Programmation HPC

Martin Quinson

Présentation du module HPC

- Ce cours vise à vous donner une bonne pratique de la programmation, en renforçant votre compréhension du fonctionnement des systèmes informatiques.
- Ce n'est pas un cours où on va apprendre beaucoup de concepts, mais plutôt de la pratique. L'objectif est qu'à chaque séance, il y ait une partie de cours assez courte au début, puis au moins une heure de pratique ensuite. Il y aura aussi besoin de programmer d'une fois sur l'autre.
- On va faire du MPI, qui est une façon de répartir les calculs qui sont trop gros/long pour tenir sur un seul coeur de calcul.
  - L'un des points importants du modèle de programmation est ce qu'on appelle SPMD (single program, multiple data).
  - Mais on a besoin d'un petit retour sur l'architecture des ordinateurs + les modèles de programmation pour comprendre ce que veut dire SPMD

I) Performance des ordinateurs et parallélisme

I.1) Motivation: parallélisme et distribution

- Certains systèmes sont géographiquement répartis
  - Accéder à des ressources distantes, ou faire collaborer des humains loin les uns des autres
  - Le web et internet, IoT, sensor networks
  - Dans ce cas, on n'a pas le choix, on doit faire collaborer des entités travaillant en parallèle
- Mais parfois, on utilise plusieurs ressources pour des questions de performance : c'est le parallélisme
  - Calculer plus vite / plus gros: **Calcul à Haute Performance (HPC)**
    - On découpe les choses à calculer pour fédérer la puissance de calcul de plusieurs unités

    - Science computationnelle, ingénierie par simulation, météo, nucléaire, médical, astro
  - Répondre plus vite: **Services cloud, etc**
    - Les requêtes arrivent dans une file d'attente, et plusieurs serveurs servent les requêtes dans l'ordre
    - Dépendances entre requêtes => graphe de micro-services
    - Context cloud, fog, microservices
- Problématique de l'énergie qui apparaît, car ces ressources sont extrêmement energivore
  - Optimiser pour le temps ou économiser des ressources est "facile", mais on ne sait pas bien optimiser pour l'énergie
  - Parfois, calculer le plus vite possible pour éteindre la ressource. Parfois, non.
  - Le matériel ne nous aide pas dans cette réflexion, qui est mal défrichée. En plus, effet rebond.
  - Ce qui semble sûr, c'est qu'il faut arrêter la course au gigantisme. Mais ce n'est pas à la mode

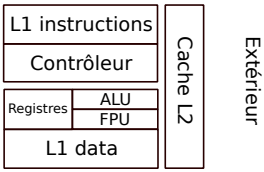
I.2) Architecture d'un ordinateur

- Rappel du cours de PCR/arcsys de l'an passé
- un ordinateur = unité de cacul (CPU) + mémoire et périphériques + bus d'interconnexion

I.2.a) CPU

- Composants du CPU
  - Unités fonctionnelles, qui font les calcul.
    - ALU: Arithmetic/Logical Unit: logique booléenne et travail sur les entiers
    - FPU: Floating Point Unit: calculs sur les nombres à virgule flottante
  - Contrôleur, qui coordonne les calculs
  - Zones de stockage, certaines pour les instructions, d'autres pour les données, ou encore mixtes

- Principe général (à 3 Ghz)
  - *Fetch:* Le contrôleur récupère l'instruction suivante du programme
  - *Decode:* Le contrôleur décode l'instruction, sépare les opérandes
  - *Execute:* L'ALU ou FPU exécute l'instruction
  - Le résultat est stocké dans un registre



