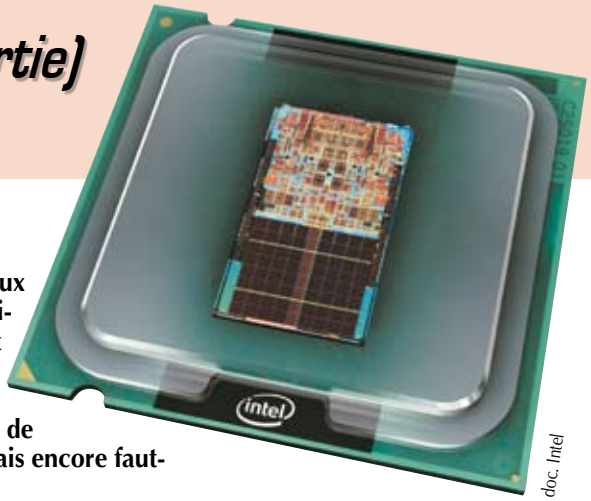
Processeurs multicoeurs et parallélisme (Première partie)

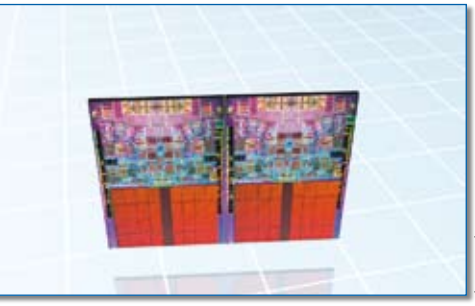


doc. Intel

Les fondeurs de microprocesseurs, depuis la fin de la course aux mégahertz, se sont lancés dans la fabrication de puces dites « multi-cœurs », qui réunissent sur le même « dé » deux, quatre et bientôt huit CPU, partant du principe qu'il était plus facile de gagner de la performance en multipliant les travailleurs plutôt qu'en demandant à un seul d'accélérer jusqu'à la rupture. Jusqu'ici réservé aux machines de haut de gamme, le parallélisme est maintenant à disposition de tous. Mais encore faut-il savoir comment en profiter.

Lorsque le fondeur Intel annonça la fin de son architecture Pentium classique (jusqu'au Pentium III) pour la remplacer par la toute nouvelle architecture dite NetBurst, incarnée par la série Pentium 4, ce dernier voyait déjà des vitesses d'horloge de l'ordre d'une dizaine de gigahertz à l'horizon 2000… Las ! Les difficultés technologiques ont rapidement rappelé à la réalité les ingénieurs un peu trop optimistes, et, dans la pratique, les processeurs de cette série n'ont jamais pu dépasser, à grand peine, les 3 GHz.

Confrontés à cette impasse, les fondeurs ont dû faire marche arrière, et revoir leur stratégie.


doc. Intel

Plutôt que d'obliger les transistors à fonctionner toujours plus vite, et donc à consommer toujours plus *1*, ils se sont tournés vers une technologie déjà connue, puisqu'utilisée dans les processeurs dits RISC *2* : les processeurs « superscalaires », c'est-à-dire regroupant sur le même silicium plusieurs unités d'exécution indépendantes.

C'est ainsi que l'on a vu apparaître chez Intel, pour ne citer que lui (mais son rival AMD n'est pas en reste), la série des processeurs Core™ puis Core 2™, dans des versions simples (dites Solo), doubles (dites Duo) et quadruples (Quattro). En attendant les futures versions à huit CPU. Toutefois, si la multiplication des unités d'exécution constitue une façon élégante de sortir de l'ornière liée à la surconsommation, les améliorations en terme de performance ne sont pas liées de manière simple et directe au nombre de celles-ci. Globalement, un processeur type Core Duo ne va que 1,5 à 1,7 fois plus vite qu'un Core Solo, et pour le Core Quattro l'accélération attendue avoisine les 3 plutôt que les 4.

