Introduction au parallélisme

Jean-Christophe Le Lann

jean-christophe.le_lann@ensta-bretagne.fr

Novembre 2024

Plan

1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

Parallélisme

Caractéristique ou capacité de certains systèmes d'effectuer plusieurs actions *simultanément*.

- Mise en jeu de ressources *spatiales*, utilisées de manière apropriée *au cours du temps*.
- Dans la Nature, le parallélisme est omiprésent : systèmes physico-chimiques, (auto-)organisation, etc
- En Informatique, le parallélisme est :
  1. à extraire : on parle de parallélisme *intrinsèque* à une application, un algorithme, etc.
  2. à inventer : reformulation d'un problème.
  3. à réinventer : le parallélisme est imposé par la machine !

Parallélisme et ordinateurs

Temps et parallélisme sont difficiles d'accès en Informatique.

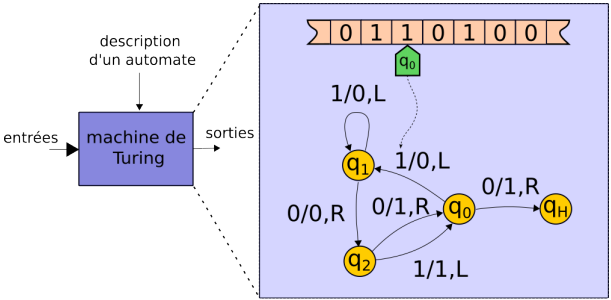
1– Introduction

Rappels historiques : avènement des ordinateurs

- 18ieme siècle
  - Blaise Pascal et la pascaline
  - Vaucanson et ses automates
- 19ieme siècle
  - Babbage
  - Boole
- Avant guerre
  - Turing et la décomposition algorithmique
  - Atanasoff : ordinateur digital électronique
- Seconde guerre mondiale
  - Bletchley park : La Bombe et Colossus
- Après guerre
  - Bell Labs : transistor (1958)
- 1970 et suivantes
  - Intel 4004 (1971)

Machine de Turing (1/2)

- Machine théorique explorant la *calculabilité*



Machine de Turing (2/2)

Following Hopcroft & Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Γ is a finite, non-empty set of *tape alphabet symbols*;
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation);
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*, that is, the set of symbols allowed to appear in the initial tape contents;
- Q is a finite, non-empty set of *states*;
- $q_0 \in Q$ is the *initial state*;
- $F \subseteq Q$ is the set of *final states* or *accepting states*. The initial tape contents is said to be *accepted* by M if it eventually halts in a state from F.
- $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. If δ is not defined on the current state and the current tape symbol, then the machine halts;[19] intuitively, the transition function specifies the next state transited from the current state, which symbol to overwrite the current symbol pointed by the head, and the next head movement.

The following table is Turing's very first example (Alan Turing 1937):

"1. A machine can be constructed to compute the sequence 0 1 0 1 0 1..." (0 <blank> 1 <blank> 0...)
(*Undecidable* p. 119)

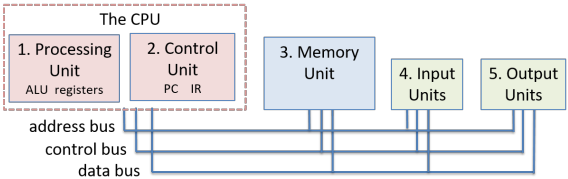
Configuration		Behavior	
m-configuration (state)	Tape symbol	Tape operations	Final m-configuration (state)
b	Blank	P0, R	c
c	Blank	R	e
e	Blank	P1, R	f
f	Blank	R	b

Source : wikipedia.

Modèle de Von Neumann

Le modèle de Von Neumann comprend 5 parties :

1. unité de calcul (registres et ALU)
2. unité de contrôl
3. mémoire (instruction & data)
4. périphériques d'entrée
5. périphériques de sorties



Von Neumann :

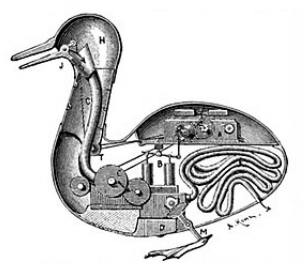
- Instructions et données résident dans la même mémoire.
- Les périphériques sont mappés dans l'espace d'adressage de la mémoire.

Harvard :

- Instructions et données résident dans des mémoires séparées.
- Parallélisme d'accès : data & instruction

Réductionisme

Approche philosophique, typique des 18 et 19ième siècles, consistant à reformuler une notion en d'autres notions plus fondamentales, et ayant structuré l'approche scientifique.



INTERIOR OF VAUCANSON'S AUTOMATIC DUCK.

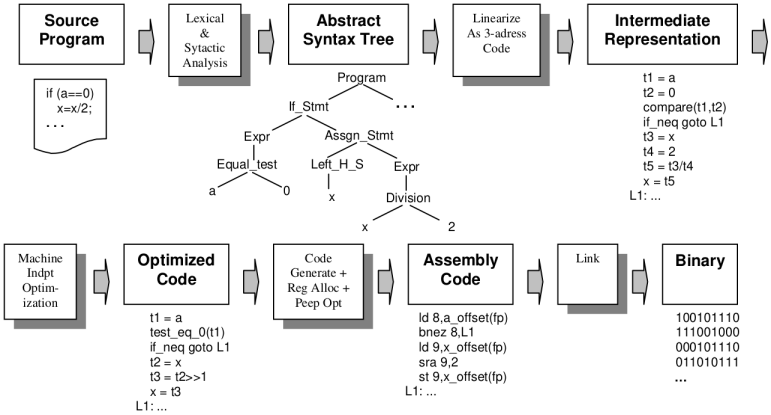
A, clockwork; *B*, pump; *C*, mill for grinding grain; *F*, intestinal tube; *J*, bill; *H*, head; *M*, feet.

La mécanisation du traitement de l'Information a été imaginée, conçue et conduite comme une **liste d'actions/instructions** :

1. séquentielles
2. atomiques
3. autonomes

Machiavel : *"Divide et impera"*, 'divise et règne'.

Rappel : compilateur



Compilation vers une machine concrète

Jeu d'instructions de `Hypocamp`

(accumulator-based toy processor)

opcode	meaning
load a	ACCU ← mot à l'addresse *a* (mem[a])
loadl c	ACCU ← constante *c*
store a	mem[a] ← ACCU
add a	ACCU ← ACCU + mem[a]
sub a	ACCU ← ACCU - mem[a]
mul a	ACCU ← ACCU * mem[a]
div a	ACCU ← ACCU / mem[a]
jump a	PC ← *a*
jumpz a	PC ← *a* si ACCU=0
receive	ACCU ← *user input*
send	*user output* ← ACCU
halt	arrêt

Calcul de la droite `y=a*x+b` :

```
        receive
        store 1      # a
        receive
        store 3      # b
Loop :  receive
        store 2      # x
        mul 1        # x*a
        add 3        # x*a+b
        store 0      # y
        send
        jump Loop
```

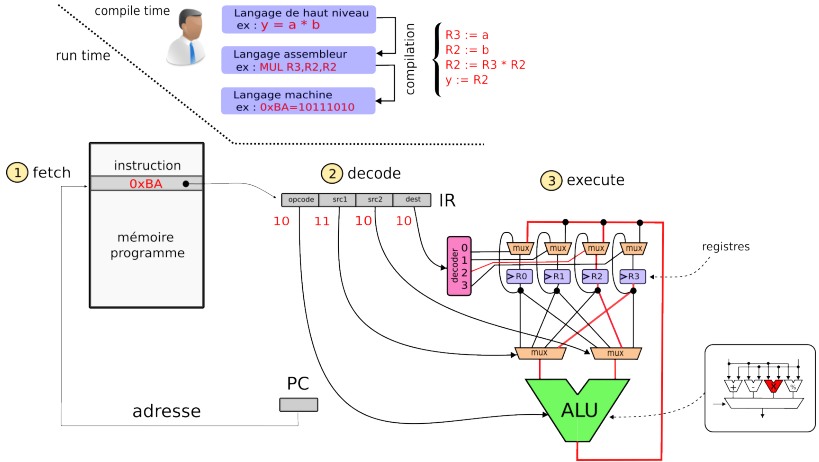
2– Parallélisme des machines

Langages séquentiels = absence du parallélisme ?

Notre propos suivant sera de montrer que :

1. L'exécution des langages séquentiels classiques repose *déjà* sur de nombreux **mécanismes parallèles** :
   - Au niveau de l'électronique des opérateurs aritmétiques et logiques
   - Au niveau du **pipeline** du processeur
   - Au niveau des **transferts de registres** intra-processeur
   - Au niveau des transferts de données dans la **hiérarchie mémoire**
2. D'autres organisations de **machines parallèles** ont été envisagées
   - dont certaines nécessitent d'autres langages...
   - et de nouvelles manières de *penser le calcul*
3. Il s'agit d'un **domaine ouvert** à la recherche

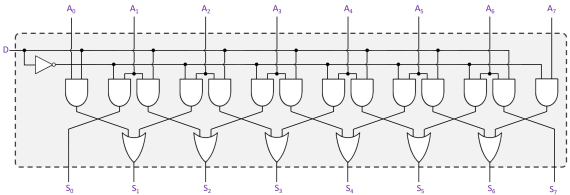
Jeu d'instructions, microarchitecture



- De nombreux fils "oscillent" en parallèle.
- Rappel : *wiring* : $P_{dyn} = \frac{1}{2} C U^2 f$

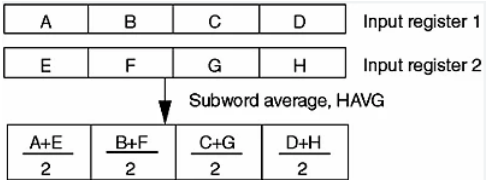
Parallélisme **au niveau bit**

- Un grand nombre d'instructions de calculs simples reposent sur un parallélisme à **grain fin**, au niveau bit
  - Opérations bit-à-bit (bitwise) : `and`, `or`, `not`,...
- *bit-level parallelism*
- exemple d'une instruction de `shift` :

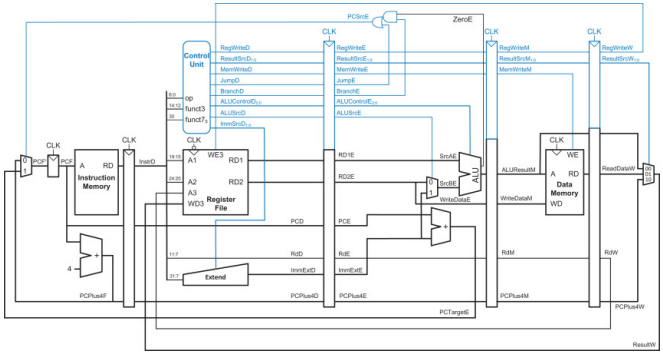


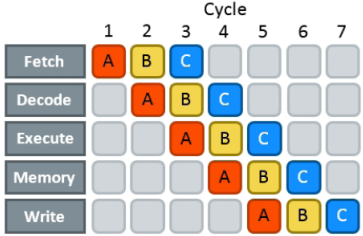
Parallélisme **intra-mot**

- Certains jeux d'instructions permettent une manipulation de subdivision de mots
- *subword parallelism*
- exemple de calcul de 4 moyennes de 2 octets, à l'aide de mots de 32 bits :