I.2.b) La mémoire

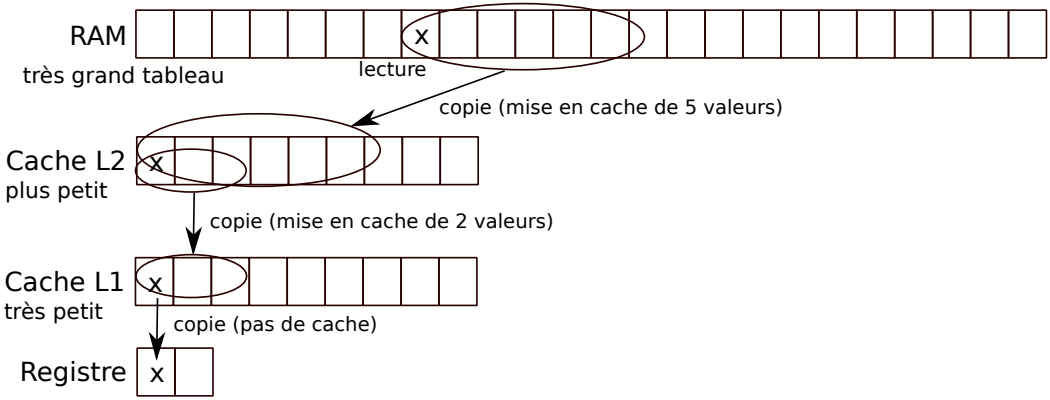
- Pyramide d'accès: On peut représenter les types de mémoire par un triangle avec les ordres de grandeurs suivants
  - Registre: <512 octets pour <1ns
  - Caches: ko-Mo pour 5ns
  - RAM: Go pour 10-30 ns
  - Stockage de masse: To-Po pour 10ms
- Memory wall:
  - Latences mises à l'échelle, pour les ordre de grandeur. Imaginons qu'un cycle fasse une seconde

1 cycle CPU	0,3ns	1s
cache L1	1ns	3s
cache L2	3ns	9s
cache L3	13ns	43s
RAM	120ns	6mn
SSD disk	50-150μs	2-6 jours
disque rotationel	1-10ms	1-12 mois
Internet Oslo/Madrid ou SF/NY	40ms	4 ans
Internet transcontinental	180ms	18 ans
TCP retransmit	1-3s	100-300 ans

  - Le CPU va très, très très vite: 0,3ns par cycle, ça fait 10 cycles quand la lumière parcourt 1m. Le défi est donc de nourrir le CPU de données sans pause
  - C'est aussi pour ça que les registres ont une petite capacité: ils doivent être petits en taille pour être proche des unités fonctionnelles. S'ils étaient plus gros, la lumière mettrait plus de temps
  - Le memory wall s'aggrave: Vitesse CPU croit de 55%/an et celle mémoire de 10%/an.

I.2.c) Les mémoires caches

- L'étymologie vient des trappeurs nord-américains. Ils apportaient du matériel sur un chariot jusqu'à une sorte de camp 0, puis ils cachaient ce qu'ils ne pouvaient pas porter pour l'expédition.
- Dans l'ordi, on a un pb de latence entre la mémoire et l'ALU, mais pas de bande passante. On va donc rapatrier plus de données que demandées, au cas où on aurait ensuite besoin de la donnée juste après en mémoire.
- une mémoire cache est une petite réserve de mémoire proche du coeur du CPU



- La fois suivante qu'on accède aux données, on regarde d'abord si elle n'est pas déja stockée dans un niveau de cache intermédiaire. Si oui, on a économisé la latence. Si non, on charge la ligne de cache nécessaire, en virant une autre ligne au besoin. Pleins d'algos différents de gestion des caches, mais LRU (least recently used eviction) marche bien en pratique.
- "Coup de chance", les accès mémoire sont effectivement souvent linéaires. En fait, on programme exprès pour, et le compilo tente de démeller les accès et les données pour fluidifier.
  - En HPC, on cherche même à faire des structures de données *cache oblivious*, cad optimisées pour n'importe quelle taille de cache.
  - C'est ce qui explique la différence de code entre les deux programmes de copie de matrice (4ms vs. 80ms pour une matrice 2048): l'un utilise très efficacement les caches, l'autre passe son temps à les invalider après chaque accès.
- Pour les écritures en mémoire, qques subtilités si le temps le permet
  - writeback: on n'écrit que dans le premier niveau de cache sans propager plus loin. Ca va plus vite à l'écriture, mais il faut faire attention quand on élimine une ligne de cache, et on va s'attirer des problèmes de cohérence en multicore. Surtout avec des architecture NUMA (non-uniform memory access), voir plus

loin.

- writethrough: on écrit à la fois dans le cache (au cas où on voudrait relire la donnée ensuite) et dans la mémoire centrale. C'est bien plus lent, mais il n'y a plus de probleme de cohérence.

