Chapitre 2

Architectures à haute performance

Ce chapitre décrit les architectures des ordinateurs conçus dans le but d'obtenir de grandes performances. Il s'agit bien sûr des architectures parallèles comme les machines SIMD et MIMD, mais aussi des ordinateurs vectoriels qui ont joué un rôle important dans le développement du calcul scientifique à haute performance. On mentionnera aussi les solutions architecturales adoptées dans les microprocesseurs modernes, notamment le parallélisme au niveau des instructions (ILP) qui est une caractéristique des processeurs superscalaires.

D'autres types d'architectures seront aussi brièvement discutées ici. C'est le cas notamment des machines *Dataflow* qui fonctionnent selon un principe différent des architectures von Neumann.

Nous commencerons ce chapitre en parlant de la classification de Flynn des architectures parallèles et nous passeront en revue les differentes architectures à haute performance. Un élément architectural important sera discuté en détail au chapitre 3. Il s'agit des réseaux d'interconnexion qui permettent de relier les processeurs entre-eux. Dans ce chapitre nous nous contentrons d'une vision simplifiée dans laquelle les processeurs sont interconnectés entre-eux soit à travers des câbles et des routeurs, soit à travers un switch.

2.1 Classification de Flynn

Depuis les années 70, les architectures ont été classées selon le diagramme de Flynn, en fonction de la multiplicité du flot de données et flot d'instructions. Si plusieurs données son traitées en parallèle par le système, on dira que le flot d'instruction est multiple. Sinon, il est unique (single, en anglais). De même, si plusieurs flot d'instructions sont générés simultanément par les différents processeurs, on dira que ce flot est multiple.

Le modèle séquentiel (ou de von Neumann) correspond à un flot d'instructions et de données unique. Dans la taxonomie de Flynn, l'architecture séquentielle s'appelle donc SISD (Singl Instruction flow, Single Data flow).

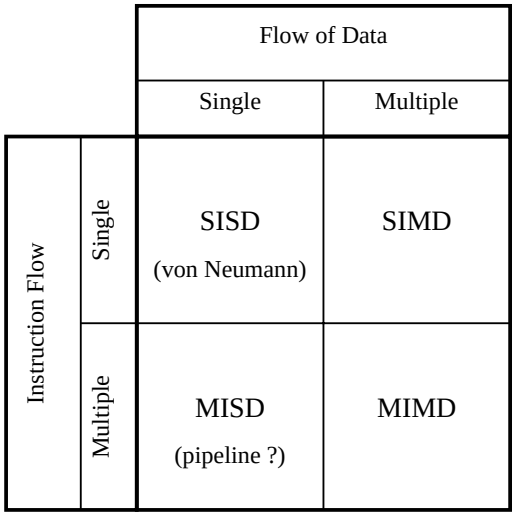

FIGURE 2.1 – *Diagramme représentant la classification de Flynn*

On a ensuite l'architecture SIMD (Single Instruction flow, Multiple Data flow) qui est l'architecture parallèle la plus simple. Elle est décrite en détail dans le paragraphe 2.2.

L'architecture MIMD (Multiple Instruction flow, Multiple Data flow) est la catégorie qui contient la grande partie de machines parallèles actuelles. Chaque processeur exécute en effet un flot d'instruction potentiellement different des autres.

Finalement, la catégorie MISD n'a pas de représentant parmi les machines qui ont été construites : elle consisterait à avoir plusieurs instructions simultanées qui traitent la même donnée. Cependant, certaines personnes ont voulu voir sous l'acronyme MIDS, le modèle de l'architecture "pipeline". Cette intreprétation est controversée.

2.2 Architecture SIMD

Actuellement, les architectures SIMD ne sont plus envisagées comme une architecture parallèle d'usage général. Les progrès technologiques permettent aujourd'hui de construire des machines MIMD offrant plus de souplesse et adaptées à un plus grand nombre d'applications. Néanmoins, les architectures SIMD restent très efficaces dans certains problèmes, et cela peut justifier leur fabrication pour un domaine d'application particulier ayant des besoins de calculs spécifiques, comme par exemple le traitement d'images ou du signal. Ainsi, certaines parties des processeurs modernes et des cartes graphiques fonctionnent sur le principe SIMD afin d'optimiser certains calculs répétitifs.

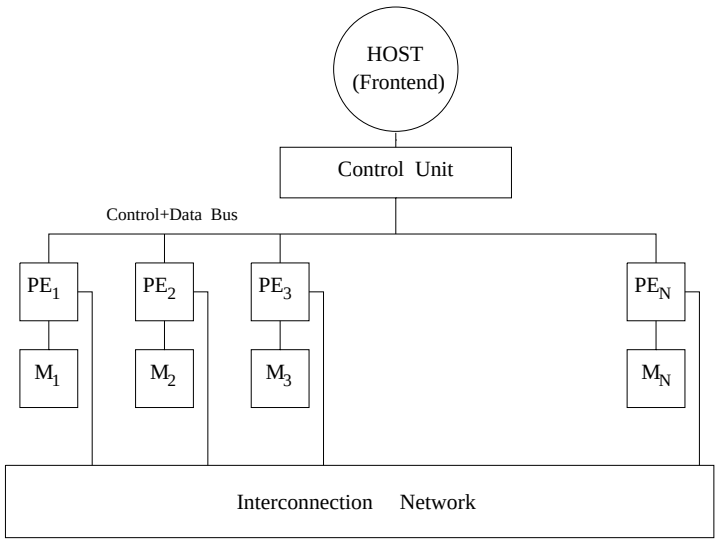

FIGURE 2.2 – *Schéma élémentaire d'une architecture SIMD.*

2.2.1 Architecture de base

Dans la classification de Flynn, l'architecture SIMD (Single Instruction, Multiple Data) se réfère à un système composé de plusieurs processeurs interconnectés, exécutant tous la *même* instruction au même moment, mais chacun sur des données qui peuvent être différentes. Ces instructions proviennent d'un programme **unique**, et l'exécution se fait de manière **synchronisée** par l'ensemble des PE (abréviation de *Processing Elements* dénotant les processeurs de calcul d'une machine parallèle).

A première vue, le modèle de calcul SIMD peut paraître quelque peu restrictif. A-t-on vraiment besoin d'avoir plusieurs processeurs qui effectuent la même tâche au même instant ? Il s'avère en fait que beaucoup de problèmes pratiques se décomposent de manière naturelle en un grand nombre d'opérations identiques qui peuvent s'effectuer simultanément sur des données différentes : c'est le parallélisme de données qui offre en général un degré de parallélisme important et de bonnes perspectives de gain en performance.

Du point de vue technologique, l'architecture SIMD a le grand avantage qu'elle reste simple dans son principe, grâce à la synchronisation des différents processeurs. Ces derniers sont connectées entre eux par un réseau d'interconnexion le plus souvent statique, comme par exemple une grille ou un hypercube. Un schéma type de machine SIMD est montré sur la figure 2.2.

Chaque processeur dispose en général d'une mémoire locale qui lui est propre et dans laquelle les données qu'il faut traiter sont conservées. Il s'agit donc d'une architecture à mémoire distribuée. Les processeurs d'une machine SIMD sont contrôlés par l'unité de contrôle (appelée CU sur la figure 2.2). Cette unité est

chargée de transmettre, à travers un bus de contrôle, les instructions qui seront exécutées simultanément par les PE. Le CU envoie (broadcast) aussi l'adresse dans la mémoire locale des PE, où les données relatives à l'instruction sont placées. De plus, les données scalaires sont aussi transmises aux PE par le CU. La liaison CU-PE doit donc être très rapide.

