la taille du type size_t est supérieure à celle du type long. De plus, %x qui permet de formater l'affichage en hexadécimal ne prend en compte que les valeurs 32 bit. On pourrait corriger la fonction d'affichage pour le programme précédent en remplaçant %x par %I64x pour l'architecture x64. Essayons à présent de corriger l'opération d'affectation, on pourrait songer à remplacer le long par le type *long long*. Le programme fonctionne correctement pour l'architecture x64, mais il ne retourne plus la bonne valeur pour l'architecture x86 ! Ceci était prévisible puisque *long long* est un entier 64 bit. À l'évidence, la solution la plus simple serait de remplacer le long par size_t. Ce petit programme de quelques lignes montre que la correction de ce type d'erreurs est loin d'être aisé.

```
#ifdef _WIN64
  #define xformat "I64"
#else
  #define xformat ""
#endif

int main(int argc, char* argv[])
{
 ///Le type size_t est 64bit pour le
 ///compilateur 64bit.
 size_t size = sizeof(myObject);
 ///L'affectation peut engendrer une
 ///perte de données.
 size_t a = size;

 ///%x assume que le type est 32 bit
 printf("%x" xformat "\n",a);
 printf("%x" xformat "\n",size);

 return 0;
}
```

Les deux erreurs dans le programme précédent sont causées par le changement de taille pour le type size_t. Ci-après une liste de

quelques types prédéfinis en C++ et leurs tailles respectives pour les architectures 32 et 64 bit.

	x86	x64
Int	32	32
Long	32	32
time_t	64	64
ptrdiff_t	32	64
Size_t	32	64

Le tableau ci-dessus montre qu'une classe contenant des attributs de type pointeur par exemple, n'occupe pas la même taille en mémoire pour un programme 64 bit et 32 bit.

Structures de données

Malgré la puissance brute des processeurs modernes, certains programmes compilés en 64 bit peuvent avoir une baisse de performance accompagnée généralement d'une augmentation importante en consommation de mémoire.

L'augmentation en consommation mémoire n'est pas seulement due au changement de taille pour les types de base. En effet, il y a de fortes chances qu'un mauvais alignement des structures soit la cause de cette augmentation. Considérons un cas ou ces effets secondaires se manifestent.

Soit la structure suivante :

```
struct heavyStruct
{
  char b;
  int *a;
  int  u;
};
```

En 32 bit, cette structure occupe une taille de 12 octets en mémoire. Par contre, en 64 bit elle aura une taille de 24 octets. Observons le cas d'une structure qui a exactement les mêmes attributs que « heavyStruct », mais qui ne sont pas dans le même ordre. On pourrait s'attendre que cette dernière ait naturellement une taille identique à « heavyStruct » mais il n'en est rien ! En effet en 64 bit elle aura une taille 16 octets et en 32 bit elle reste égale à 12 octets. Retenons donc ceci : la taille d'une structure n'est pas égale à la somme des tailles de ses attributs.

Dans la plupart des cas, ceci n'a pas un impact perceptible sur la consommation en mémoire. Toutefois, l'ordre des attributs à l'intérieur d'une structure peut limiter la consommation en mémoire de l'application, notamment lorsqu'il s'agit de créer un tableau de grande taille.

```
struct lightStruct
{
  int *a;
  int  u;
  char b;
};
```

La taille d'un struct n'est pas égale à la somme des tailles de ses attributs comme le montre la figure suivante, les cases en rouge sont celles occupées par les données de la structure tandis que celles en gris représentent l'espace gaspillé.

heavyStruct

	Octets							
	1	2	3	4	5	6	7	8
caractère	●	•	•	•	•	•	•	•
pointeur	●	●	●	●	●	●	●	●
entier	●	●	●	●	•	•	•	•

Un agencement meilleur des attributs permet d'optimiser l'usage de mémoire.

lightStruct

	Octets							
	1	2	3	4	5	6	7	8
pointeur	●	●	●	●	●	●	●	●
Entier+caractère	●	●	●	●	●			

Conclusion

Les erreurs dues au changement de taille des types prédéfinis sont très fréquentes. Et comme nous l'avons vu il y a de fortes probabilités que ces erreurs aient des conséquences dramatiques sur le comportement de l'application.