I.3) Stratégies historiques des CPUs plus performants

I.3.a) Miniaturisation

- Pour que la lumière porte l'info plus vite.
- Moteur principal jusqu'au années 1990, moins efficace maintenant car c'est vraiment petit. C'était une solution hardware, trobien.
- Gravure de plus en plus fine (`Wikipedia:32_nm_process`)
  - 1970: 10μm ($10 \cdot 10^{-6}$); 1982: 1,5μm; 2001: 130nm; 2014: 14nm; 2017: 10nm; 2020: 5nm
  - Arduino à 40nm est une technologie de 2008 pour Intel, d'où le prix actuel
  - Un cristal de silicium est de l'ordre du nanomètre
  - En dessous de 7nm, les électrons fuitent des transistors par effet tunnel...
  - Les CPU modernes sont multi-couches, ce qui rend le pb de placement encore plus complexe

I.3.b) Fréquence

- Augmenter la fréquence augmente mécaniquement le nombre de top d'horloge. Encore une solution hardware.
  - Quand la fréquence croit, la quantité de calculs croit linéairement
  - Mais la quantité d'énergie croit quadratiquement. 50% plus vite, 2 fois plus d'énergie.
- L'énergie est un problème cruel en informatique hautes performances

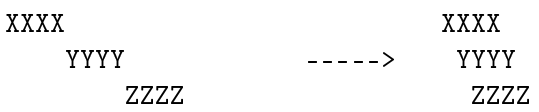
I.3.c) Nombre de transistors

- Loi de Moore (ex-président d'Intel) = Le nombre de transistors double tous les 18 mois.
- C'est une prédiction autoréalisante puisque Moore est un ex-PDG d'Intel et que la companie a comme business plan que tous les ordinateurs sont remplacés tous les 18 mois.
- 1971: 2000 transistors sur un 4004; 2010: 2.6 milliards sur xéon; 2020: 40 milliards AMD epyc

- Les transistors supplémentaires sont utilisés par exemple pour ajouter des instructions (de mots au vocabulaire du processeur).
  - Plus d'instructions: plus de séquences directement inscrites dans le silicium, donc plus rapide et plus gros/complexe
  - C'est exactement la différence entre RISC vs. CISC: Reduced/Complex Instruction Set CPU.
  - Un RISC est souvent plus petit et moins energivore qu'un CISC, mais on ne peut pas dire que l'un est mieux que l'autre : c'est un autre compromis entre complexité du CPU vs. complexité des applications
  - FPGA: aucune instruction inscrite dans le silicium, tout est interprété
    * permet des performances intermédiaires entre interprétation pure par un processeur générique et ISA spécifique dans le silicium

I.3.d) Architectures superscalaires

- **Pipelining** = faire plusieurs opérations en même temps au niveau du silicium
  - Parfois, les instructions CPU prennent plus d'un cycle (surtout en CISC mais pas que). Imaginons que la multiplication prenne 4 cycles. Si on veut faire 3 multiplications (X Y et Z ci dessous), ça fait 12 cycles.
  - On découpe ces opérations complexes en morceau qui prennent un cycle chacun. Les sous-instructions sont chainées, mais du coup on peut commencer l'opération suivante avant la fin de la première, en réutilisant cette zone du circuit. On fait alors les 3 multiplications en 6 cycles.
  - On peut aussi raccourcir les cycles (augmenter la fréquence) en découpant le Fetch/Decode/Execute en 3 étapes séparées qui rentre dans le pipeline



- La difficulté est ensuite d'assurer que les pipelines restent bien pleins, ie d'éviter les *pipe holes* qui apparaissent quand on a besoin de la fin d'un calcul pour savoir quoi faire ensuite.
- **Exécution spéculative**. On tente de prédire le résultat d'un branchement et commencer les calculs suivants. Si on a bien prédit, on a déjà commencé le calcul suivant. Si on s'est trompé, on annule le calcul spéculatif et c'est pas grave.
- Les compilateurs font énormément de transformations du code pour rendre le code plus sympa pour les pipelines, et certains savent même benchmarker quelques exécutions pour faire des binaires où la prédiction de branches est pré-

cise pour aller plus vite. Ce n'est plus une solution hardware, mais améliorer les compilateurs profite à toutes les applications.