L'unité de contrôle est à son tour reliée à une machine hôte, en général un ordinateur à architecture classique (le frontal ou "front end"). Typiquement, le programme utilisateur réside dans le frontal et c'est là qu'il est exécuté. Alors que les instructions parallèles sont transmises aux PE par l'intermédiaire du CU, les parties scalaires du code sont entièrement exécutées par le frontal. Le programme utilisateur n'est donc pas décodé par les PE, ni stocké dans les mémoires locales.

Le type de processeur qui va constituer une architecture SIMD peut varier d'une machine à l'autre et va dépendre de l'utilisation à laquelle l'ordinateur est destiné. Ce sont typiquement des processeurs simples, comparables à des unités arithmétiques et logiques. Ils disposent en général d'un mécanisme leur permettant d'être actifs ou inactifs, c'est à dire de participer ou non à une opération dictée par le CU. Ce choix est déterminé dynamiquement, selon l'état courant des données placées en chaque processeur. De plus, dans les machines perfectionnées, les PE ont la possibilité de faire de l'adressage indirect dans leur mémoire locale, c'est à dire de travailler sur une donnée stockée à une position mémoire contenue à l'adresse unique spécifiée par le CU.

Clairement, le parallélisme SIMD, tel qu'il à été mis en oeuvre par les constructeurs, est un parallélisme à grain fin, avec un grand nombre de processeurs simple. Les machines SIMD les plus célèbres sont : l'ILLIAC IV (64 PE, années 70), le DAP (4096 PE, années 80-90), la Connection Machine CM-2/200 (jusqu'à 65536 PE, fin 80 à début 90) et la MasPar (jusqu'à 16384 PE, début années 90).

2.2.2 Communications

Les communications se font grâce aux routeurs plus ou moins complexes qui sont associés à chaque PE. En général, les routeurs sont pilotés directement par les PE mais on peut aussi avoir un contrôle du réseau d'interconnexion par l'unité de contrôle. On peut aussi avoir un bus de données qui permet un accès direct du CU aux mémoires individuelles. On distingue en général trois types de communications dans un système SIMD : locales, générales et globales. Ces communications sont faites de manière synchrone, en suivant pas à pas l'algorithme de routage approprié. Aucun calcul ne peut être réalisé par un processeur tant que la dernière donnée n'est pas arrivée à destination.

Les **communications locales** concernent le transfert d'information entre processeurs plus proches voisins. Elles sont très fréquentes pendant l'exécution de la plupart des algorithmes parallèles et utilisent des liens directs (fils de connexions entre processeurs) pour être réalisées.

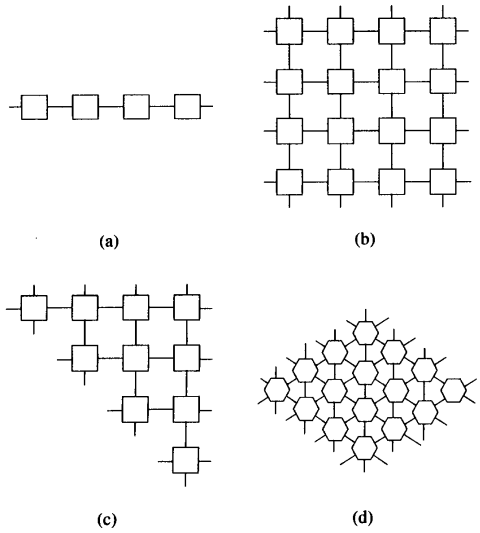

FIGURE 2.3 – *Exemples de réseaux systoliques.*

Les **communications globales** permettent d'obtenir une information globale sur le contenu de l'ensemble des processeurs d'une machine parallèle. Par exemple, il est courant de vouloir obtenir la somme des valeurs contenues dans chaque processeur. Ce sont des opérations collectives qui nécessitent un mécanisme global (mais régulier) de communication par lequel les informations contenues dans chaque processeur sont combinées et, si nécessaire, transmises à l'ordinateur frontal.

Finalement, les **communications générales** permettent des échanges quelconques entre processeurs, sans respecter un schéma de communication régulier ni prévisible. Les communications générales font appel aux mécanismes de routage des messages à travers les liens physiques du réseau. Ce type de communication est important pour assurer un maximum de souplesse aux architectures SIMD. En revanche, il est moins performant car plus général.

2.2.3 Réseaux systoliques

Un réseau systolique est une architecture SIMD de type particulier. Il consiste en un ensemble régulier de cellules simples qui sont capables d'exécuter un nombre restreint d'opérations telles des additions et multiplications. De plus, chaque cellule dispose de quelques registres internes qui font office de mémoire temporaire. Les cellules adjacentes sont typiquement interconnectées, comme le montre la figure 2.3 (cf Kumar, fig 12.1, p. 492).

Ces cellules travaillent en parallèle, de manière synchrone, comme pour une machine SIMD classique. Leur utilisation est toutefois beaucoup plus restrictive et dédiée à un algorithme spécifique. En particulier, les communications se font

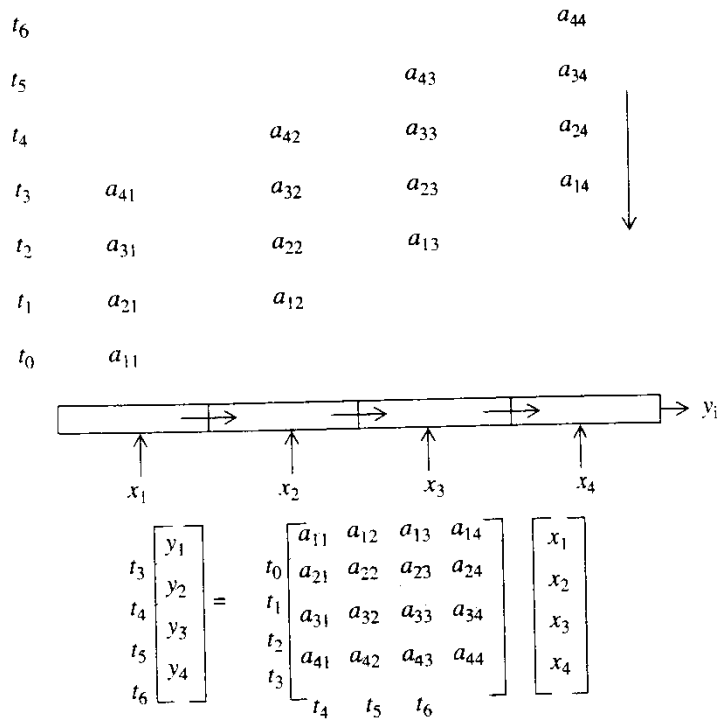

FIGURE 2.4 – *Multiplication systolique matrice-vecteur sur un réseau linéaire à quatre cellules.*

de manière très régulière et selon une structure bien précise qui correspond à l'algorithme mis en oeuvre. Le mot systolique a d'ailleurs été choisi pour décrire le flot rythmique des données à travers les cellules.

Les algorithmes systoliques sont nombreux, notamment dans le traitement de signaux et les multiplications matricielles. La particularité de ces algorithmes est que les données sont injectées par les bords du réseau systolique, puis se combinent entre elles à mesure qu'elles traversent chaque cellule. Il est donc essentiel de synchroniser le passage des données pour que les valeurs adéquates se rencontrent au bon endroit et au bon moment. La figure 2.4 (Moldovan fig. 4.28, p. 221) illustre le principe du calcul systolique pour une multiplication matrice-vecteur dans le cas d'un réseau systolique linéaire à quatre cellules.