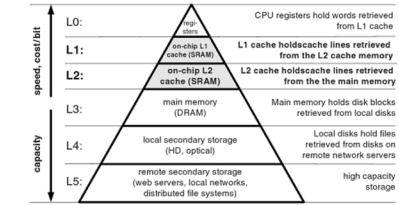


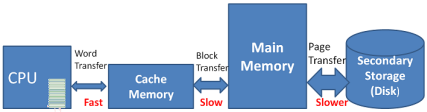
- Opérations simultanées





Parallélisme dans la **hiérarchie mémoire**



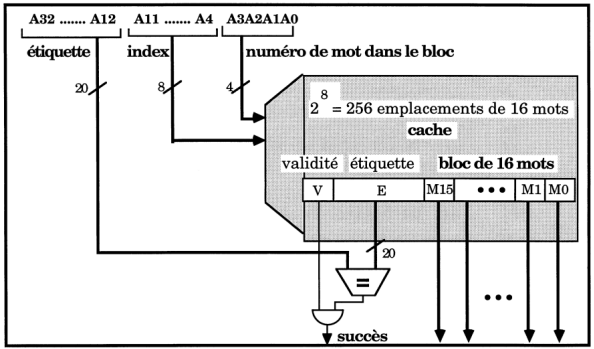


Le simple mécanisme de hiérarchie mémoire exhibe :

1. un parallélisme de flux (streaming)
2. un parallélisme de tâches

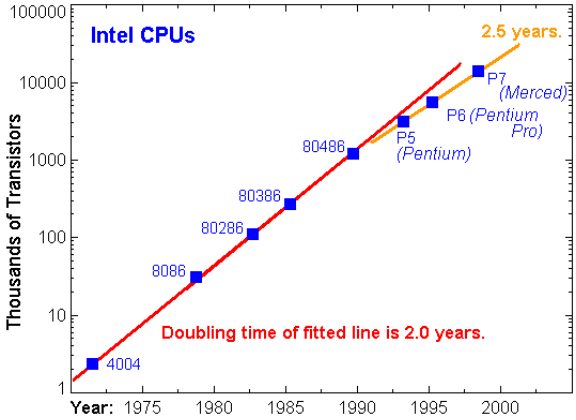
(...et toujours pas d'applications parallèles...)

Parallélisme dans la **hiérarchie mémoire**



- Adresse découpée en étiquette, index de ligne, et index de mot.
- activités électriques hautement parallèles

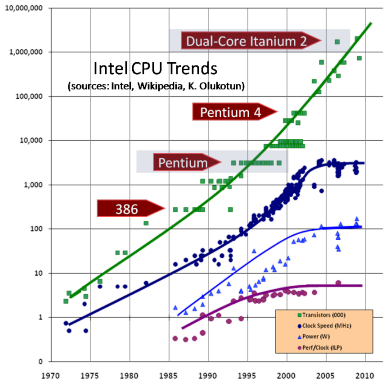
Loi de Moore



- Enoncée en 1957 par Gordon Moore (confondateur Intel)
- Doublement de la capacité d'intégration en 18 mois.
- Les processeurs bénéficient de progrès *gratuits*...

2005 : *Free lunch is over*

Dans un article célèbre, **Herb Sutter** annonce la fin des bénéfices gratuits de l'élévation en fréquence.



- Incapacité de la microélectronique à élever la fréquence.
- Limites physiques proches.

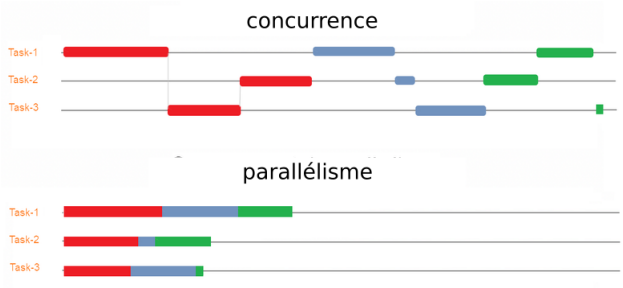
Le sous-titre annonce clairement (?) les futurs enjeux :

"A Fundamental Turn Toward Concurrency in Software".

Notion de concurrence

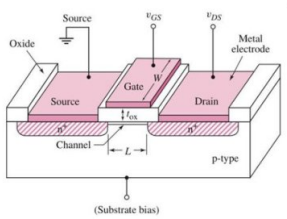
Se réfère à une compétition d'accès à une ou plusieurs ressources partagées (processeurs, mémoires, ...)

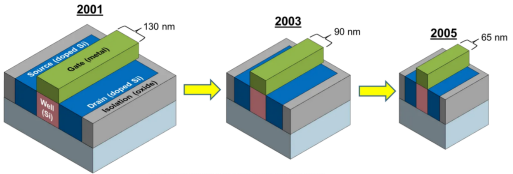
- Trois tâches 1,2,3
  - exécutent chacune `R;B;G`
- Accès à un **unique** processeur.
- Accès à **trois** processeurs.



Le mot de Herb Sutter (spécialiste c++) révèle l'inclinaison du *software* (langages,...,OS) à penser *concurrence* plutôt que *parallélisme*. Ici par exemple, la concurrence n'apporte rien en terme de vitesse (un seul processeur) !

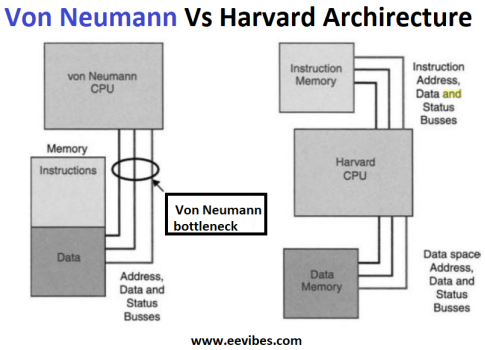
- Technologie 2023 : environ **3nm**
  - Attention : depuis 2008, ce chiffre n'indique plus la taille de la grille.
  - ...reflète plutôt une densité de transistor.
  - En général donné pour des points mémoires DRAM.
- Pour la techno 3nm, la largeur d'un transistor est environ 13nm.
- Comparaison
  - Un Ø cheveu humain entre 50 et 100 µm soit...
  - ...10000 transistors sur la circonférence d'un cheveu





Idée

Améliorer le modèle de Von Neuman



1. Permet des accès simultanés aux deux mémoires
   - Accès à la prochaine instruction
   - Ecriture des données en mémoire

Idée 1 : Instruction-level parallelism (ILP)

Exécuter simulanément plusieurs instructions d'un même programme mono-thread sur le même processeur.

- Existence d'instructions indépendantes
- pouvant s'exécuter sur des ALU différentes, *si disponibles*
- Cette disponibilité peut être vérifée/établie :
  - **Dynamiquement**, par le processeur dit *superscalaire*, supportant ce parallélisme d'instruction.
  - ou **statiquement**, dans l'application, par le compilateur (VLIW).
- Point d'achopement : prééminence de la machine ou de l'application ?

superscalaire

Un processeur superscalaire analyse le programme pendant son exécution pour trouver des ensembles d'instructions qui peuvent être exécutées en parallèle.

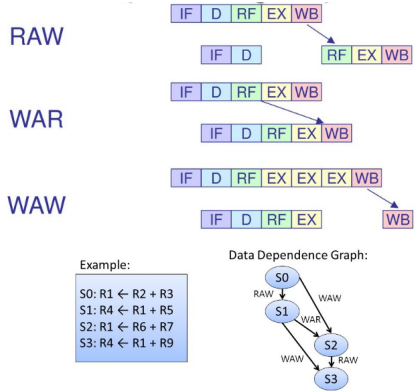
La microarchitecture d'un tel processeur inclue :

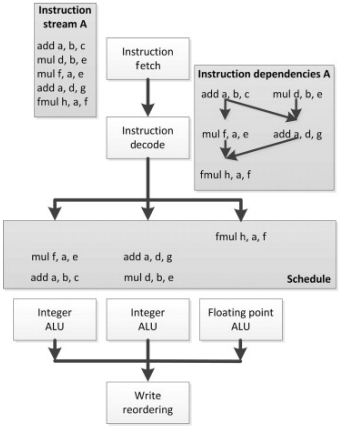
1. plusieurs ALUs et registres (totalement ou partiellement partagés)
2. des files d'attente et de dispatch d'instructions
3. des mécanismes de prédiction de branchements avancés et de spéculation
4. des mécanismes d'exécution dans le désordre (*out-of-order* execution)
5. des mécanismes de renommages de registres

- Mécanisme inclu dans x86 depuis le Pentium.
- Les processeurs superscalaires ne sont pas autant répandu dans le monde de l'Embarqué que dans Desktop/Serveurs.
- Complexité importante de l'architecture.

Un processeur superscalaire détecte les dépendances en analysant les instructions en cours de traitement. Les dépendances sont classées en **trois catégories** :

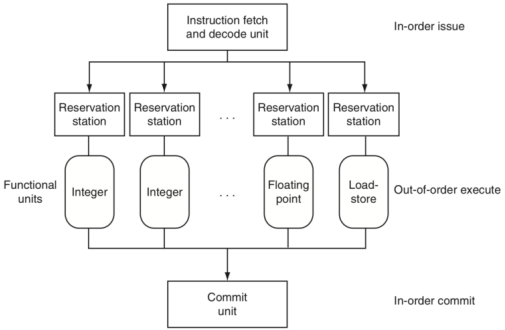
- **RAW - Read After Write** : Dépendance classique. Une instruction a besoin du résultat d'une instruction située en amont.
- **WAR - Write After Read** : Antidépendance. Une instruction détruit une donnée utilisée dans une instruction située en amont.
- **WAW : write after write** : Dépendance de sortie. Une instruction écrase une donnée avant qu'une instruction précédente n'ait fini de la lire.



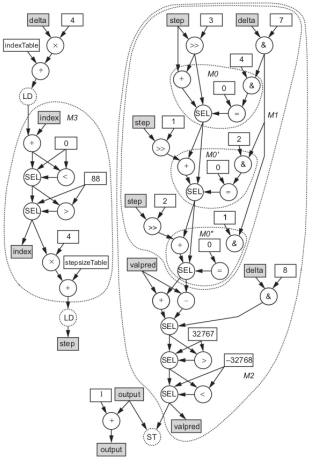


- Ce sont des registres alloués par le compilateur qui décrivent initialement les dépendances.
- Le **matériel** cherche à maintenir ces dépendances lors de l'exécution.
- Les instructions s'exécutent dans le désordre : exécution dite **out-of-order** (OoO)
- Il doit souvent procéder en **renommant les registres**.

Parallélisme d'instructions : architecture superscalaire



1. The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
2. Each functional unit has buffers, called reservation stations, which hold the operands and the operation.
3. As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.
4. When the result is completed, it is sent to any reservation stations waiting for this particular result as well as to the commit unit, which buffers the result until it is safe to release the result of an operation to programmer-visible registers and memory.
5. The buffer that holds the results is called reorder buffer. Stores to memory must be buffered until commit time either in a store buffer or in the reorder buffer.
6. Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline. The commit unit allows the store to write to memory from the buffer when the buffer has a valid address and valid data, and when the store is no longer dependent on predicted branches.



Benchmark ADPCM.