Et ces chiffres représentent un maximum qu'il est difficile d'atteindre dans la réalité. Pourquoi ? Parce que tirer partie du parallélisme est substantiellement plus complexe que tirer partie d'une accélération d'horloge : dans ce dernier cas, il suffit simplement de modifier le matériel pour qu'il s'adapte au rythme d'exécution plus enlevé (le logiciel demeure inchangé), alors que le parallélisme exige non seulement des modifications matérielles, mais également logicielles.

Dans un premier temps, nous allons évoquer les différentes formes de parallélisme qui existent (on parle de grains), puis nous verrons, dans un second temps, quelles sont les étapes logicielles indispensables lorsque l'on désire tirer pleinement profit de ces architectures superscalaires.

PARALLÉLISME ET GRAIN

Tout comme le riz, le parallélisme se classe par grosseur de grain : on parle ainsi de parallélisme à gros grain ou à grain fin. Le grain représente, ici, la taille élémentaire du code parallélisable. Nous allons commencer par examiner, comme au microscope, le plus gros avant de finir par le plus petit.

Parallélisme à gros grain

On classe dans cette catégorie les logiciels dont l'exécution est parallélisable tout simplement parce qu'ils opèrent, à chaque exécution, sur des données différentes. Le cas typique ici, est représenté par la recherche d'informations sur des moteurs comme Google ; chaque requête, indépendante de toutes les autres, est traitée par la même machine, ou par une autre, en totale isolation : il n'y a aucune subtilité. L'affectation des tâches est opéré (du moins dans le domaine Internet) par

1. La consommation statique d'un processeur en technologie CMOS est quasi-nulle. En revanche, elle augmente en raison du carré de sa fréquence de fonctionnement, un phénomène qui s'explique par les fuites de courant lors des transitions d'état.
2. RISC : Reduced Instruction Set CPU : processeur à jeu d'instruction réduit. Une technologie qui a eu un certain succès dans les années 90, mais ne subsiste maintenant plus que dans les technologies embarquées de type ARM (Acorn RISC Machine). Tous les processeurs type x86 actuels s'en sont inspiré.

un ordinateur frontal appelé en anglais load balancer (répartiteur de charge), qui connaît en permanence le nombre de requêtes que chaque machine traite, et aiguille ainsi les communications entrantes de façon à équilibrer la charge totale moyenne. En dehors de cette opération de répartition, il n'y a aucune communication entre les processus exécutés.

Ce type de parallélisme s'applique essentiellement aux recherches en bases de données, ainsi qu'aux applications en mode ASP. Mais c'est aussi, à un niveau plus matériel, la façon dont opèrent les pilotes de périphériques : chacun surveille un bout du matériel, a priori indépendamment du reste.

Parallélisme à grain moyen

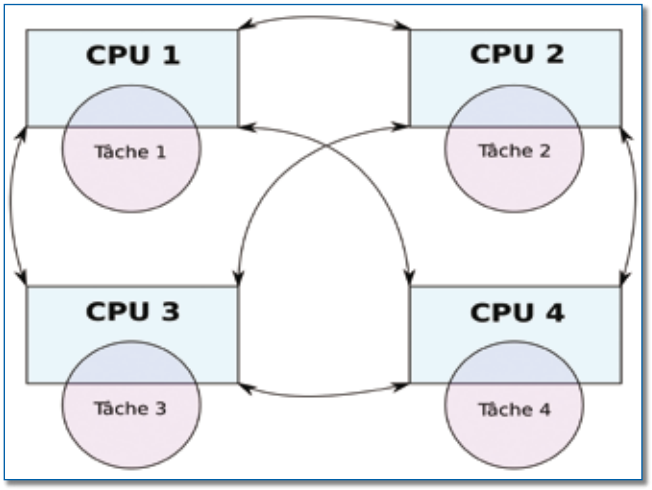
Nous entrons ici véritablement dans le vif du sujet : le parallélisme à grain moyen implique une coordination entre plusieurs routines logicielles qui peuvent s'exécuter en parallèle, mais sont obligées d'échanger des données. C'est le modèle logiciel que l'on appelle en anglais threading, que l'on pourrait traduire par effilochage : le programme principal est constitué d'un ensemble de brins autonomes, mais que l'échange de données tresse pour obtenir un résultat homogène (toron).