Systèmes multi-cœurs

Nous avons vu quelques pièges à éviter lors de la migration d'applications pour une architecture 64 bit. Malheureusement, cela ne suffit plus à tirer avantage des processeurs à cœurs multiples. En effet, si notre application est monothread, son processus sera exécuté sur un seul cœur.

Mais comment ferait-on pour paralléliser son application existante, sans être amené à la réécriture du code source ?

Bien entendu, on peut utiliser les threads Windows ou ceux de POSIX, mais pour la migration d'application on préfère utiliser la technologie la moins intrusive. La solution qui répond le plus à ce besoin est la technologie OpenMP. Car elle ne nécessite pas de modifier la structure de l'application.

Et c'est une technologie standard multi-plate-forme supportée par plusieurs compilateurs C/C++.

Supposons que nous ayons à notre disposition une technologie qui nous permette de paralléliser assez rapidement des portions de notre code existant. Si par exemple on parvient à paralléliser 60 % de notre code. Quel est le gain de performance maximal que l'on peut obtenir ?

La loi d'Amdahl, très utilisée en informatique, permet d'avoir une idée sur la performance maximale que l'on peut atteindre en parallélisant une portion P du code sur N cœurs :

$$\text{performance maximale} = \frac{1}{1 - P + \frac{P}{N}}$$

Si l'on parvient par exemple à paralléliser 60 % (P = 0,6) du code sur un système à quatre processeurs (N = 4) alors on peut s'attendre à une augmentation des performances allant jusqu'à deux fois la vitesse pour un seul processeur.

Pour un ordinateur ayant 16 processeurs en parallèle, le gain peut atteindre 2.35 fois la performance pour un seul processeur. En revanche si 100 % du code est parallèle, alors sur un système à 16 processeurs le gain est multiplié par 16. En règle générale plus il y a de proportion du code source tournant en parallèle plus le programme est extensible (scalable).

Granularité des threads

Lorsqu'il s'agit de paralléliser son application, la solution la plus simple serait de décomposer le programme en tâches indépendantes. La décomposition est souvent fonctionnelle. Par exemple, dans un jeu vidéo on peut exécuter le système sonore, l'intelligence artificielle et le rendu sur des threads séparés.
Cette solution est assez répandue, néanmoins, lorsque le nombre de cœurs dépasse le nombre de blocs parallèles, les performances ne seront plus améliorées avec l'ajout d'autres cœurs au système. Une granularité grossière des threads ne permet pas toujours d'augmenter l'extensibilité du programme.
Une granularité fine des threads permet une plus grande extensibilité. On doit créer les threads, assigner à chaque thread la tâche qu'il doit exécuter et finalement terminer leurs exécutions. L'implémentation manuelle de cette méthode peut apporter des gains considérables en termes de performances, mais elle est généralement assez difficile à réaliser.
Encore une fois, la technologie OpenMP peut simplifier les choses en proposant d'automatiser la gestion des threads.

Le couteau suisse du développeur

La migration peut être facilitée à l'aide des outils d'analyse et d'inspection automatique. Ils permettent de détecter plusieurs erreurs classiques en programmation parallèle, telles que les *deadlocks* et *race conditions*. En outre, à l'aide de la visualisation graphique, on peut détecter facilement les goulots d'étranglement dans une application. Avec Intel Parallel Studio, on trouve une gamme d'outils incontournables.

Exemple d'application

Afin de mieux présenter les spécificités des processeurs de nouvelles générations, nous avons choisi de réaliser un simple système de particule. Car il présente un cas typique ou une région du code ralentit l'ensemble du programme.
J'aimerais rappeler qu'un système de particule est une simulation d'un grand nombre de points dans l'espace soumis à un ensemble de forces comme la gravité. Il est souvent utilisé en infographie pour simuler l'apparence de nombreux phénomènes naturels et pour créer des effets spéciaux.
Nous n'avons pas la prétention de créer un système complexe de A jusqu'à Z, mais seulement de retenir les détails les plus importants de la migration.
L'idée de base était de trouver un cas d'application ou il y a un goulot d'étranglement au niveau des calculs effectués par le CPU. Les données relatives aux particules sont conservées dans la mémoire à haute performance du GPU.
Les données sont encapsulées dans des "buffer objects" pour minimiser le transfert de données entre processeurs et carte graphique.
À chaque frame, on calcule la nouvelle position, vitesse et accélération de chaque particule.
Ce temps de calcul est plus important que le temps pris par le rendu des particules par la carte graphique. Pour peu qu'on arrive à optimiser cette partie du code source, la vitesse d'exécution sera nettement améliorée. Mais avant d'entamer l'étape d'optimisation, examinons de plus près notre code pour y déceler d'éventuelles erreurs.La boucle permettant de mettre à jour les particules est sous la forme :

```
 for (ptrdiff_t i = 0;i<particleSystem->getParticleNumber();i++)
   particleSystem->Update(i);
```