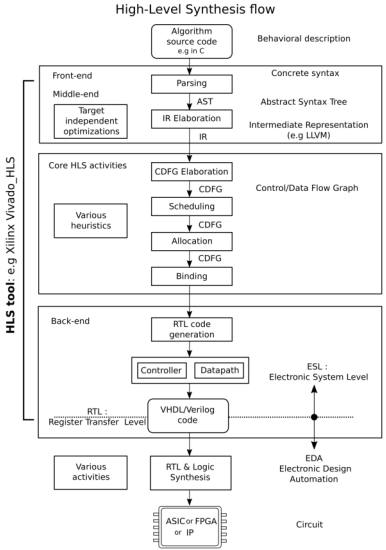
VLIW

Le parallélisme est géré par le compilateur

1. Le compilateur traque les dépendances de données dans l'application
2. Ordonnance et regroupe des instructions parallèles
3. Les opcodes multiples pilotent plusieurs ALUs

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

(Exemple non-lié à ADPCM)

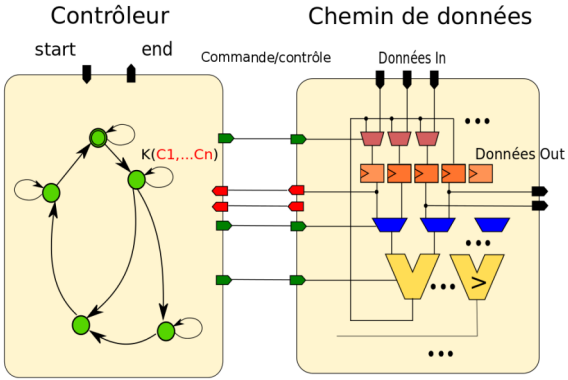


HLS : High-level synthesis

Comme pour VLIW, le parallélisme est *extrait* par le compilateur, mais ce dernier **génère** l'architecture qui le réalise.

- AST puis IR (LLVM)
- Représentation spécifique CDFG : expose contrôle et dépendance de données.
- Allocation : prise en compte de contraintes de ressources.
- Ordonnancement
- Assignation des ressources (binding)
- Génération de code RTL

Architecture cible :

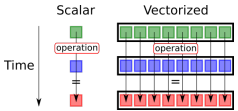


Microarchitecture RTL : register-transfer level

Ensemble de composants matériels réalisant une fonction séquentielle. On parle de FSMD explicite : Finite-state machine + Datapath.

Principe SIMD : single instruction, multiple data

Trouver des calculs identiques, sur des données différentes.



Scalar approach:

```
for (i = 0; i < 1024; i++)
{
   C[i] = A[i]*B[i];
}
```

Vectorized approach:

```
for (i = 0; i < 1024; i+=4)
{
   C[i:i+3] = A[i:i+3]*B[i:i+3];
}
```

1. Jeux d'instruction spécialisés
   - x86 : MMX, SSE*x*, AVXy
   - PowerPC : AltiVec
   - ARM : VFPz, NEON
2. Particulièrement adapté au traitement du sigal
3. Les compilateurs réalisent de mieux en mieux cette vectorisation

Vectorisation : cas problématiques

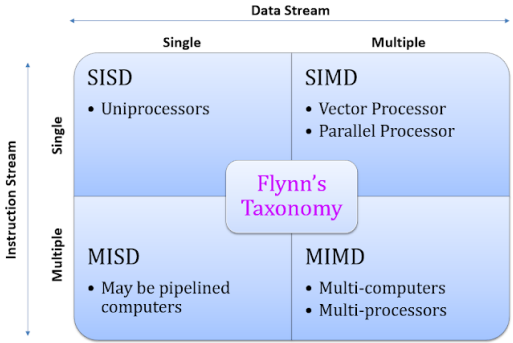
```
void foo(double *y)
{
  for(int i=1;i<10;i++) {
  // a loop that puts sequential numbers into array y
    y[i] = y[i-1]+1;
  }
}
void bar(double *y)
{
  for(int i=1;i<10;i++) {
  // a loop that puts sequential numbers into array y
    y[i] = y[0]+i;
  }
}
```

```
int function(int n)
{
  return (n*n-1);
}
void unsupported_loop_structure(double *y, int n)
{
  for (int i=0; i<function(n); i++) {
    *y = *y * 2.0;
  }
}
```

1. Le premier cas ne se vectorise pas : dépendance entre tours de boucle.
2. Le second cas se vectorise
3. Le troisème cas ne se vectorise pas : calcul sur l'index de boucle.

Idée 2 : combiner plusieurs processeurs

Trouver des organisations de processeurs interconnectés



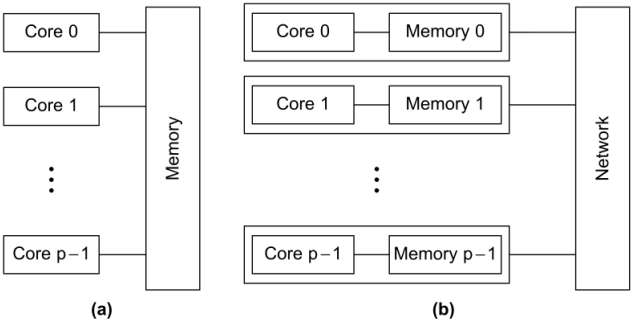
- SISD : The standard von Neumann model.
- SIMD : Thinking Machines, MasPar.
- MISD : *Aucun système digne d'intérêt connu*
- MIMD : Processeurs massivement parallèles, clusters de calculs, shared-memory SMPs.

[Flynn72] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. computers, vol. C-21, No. 9, 1972.

taxonomie décriée...

Architecture à mémoire partagée ou à mémoire distribuée

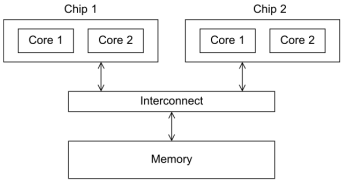
- Les types d'architectures MIMD les plus usuelles.
- Une classification alternative, qui focalise sur l'accès des *coeurs* à la mémoire.



- a) mémoire partagée
- b) mémoire distribuée

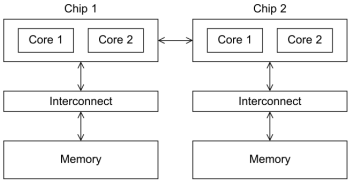
- SMP : symmetrice multi-processing.
  - architecture à mémoire partagéé s'appuyant sur des *processeurs identiques*.

Architecture à mémoire partagée : UMA et NUMA



1. UMA : uniformed memory access.
   - les temps d'accès à la mémoire sont identiques pour tous les coeurs.
2. NUMA : non-uniformed memory access
   - les temps d'accès dépendent du chemin à parcourir pour accéder à la mémoire.

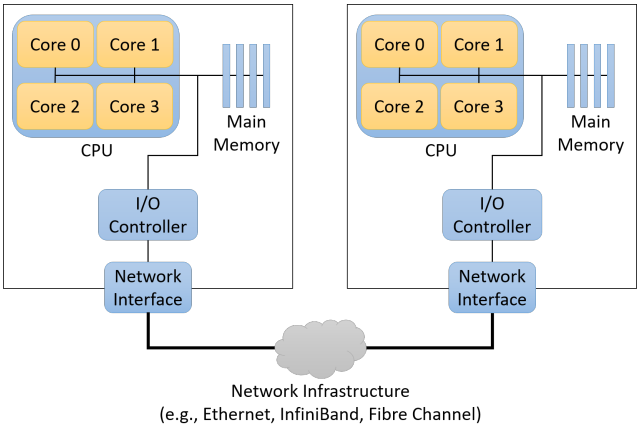
Dans le cas UMA, les temps d'accès à la mémoire sont identiques pour tous les coeurs.



Features	UMA	NUMA
Full Forms	UMA is an abbreviation for Uniform Memory Access.	NUMA is an abbreviation for Non-Uniform Memory Access.
Memory Controller	It contains a single memory controller.	It contains several memory controllers.
Memory Access Time	It contains balanced or equal memory access time.	Its memory access time changes according to the distance of the microprocessor.
Memory Access	Its memory access is slow.	Its memory access is faster.
Suitability	It is mainly utilized in time-sharing and general-purpose applications.	It is mainly utilized in time-critical and real-time apps.
Bandwidth	It has limited bandwidth.	It has more bandwidth.
Bus type	It employs single, multiple, and crossbar buses.	It employs hierarchical and tree-structured buses and network connections.

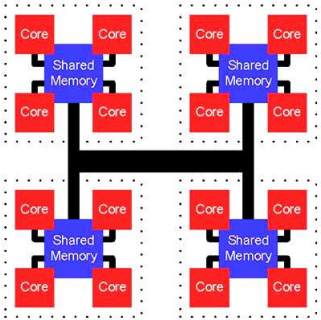
Cluster

Les clusters sont la forme la plus courante d'architecture à mémoire distribuée, où généralement plusieurs PC communiquent, par exemple, via Ethernet.



Hybride

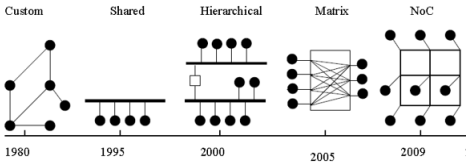
Architecture présentant des motifs d'organisation où mémoires distribuées et mémoire partagée cohabitent.



1. Des groupes de 4 coeurs partagent une mémoire commune.
2. Mais chacun des coeurs, n'appartenant pas à un groupe donné, peut accéder à la mémoire de ce groupe.

Réseau d'interconnexion

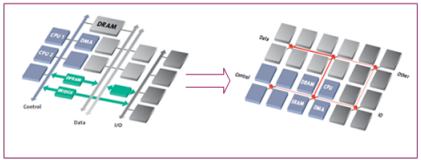
- On constate l'importance du réseau de communication/interconnexion.
- La performance pure des processeurs n'est pas seule responsable de la performance d'ensemble.
- Le réseau d'interconnexion peut en faciliter ou pénaliser la performance.

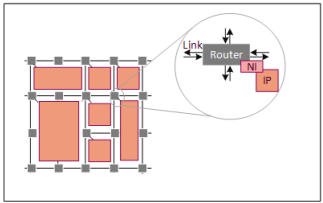


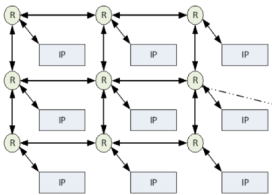
1. Interconnect pour mémoire partagée
   - traditionnellement : bus
   - Bus hiérarchique
   - remplacé par réseau commuté (ex : crossbar)
   - Network-on-chip (NoC)
2. Interconnect pour mémoire distribuée
   - topologies variées

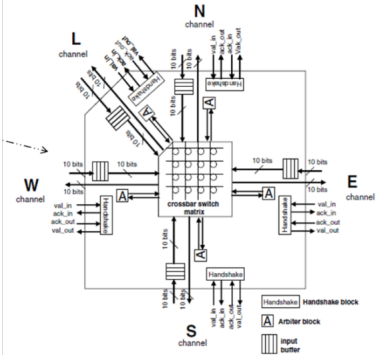
NoC

Le Network-on-Chip constitue un changement de paradigme majeur.



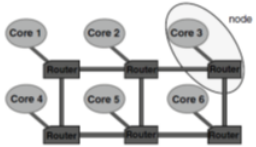




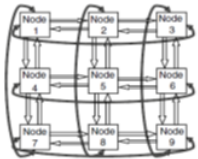


Topologies directes vs indirectes

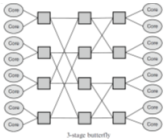
Selon la proximité immédiate ou non du router aux processeurs.

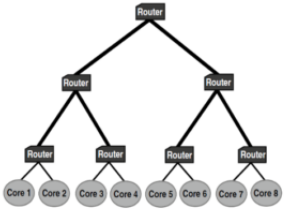


2-D grid



2-D torus



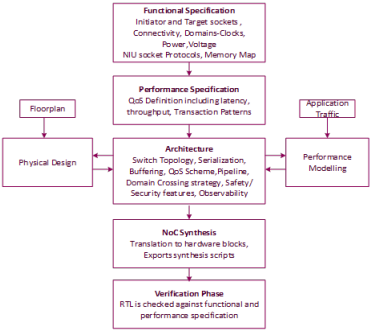


Fat-tree

source : ignitarium website

Un mot sur ...l'EDA

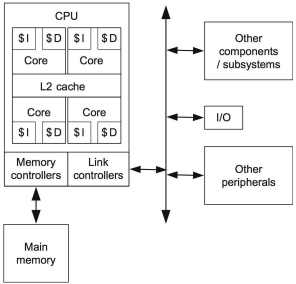
Electronic Design Automation : ensemble des outils de conception permettant la réalisation d'un *chip*.