- **Hyperthreading**: Notion du marketing Intel (=solution hardware, parfois survendue)
  - L'idée générale est d'avoir plusieurs flots de calcul applicatifs en même temps sur le même coeur. On ne dédouble que 15% de la surface du coeur (registres de contrôle et registres généraux), et on partage les pipelines de calcul
  - Cela permet d'optimiser le remplissage des pipelines avec les instructions des deux applications.
- **Architectures superscalaires en désordre** (out of order)
  - Le CPU contient un moteur d'exécution qui analyse les dépendances entre les instructions et les réordonne à chaud pour optimiser le remplissage
- C'est quand les infrastructures deviennent aussi complexes qu'on peut avoir des "bug matériel" comme Meltdown qui était une race condition dans l'exécuteur out-of-order de Intel entre la lecture mémoire et la vérification des droits, exploitable avec un side-channel permettant de deviner l'état de la mémoire d'après l'état des caches, partagés entre les applications. Cela force l'OS à ne plus avoir confiance dans la protection de la mémoire, et perdre 5 à 30% de son temps à effacer la mémoire sensible des lignes de cache.

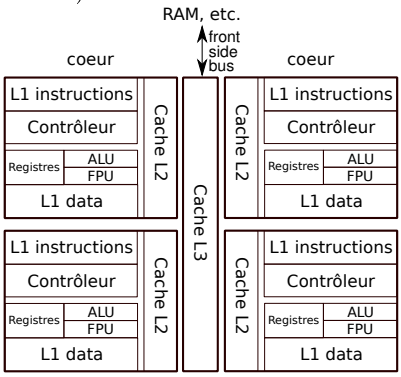
I.3.e) Architectures vectorielles

- On fait appliquer le même calcul à plusieurs données à la fois. 32bits vs 64bits ressemble un peu, mais on parle là d'ajouter deux vecteurs entre eux par exemple.
- Processeur x86.
  - on a des registres plus grands SSE=128bits; AVX=256bits; AVX512=512bits
  - On applique les opérations au registre directement
- GPU: unité de calcul spécialisée dans le calcul vectoriel sur nombres flottants. Parallélisme de masse.
  - c'est très dépendant du vendeur, y'a bcp de technologies que je ne maîtrise pas
  - Contient des ALU mais vraiment pas efficace pour les calculs génériques. Comparable à une formule 1: très rapide si le calcul est très régulier, nul sinon.

I.3.f) Architecture parallèle

- Puisqu'on peut plus faire d'unité plus rapide, multiplions les unités

  - idée inverse de la fréquence: deux coeurs underclockés de 20% consomment autant qu'un seul coeur, mais produisent 150% de calculs
  - c'est une solution terriblement software, car ça demande de reprendre l'intégralité des programmes. C'est d'ailleurs tout le sujet de ce module, même si on ne va pas explorer la complexité algorithmique posée
- SMP: symetrical multi-processor.
  - Plusieurs processeurs identiques, chacun ses caches, RAM pour tous.
  - Exécution en parallèle de programmes (ou threads) différents
- Multicoeur: on ne dupplique pas entièrement les CPU, les caches peuvent être partagés
  - L'hyperthreading peut être vu comme une forme de multicoeur au sein du coeur
  - Caches partagés entre les cores → pb de cohérence
  - NUMA (non uniform mem archi): mémoire plus ou moins proche: on connecte des cartes mères.