Les coefficients a_{ij} de la matrice sont injectés dans le réseau à chaque cycle, de la manière indiquée dans la figure 2.4. Au premier cycle, a_{11} rencontre x_1 dans la première cellule, où ils sont multipliés. Au deuxième cycle, d'une part a_{21} est multiplié avec x_1 dans la première cellule et, d'autre part, le résultat précédemment obtenu arrive dans la deuxième cellule, juste au moment où a_{12} et x_2 sont combinés. Le résultat $a_{11}x_1 + a_{12}x_2$ est alors transmis à la troisième cellule où le processus continue. Au cinquième cycle, la valeur de y_1 est obtenue à la sortie du réseau. On remarque que ce processus de calcul est efficace lorsque

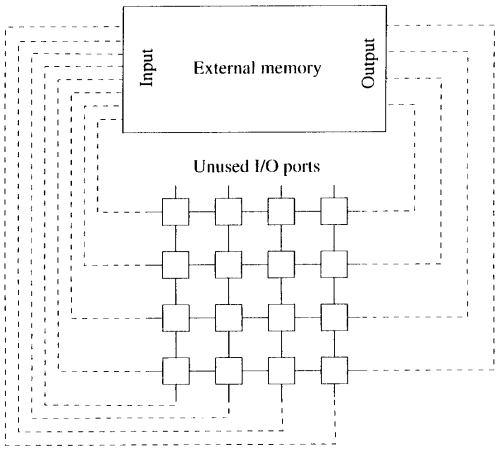

FIGURE 2.5 – *Connexions d'un réseau systolique avec l'extérieur.*

toute les cellules sont actives avec des données.

Le calcul systolique est donc essentiellement du calcul pipeline généralisé à des réseaux de processeurs. Seuls les bords du réseau sont reliés au monde extérieur, comme l'indique la figure 2.5 (cf Kumar, fig 12.2, p 493). Il s'agit toutefois d'un pipeline de données et non d'instructions, comme dans le cas des processeurs vectoriels, puisque c'est la même instruction qui est exécutée par chaque cellule.

Les réseaux systoliques sont apparus dans les années 80 lorsque les progrès dans la technologie des semi-conducteurs ont permis, grâce à l'intégration VLSI, de concevoir des circuits mettant en oeuvre des algorithmes directement au niveau du hardware.

Le calcul systolique est recommandé pour les problèmes impliquant des calculs intensifs. Les temps de communication, en raison de la spécificité des échanges de données et de l'implantation directe dans le silicium, sont souvent faibles ou négligeables.

Les algorithmes systoliques s'adaptent naturellement sur les machines SIMD. Cependant, dans ce cas, les communications sont des facteurs dont il faut tenir compte, de même que le mapping des données sur les processeurs.

2.3 Architecture MIMD

Un ordinateur MIMD (Multiple Instruction Multiple Data) se caractérise par un ensemble de processeurs pouvant communiquer de manière efficace et disposant chacun de son propre flot d'instructions (programme) qui agit sur des données qui peuvent différer d'un processeur à l'autre. Le modèle MIMD offre donc une programmation beaucoup plus riche que les machines SIMD qui n'exécutent qu'une seule et même instruction à la fois. Une autre différence importante entre ces deux architectures est que les machines MIMD sont en général asynchrones,

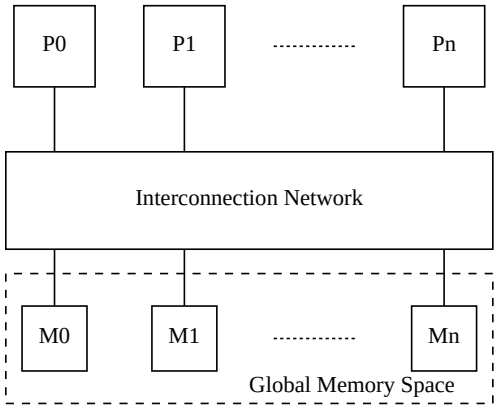

FIGURE 2.6 – *Schéma type de l'architecture à mémoire partagée.*

ce qui implique que l'état d'avancement du travail dans un processeur n'est pas prévisible par la connaissance de ce qui se passe dans un autre processeur. Deux exécutions successives d'un même programme peuvent ainsi se comporter différemment et on parle souvent de comportements **non déterministes**.

Il y a deux principales classes d'architecture MIMD : les machines à **mémoire partagée** et celles à **mémoire distribuée**. À cela vient s'ajouter les architectures à mémoire partagée **virtuelle** (NUMA et COMA) qui cherchent à concilier les avantages des deux approches : scalabilité des architectures distribuées et simplicité de programmation des systèmes à mémoire partagée.

2.3.1 Systèmes à Mémoire Partagée

Les architectures à mémoires **partagées** offrent un espace d'adresse uniforme et un accès direct à l'ensemble de la mémoire à partir de n'importe quel processeur présent. Cette architecture est représentée sur la figure 2.6. Un ordinateur à mémoire partagée est souvent dénommé **multiprocesseur**.

La mémoire se compose typiquement de plusieurs bancs, symétriquement accessibles par tous les processeurs, comme l'indique la figure 2.6. Ces différents modules de mémoire sont connectés aux processeurs à travers un réseau d'interconnexion dynamique ou commutateur (switch) dont la nature est typiquement celle d'un crossbar ou d'un réseau multiétages (voir chapitre 3).

Mais la mémoire partagée la plus simple et la moins onéreuse se base sur une architecture à **bus partagé**, à travers lequel les différents processeurs accèdent à une mémoire commune. Cette solution est typiquement adoptée dans les systèmes comportant peu de processeurs.

Les SMP (Symmetric MultiProcessors) représentent l'architecture parallèle la plus courante et la plus répandue. Les SMP ont connu un succès commercial important mais pas tellement pour leur capacité à travailler en parallèle (c'est à dire à accélérer l'exécution d'un programme unique) que par le fait qu'ils permettent

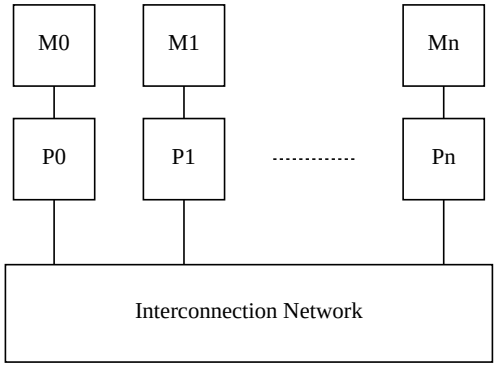

FIGURE 2.7 – *Schéma type de l'architecture à mémoire distribuée. Les processeurs ont chacun leur propre mémoire et sont connectés par un réseau d'interconnexion rapide. Chaque processeur dispose d'une unité de contrôle et peut exécuter un programme différent de son voisin.*

de mettre en oeuvre de la vraie multi-programmation. Le multi-tâches permet d'augmenter la productivité (le "throughput") en offrant à plusieurs utilisateurs la possibilité d'exécuter simultanément leurs programmes, dans un environnement où les ressources sont naturellement partagées.

2.3.2 Architecture à Mémoire Distribuée

Dans les systèmes à mémoires **distribuées** la mémoire est répartie sur chaque processeur sous forme de blocs mémoire locaux et privés, comme indiqué sur la figure 2.7. L'accès mémoire n'est possible que pour le processeur détenteur. Les communications se font exclusivement par des protocoles d'échange de messages (*message passing*) au cours desquels un processeur envoie, à travers le réseau, une de ses données propres à un autre processeur. Un ordinateur à mémoire distribuée est aussi qualifié de **multi-ordinateurs** ou **multicomputer** car il ressemble à une réunion de plusieurs ordinateur indépendants, couplés par un réseau.