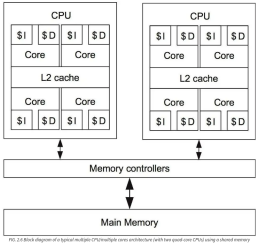


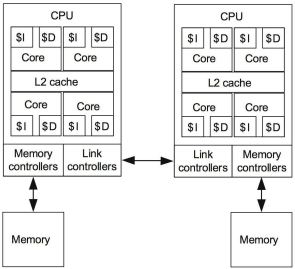
Outils cruciaux ! Ici cas de FlexNoc (Arteris)

1. Spécification fonctionnelle
2. Spécification des performances
3. Définition de l'architecture
4. Synthèse automatique du NoC
5. Verification

Architecture des *multicoeurs*







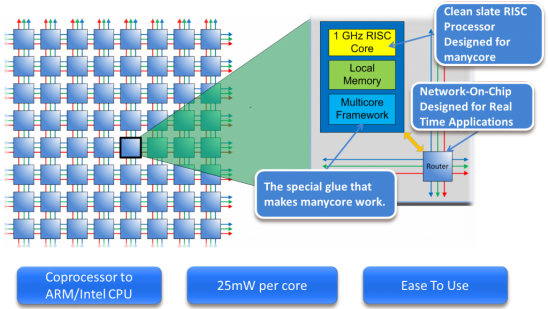
- Quad-core
- Cache L1 instructions et données
- Cache L2 partagé.
- Organisés en mémoire partagée ou distribuée.

Cardoso et al. Embedded Computing for High performance

Manycores

Organisation 2D de coeurs simplifiés, présents en grand nombre.

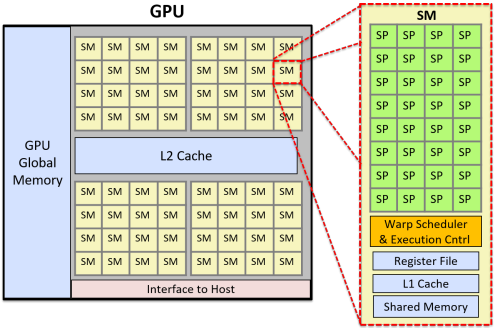
- Exemple de Adapteva Epiphany



1. Network-On-Chip, distributed memory system
2. Epiphany V (2016) : 1024 registers, L1 cache, and shared memorycoeurs RISC 64 bits FP (double précision)
3. 64-core Epiphany-IV made with 28 nm estimated as 50 GFLOPS/W (single-precision)

GPU

Des applications 3D graphiques au parallélisme *embarassant*



1. single instruction/multiple thread (SIMT), variation de SIMD
2. SIMT : le nombre de threads peut-être supérieur au nombre d'unités de calcul.
3. nécessité de politique d'ordonnancement de (groupe de) threads
4. Plusieurs SM : streaming multiprocessors, chacun ayant
   - registers, L1 cache, and shared memory
   - plusieurs processeurs scalaires (SP)

1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

3– Parallélisme des applications

Parallélisme intrinsèque

Potentiel de l'application à s'exécuter en parallèle.

1. Parallélisme **implicte**
   - Le compilateur devra exhiber ce parallélisme.
   - Maintenir une représentation interne adéquate
   - Cette parallélisation n'est pas toujours possible :
     - aliasing de pointers par exemple
     - Questionnement sur l'adéquation langage-parallélisation.
2. Parallélisme **explicite**
   - Charge laissée au programmeur.
   - Questionnement sur le *moyen* de cette explicitation.
   - Questionnement sur l'adéquation du parallélisme applicatif et architectural.
3. Parallélisme **guidé**
   - *Indications* laissées par le programmeur.
   - Délégation des détails au compilateur.
   - Questionnement sur l'interaction outil-programmeur.

Lequel choisissez-vous ?

Reflexions

Existence de cas délicats.

limites du parallélisme :

- les dépendances de données :

Dépendance de flot

Dépendance de sortie

=> les instructions doivent être indépendantes :

- exécuter dans un ordre quelconque
- simultanément

- les dépendances de contrôle :

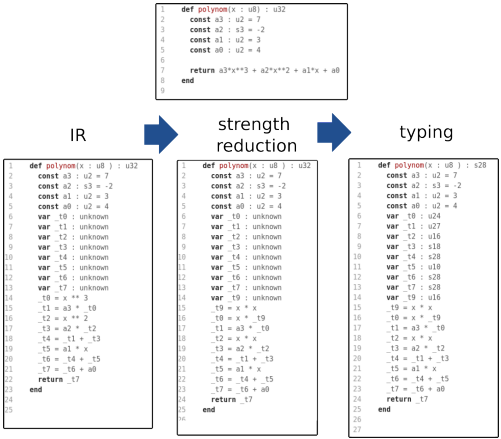
```
I1   a=b+c;          I1 et I4 sont à priori indépendantes
I2   if (a<0)        mais selon la valeur de la variable 'a'
I3      {d=e+f;}     I3 peut être exécuté ou non
I4   g=d+h;          entraînant une dépendance entre I3 et I4
```

[Aurélia Marchand]

Exemple de NangaHLS

Outil pédagogique pour la synthèse de haut-niveau (HLS).

- HLS : génération automatique du processeur *cablé*
  - exemple de parallélisation automatique au niveau instructions.
- idem ASIP : génération automatique du processeur + ISA spécialisée.
- Différentes étapes de compilation illustrées
  1. Elaboration d'une IR
  2. Optimizations
  3. Elaboration d'un graphe *flot-de-données*

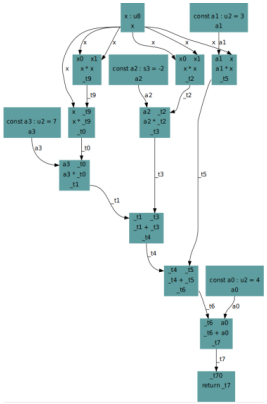


Exemple de NangaHLS

Outil pédagogique pour la synthèse de haut-niveau (HLS).

- DFG : graphe *flot-de-données*
- En fait : instructions connectées.
- Exhibe le parallélisme intrinsèque *grain fin*, s'il existe.

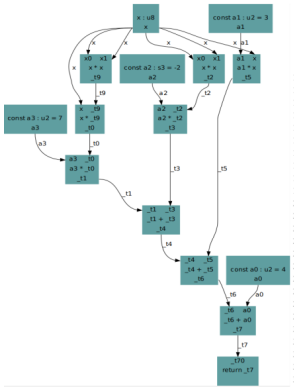
```
1  def polynom(x : u8 ) : s28
2    const a3 : u2 = 7
3    const a2 : s3 = -2
4    const a1 : u2 = 3
5    const a0 : u2 = 4
6    var _t0 : u24
7    var _t1 : u27
8    var _t2 : u16
9    var _t3 : s18
10   var _t4 : s28
11   var _t5 : u10
12   var _t6 : s28
13   var _t7 : s28
14   var _t9 : u16
15   _t9 = x * x
16   _t0 = x * _t9
17   _t1 = a3 * _t0
18   _t2 = x * x
19   _t3 = a2 * _t2
20   _t4 = _t1 + _t3
21   _t5 = a1 * x
22   _t6 = _t4 + _t5
23   _t7 = _t6 + a0
24   return _t7
25 end
26
27
```



Suite des *passes* sur le DFG

- ordonnancement
- allocation des ressources
  - calculs et registres
- *binding/mapping* des opérations aux opérateurs

```
|--[+] register allocation
  |--[+] compute variables lifetimes
   x : [=]-----
  $9 : -[]-----
  $2 : -[]-----
  $5 : -[===]--
  $0 : --[]----
  $3 : --[=]---
  $1 : ---[]---
  $4 : ----[]--
  $6 : -----[]-
  $7 : ------[]
   |--[+] building compatibility graph
   |--[+] clique partitioning
   |--[+] mapping [Nanga::Var]$9@reg_0 to register 0
   |--[+] mapping [Nanga::Var]$3@reg_0 to register 0
   |--[+] mapping [Nanga::Arg]x to register 1
   |--[+] mapping [Nanga::Var]$0@reg_1 to register 1
   |--[+] mapping [Nanga::Var]$1@reg_1 to register 1
   |--[+] mapping [Nanga::Var]$4@reg_1 to register 1
   |--[+] mapping [Nanga::Var]$6@reg_1 to register 1
   |--[+] mapping [Nanga::Var]$7@reg_1 to register 1
   |--[+] mapping [Nanga::Var]$2@reg_2 to register 2
   |--[+] mapping [Nanga::Var]$5@reg_3 to register 3
```

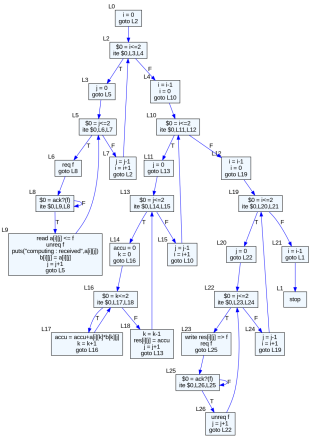



```
1  def polynom(x : u8 ) : s28
2    const a3 : u2 = 7
3    const a2 : s3 = -2
4    const a1 : u2 = 3
5    const a0 : u2 = 4
6    var _t0 : u24
7    var _t1 : u27
8    var _t2 : u16
9    var _t3 : s18
10   var _t4 : s28
11   var _t5 : u10
12   var _t6 : s28
13   var _t7 : s28
14   var _t9 : u16
15   cstep 0 {
16     x : u8
17   }
18   cstep 1 {
19     _t9@reg_0 = x@reg_1 *@mul_0 x@reg_1
20     _t2@reg_2 = x@reg_1 *@mul_1 x@reg_1
21     _t5@reg_3 = a1 *@mul_2 x@reg_1
22   }
23   cstep 2 {
24     _t0@reg_1 = x@reg_1 *@mul_0 _t9@reg_0
25     _t3@reg_0 = a2 *@mul_1 _t2@reg_2
26   }
27   cstep 3 {
28     _t1@reg_1 = a3 *@mul_0 _t0@reg_1
29   }
30   cstep 4 {
31     _t4@reg_1 = _t1@reg_1 +@add_0 _t3@reg_0
32   }
33   cstep 5 {
34     _t6@reg_1 = _t4@reg_1 +@add_0 _t5@reg_3
35   }
36   cstep 6 {
37     _t7@reg_1 = _t6@reg_1 +@add_0 a0
38   }
39   cstep 7 {
40     return _t7@reg_1
41   }
42 end
```

Exemple de Archipel

Outil de *codesign HW/SW*.

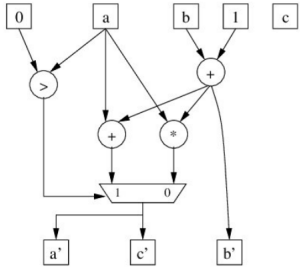
- CFG : graphe *flot-de-contrôle*
- Basic-blocs interconnectés
- Permet d'envisager un parallélisme inter-blocs.
  - Non-représenté ici.