Typiquement, ce genre de modèle est appliqué au calcul scientifique, où les problèmes peuvent se décomposer en sous-problèmes plus élémentaires indépendants. Ainsi, la multiplication matricielle de deux matrices carrées à nombre d'éléments pairs :

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

se décompose, si l'on coupe chaque matrice en quatre, en quatre opérations élémentaires, chacune indépendante des autres, qui peuvent donc être exécutées en parallèle. À la fin du calcul simultané, le logiciel exécutant la tâche principale n'a plus qu'à récupérer les résultats et les assembler correctement, avant de continuer.

Pour paralléliser de cette manière le calcul matriciel, on va donc créer quatre tâches identiques, leur communiquer, de la façon la plus efficace, les sous-matrices dont elles ont besoin, les laisser s'exécuter puis récupérer leurs résultats. Ce processus peut-être itératif : chaque sous-tâche se décompose éventuellement en sous-sous-tâche, et ainsi de suite jusqu'à atteindre une opération élémentaire *3*.



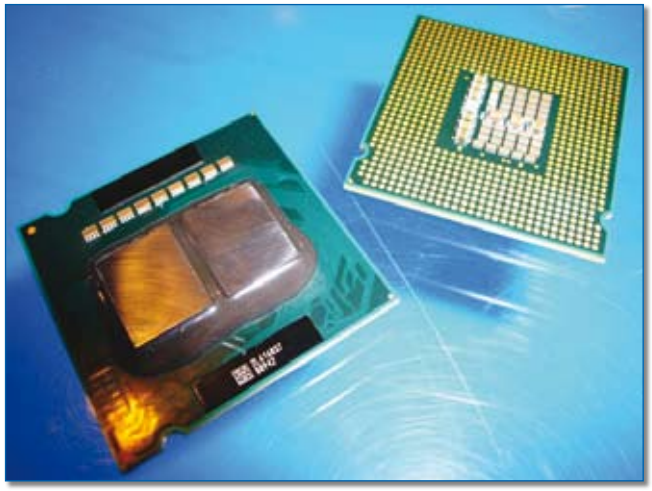
Dans cette figure, on illustre le parallélisme à gros grain. Sur quatre processeurs différents, et pas nécessairement situés sur la même machine, s'exécutent quatre tâches totalement indépendantes. Les processeurs dialoguent entre eux par des primitives de communication qui peuvent faire, ou non, intervenir un réseau de communication.

Un autre exemple : exporter des clichés raster de grande taille vers un format hautement compressé est une opération rarement urgente, mais qui peut parfois se révéler très gourmande en ressource machine. On affecte donc cette opération dans une tâche particulière, qui tourne en fond, de manière autonome, et pendant ce temps le logiciel continue à s'exécuter normalement, sans que l'utilisateur ne soit obligé d'attendre la fin de la création du fichier.

Ces tâches élémentaires, threads ou fils d'exécution, peuvent s'exécuter sur le même CPU, sur le même processeur, sur plusieurs CPU séparés, ou sur une machine différente. Ils sont typiquement créés, sur ordre du programme principal, par le système d'exploitation, qui procure les API abstraites nécessaires.