Les MPP et Beowulfs

Une architecture à mémoire distribuée qui a été conçue dès le début pour un parallélisme à couplage fort et qui utilise de nombreux composants dédiés s'appelle aussi MPP (Massively Parallel Processor). La vitesse des échanges entre les processeurs est particulièrement importante (latence faible, bande passante élevée) ainsi que le degré d'élaboration des primitives de communications.

Les MPP s'opposent aux systèmes appelés **Beowulf**. Ces derniers représentent une classe d'architecture à mémoire distribuée caractérisée par un excellent rapport coût/performance. Ce sont typiquement des machines faite à partir de composants du marché, tels des PC interconnectés par un réseau fast-ethernet. Le

logiciel est lui aussi largement répandu et en général du domaine publique : Linux avec des outils GNU et MPICH ou PVM pour les échanges de message.

Beowulf est le nom d'un projet du milieu des année 90, visant à produire une machine à haute performance avec un budget modeste. L'origine du nom Beowulf provient d'un héro de la littérature anglaise du 8eme siecle qui libéra les populations d'un monstre qui les terrorisait. Dans le cadre du projet, l'analogie vient du fait que la machine plannifiée allait libérer les scientifiques de la tâche oppressante de continuellement porter leurs applications sur de nouvelles architectures à haute performance.

Il n'y a pas de définition précise quant à l'architecture d'un Beowulf si ce n'est que c'est une architecture à mémoire ditribuée. La topologie d'interconnexion est choisie selon les composants du marché disponibles et le budget à disposition.

La réalisation la plus courante comprend des PC reliés par un giga-switch ethernet. Ces switches sont actuellement courants et pour les systèmes avec beaucoup de processeurs, plusieurs switches peuvent être mis en cascade. Mais d'autres solutions sont aussi envisagées, comme des liaisons par des protocoles plus performants que TCP/IP. Myrinet est un exemple de technique d'interconnexion qui utilise le wormhole pour transmettre les messages entre les noeuds. Infiniband est une autre technologie d'interconnexion qui est bien utilisée.

Actuellement, un système Beowulf se rapproche de plus en plus d'un MPP par ses performances CPU et la richesse du logiciel. Cependant, le Beowulf reste souvent limité par ses capacités de communication s'il n'utilise pas des techniques de communication rapides.

2.3.3 Mémoire Partagée Virtuelle

Les systèmes à mémoire partagée **virtuelle**, cherchent à concilier les avantages des deux solutions "mémoire partagée" et "mémoire distribuée", notament la scalabilité des derniers et l'espace mémoire unique des premiers.

La mémoire est physiquement distribuée mais des mécanismes hardware permettent un accès à la totalité de l'espace des adresses mémoires, de manière plus au moins transparente pour l'utilisateur. L'architecture dite COMA (Cache Only Memory Access) offre une mise en oeuvre d'une mémoire partagée virtuelle en utilisant une architecture uniquement à base de mémoires caches. Elle est décrite sur la figure 2.8.

Dans un système COMA, chaque processeur dispose d'une mémoire cache locale, de taille comparable à une mémoire centrale. Ces mémoires sont appelées mémoires cache car elles sont associatives et donc adressables par le contenu. Le principe de base est que les données n'ont pas d'emplacement fixe. Elles n'appartiennent à aucun processeur en particulier. Au contraire, elles "flottent" d'une mémoire cache à l'autre, selon les processeurs qui les demandent. Alors que dans un cache usuel, les données qui s'y trouvent proviennent de la mémoire centrale,

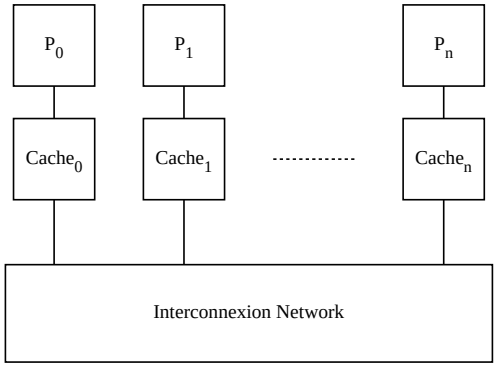

FIGURE 2.8 – *Schéma type de l'architecture COMA (Cache Only Memory Access).*

ici, elles proviennent d'un autre cache. Il y a donc des possibilités de **migration** des données.

L'adresse physique est la clé d'accès des données et la gestion des modules caches est réalisée au niveau du hardware par un dispositif particulier. Notamment, une question primordiale est la **cohérence des données** dans les caches car une même information peut être répliquée plusieurs fois. Un protocole (p.ex. le protocole MESI) implanté au niveau matériel, garantit que les modifications des données se font de façon consistante par les différents processeurs du système.

Il faut remarquer que ce problème de cohérence des données est aussi présent dans les architectures à mémoire partagée dans lesquelles chaque processeur dispose d'une mémoire cache traditionnelle. En effet, dans ce cas aussi, une même donnée peut exister dans plusieurs processeurs qui pourraient potentiellement la modifier de façon inconsistente.

On trouve aussi des architectures dites NUMA (Non Uniform Memory Access). L'accès aux mémoires est **non-uniforme** en ce sens que le temps nécessaire pour lire ou écrire en mémoire varie s'il s'agit d'une mémoire locale ou non. L'architecture NUMA s'apparente beaucoup aux systèmes à mémoire distribuée, si ce n'est qu'elles offrent des mécanismes hardware de lecture et écriture dans les mémoires locales des autres processeurs (Remote-fetch). Dans le cas d'une machine purement à mémoire distribuée, ces mêmes transferts se feraient explicitement par échange de messages. L'architecture NUMA est schématisée sur la figure 2.9.

2.3.4 Architectures hybrides

Une autre classe d'architectures parallèles est le modèle hybride combinant hiérarchiquement, au niveau matériel, les aspects mémoires partagées et mémoires distribuées. Il s'agit de ce qu'on appelle communément des clusters de SMP ou

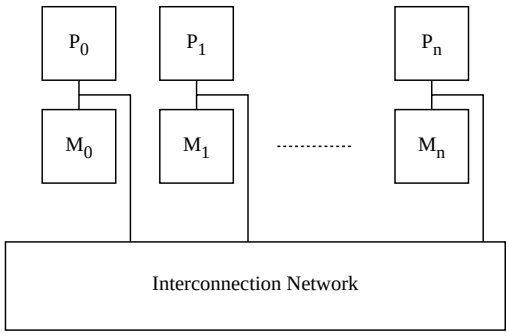

FIGURE 2.9 – *Schéma type de l'architecture NUMA (Non Uniform Memory Acces).*

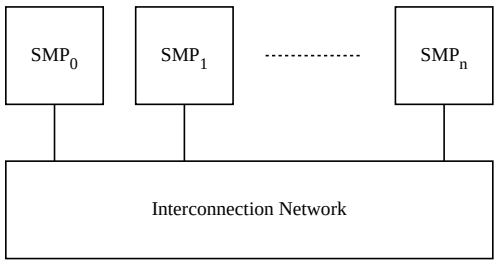

FIGURE 2.10 – *Schéma type de l'architecture hybride faite d'un ensemble de SMP ou de multi-coeur mis en réseau.*

des clusters de multicores, comme le montre la figure 2.10.

Au niveau global, c'est une architecture distribuée mais dont chaque noeud est lui même, au niveau local une machine parallèle à mémoire partagée. On combine ainsi l'avantage de la scalabilité des systèmes distribués avec la possibilité qu'offre les SMP d'avoir des noeuds individuels très puissants.