Execution spéculative

Calculer, quand c'est possible, même sans savoir si le résultat servira.

```
int a, b, c;

void fct()
{
    b++;
    if (a > 0)
        c = a + b;
    else
        c = a * b;
    a = c;
}
```



1. 4 opérations ordonnançables au cstep 1.
2. Supposons les 4 exécutables au cstep 1 (ressources disponibles)
3. Entre + et x, une seule servira, selon le résultat de la comparaison.

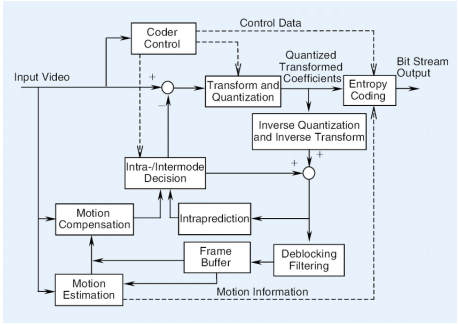
Parallélisme explicite

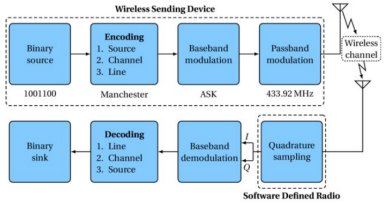
Pré-connaissance d'une structure *coarse-grain*.

- Connaissance métier primordiale.
  - Traitements intrinsèquement parallèles.
  - Fréquents en traitement du signal
  - Ici : encodeur H264 & chaîne SDR

Toutefois la question de l'exécution demeure...

- Calcul *spatial* ?
  - Un processeur par calcul ?
- Calcul *temporel* ?
  - Quelles ressources sont partageables effectivement ?

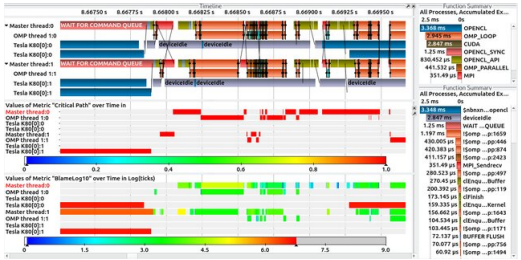




Parallélisation au niveau tâches ?

Extraction automatique de tâches non-exprimées ?

- Sujet de recherche
- Serpent de mer
- Simple *analyse* d'une exécution déjà difficile...



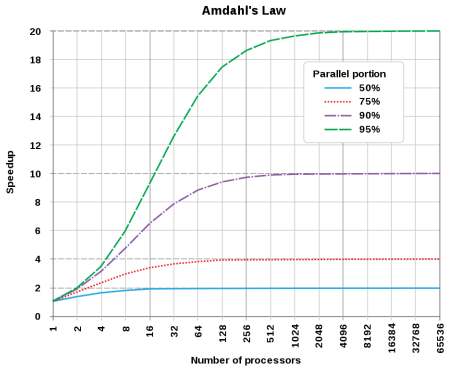
Outil Gromacs pour la simulation de la dynamique moléculaire (MD).

The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law[15]

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}} = \frac{s}{s + p(1-s)}$$

where

- S_{latency} is the potential speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the parallelizable part of the task;
- p is the percentage of the execution time of the whole task concerning the parallelizable part of the task *before parallelization*.



Loi d'Amdhal

La *partie* non-parallélisable entrave fortement la version parallélisée de l'application.

1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

4– API et Langages pour le parallélisme

API retenues

Application Programming Interface pour C/C++

Parallel Architecture	Shared Memory	Distributed Memory	Accelerators	Hybrid Architectures
Parallel Programming Model	Pthreads, OpenMP	MPI	OpenACC/ OpenCL/ CUDA	MPI+OpenMP

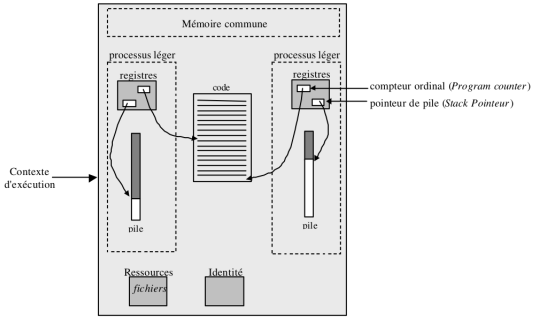
Overview

1. OpenMP
   - Type : Parallélisme à mémoire partagée
   - Focus : Multithreading
   - Utilisation : Paralléliser des boucles, des tâches et des sections
   - Avantages : Facile à utiliser, portable sur différentes plates-formes
   - Limitations : Limité aux architectures à mémoire partagée
2. MPI (Interface de Passage de Messages)
   - Type : Parallélisme à mémoire distribuée
   - Focus : Passage de messages entre les processus
   - Utilisation : Évolutif pour des systèmes parallèles à grande échelle
   - Avantages : Hautes performances sur des architectures distribuées
   - Limitations : Programmation complexe pour la communication
1. OpenCL
   - Type : Parallélisme hétérogène
   - Focus : Programmation multiplateforme pour divers accélérateurs
   - Utilisation : Calcul général sur GPU, CPU, FPGA
   - Avantages : Indépendance de la plate-forme, prend en charge plusieurs dispositifs
   - Limitations : Courbe d'apprentissage abrupte, moins spécialisé pour les GPU
2. CUDA
   - Type : Parallélisme GPU
   - Focus : Programmation GPU NVIDIA
   - Utilisation : Paralléliser explicitement le code pour les GPU NVIDIA
   - Avantages : Hautes performances, contrôle détaillé
   - Limitations : Spécifique au fournisseur (NVIDIA), non portable

Qu'est-ce qu'un Thread ?

Un thread est une unité d'exécution légère, un sous-processus d'un processus. Les threads partagent le même espace mémoire que le processus parent.

1. "processus léger" et très performants.
2. Pas d'intervention du noyau : pas de changement de contexte (coûteux)
3. 10 à 100 fois plus rapides.
4. Lire document Pierre Deiber.



[Deiber 1999]

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_1(void *arg) {
    int *i = (int *) arg;
    (*i)++;
    // Arrêt propre du thread
    pthread_exit(NULL);
}

int main(void) {
    // Création de la variable qui va contenir le thread
    int i = 1;
    pthread_t thread1;
    printf("Avant la création du thread, i = %i.\n", i);
    // Création du thread
    pthread_create(&thread1, NULL, thread_1, &i);
    pthread_join(thread1, NULL);
    printf("Après la création du thread, i = %i.\n", i);
    return EXIT_SUCCESS;
}
```

```c
typedef struct mutex_data {
    int data;
    pthread_mutex_t mutex;
} mutex_data;
```

- pour initialiser un *mutex*:

```c
int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr )
```

- pour verrouiller un *mutex*:

```c
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- pour déverrouiller un *mutex*:

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

- pour détruire un *mutex*:

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

// Nombre total de thread
#define NB_THREAD 2
// Limite de l'incrément
#define INCREMENT_LIMIT 10

// Tableau contenant les threads
pthread_t threads[NB_THREAD];

// Structure de données contenant le mutex
typedef struct mutex_data {
    int data;
    pthread_mutex_t mutex;
} mutex_data;

// Fonction exécutée dans le thread
void * job(void *arg) {
    mutex_data *md = (mutex_data*) arg;
    pthread_t tid = pthread_self();
    while ((*md).data < INCREMENT_LIMIT) {
        // Vérouillage du mutex
        pthread_mutex_lock(&(*md).mutex);
        (*md).data++;
        // Dévérouillage du mutex
        pthread_mutex_unlock(&(*md).mutex);
        printf("thread [ %ld ] data [ %i ]\n", tid, (*md).data);
        // Pause l'exécution du thread pendant 1 seconde
        sleep(1);
    }
    printf("Fin du thread %ld\n", tid);
    pthread_exit(NULL);
}
```

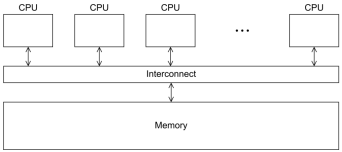
```c
// Fonction principale
int main() {
    // Création du mutex
    mutex_data md;
    // Initialisation de la donnée
    md.data = 0;
    // Initialisation du mutex
    if (pthread_mutex_init(&md.mutex, NULL) != 0) {
        printf("\n mutex init failed\n");
        return EXIT_FAILURE;
    }
    // Boucle de création des threads
    for (int i = 0; i < NB_THREAD; i++) {
        // Création du thread et passage de la structure par référence
        int err = pthread_create(&threads[i], NULL, job, &md);
        if (err != 0) {
            printf("Echec de la création du thread: [%s]", strerror(err));
            break;
        }
        printf("Création du thread numéro %ld\n", threads[i]);
    }
    // En attente des threads
    for (int i = 0; i < NB_THREAD; i++) {
        pthread_join(threads[i], NULL);
    }
    // Destruction du mutex
    pthread_mutex_destroy(&md.mutex);
    return EXIT_SUCCESS;
}
```

[wikipedia]

Utilisation complexe.

OpenMP : un standard

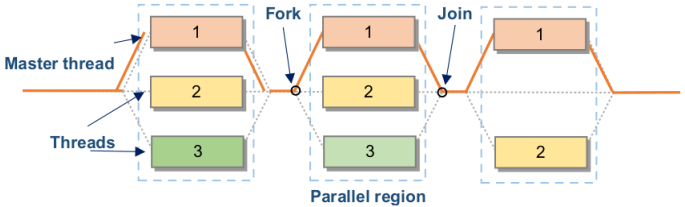
- Vise à simplifier le recours aux threads par un **ensemble d'annotations**
  - code initial non-altéré.
  - applications scientifiques, HPC : Fortran et `C & C++`
- Architectures à mémoire partagée.



- la protection des variables est laissée à la charge du programmeur.
  - *The OpenMP memory model consists of two components : there is a global memory shared by all threads, and each thread has its own temporary view of the memory. Each read and write operation refers to the temporary view.*
  - *The temporary view may or may not be consistent with the global memory.*
- Plusieurs versions
  - 1997 : openMP version 1.0
  - 2021 : openMP version 5.2

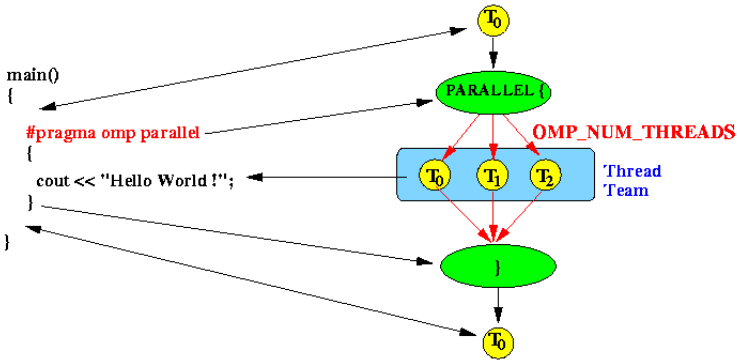
OpenMP

- Utilisation Principale
  - Parallélisation de boucles, tâches et sections du code.
  - Ajout de directives spécifiques au code pour indiquer les parties parallèles.
  - Standard conséquent. Nombreuses possibilités
  - Uniquement survolé ici...
- Premières directives OpenMP
  - `#pragma omp parallel` : Crée une équipe de threads.
  - `#pragma omp for` : Parallélise une boucle.
  - `#pragma omp sections` : Définit des sections parallèles.