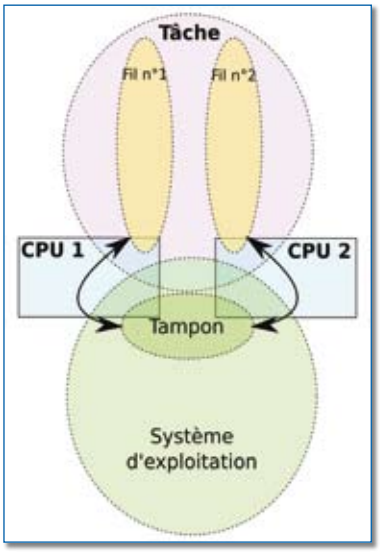
Il revient donc au concepteur de réfléchir sur son logiciel, d'identifier les segments parallélisables, puis d'ajouter le code nécessaire pour créer les fils, les approvisionner en données, puis collecter leurs résultats. En aucun cas ce travail ne peut être pris en charge par les outils de programmation comme les compilateurs, dont la vision est très proche du matériel.

Lorsque les fils sont répartis sur différentes machines (notion de cluster, ou amas de machines), les données sont généralement échangées sur un réseau rapide (Gigabit Ethernet). Pour que le gain soit sensible, il faut que le temps d'échange réseau (majoré du temps requis par le système d'exploitation pour mettre en place l'environnement d'exécution) ne représente qu'une fraction négligeable du temps d'exécution de chaque fil : il n'est par conséquent pas rentable de poursuivre le découpage en sous-tâches au-delà d'un certain seuil dépendant de l'architecture (matériel + système d'exploitation).



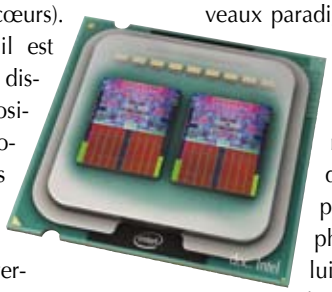
doc. Intel

3. Ce principe, appelé en anglais *Divide-and-conquer*, appliqué à la multiplication matricielle, est à l'origine de la méthode dite de *Strassen*, qui permet de limiter le nombre d'opérations à $n^{2,7}$ alors que la méthode classique (celle que l'on apprend en classe) demande n^3 opérations, où n est l'ordre de la matrice (supposée ici carrée).



Dans cette figure, une tâche donnée a créé deux fils distincts, qui s'exécutent chacun sur un processeur. Ces fils sont physiquement indépendants, ils ne partagent pas de ressources. Ils dialoguent au travers de primitives système de synchronisation et/ou d'échanges, qui mettent en jeu de la ressource mémoire système.

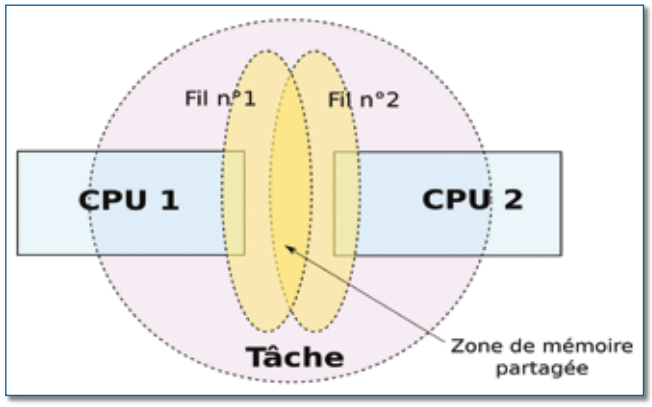
Lorsque tous les fils s'exécutent sur la même machine, la communication entre les tâches peut se faire soit par l'intermédiaire de primitives système spécialisées, soit au travers de zones de mémoire partagées, si la machine le permet (c'est typiquement le cas des processeurs multicœurs). Dans ce cas, il est nécessaire de disposer de dispositifs de synchronisation appelés sémaphores. Ces derniers permettent de verrouiller les zones d'échange, de sorte à assurer la cohérence des données (par exemple, éviter qu'une lecture ne soit interrompue par une mise à jour, auquel cas la lecture lirait une partie de données anciennes et une partie de données nouvelles). Le principe de la zone mémoire d'échange est particulièrement efficace si cette dernière n'est pas trop importante, ce qui lui permet de tenir dans un cache mémoire commun aux processeurs, cache dont les temps d'accès sont beaucoup plus rapides que ceux de la mémoire traditionnelle.