En principe, on peut imaginer que ces machines aient aussi un modèle de programmation mixte : une gestion multithread au niveau des noeuds et un échange de message entre les noeuds. En pratique, toutefois, c'est souvent MPI qui est utilisé déjà au niveau des processeurs et la structure hybride n'est plus visible pour le programmeur.

2.3.5 GPU

Les GPU (Graphical Processing Units) sont devenues une architecture très populaire pour le calcul HPC. Initiallement prévues pour le rendu d'images dans des applications graphiques exigentes (industrie du jeu), les cartes GPU ont un

FIGURE 2.11 – Architecture globale d'un GPU de Nvidia (source : documentation Nvidia).

degré de parallélisme élevé. Pour cette raison, dès le début des années 2000, des chercheurs les ont utilisées pour du calcul scientifique général (GPGPU, pour General Purpose GPU). Nvidia a par la suite encouragé cette direction en rendant la programmation plus aisée (CUDA) et en fournissant une architecture intégrant des opérations en double précision. Par ailleurs, l'environnement OpenCL offre un langage de programmation pour GPU qui n'est pas associé à l'architecture Nvidia, rendant ainsi l'utilisation des GPU plus portable.

D'un point de vue architectural, une carte GPU est un modèle hybride entre le SIMD et le MIMD, avec des mémoires partagées. Elle offre cependant une combinaison souvent peu claire de ces différents modèles, reflètant une origine historique où le rendu d'image est optimisé au dépend d'un modèle de programmation parallèle sûr et bien défini.

En gros, un GPU est une machine MIMD à mémoire partagée, dont les noeuds de calcul (baptisés SMX chez Nvidia) sont des machines SIMD à mémoire paratagée (voir figure 2.11). Il y a donc plusieurs hiérarchies de mémoire.

Entre-eux, les SMX sont asynchrones et partagent une mémoire globale commune. Cependant, les opérations atomiques contrôlant l'accès à cette mémoire ont été ajoutées seulement récemment. Une synchronisation entre les SMX a lieu lorsque les kernels (programmes tournant sur le GPU) se terminent, ce qui reste souvent la façon la plus courante de coordonner le parallélisme entre eux.

Les SMX (voir figure 2.12) sont des machines SIMD comprennant typiquement 192 coeurs synchronisés. Quatre groupes de 32 threads (appelés *warps* dans ce contexte) peuvent être exécutés en même temps, et deux instructions par thread peuvent être exécutée simultanément sur 2 coeurs, si les dépendances le

FIGURE 2.12 – Vue détaillée d'un SMX et de ses coeurs de calculs.

permettent. On notera aussi que des variables partagées ou privées peuvent être utilisées.

La programmation d'un GPU est définie par un code qui s'applique à un élément de donnée générique (par exemple un point d'une grille 2D). Dans ce code (appelé *kernel* comme indiqué plus haut), on a accès à toutes les autres variables, comme dans un modèle à mémoire partagée, mais sans pour autant avoir des primitives de coordinations garantissant par exemple une exclusion mutuelle dans une section critique.

2.4 Les ordinateurs vectoriels

Actuellement, on parle souvent d'exécution vectorielle dans les processeurs modernes. Il s'agit typiquement de la possibilité qu'offrent ces processeurs d'exécuter la même instruction simultanément sur plusieurs données, typiquement 4 éléments d'un vecteur à la fois. Dans la terminologie adoptée ici, cela s'apparente plutôt à une exécution SIMD.

Dans ce qui suit on va présenter l'architecture vectorielle à travers son évolution historique, ce qui la distingue clairement de la vision SIMD.

Les architectures vectorielles ont connu un grand succès depuis leur apparition en 1976 (Cray-1). Ce type de machine a été unanimement considéré comme le

standard de l'ordinateur à haute performance jusqu'à la fin des années 80. Les machines vectorielles ont été longtemps sans concurrence du point de vue de leur performance et il a fallu attendre l'avènement des grosses machines parallèles pour inverser cette tendance.

Il n'en reste pas moins que l'architecture vectorielle possède des caractéristiques importantes, particulièrement utiles et bénéfiques dans le domaine du calcul scientifique.

Les excellentes performances obtenues avec les machines vectorielles sont en partie dues à la technologie de pointe utilisée. Cela se reflète dans le prix élevé de ces machines qui ne sont accessibles qu'à des centres de recherche importants avec de gros besoins numériques. Un exemple célèbre d'une telle architecture est le Cray Y-MP qui combine le côté vectoriel avec la présence de plusieurs processeurs (habituellement utilisés pour augmenter le throughput du système plutôt que pour faire du parallélisme au niveau d'un même programme) Dans le même esprit, le Cray C-90 et le NEC SX-4 sont d'autres exemples d'architectures vectorielles multiprocesseurs qui ont eu un grand succès à la fin du 20ème siècle.

Cependant, la raison majeure des performances remarquables fournies par les processeurs vectoriels provient des particularités architecturales de ces machines. On peut distinguer trois éléments essentiels qui définissent une architecture vectorielle :

— Les données sont organisées en vecteurs comprenant typiquement 128 ou 256 valeurs ou mots machines (de 64 bits, en général). Ces vecteurs sont traités comme des entités élémentaires par le flot de contrôle. Le travail relatif à chaque instruction est donc important puisqu'il est répété pour chaque élément du vecteur. Par conséquent, le débit d'instructions nécessaire est réduit et il n'y a pas de risque de goulet d'étranglement dans le flot d'instructions. Pour cette raison, une architecture de type RISC ne s'est pas développée pour les processeurs vectoriels.
— Chaque opération qui est effectuée dans une machine vectorielle tire parti au maximum de la mise en pipeline.
— Un parallélisme au niveau des unités fonctionnelles est présent, comme dans les architectures superscalaires. En particulier, il y a des unités FP (floating point) et FX (integer) séparées. Il peut aussi y avoir plusieurs paires de ces unités.

Toutes ces spécificités architecturales additionnées permettent un speedup considérable, souvent supérieur à 100. Cependant, ce speedup est obtenu au prix d'une technologie de compilation avancée et coûteuse.

Globalement, une architecture vectorielle s'avère très complexe. Ici, nous nous contenterons de décrire en quelques détails les principes de base énoncés ci-dessus et d'illustrer les problèmes fondamentaux qui font obstacle à la vectorisation.

2.4.1 Vecteurs

Dans une architecture scalaire classique, le temps nécessaire pour calculer l'adresse en mémoire des données, les faire venir au CPU et, finalement les replacer en mémoire est de loin supérieur au temps de calcul proprement dit.

Si la même opération est répétée sur plusieurs données, on peut gagner un temps considérable en groupant les données en vecteur : on ne fait qu'un accès mémoire de toutes les données à la fois dans un registre vectoriel, comme présenté sur la figure 2.13. S'il y a plus de données dans le problème que ne peut en contenir le registre, l'opération est alors répétée sur les blocs de données correspondant à la taille du registre (par exemple 256 mots). Il faut remarquer que des registres scalaires supplémentaires existent aussi pour réaliser toutes les opérations non vectorielles.

La figure 2.14 illustre le gain de temps obtenu par l'emploi de registres vectoriels pour accéder à la mémoire et masquer le temps de latence.

L'accès «simultané» à toutes les données d'un vecteur n'est possible que si les données sont placées dans des **bancs** mémoire différents (c'est à dire des portions distinctes de la mémoire centrale qui peuvent être accédées en même temps). Si ce n'est pas le cas, on parle de conflit de banc et la vectorisation est entravée. En général, le compilateur se charge de bien répartir les données à traiter dans la mémoire, mais cela n'est pas toujours possible si les mêmes données sont utilisées de façon différente dans plusieurs parties du programme (par exemple, on n'accède pas forcément les données consécutivement, avec un pas de 1).