Le type « ptrdiff _t» a une taille de 32 bit pour les systèmes 32 bit et 64 bit pour les systèmes 64 bit.
Ce qui le rend idéal pour les index des tableaux. Par ailleurs, il permet d'éviter de sacrifier inutilement de la mémoire pour les systèmes 32 bit. Et peut-être utilisé invariablement, quel que soit le système.
La boucle peut être réécrite de la façon suivante, dans l'intention d'expliciter les instructions de la fonction de mise à jour.

```
void particleSystem::Update()
{
 ptrdiff_t i;
 for (i = 0;i<particleNb;i++)
 {
   ForceZ += sinf(ForceZaxis[i]);
   //L'acceleration est la somme des forces divisée par
   //la masse de la particule.
   particle[i].acceleration = ForceZ / particle[i].mass * ez;
   //On met à jour la vitesse de la particule.
   particle[i].velocity += particle[i].acceleration;
   //On met à jour la position de la particule.
   particle[i].position += particle[i].velocity;
 }
}
```

La fonction de mise à jour pour une particule donnée ne nécessite pas la connaissance de l'état des autres particules. Elles agissent indépendamment et par conséquent on peut imaginer que cette tâche peut être répartie sur plusieurs threads.
Chaque thread mettra à jour un certain nombre de particules. Le problème ne nécessite aucune solution de synchronisation avancée. Sans doute, la solution doit être très simple, dans le cas d'OpenMP elle l'est, mais dans bien d'autres cas, elle nécessite la modification du code.
Pour indiquer au compilateur que la boucle peut être parallélisée, il suffit d'ajouter la directive :

```
pragma omp parallel for shared(particle)
```

Notons au passage que dans le cas où le support d'OpenMP est désactivé, le compilateur ignore tout simplement cette ligne et on obtiendra la version sérielle de notre application.

```
 #pragma omp parallel for shared(particle)
for (i = 0;i<particleNb;i++)
 {
   ForceZ += sinf(ForceZaxis[i]);
   //L'acceleration est la somme des forces divisée par
   //la masse de la particule.
   particle[i].acceleration = ForceZ / particle[i].mass * ez;
   //On met à jour la vitesse de la particule.
   particle[i].velocity += particle[i].acceleration;
   //On met à jour la position de la particule.
   particle[i].position += particle[i].velocity;
 }
```

La granularité des threads est un facteur important lors de la parallélisation de l'application. L'ajustement de la granularité en OpenMP se fait par le biais de la clause : schedule (type, nombre)

```
schedule(static, 100)
```

On indique que la capacité du lot est égale à 100. Par conséquent à chaque thread on affecte 100 itérations.

```
#pragma omp parallel for schedule(static, 100) private(i)
\ shared(particle, ForceZ)
for (i = 0;i<particleNb;i++)
 {
   ForceZ += sinf(ForceZaxis[i]);
   //L'acceleration est la somme des forces divisée par
   //la masse de la particule.
   particle[i].acceleration = ForceZ / particle[i].mass * ez;
   //On met à jour la vitesse de la particule.
   particle[i].velocity += particle[i].acceleration;
   //On met à jour la position de la particule.
   particle[i].position += particle[i].velocity;
 }
```

Considérons à présent l'instruction suivante permettant d'effectuer la somme des éléments d'un tableau :

```
 ForceZ += sinf(ForceZaxis[i]);
```