L'affectation des fils aux différents processeurs de la machine est une tâche dévolue à une routine système particulière, le scheduler ou ordonnanceur. À chaque quantum de temps machine (généralement 10 ms), ce bout de code prend la main, enregistre le contexte courant, parcourt une table des tâches (fils) et choisit quelle sera la prochaine qui sera exécutée durant le quantum suivant, et par quel processeur.

Ce dernier représente donc une sorte de répartiteur de charge, à ceci près qu'il est possible de le contrôler plus finement, par exemple en affectant une priorité différente à chaque fil : le fil le plus prioritaire sera exécuté plus fréquemment, et donc plus vite, que les fils de priorité inférieure.

Font également partie de ce type de parallélisme les nouveaux paradigmes CPU/GPU de type OpenCL ou Cuda, où le programmeur choisit d'envoyer sur le processeur graphique, composé lui-même de (parfois) plusieurs centaines de calculateurs flottants élémentaires des tâches complexes : des primitives particulières, intégrées au pilote graphique, se chargent de copier les données et le code vers la mémoire vidéo propre au GPU, puis de programmer son séquenceur interne pour effectuer les opérations demandées. Enfin, un système de sémaphore



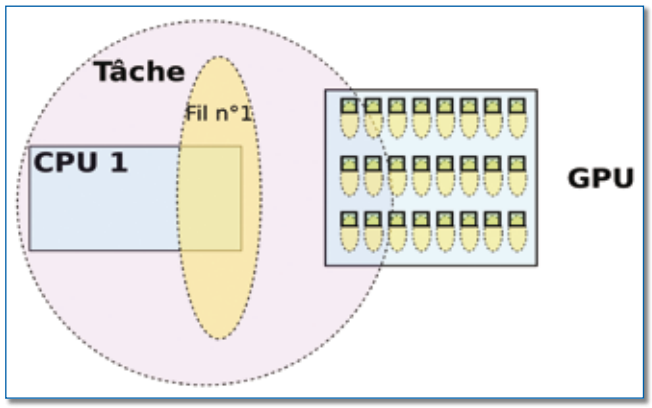
Même situation que dans la figure précédente, mais ici, les deux fils partagent des ressources mémoires (par exemple sous forme de variables). Ils peuvent donc s'échanger des informations et se synchroniser directement, sans faire appel aux services proposés par le système d'exploitation.

prévient de la fin de l'opération parallèle décentralisée. Dans ce cas, l'ordonnanceur n'intervient pas, puisque les fils sont créés dans une mémoire à part et exécutés par un processeur spécialisé.

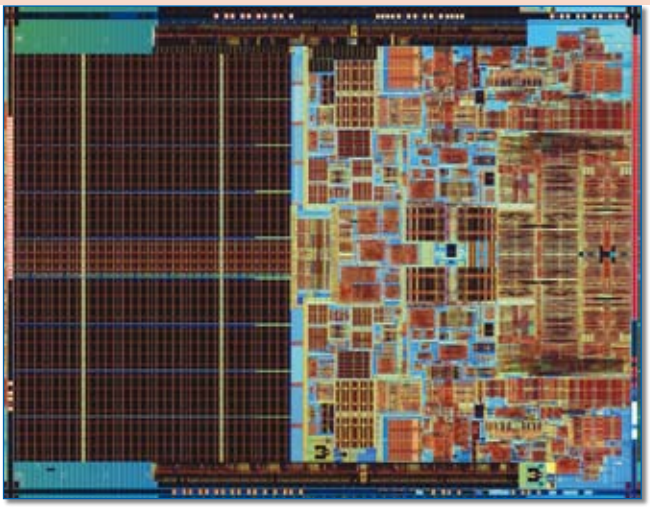
Parallélisme à grain fin

Après avoir survolé le parallélisme à gros grain (échelle du processus), à grain moyen (échelle du fil d'exécution), nous descendons maintenant au niveau le plus élémentaire, celui des instructions processeur. Chaque CPU d'un processeur, même multicœur, est lui-même composé de plusieurs unités d'exécution qui fonctionnent simultanément, et qui partagent toutes les ressources mémoire. Cas typique, un processeur Core 2 est capable d'exécuter deux instructions « entières » (comprendre : qui travaillent sur des registres internes de type entier), plus une instruction « flottante », qui met en jeu des nombres décimaux.