Si T_{fetch} et T_{store} sont les temps pour prendre et stocker une donnée en mémoire, et T_{cpu} le temps nécessaire pour faire un calcul sur cette donnée, la répétition de l'opération sur N valeurs par un processeur scalaire prend un temps

$$T_{scalar} = NT_{fetch} + NT_{cpu} + NT_{store}$$

Par contre, dans une architecture vectorielle, seul le temps de calcul est répété N fois. On obtient donc

$$T_{vector} = T_{fetch} + NT_{cpu} + T_{store}$$

ce qui est d'autant plus favorable que T_{fetch} et T_{store} sont grands par rapport à T_{cpu}. Par exemple, le speedup obtenu, T_{scalar}/T_{vector} peut être estimé ainsi

$$\frac{T_{scalar}}{T_{vector}} = \frac{NT_{fetch} + NT_{cpu} + NT_{store}}{T_{fetch} + NT_{cpu} + T_{store}} \approx \frac{T_{fetch} + T_{store}}{T_{cpu}}$$

en supposant que $T_{fetch} \approx T_{store} >> T_{cpu}$ et que, avec N assez grand, $NT_{cpu} >> T_{fetch}$.

2.4.2 Pipeline

Dans une architecture vectorielle, la technique du pipeline est utilisée partout où elle est possible. Chaque instruction complexe est subdivisée en instructions

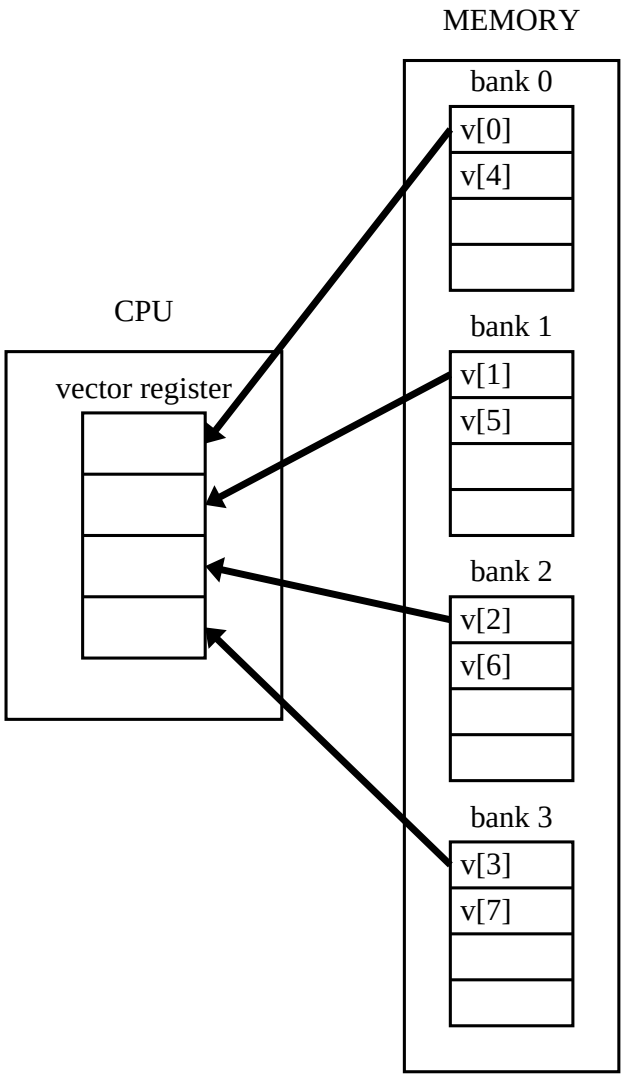

FIGURE 2.13 – Illustration d'un registre vectoriel qui peut recevoir en une fois plusieurs composantes d'un vecteur de donnée, grâce à une accès parallèle aux données réparties en mémoire dans différents bancs mémoire. Ici on considère un registre vectoriel à 4 positions, une mémoire structurée en 4 bancs et un vecteur de 8 éléments. Dans les architectures vectorielle haut de gamme, ces tailles sont bien plus grandes, avec des possibilités de traiter des vecteurs de taill 256 ou 512. Cette figure montre une opération de lecture en mémoire, mais en reversant les flèches, on obtient une écriture simulatanée en mémoire.

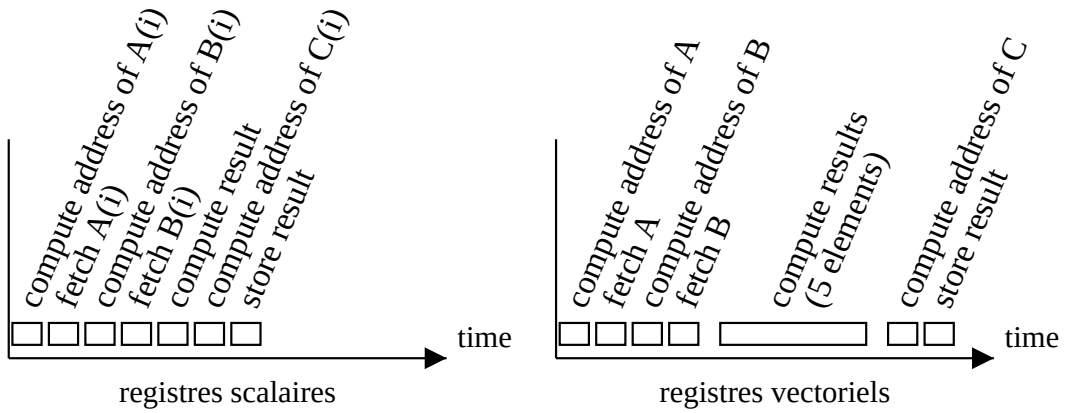

FIGURE 2.14 – *Dans une architecture vectorielle, l'accès à la mémoire se fait par blocs de données, grâce à des registres vectoriels. Cette technique, applicable si la même opération est effectuée sur de nombreuses données, permet de ne payer la pénalité d'un accès mémoire qu'une seule fois pour tout le vecteur. Ici, on compare les performances réaslisées pour l'opération* `C=A+B` *pour un accès par registre scalaire et vectoriel. A gauche, on calcule une seule valeur de* `C` *alors qu'à droite on calcule tout le vecteur (qui, sur la figure comporte 5 éléments).*

élémentaires prenant si possible un cycle d'horloge chacune. Chaque opération est ainsi un enchaînement de q instructions de base qui se suivent et qui forment le pipeline.

Si plusieurs données doivent être traitées à la suite, elles peuvent s'enchaîner (recouvrement d'instructions) dans le pipeline sans attendre que ce dernier soit entièrement vidé de la précédente donnée avant d'entrer.

Donc, si le pipeline a q niveaux, on aura jusqu'à q valeurs simultanément présentes dans le pipeline et, après les q premiers cycles nécessaires à remplir le pipeline, on aura un nouveau résultat par cycle.

La difficulté pour transformer une opération donnée sous forme d'un pipeline est de s'assurer que la subdivision peut se faire en sous-opérations de longueurs identiques. Si ce n'est pas le cas, il faut subdiviser plus finement la sous-opération la plus lente ou, si cela n'est pas possible, utiliser des solutions technologiques ou architecturales pour accélérer les étages trop lents.