On peut indiquer à OpenMP d'optimiser l'opération de sommation. En spécifiant le nom de la variable et l'instruction. L'opération de réduction permet de se débarrasser des sémaphores de synchronisation. Dans le code suivant la variable ForceZ ne nécessite pas de primitive de synchronisation grâce à l'opération de réduction.

```
#pragma omp parallel for schedule(static, 100) private(i)
\ shared(particle) reduction(+:ForceZ)
for (i = 0;i<particleNb;i++)
 {
   ForceZ += sinf(ForceZaxis[i]);
   //L'acceleration est la somme des forces divisée par
   //la masse de la particule.
   particle[i].acceleration = ForceZ / particle[i].mass * ez;
   //On met à jour la vitesse de la particule.
   particle[i].velocity += particle[i].acceleration;
   //On met à jour la position de la particule.
   particle[i].position += particle[i].velocity;
 }
```

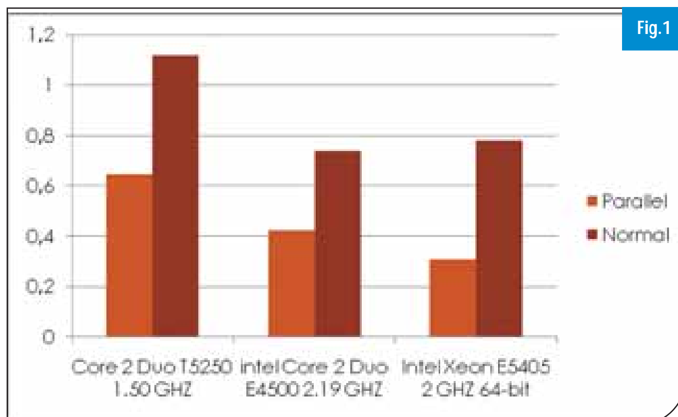

Fig.1

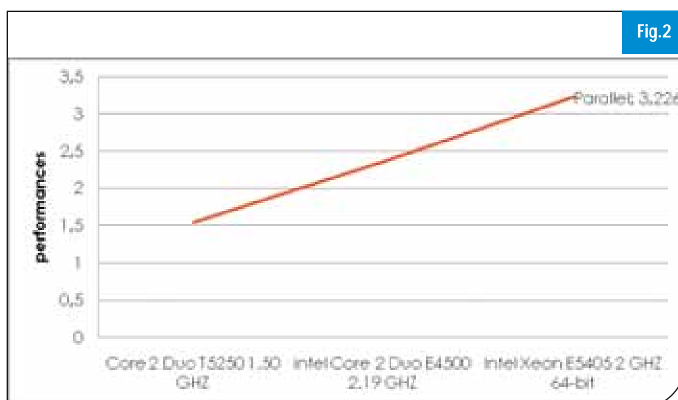

Fig.2

Résultats

Nous avons testé le temps d'exécution de l'application sur différents processeurs.

Nous remarquons une baisse de performance pour la version sérielle de l'application pour le processeur Intel Xeon, malgré qu'il soit le plus puissant des processeurs testés.

La version parallèle de l'application est nettement plus performante, en effet, elle est 2.26 fois plus rapide que la version sérielle pour le processeur Intel Xeon. [Fig.1]

La courbe suivante montre l'extensibilité de notre application. La version parallèle permet en effet d'exploiter la puissance des processeurs testés. [Fig.2]

Conclusion

Nous avons vu dans cet article quelques pièges à éviter lors de la migration d'applications pour les processeurs modernes. Une des technologies clefs pour une migration progressive est la technologie OpenMP. Ceci étant, il reste encore de nombreuses voies à explorer afin d'exploiter au mieux les capacités des ordinateurs modernes et notamment la puissance de calcul du GPU.

Le standard OpenCL comme le fut OpenMP ouvre de nouvelles possibilités aux développeurs pour paralléliser leurs applications.

Références et ressources

http://developer.amd.com/documentation/articles/pages/6220067.aspx

http://software.intel.com/en-us/blogs/2010/02/16/search-of-64-bit-errors-in-array-implementation/

http://software.intel.com/en-us/articles/granularity-and-parallel-performance/

http://msdn.microsoft.com/en-us/magazine/cc300794.aspx

http://msdn.microsoft.com/en-us/library/3b2e7499%28VS.80%29.aspx

http://www.viva64.com/content/articles/64-bit-development/?f=64-bit-migration-7-steps.html&lang=en&content=64-bit-development

■ Ghassen **Hamrouni**

Expert C++ chez IP-Tech : SSII offshore en Tunisie (www.iptech-offshore.com).