Cependant, pour que cette machinerie très complexe puisse fonctionner à plein régime, il faut s'assurer que les instructions consécutives aient le moins d'interdépendance possible. Pourquoi ? Supposons que notre CPU, qui dispose de deux unités d'exécution U1 et U2, ait à exécuter le code suivant :



Les nouveaux processeurs graphiques des principaux fondeurs (Nvidia/ATI) comportent tous des unités de calcul d'ombrage qui sont également exploitables en tant qu'unités flottantes généralistes. Il est ainsi possible de lancer des travaux de calcul massivement parallèles sous forme de petits fils exécutés indépendamment par chaque calculateur graphique élémentaire.



doc. Intel

1. Lire A
2. Ajouter 1 à A
3. Stocker A
4. Lire B
5. Ajouter 2 à B
6. Stocker B

On conçoit aisément qu'il est impossible d'exécuter les instructions 1 et 2 simultanément : si l'unité d'exécution U1 exécute 1, U2 ne peut exécuter 2 puisque 1 n'est pas terminée et la valeur de A n'est donc pas connue. Idem, quand 2 s'exécute (ensuite) sur U1 ***4***, U2 ne peut pas non plus exécuter 3, seul U1 pourra le faire au coup suivant. Au total, U2 est restée totalement inactive pendant deux cycles. En revanche, 3 et 4 sont deux instructions indépendantes qui peuvent donc tourner en parallèle (dans la mesure où le système d'accès à la mémoire le permet), l'une sur U1 l'autre sur U2. Mais, une nouvelle fois, 5 et 6 ne pourront se dérouler ensemble. Au total, il nous faudra donc cinq cycles élémentaires pour exécuter six instructions sur les deux unités d'exécution, là où l'on aurait attendu 6÷2 = 3 cycles.

Y a-t-il un moyen d'atteindre ce résultat idéal. Oui ; il suffit pour cela de réorganiser les instructions du code ainsi :

1′ : Lire A
2′ : Lire B
3′ : Ajouter 1 à A
4′ : Ajouter 2 à B
5′ : Stocker A
6′ : Stocker B

Dans cette nouvelle organisation, deux instructions consécutives sont indépendantes (elles ne portent pas sur les mêmes données) : 1′ et 2′ peuvent donc être exécutées en même temps sur U1 et U2, au coup suivant, ce sera le tour de 3′ et 4′, puis enfin de 5′ et 6′.

Nous avons donc réussi, en trois cycles, à dérouler les six instructions.

En règle générale, rares sont les programmeurs qui descendent au niveau de l'assembleur pour écrire leurs logiciels. Heureusement, ces principes d'interdépendance sont assez bien modélisés et implémentés dans les compilateurs modernes. Ainsi, pour réaliser la séquence C suivante :

a ++ ; b += 2 ;

tous les compilateurs modernes choisiront directement l'agencement optimisé plutôt que la transcription « littérale » du code source.