L'avantage en performance d'un pipeline est un speedup d'un facteur q. En effet, une instruction prenant q cycles sur une machine scalaire d'horloge τ demandera un temps

$$T_{scalar} = Nq\tau$$

si elle est répétée N fois. Par contre, si ces q cycles sont subdivisés en q niveaux d'un pipeline, un processeur vectoriel de même cycle horloge ne prendra que

$$T_{vector} = q\tau + (N-1)\tau$$

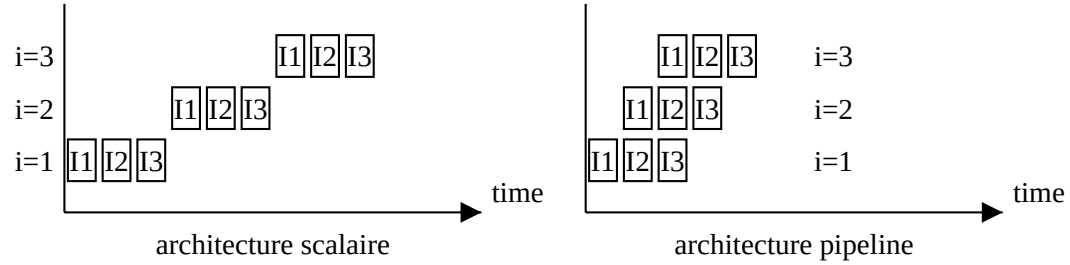

FIGURE 2.15 – *La technique du pipeline permet d'enchaîner les opérations sur plusieurs données. Ici on considère le cas d'une opération qui est décomposée en 3 instructions de base notées* `I1`*,* `I2` *et* `I3`*. On suppose que la même opération est répétée pour trois données, indicées par* `i=1`*,* `i=2` *et* `i=3`*. Dans une machine scalaire, on attend que ces 3 sous-instructions soient terminées avant de considérer la valeur suivante. Dans l'architecture vectorielle, les données peuvent se superposer dans le CPU.*

On attend q cycles pour la première valeur et, ensuite, à chaque cycle, on récolte un autre des $N-1$ résultats attendus. Pour N grand, $T_{vector} \approx (N-1)\tau$ et on obtient bien le speedup annoncé.

La figure 2.15 illustre l'avantage du pipeline dans le cas d'une opération se décomposant en $q = 3$ niveaux. On remarque sur cette figure que si le résultat du calcul pour $i = 1$ est nécessaire pour commencer celui pour $i = 2$ (c'est à dire qu'il y a dépendance entre les opérations), la structutre en pipeline n'est plus possible : l'enchaînement est rompu et le pipeline doit se vider et se reremplir.

2.4.3 Unités fonctionnelles

Une architecture vectorielle se caractérise par la présence de plusieurs unités fonctionnelles qui peuvent être actives en même temps. Il y aura ainsi une ou plusieurs unités FPU pour les opérations en virgule flottante et une ou plusieurs unités FX pour les calculs sur les entiers (virgule fixe). Ce parallélisme au niveau du processeur permet de traiter simultanément des segments différents du programme et aussi de partager le calcul sur un grand vecteur sur deux unités.

Il faut aussi remarquer que les additions et multiplications se font sur des unités FP distinctes, ce qui permet d'enchaîner sans rompre le pipeline une opération du type `D(i)=C(i) + B(i)*A(i)`. Cette particularité est très utile lors de calculs intensifs sur les matrices.

Notons encore que les opérations scalaires sont réalisées par des registres et unités d'exécution spécifiques, ce qui augmente encore le degré de parallélisme dans le processeur.

La figure 2.16 montre un diagramme schématique d'une architecture vectorielle.

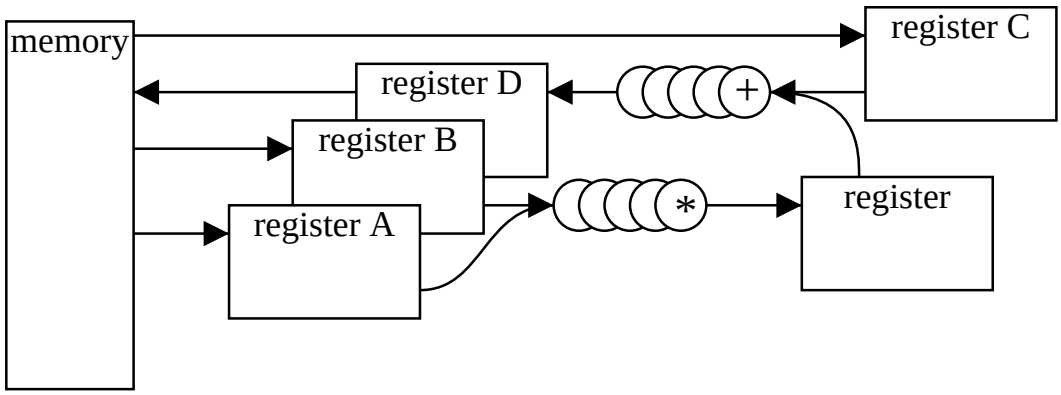

FIGURE 2.16 – *Schéma d'une architecture vectorielle avec plusieurs registres vectoriels et deux unités FP (en pipeline) actives en même temps, l'une pour la multiplication et l'autre pour l'addition. Cette figure illustre le calcul de* `D=A*B+C`*, où* `A`*,* `B`*,* `C` *et* `D` *sont des vecteurs.*

2.4.4 Problèmes liés à la vectorisation

Pour pouvoir tirer parti des ressources d'un processeur vectoriel, il faut que le problème y soit adapté. C'est le cas de nombreux problèmes de calcul scientifique qui nécessitent le traitement de grandes structures de données régulières.

Pourtant, même dans ce cas, les obstacles à la vectorisation sont nombreux. Heureusement il existe souvent des astuces de programmation qui permettent de contourner les difficultés. Ce paragraphe se propose de décrire certaines situations typiques et la façon de les réexprimer pour favoriser la vectorisation. Précisons à cet effet que les compilateurs modernes qu'on trouve sur les machines vectorielles haut de gamme sont souvent très élaborés et corrigent automatiquement la formulation du programmeur pour permettre la vectorisation du code.

Branchement : Une situation courante qui empêche la vectorisation est l'interruption du flot de données due à la présence d'un branchement `if`, par exemple. L'exemple ci-dessous en est une illustration :

```
do i=1,n
   if A(i)>0 then
       B(i)=sqrt(A(i))
   else
       B(i)=0
   endif
enddo
```

L'instruction `if` empêche le programme de traiter la totalité des éléments de `A` de façon identique (il faut un traitement différent pour les valeurs négatives et

positives), et cela est contraire au bon fonctionnement des unités vectorielles.

La solution est de réécrire ce fragement de code de la façon suivante

```
do i=1,n
   cond(i)= A(i)>0
enddo

do i=1,n
   B(i)=sqrt(cond(i)*A(i))
enddo
```

De cette façon, on a décomposé la boucle en deux parties qui chacune assure un traitement identique des données et, donc, une mise en vecteur possible.

Dépendances de données : Un autre problème qui empêche la vectorisation d'une boucle est l'existence de **dépendances** entre les données d'un vecteur. Par exemple, le code suivant (qui correspond à une multiplication matrice-vecteur)

```
do i=1,n
   do j=1,m
      y(i)=y(i) + A(i,j)*x(j)
   enddo
enddo
```

contient une dépendance car la valeur de `y(i)` à l'itération `j+1` dépend du résultat de l'itération `j`. Il faut donc totalement terminer le calcul de l'itération `j` avant de pouvoir commencer celui de l'itération `j+1`. Cela rend évidemment impossible un calcul en pipeline sur des vecteurs.