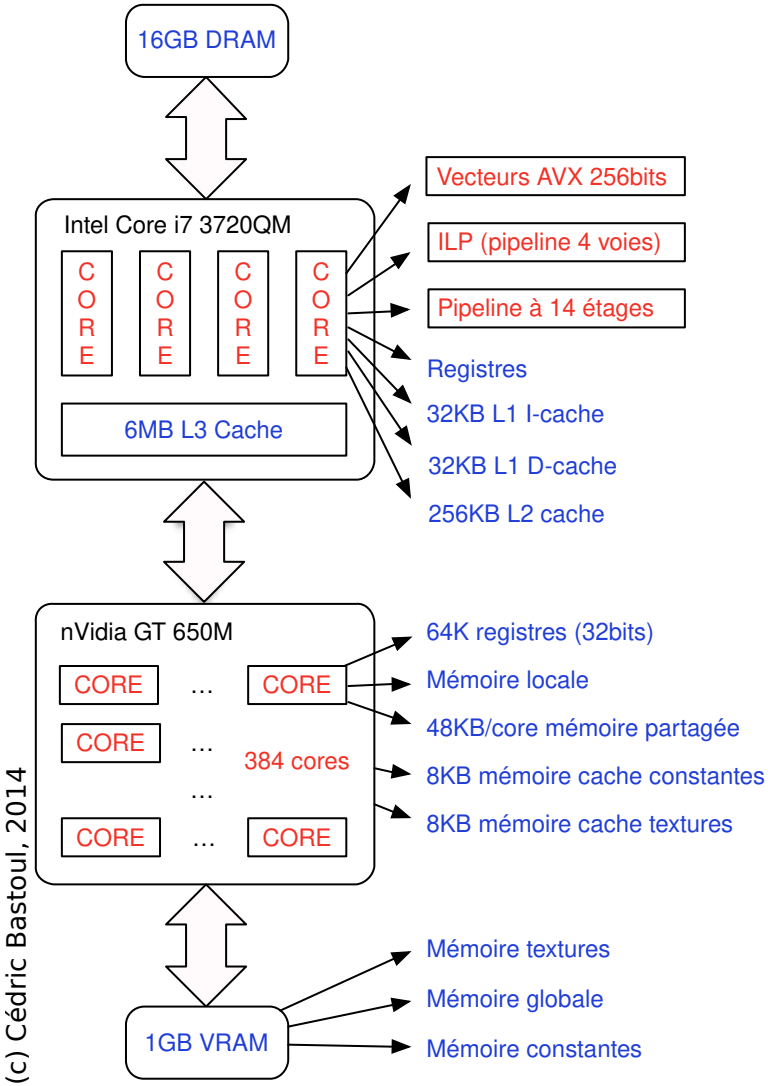


- Multicore hétérogène: le processeur Cell (2005) pour Playstation3: un PowerPC peu puissant avec 8 gros coeurs type GPU. De beaux jouets, mais ultra durs à programmer efficacement. La PS4 a un design plus classique (CPU + GPU classiques sur même puce)
  - Intel/AMD voudraient fondre des systemes tjs plus gros (SoC: System on Chip) mais on a du mal à les programmer
- Grappe de calcul / cluster : réseau d'ordinateurs collaborant à une tâche
  - On peut faire ça avec des ordis de bureau (cluster) ou des noeuds spécialisés (supercomputer ou cloud)
- Aller plus loin: `http://polaris.imag.fr/arnaud.legrand/teaching/2015/M2R_PC.php`

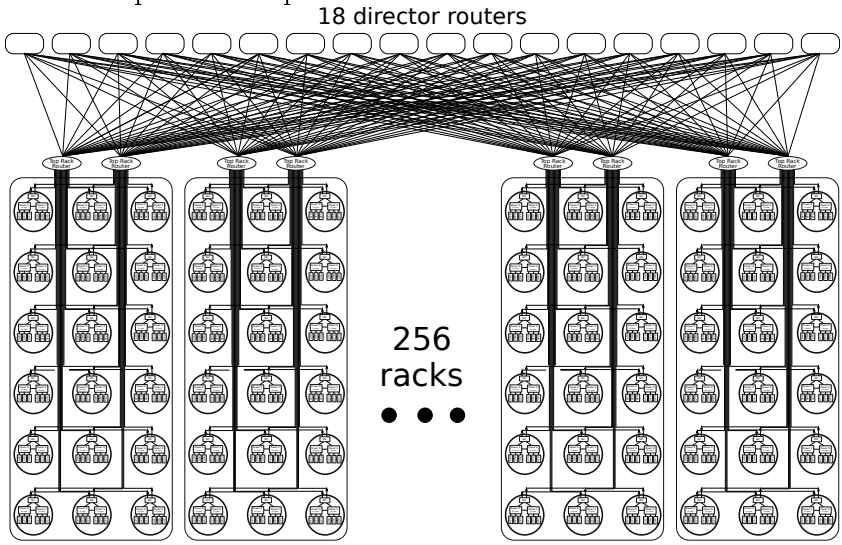
I.4) Quelques exemples d'ordinateurs parallèles