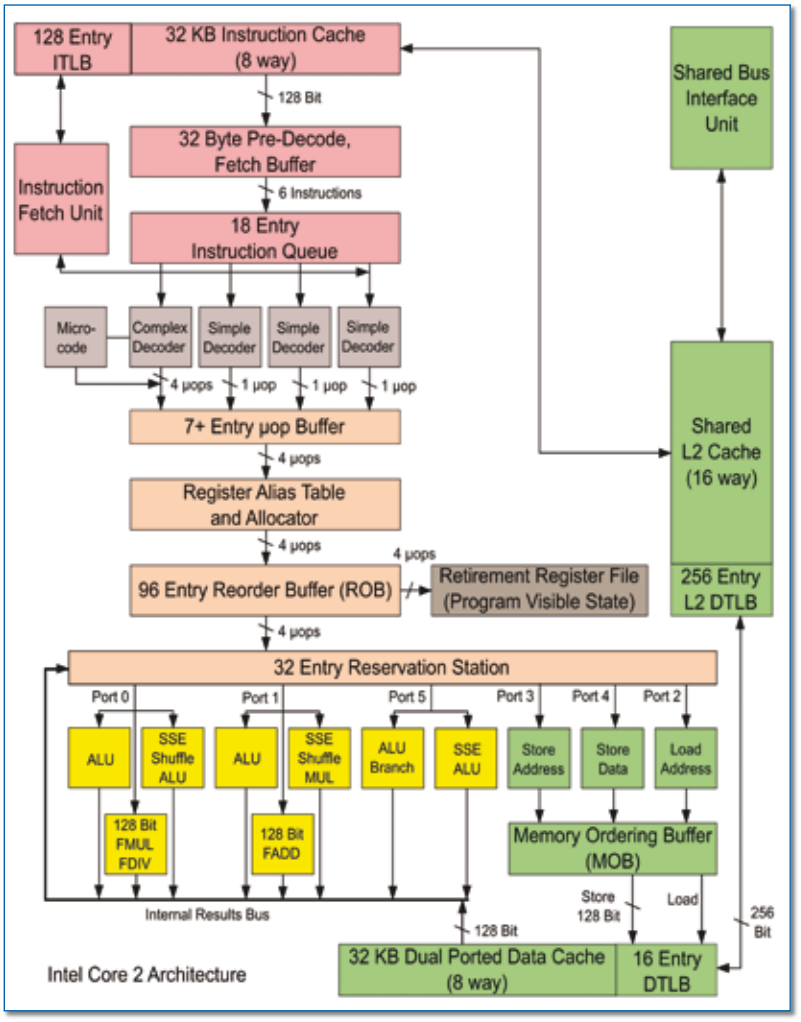
Malgré tout, un tel concept à ses limites. D'une part, il est impossible de réorganiser tout un programme pour que les instructions soient indépendantes deux à deux. Exemple :

1. Charger A
2. Charger B
3. Si A > 10 alors B = 1
4. Incrémenter B
5. Sauver B

1 et 2 peuvent s'exécuter ensemble, mais dès que l'on rencontre un test qui réalise une opération conditionnelle, il faut nécessairement attendre son résultat avant de poursuivre ***5***. Comme 3, 4 et 5 sont dépendantes, un processeur à deux unités d'exécutions ne fait guère mieux qu'un processeur simple.

En outre, multiplier les unités d'exécution n'a guère de sens au-delà de deux. D'une part, il faut pouvoir récupérer simultanément en mémoire autant d'instructions qu'il y a d'unités et, d'autre part, il devient très difficile de réorganiser le code quand le nombre d'unités augmente (sans évoquer la multiplication de circuits de synchronisation matérielle ***6*** sur la puce destinés à gérer les accès simultanés aux ressources par les différentes unités).

A suivre dans le prochain numéro.



Architecture des processeurs x86 de la série Core™.
Source Wikipedia/Appaloosa Licence GPL.

4. En toute rigueur U1 ou U2. Mais cela ne change rien à l'exposé.
5. Les processeurs modernes possèdent des unités d'exécution dites spéculatives, qui choisissent *a priori* une branche (généralement la plus fréquente ou probable) et l'exécutent avant de connaître le résultat du test, quitte à revenir en arrière s'il s'avère que le choix était mauvais. Cela permet de gommer, dans une certaine mesure, le temps perdu à attendre que la condition soit évaluée.
6. Appelés scoreboards ou tableaux noirs.