La façon de contourner le problème dans ce cas est d'inverser l'ordre des boucles sur `i` et `j`

```
do j=1,m
   do i=1,n
      y(i)=y(i) + A(i,j)*x(j)
   enddo
enddo
```

Sous cette forme, il n'y a pas de dépendance dans la boucle sur `i` et la vectorisation peut se faire. Dans cette manière de faire, on calcule les contributions du produit matrice-vecteur, progressivement pour chaque élément du vecteur colonnne `y`. Par contre dans la première disposition des boucles, chaque élément est totalement calculé avant de passer au suivant.

D'une façon générale, il est important de remarquer que l'ordre des indices de deux boucles imbriquées influence aussi les performances par rapport à l'accès mémoire. En Fortran par exemple, les éléments d'un tableau bidimensionnel

sont stockés en mémoire par colonnes (`A(i,j)` et `A(i+1,j)` sont consécutifs en mémoire) et la forme en "`ji`" est donc largement préférable et évitera des conflits de bancs au moment des chargements des registres vectoriels.

Déroulement de boucles : L'opération de réduction

```
s=0
do i=1,n
   s=s+y(i)
enddo
```

est fréquente dans de nombreuses applications. Elle ne permet pas la vectorisation, de nouveau en raison de la dépendance de la variable `s` au cours de la boucle.

La technique dite de déroulement de boucle (*loop unrolling*) est utilisée pour permettre néanmoins une certaine part de vectorisation. On réécrit la boucle de la façon suivante

```
s=0

do i=0,n,q
   do k=1,q
     t(k)=t(k)+y(i+k)
   enddo
enddo

do k=1,q
   s=s+t(k)
enddo
```

La boucle extérieur progresse par pas de `q` et la variable `t` accumule, sur `q` positions, les morceaux de somme, comme indiqué sur le schéma de la figure 2.17.

La boucle intérieure sur `k` se vectorise maintenant car il n'y a plus de dépendance. Par contre, la dernière boucle, celle qui calcule `s` ne se vectorise toujours pas. Mais elle est beaucoup plus courte qu'avant si `q` est beaucoup plus petit que `n`. On a donc intérêt à choisir `q` aussi petit que possible. Cependant `q` ne doit pas être choisi inférieur au nombre d'étages q_+ du pipeline de l'unité qui réalise l'addition, sans quoi, on ne tirerait pas pleinement parti de la vectorisation de la boucle interne. En général, `n` est ainsi largement supérieur à `q` et l'opération de réduction se fait plus efficacement avec le déroulement de la boucle.

Une façon encore plus astucieuse de faire est de choisir `q` grand (afin de mieux vectoriser la boucle) et ensuite de répéter la même procédure plusieurs fois sur la boucle non-vectorisable :

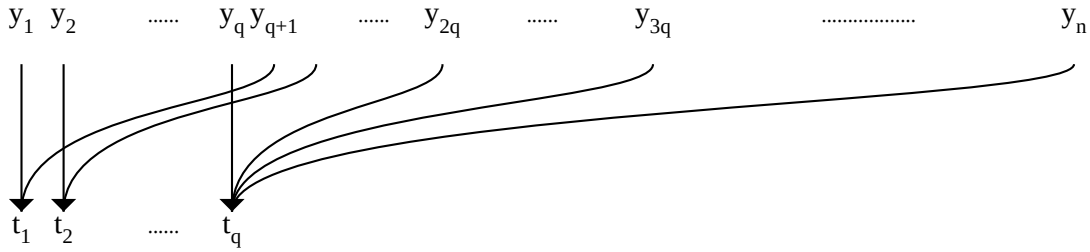

FIGURE 2.17 – *Manière de formuler une opération de réduction avec la technique de déroulement de boucle. La variable* `t` *stocke les valeurs de* `y` *selon la structure indiquée par les flèches.*

```
do k=1,q
   s=s+t(k)
enddo
```

En effet, en choisissant par exemple `q=n/2` on se ramène au même problème de réduction, mais deux fois plus petit. On peut donc recommencer la démarche de déroulement avec `q=n/4` et ainsi de suite, jusqu'à ce que `q` atteigne la valeur limite de vectorisation $\texttt{q=q}_+$.

2.5 L'architecture Data-flow

Les ordinateurs les plus courants sont basés sur une architecture dite **control-driven**, c'est à dire que l'exécution suit l'ordre des instructions spécifiée par le programme. C'est l'idée de base du modèle de von Neumann qui nous est très familière et qui correspond aussi au modèle de base des architectures parallèles SIMD et MIMD.

Mais ce n'est pas la seule solution et d'autres modèles calculatoires ont été réalisés en hardware, comme les architectures *data-driven* (data-flow) et *demand-driven* (machine à réduction) qui toutes deux font jouer un rôle prépondérant aux données.

Dans l'architecture data-driven, c'est la disponibilité des données qui déclenche les instructions correspondantes. Ainsi, l'instruction

$$\texttt{a} = \texttt{b} + \texttt{c}$$

sera exécutée dès que les valeurs de `b` et `c` seront calculées. Ensuite, ce résultat est dupliqué autant que nécessaire pour toutes les autres intructions qui en ont besoin. Il n'y a plus de compteur d'instruction qui, à un moment déterminé, décide de l'exécution et l'ordre des instructions du programme n'est pas respecté.

Une structure en arbre, avec des opérations arithmétiques en chaque noeud et des données placées sur les feuilles est une bonne façon de représenter un calcul

dirigé par les données : les données "s'écoulent" des feuilles vers les noeuds et sont combinées lorsqu'elles s'y croisent.

Cette approche est intéressante pour exploiter naturellement le parallélisme à grain fin entre les intructions. Il n'y a pas d'analyse de dependance explicite à faire puisque que la présence des données implique que l'opération est valide. De la sorte, toutes les instructions réalisables en même temps peuvent être activables.

Mais, d'un point de vue hardware, il faut réaliser un mécanisme qui détecte la disponiblité des données et leur faire correspondre les instructions où elles sont impliquées. Une solution est de repérer chaque instruction par un tag relatif aux données qu'elle met en jeu. Quelques réalistions pratiques de machine data-flow existent, tel le *tagged-token computer* du MIT développé au début des années 80.

Les machine demand-driven (aussi appelées machines à réduction) sont un peu semblables si ce n'est qu'elles déclenchent les opérations lorsqu'un résultat est requis. Par exemple, l'instruction

$$\mathtt{a} = \mathtt{b} + \mathtt{c}$$

ne sera considérée que quand la valeur de `a` est demandée par une autre instruction.

De nouveau, une structure en arbre, avec des opérations arithmétiques en chaque noeud et des données placées sur les feuilles permet de représenter un calcul régit par la demande : le résultat est d'abord demandé pour la racine, ce qui implique que les deux valeurs filles soient connues. Récursivement, cela déclenche les opérations depuis les feuilles.

La figure 2.18 illustre le calcul de l'espression

$$\mathtt{a} = (\mathtt{b} + 1) * (\mathtt{b} - \mathtt{c})$$

en modes control-driven, data-driven et demand-driven. On voit que le parallélisme disponible entre les instructions

$$\mathtt{i1} : \mathtt{b} + 1 \qquad \text{et} \qquad \mathtt{i2} : \mathtt{b} - \mathtt{c}$$

est naturellement pris en compte dans les systèmes data-driven et demand-driven.

2.6 Parallélisme interne pour les microprocesseurs

De nombreuses techniques **architecturales** ont été mises en oeuvre dans les processeurs afin d'en augmenter la performance. Par exemple, la mémoire cache est un composant qui permet de réduire le goulet d'étranglement de von Newmann en diminuant les besoins d'accès à la mémoire centrale.

Une autre direction est d'amener du parallélisme directement dans le CPU afin d'exécuter plusieurs instructions simultanément. Les architectures qui permettent