- On retrouve TOUT les mécanismes en même temps, c'est l'enfer à utiliser efficacement
- Mon ordinateur (achat décembre 2019):
  - processeur Core i7 "Coffee-Lake": 6 coeurs, 16 threads
    - L1 cache par coeur (partagé entre les threads): 32 KiB instruction, 32 KiB data

    * L2 cache par coeur: 256 KiB
    * L3 en ring entre les coeurs: 12 Mib
    * Vecteurs AVX512, etc. out-of-order, bien sûr
    * `https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake`
- NVIDIA T1000M
    * 896 cores
    * 4Gb mémoire en 128bits de large
    * 4 700 millions de transistors, gravé à 12nm
    * 50 W
    * `https://www.techpowerup.com/gpu-specs/t1000.c3797`
- 16 Gb de DRAM
- Disque dur SSD NVMe (non-volatil memory express)


16GB DRAM
Intel Core i7 3720QM
CORE
CORE
CORE
CORE
6MB L3 Cache
Vecteurs AVX 256bits
ILP (pipeline 4 voies)
Pipeline à 14 étages
Registres
32KB L1 I-cache
32KB L1 D-cache
256KB L2 cache
nVidia GT 650M
CORE
CORE
CORE
CORE
CORE
CORE
384 cores
64K registres (32bits)
Mémoire locale
48KB/core mémoire partagée
8KB mémoire cache constantes
8KB mémoire cache textures
1GB VRAM
Mémoire textures
Mémoire globale
Mémoire constantes
(c) Cédric Bastoul, 2014

- Le supercalculateur Summit, top 4 en juin 22 (#1 11/18 - 11/19)
  - 1 noeud = 2 cpu Power9 (22 coeurs ARM, 4 threads/coeur) + 6 GPU Volta + 512Gb mém SMP + 96 Gb mém NVIDIA + 1.6Tb NVMe
  - 1 rack = 18 noeuds interconnectés en infiniband
  - Ordinateur = 256 racks de calcul, 18 racks de switchs, 40 de stockage
  - Total: 4608 noeuds, 2 414 592 coeurs ARM, 27 648 GPU Volta. 150Pflops Rmax.
  - 340 tonnes (comme un boing), 2 terrains tennis au sol, 300km de fibre optique
  - 15m³ d'eau par minute pour évacuer 13MW de chaleur



- Fugaku, top2: 160k noeuds, 7.6M coeurs, pas de GPU. 440Pflops Rmax, 26MW
  - Processeur spécifique: A64FX (4 NUMA de 12 coeurs ARM chaque) vectoriel 512bits
  - 1 milliard de dollars, contre 300M pour Summit et ses amis
- Vieillissement des supercalculateurs: exponentiel
  - TOP5 déconnecté du reste (juste des courses de formule 1)
  - Le premier le reste un an ou deux, puis passe dernier en ≈ 8 ans
  - Mon PC a les perfs d'un ordi du Top500 y'a 10 ans
  - Mon téléphone celles d'un ordi du Top500 y'a 20 ans
- Mais on a dit qu'on arrêtait le gigantisme

II) Programmer pour les performances

II.1) Algorithmiques distribuée, parallèle et concurrente

- Quelle est la différence entre distribué, parallèle et concurrent ? Ce n'est pas du tout la même algorithmique.
- $\frac{\text{temps calcul}}{\text{temps comm}} \approx 0$: algorithmique distribuée
  - Le temps de calcul est négligeable devant les temps de communication.
  - Bcp de recherche car l'algorithmique est belle, c'est le cadre du module d'algo dist de 1A
  - Métrique classique: nombre de messages en fct nombre de noeuds (assez théorique)
- $\frac{\text{temps calcul}}{\text{temps comm}} \approx \infty$: algorithmique concurrente
  - Le temps de communication est négligeable devant les temps de calcul.
  - Application multithreadée. Y'a bcp d'usage, car les processeurs sont multi-coeurs.
  - L'algorithmique est assez compliquée, car ce n'est pas reproductible. Tester est difficile
- $\frac{\text{temps calcul}}{\text{temps comm}} \approx 1$: algorithmique parallèle
  - C'est le cadre de ce module.
  - La pratique est souvent plus difficile que l'algorithmique, qui est un peu facile
  - métrique classique: makespan (temps d'arrêt du dernier processus)