```
#pragma openmp parallel
```

- Création d'un *threads team*



- `gcc -Wall -fopenmp -o omp_test omp_test.c`
- `.omp_test 4`

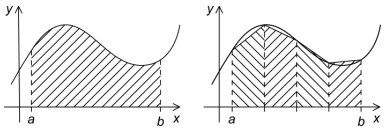
```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

or

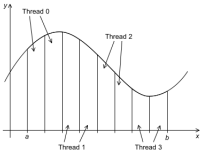
```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n",
          my_rank, thread_count);

}  /* Hello */
```

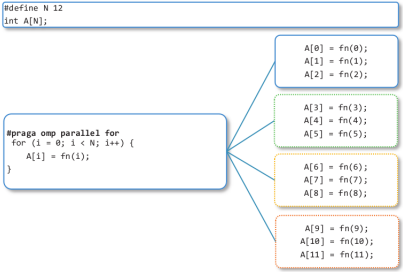
OpenMP : exemple d'application




```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```




```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     /* We'll store our result in global_result: */
9     double  global_result = 0.0;
10    double  a, b;  /* Left and right endpoints   */
11    int     n;     /* Total number of trapezoids */
12    int     thread_count;
13
14    thread_count = strtol(argv[1], NULL, 10);
15    printf("Enter a, b, and n\n");
16    scanf("%lf %lf %d", &a, &b, &n);
17 #  pragma omp parallel num_threads(thread_count)
18    Trap(a, b, n, &global_result);
19
20    printf("With n = %d trapezoids, our estimate\n", n);
21    printf("of the integral from %f to %f = %.14e\n",
22       a, b, global_result);
23    return 0;
24 }  /* main */
25
26 void Trap(double a, double b, int n, double* global_result_p) {
27    double  h, x, my_result;
28    double  local_a, local_b;
29    int     i, local_n;
30    int my_rank = omp_get_thread_num();
31    int thread_count = omp_get_num_threads();
32
33    h = (b-a)/n;
34    local_n = n/thread_count;
35    local_a = a + my_rank*local_n*h;
36    local_b = local_a + local_n*h;
37    my_result = (f(local_a) + f(local_b))/2.0;
38    for (i = 1; i <= local_n-1; i++) {
39      x = local_a + i*h;
40      my_result += f(x);
41    }
42    my_result = my_result*h;
43
44 #  pragma omp critical
45    *global_result_p += my_result;
46 }  /* Trap */
```



- Notes :
  - assignation gérée par le système
  - le nombre de threads peut être précisé
  - attention aux accumulateurs, qui devront être *réduits* comme dans l'exemple suivant

```
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
    for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
    approx = h*approx;
```

```
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
#   pragma omp parallel for num_threads(thread_count) \
       reduction(+: approx)
    for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
    approx = h*approx;
```

OpenMP : sections

```c
#include <stdio.h>
#include <omp.h>
int main() {
    // Code séquentiel
    printf("Cette partie du code est séquentielle.\n");
    // Parallélisation de sections
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            printf("Section 1 traitée par le thread %d\n", omp_get_thread_num());
        }
        #pragma omp section
        {
            printf("Section 2 traitée par le thread %d\n", omp_get_thread_num());
        }
    }
    // Code séquentiel
    printf("Cette partie du code est à nouveau séquentielle.\n");
    return 0;
}
```

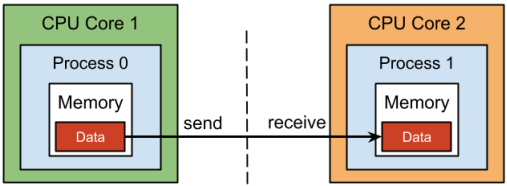
- Static ou dynamique

```c
#include <stdio.h>
#include <omp.h>
int main() {
    const int size = 10;
    int array[size];
    // Initialisation du tableau
    for (int i = 0; i < size; ++i) {
        array[i] = i;
    }
    // Parallélisation de la boucle avec ordonnancement statique
    #pragma omp parallel for schedule(static, 2)
    for (int i = 0; i < size; ++i) {
        printf("Thread %d traite l'indice %d avec la valeur %d\n", omp_get_thread_num(), i, array[i]);
    }
    return 0;
}
```

```
Thread 0 traite l'indice 0 avec la valeur 0
Thread 0 traite l'indice 1 avec la valeur 1
Thread 4 traite l'indice 8 avec la valeur 8
Thread 4 traite l'indice 9 avec la valeur 9
Thread 1 traite l'indice 2 avec la valeur 2
Thread 1 traite l'indice 3 avec la valeur 3
Thread 3 traite l'indice 6 avec la valeur 6
Thread 3 traite l'indice 7 avec la valeur 7
Thread 2 traite l'indice 4 avec la valeur 4
Thread 2 traite l'indice 5 avec la valeur 5
```

Ici, un thread gère 2 indices successifs.

- Acronyme : Message Passing Interface
  - `send/receive`
- Programmation des systèmes à mémoire distribuée.
  - Fonctionne toutefois sur mémoire partagée.
- Essentiellement clusters de calculs.
- Utilisation plus variée que OpenMP
- Mais à nouveau basé sur `C/C++`



MPI : fonctions usuelles

Function Purpose	C Function Call
Initialize MPI	int **MPI_Init**(int *argc, char **argv)
Determine number of processes within a communicator	int **MPI_Comm_size**(MPI_Comm comm, int *size)
Determine processor rank within a communicator	int **MPI_Comm_rank**(MPI_Comm comm, int *rank)
Exit MPI (must be called last by all processors)	int **MPI_Finalize**()
Send a message	int **MPI_Send** (void *buf,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Receive a message	int **MPI_Recv** (void *buf,int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

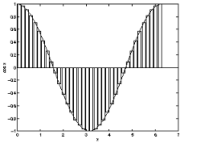
```c
/* C Example */
#include <stdio.h>
#include <mpi.h>


int main (argc, argv)
     int argc;
     char *argv[];
{
  int rank, size;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

MPI : exemple

```c
#include <mpi.h>
#include <math.h>
#include <stdio.h>
float fct(float x){ return cos(x);}
/* Prototype */
float integral(float a, int n, float h);
void main(int argc,char* argv[]){
      int n, p, i, j, ierr,num;
      float h, result, a, b, pi;
      float my_a, my_range;

      int myid, source, dest, tag;
      MPI_Status status;
      float my_result;

      pi = acos(-1.0);  /* = 3.14159... */
      a = 0.;           /* lower limit of integration */
      b = pi*1./2.;     /* upper limit of integration */
      n = 100000;       /* number of increment within each process */

      dest = 0;         /* define the process that computes the final result */
      tag = 123;        /* set the tag to identify this particular job */

/* Starts MPI processes ... */

      MPI_Init(&argc,&argv);            /* starts MPI */
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);  /* get current process id */
      MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

      h = (b-a)/n;    /* length of increment */
      num = n/p;  /* number of intervals calculated by each process*/
      my_range = (b-a)/p;
      my_a = a + myid*my_range;
      my_result = integral(my_a,num,h);

      printf("Process %d has the partial result of %f\n", myid,my_result);

      if(myid == 0) {
        result = my_result;
        for (i=1;i<p;i++) {
          source = i;           /* MPI process number range is [0,p-1] */
          MPI_Recv(&my_result, 1, MPI_REAL, source, tag,
                   MPI_COMM_WORLD, &status);
          result += my_result;
        }
        printf("The result =%f\n",result);
      }
      else
        /* send my result to intended dest. */
        MPI_Send(&my_result, 1, MPI_REAL, dest, tag, MPI_COMM_WORLD);

      MPI_Finalize();/* let MPI finish up ... */
```

Example 1. Numerical Integration by Mid-Point rule

$$\int_a^b \cos(x)dx = \sum_{i=0}^{p-1}\sum_{j=0}^{n-1}\int_{a_j}^{a_j+h}\cos(x)dx \simeq \sum_{i=0}^{p-1}\left[\sum_{j=0}^{n-1}\cos(a_{ij}+\frac{h}{2})h\right]$$

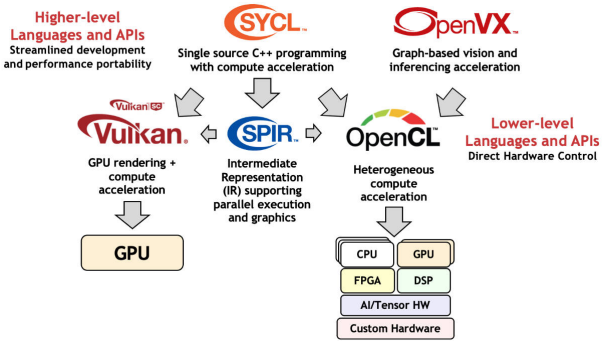


where p = # of processes
n = number of intervals per process
a = lower limit of integration
b = upper limit of integration
h = (b-a)/(n*p)
aij = a +[i*n +j]h

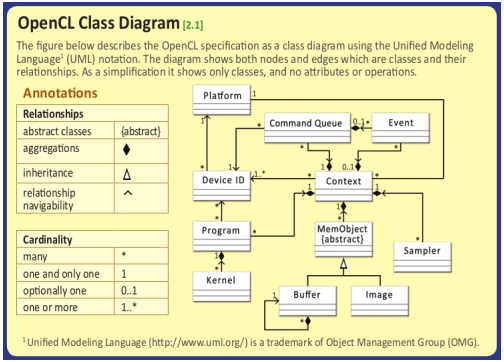
```c
float integral(float a, int n, float h){
      int j;
      float h2, aij, integ;

      integ = 0.0;                  /* initialize integral */
      h2 = h/2.;
      for (j=0;j<n;j++) {           /* sum over all "j" integrals */
        aij = a + j*h;              /* lower limit of "j" integral */
        integ += fct(aij+h2)*h;
      }
      return (integ);
}
```

OpenCl

- Issu de Apple et Khronos group.
  - Technologie C/Clang/LLVM
  - Systèmes parallèles hétérogènes
- Séparation du calcul maître (CPU) et périphériques supportant un calcul intensif :
  - GPU
  - CPU
  - FPGA etc
- Définition d'un `Kernel` de calcul
  - et son alimentation en données (Queue)
- Plusieurs défaut :
  - Connaissance fine de l'architecture nécessaire !
  - Nombre de lignes `x5` en moyenne.



OpenCl : diagramme de classes



OpenCL Class Diagram [2.1]

The figure below describes the OpenCL specification as a class diagram using the Unified Modeling Language[1] (UML) notation. The diagram shows both nodes and edges which are classes and their relationships. As a simplification it shows only classes, and no attributes or operations.

Annotations

Relationships	
abstract classes	{abstract}
aggregations	♦
inheritance	△
relationship navigability	^

Cardinality	
many	*
one and only one	1
optionally one	0..1
one or more	1..*

[1] Unified Modeling Language (http://www.uml.org/) is a trademark of Object Management Group (OMG).

Plan

1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

Prise de recul : une nécessité

- Focus jusqu'ici :
  - Architecture et APIs pour le parallélisme.
  - Manipulation de l'application exprimée ...dans un langage séquentiel.
- Quid de l'essence du parallélisme ?
  - Autres paradigmes ?
  - Autres langages ?

HPC

Le calcul scientifique qui repose sur l'algèbre linéaire...

- Exemple de la multiplication matricielle : ça commence mal !
- Plusieurs opérations possibles simultanément...

$$
\begin{aligned}
AB &= \begin{pmatrix} 0 & 1 & 3 & 0 \\ 8 & 5 & 0 & 6 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \\ 3 & 3 \\ 9 & 5 \end{pmatrix} \\
&= \begin{pmatrix} 0\times0+1\times0+3\times3+0\times9 & 0\times1+1\times0+3\times3+0\times5 \\ 8\times0+5\times0+0\times3+6\times9 & 8\times1+5\times0+0\times3+6\times5 \\ 0\times0+0\times0+1\times3+1\times9 & 0\times1+0\times0+1\times3+1\times5 \end{pmatrix} \\
&= \begin{pmatrix} 9 & 9 \\ 54 & 38 \\ 12 & 8 \end{pmatrix}.
\end{aligned}
$$

```
MatMult(A, B, C)
1:  Pour i = 1 à n :
2:      Pour j = 1 à n :
3:          Pour k = 1 à n :
4:              C_ij ← C_ij + A_ik B_kj
```

...mais un code enlisé dans un **nid de boucles**.

Prise de recul : équations et hardware

La plupart des calculs HPC peuvent se ramener à un système d'équations différentielles linéarisées et discrétisées.

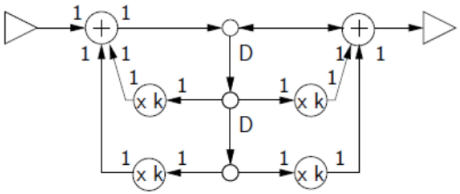
$$\frac{d\vec{x}}{dt} = f(\vec{x}, t)$$

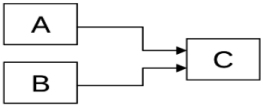
$$\begin{cases} \frac{dx_1}{dt} = f_1(x_1, x_2, ..., x_n, t) \\ \dots \\ \frac{dx_n}{dt} = f_n(x_1, x_2, ..., x_n, t) \end{cases}$$

$$\begin{cases} x_1(t + \Delta) = f_1(x_1, x_2, ..., x_n, t) \\ \dots \\ x_n(t + \Delta) = f_n(x_1, x_2, ..., x_n, t) \end{cases}$$

5– Modèles de Calcul
Models of Computation

- Algorithmes pouvant être représenté sous forme de bloc-diagrammes
- Connectés à un modèle mathématique
- Flux infinis, répétitifs
- Exemple (rappel) : IIR
  - Filtre à réponse impulsionnelle infinie





One possible interpretation is that block C is executed as soon as it gets an input sample either from block A or B.

Another is that block C becomes executable only after it gets input samples from both block A and B.

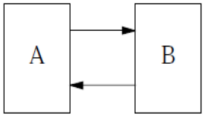
When block A is executed twice to produce two output samples to block C, either two samples are queued on the arc or the later sample overwrites the former sample.

Référence :SoohnHoi Ha

Système = Contraintes sémantiques = { relations }

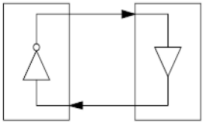
Pas si simple...

...même quand ça paraît simple !



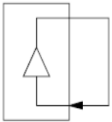
(a)

Co-dépendant :
lequel doit être évalué
en premier ?



(b)

Paradoxe :
pas de solution



(c)

Ambigu :
plusieurs solutions

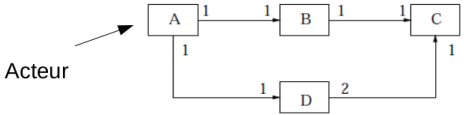
- Static analysis of the specification allows one to verify some correctness measures of the functional specification.
- The specification model can be refined to an implementation to ease the system validation by the principle of "correct-by-construction".
- A formal specification model is not biased to any specific implementation method, so allows one to explore the wider design space.
- It represents the system behavior unambiguously so that collaboration and design maintenance can be accomplished easily

Référence :SoohnHoi Ha

Ptolemy : MoC SDF

Lee Messerschmidt 1987

- On annote le nombre de « tokens » produits et consommés
- On résoud un système d'équations
- S'il y a une solution, le système est réalisable avec une mémoire bornée

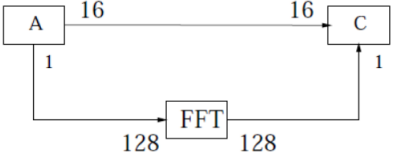


Ici : pas de solution !

Soit $n_{k,m}$ le nombre de valeurs écrites/lues par l'acteur k sur l'arc m. $n_{k,m}$ est positif quand il s'agit d'écrire sur l'arc et négatif quand il s'agit de lire. On cherche à trouver les q_k tels que :

$$\forall m, \sum_{k=1}^{K} q_k n_{k,m} = 0$$

Le vecteur $\vec{q} = (q_1, \ldots, q_K)$ est le vecteur de répétition.
on a alors :



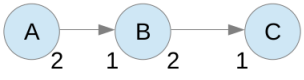
Exemple de calcul d'équations d'équilibrage

$$q_A * 16 = q_C * 16$$

Valid Schedule : A(2(B(2C)))

Synthesized Code

```
main() {
        code block for A
        for (i=0; i<2; i++){
            code block for B
             for (j=0; j<2; j++){
                  code block for C
            }
        }
}
```



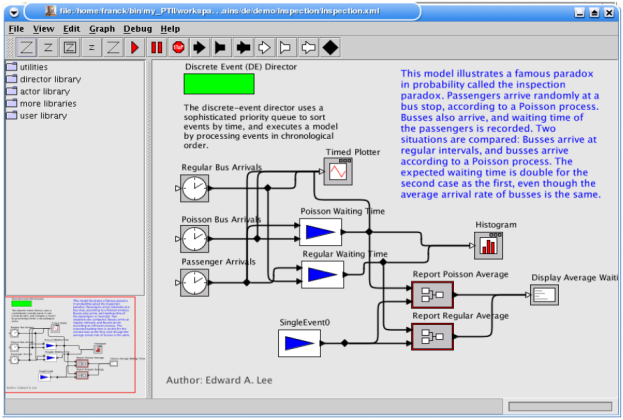
Ptolemy

et les modèles de calculs (MoC)

- Berkeley / Edward Lee
- Outil de simulation
- Etude des interactions des modèles de calculs
  - Comment composer différents MoC ?
    - SDF : synchronous dataflow (!= langages synchrones)
    - Discrete Event
    - Synchronous Reactive
    - Communication sequential process
    - Kahn process network
    - …
  - C'est-à-dire assembler/composer des composants ?
- Notion de « Franckenware »

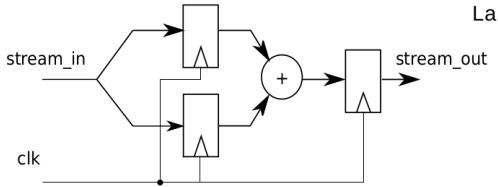
Ptolemy

et les modèles de calculs (MoC)



Un modèle de calcul est une abstraction rigoureuse des méthodes de synchronisation d'entités concurrentes, qui permet d'analyser un comportement d'ensemble du système avant sa réalisation physique

La sortie doit être paire !

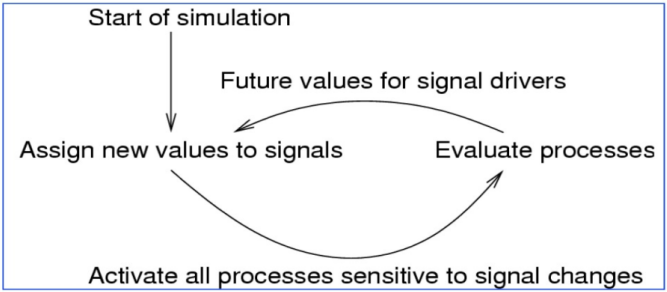


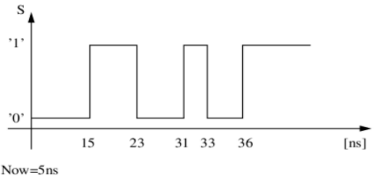

```
jcll@jcll-Latitude-E6520 ~/FORGE/jcll_sandbox/trunk/COURS/IMT-Amer/2017 $ ghdl -r buggy_tb
buggy_tb.vhd:50:7:@70ns:(report note): stream = 0
buggy_tb.vhd:50:7:@90ns:(report note): stream = 0
../../../src/ieee/numeric_std-body.v93:2098:7:@110ns:(assertion warning): NUMERIC_STD.TO_INTEGER
buggy_tb.vhd:50:7:@110ns:(report note): stream = 0
buggy_tb.vhd:52:9:@110ns:(report note): ERROR: result should be even !
buggy_tb.vhd:50:7:@130ns:(report note): stream = 1
buggy_tb.vhd:52:9:@130ns:(report note): ERROR: result should be even !
buggy_tb.vhd:50:7:@150ns:(report note): stream = 3
buggy_tb.vhd:52:9:@150ns:(report note): ERROR: result should be even !
buggy_tb.vhd:50:7:@170ns:(report note): stream = 5
buggy_tb.vhd:52:9:@170ns:(report note): ERROR: result should be even !
buggy_tb.vhd:50:7:@190ns:(report note): stream = 7
buggy_tb.vhd:52:9:@190ns:(report note): ERROR: result should be even !
buggy_tb.vhd:50:7:@210ns:(report note): stream = 9
```

```
dff_1 : process(reset_n, clk)
begin
  if reset_n = '0' then
    a <= to_unsigned(0, 8);
  elsif rising_edge(clk) then
    a <= stream_in;
  end if;
end process;

clk2 <= clk;

dff_2 : process(reset_n, clk2)
begin
  if reset_n = '0' then
    b <= to_unsigned(0, 8);
  elsif rising_edge(clk2) then
    b <= stream_in;
  end if;
end process;

dff_3 : process(reset_n, clk)
begin
  if reset_n = '0' then
    stream_out <= to_unsigned(0, 8);
  elsif rising_edge(clk) then
    stream_out <= a+b;
  end if;
end process;
```

- VHDL possède 2 sémantiques distinctes
  - Sémantique de simulation
  - Sémantique de synthèse
- Le temps n'est pas traité de la même manière dans les deux cas
  - Le simulateur procède par seul respect de la causalité : propagation de proche en proche (*delta delay infinitésimal*) de la valeur des signaux, à chaque pas de simulation.
  - Le synthétiseur remplace l'*assignation* par une ligne équipotentielle, supposée (à juste titre) propager l'information suffisamment vite au regard de l'horloge physique du circuit.
    - Le circuit sur FPGA ne rate aucune synchronisation d'horloges
    - Le circuit simulé rate des synchronisations d'horloges, tout en respectant la causalité.
      - Sorte d' « événement macroscopique » non compris par le simulateur

- Chaque process est évalué, jusqu'à un point de synchronisation (VHDL : wait)
- Les modifications futures de chaque signal sont enregistrées dans une FIFO, avec un « timestamp »
  - Temps physique ou temps infinitésimal (delta-délai)
  - Au point d'arrêt, le simulateur prend la première valeur de la première FIFO, actualise le signal correspondant, et relance les processus sensibles à ce signal
  - Quand il n'y a plus de transactions dans les FIFO : c'est la fin de la simulation
  - Fin par « Epuisement d'événements »





Valeur courante transactions

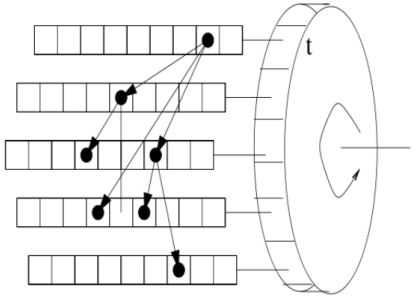
		15	23	31	33	36
S:	'0'	'1'	'0'	'1'	'0'	'1'

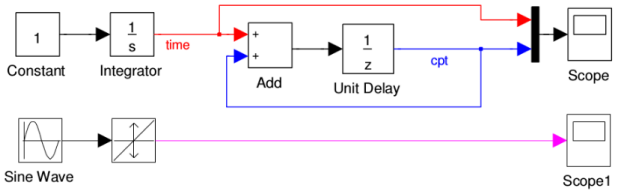
Sémantique de la plupart des simulateurs

- Hardware (VHDL,...), Software (Rhapsody,...)
- Mais aussi : simulateur de foules, de scenario militaires,...

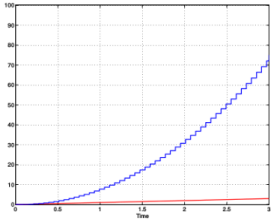
MoC qui repose sur un mécanisme central :

- échéancier

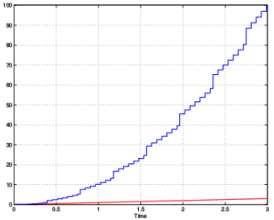




Basic model



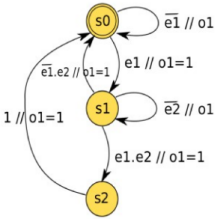
with Sine Wave



- The shape of cpt depends on the steps chosen by the solver.
- Putting another component in parallel can change the result.

2016, Marc Pouzet, Building a Hybrid Systems Modeler from Synchronous Language Principles

FSM Causale :



Pour que la FSM résultante soit considérée comme correcte ou "consistante", on doit examiner les transistions partant de chaque état :

— Les conditions sur les transitions doivent être collectivement exhaustives : il existe forcément une transition à '1'.

$$\forall s \in S : \sum_i c_i(s) = 1$$ Réactivité

— Les conditions sur les transitions doivent être mutuellement exclusives : on ne peut pas avoir deux transitions à '1' en même temps.

$$\forall s \in S, \forall (i,j), i \neq j : c_i(s).c_j(s) = 0$$ Déterminisme

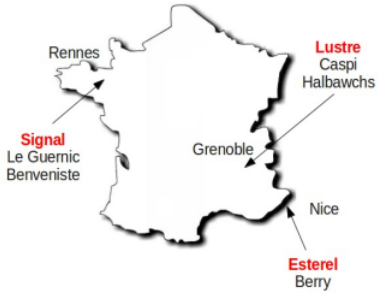
Que se passe-t-il lorsque ces impératifs ne sont pas respectés ?

Risques élévés de présence de principes implicites

1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

6– Langages Synchrones

- Esterel, Lustre, Signal
  - Langages possédant une sémantique temporelle très forte
  - Inspirés des circuits numériques, mais servant surtout à faire du logiciel
  - Idée du synchronisme parfait :
    - Les temps de calcul sont réduits à 0 en première instance
    - les signaux possèdent des indices = suites numériques
    - Ces indices correspondent à des instants de présence des signaux = horloges
  - Permettant d'analyser des comportements
    - Mathématiquement !
    - De les raffiner
    - De générer du code : processus + communications
  - Notion d'horloges logiques



Reactive Systems[HP] combine two main characteristics:

- They are continuously running systems, not intended to terminate. Thus, they do not fall into the class of traditional programs which are executed with some data, and which terminate after a while by producing a result. On the contrary, reactive systems interact continuously with the environment.

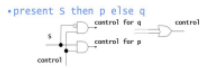
- In response to an activation, a reactive system reacts, depending on the environment state, by changing it, then it waits for the next activation, an so on without ever ending. Reactions to activations are called *instants*.

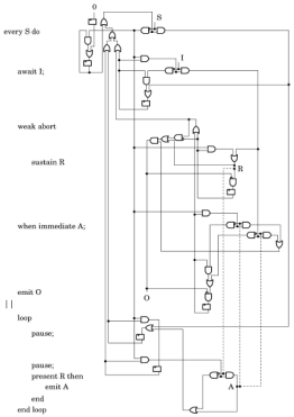
At the communication level, broadcast of information between parallel components has several advantages, compared to traditional communication mechanisms as message passing or *rendezvous*:

- It is simple, intuitive, and powerful; the same information is, for example, transmitted to several receivers in one single operation.

- It allows a modular approach, as new receivers can be dynamically added to the system without inducing any change to emitters.

Intuition à travers Esterel

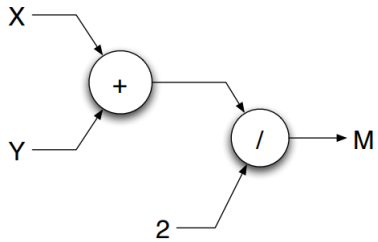




- Chaque instruction peut être traduite en un circuit / équation
- La composition des instructions correspond à l'aboutement des circuits

`emit s`	Makes signal `s` present immediately
`present s then P else Q end`	If signal `s` is present, performs `P` otherwise `Q`
`pause`	Stops the current thread of control until the next reaction
`P ; Q`	Runs `P` then `Q`
`loop P end`	Repeats `P` forever
`loop P each s`	Runs `P`, if `s` occurs while `P` is still active, then `P` is aborted and immediately restarted again; if `P` terminates before `s` occurs, one has to wait for `s` and restart `P` again
`await s`	Pauses until the next reaction in which `s` is present
`P || Q`	Starts `P` and `Q` together; terminates when both have terminated
`abort P when s`	Runs `P` up to: either `P` is finished, or a reaction (not included) in which `s` is present while `P` is not finished yet
`suspend P when s`	Runs `P` except when `s` is present
`sustain s`	Means `loop emit s; pause end`
`run M`	Expands to code for module `M`

- Approche flot de données : classique en automatique et conception de circuits




```
node Moyenne (X, Y : int)
returns (M: int);
let
    M = (X+Y)/2;
tel
```

- Interprétation synchrone : temps = IN

$$\forall t \in IN \quad M_t = \left(X_t + Y_t\right)/2$$

- affectation d'un flot de sortie ou local : `X = ...`
- Initialisation : `flot -> ...`
- Parallélisme : `EQ1 ; EQ2`
- valeur précédente : `pre(flot)`
- alternative : `if cond then exp else exp`
- gestion d'horloge : `exp when cond`
- valeurs booléennes (`bool`) : `true`, `false`, `or`, `not`, `and`, `xor`, `#`
- valeurs numériques : (`int`, `real`) `+`, `-`, `*`, `/`, `div`, `mod`, `**`, `<>`, `<`, `>`, `<=`, `>=`, `int(exp)`, `real(exp)`
- appel d'un nœud : `MyNode(F1,F2)`
- tuples `(F1,F2)=(EQ1,EQ2)` ; `(s,u)=MyNode(F1,F2);`

[Crédits : JF Susini, CNAM 2011]

```
node Max (A,B: int) returns (M: int);
let
    M = if A >= B then A else B;
tel
```

A	1	3	6	8	1	2	...
B	5	2	5	2	5	2	...
M	5	3	6	8	5	2	...

[Crédits : JF Susini, CNAM 2011]

- Sous échantillonnage avec l'opérateur when
  - Définir un flot plus lent que le rythme principal

X	***12***	***5***	***7***	***-2***	***21***	***0***
C	***vrai***	***faux***	***faux***	***vrai***	***vrai***	***faux***
X when C	***12***	***-***	***-***	***-2***	***21***	***-***

  - Quand C est faux X when C n'existe pas
  - On ne peut utiliser que des flots de même horloge
    - `X +(X when C)` est interdit

Augmenter la modularité et la compositionnalité.

In contrast to synchronous systems, polychronous specifications [16], [26] are based on a partially ordered model of time. Partially ordered time allows one to express asynchronous computations which possibly need to synchronize intermittently. As the name suggests, polychrony makes use of several clocks, which means that signals do not need to be present at all instants. Since the used clocks may not imply each other, polychronous models are not based on a linear model of time, so that the reactions of a polychronous system are only partially ordered. Two instants can be only compared on the time scale if both contain events of a shared signal x.

Another aspect of polychronous specifications is that they are *relational*, rather than *functional*. A polychronous behavior is not described in an operational way, but rather, it is constrained by relational clauses.

expression	causal dependencies
x (signal)	$\widehat{x} \xrightarrow{\widehat{x}} x$
$y := \mathrm{f}\,(x_1, \ldots x_n)$	$x_1 \xrightarrow{\widehat{y}} y, \ldots, x_n \xrightarrow{\widehat{y}} y$
$y := x$ **$ init** c	———
$y := x_1$ **when** x_2	$x_1 \xrightarrow{\widehat{y}} y$
$y := x_1$ **default** x_2	$x_1 \xrightarrow{\widehat{x_1}} y, x_2 \xrightarrow{\widehat{x_2} \wedge \neg \widehat{x_1}} y$

Plan

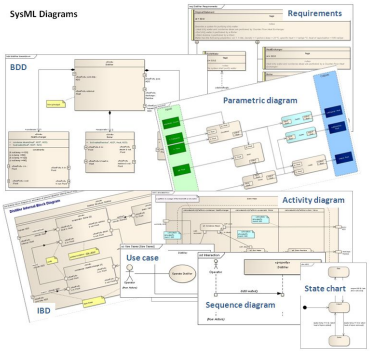
1. **Introduction**
   - Rappels historiques
   - Architecture des ordinateurs
2. **Parallélismes des machines**
   - Premiers types de parallélismes
   - Parallélisme et concurrence
   - Taxonomie de Flynn
   - Types de machines
3. **Parallélisme des applications**
   - Parallélisme d'instructions
   - Parallélisme de tâches
   - Parallélisation automatique
   - Partitionnement des données
5. **Parallélisme, langages et API**
   - Overview : threads, openMP, openCl, etc
   - Exemples en C & C++
   - Langages exotiques
6. Sujets de **recherche**
   - GPU & MPPA
   - Electronique reconfigurable
   - Co-design HW/SW & HLS
7. Travaux dirigés
   - Exo : bit-level
   - Exo : ILP et microarchitecture
   - Exo : openMP

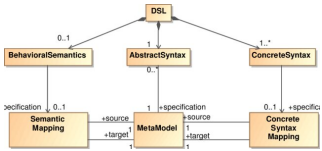
7– Sujets de Recherche
En guise de conclusion

Problème du parallélisme

Comment rendre la modélisation plus naturelle ?

- Faciliter la *capture applicative* du spécialiste.
  - Et préserver le parallélisme intrinsèque
- Plus généralement, ouvrir les environnements de compilation
  - Langages et Front-ends
  - Chaîne de compilation *spécialisée*
  - Agilité au regard de la cible. Non adhérence technologique.
- Nombreuses difficultés scientifiques & techniques & "sociologiques"
  - Standardisation de ces outils "meta" ?
  - Maîtrise effective du parallélsime ?
  - Illusion du *no-code* !

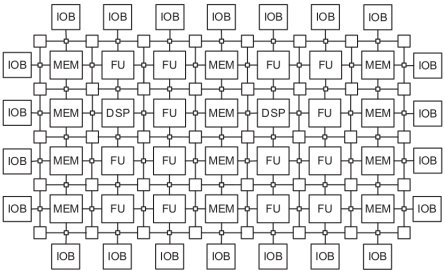




Idée

Construire sa plateforme *à façon*.

- Pas de choix de plateforme *a priori*
  - Platforme reconfigurable "nue"
- Plusieurs options :
  1. FPGA : Field Programmable Gate Array
     - Intel (Altera)
     - AMD (Xilinx)
  2. CGRA : Coarse Grain Reconfigurable Array
  3. MPPA : Massively Programmable Processor Array



ESL

Electronic System Level

- Etude de l'Adéquation algorithme-architecture
- Co-design HW/SW
- Génération d'accélérateurs spécialisés

DSA

Domain-specific Architecture

Platforme reconfigurable "nue"