II.2) Difficultés

- Raisonner sur un système demande de:
  - définir un **état** du système (de la mémoire)
  - définir un **ordre** entre les événements et actions
- Si le système est distribué géographiquement,
  - Pas de mémoire centrale, donc pas d'état connu de tous
  - Pas d'horloge commune, donc l'ordre des événements n'est pas communément admis
  - Les communications sont asynchrones (pas de borne sup sur le temps) car charge variable sur les ressources partagées
  - Il peut y avoir des pannes (on peut vouloir gérer les pannes partielles du système)
- C'est déjà le cas dans un système multicoeurs avec des caches spécifiques à chaque coeur

II.3) Taxonomie de Flynn

- On classifie les modèles de calcul en fonction des données et des instructions
- SISD (Single Instruction, Single Data)
  - Modèle séquentiel, architecture de Von Neumann
- SIMD (Single Instruction, Multiple Data)
  - Processeur vectoriel. Calcul sur tout un vecteur en même temps
- MISD (Multiple Instruction, Single Data)
  - Redondance des systèmes critiques ? Peu d'implémentations en pratique.
- MIMD (Multiple Instruction, Multiple Data)
  - Plusieurs unités de calcul, chacune avec ses occupations propres dans sa mémoire
- Dans cette taxonomie, 3 cases sont intéressantes:
  - SISD: nom compliqué pour algo séquentielle, celle que vous connaissez
  - MIMD: multithread et parallélisme asymétrique : chacun fait son truc dans son coin. C'est DUR de synchroniser correctement
    - Dans un cours d'algo dist, au moins un tiers est consacré à la notion de consensus. Faire en sorte que les acteurs se mettent d'accord
    - Ordonner les événements est une autre source de bonheur intense (ordre causal)
    - Le simple fait de tester un programme est difficile, car non reproductible
  - SIMD: une sorte de mix entre les deux: une sorte d'algorithme séquentiel, mais qui s'applique à bcp de données en même temps.
    - C'est donc une ruse pour simplifier l'écriture des programmes, qui n'ont pas à réfléchir à la correction tout en calculant plus vite
    - Cette efficacité sans problème théorique explique le succès pratique de cette approche
    - C'est le premier objectif de ce module (la première année, je sais pas jusqu'où on pourra aller)
- En MPI, ce n'est pas SIMD mais SPMD, pour dire que c'est single program, multiple data.
  - Tous les ranks exécutent le même programme, mais ce programme peut contenir des branchements en fonction du rank. C'est possible, même si ce n'est pas très idiomatique.
  - En général, le rank 0 charge les données, les envoie à tous les autres, tout le monde calcule (c'est la partie SIMD), puis à la fin le rank 0 regroupe les données et c'est fini.

III) Mémoire partagée et MPI

III.1) Motivation et historique

- Modèle à mémoire partagée: les données doivent être échangés explicitement.
  - Modèle assez complexe à programmer: il faut gérer le détail des mouvements mémoires, ce qui en fait une sorte d'assembleur où on doit contrôler trop de choses pour que ce soit agréable
- Les scientifiques sont de gros consommateurs de calculs
  - Tous les scientifiques modernes (physiciens, biologie, ingénierie, astrophysiciens, climatologues, géologues, etc) visent à établir un modèle qui sera ensuite calculé sur ordinateur. Dans la mesure du possible, la plupart des scientifiques préfèrent utiliser des outils comme matlab ou python qui leur permettent de rester concentrés sur leurs problèmes scientifiques, mais si beaucoup de calcul (ou beaucoup de données), il faut répartir.
  - Le savoir scientifique est présenté dans un article, mais détaillé et mis en oeuvre dans le code. L'article est la pointe de l'iceberg.
  - Ces codes tendent à devenir gros et complexes d'un POV informatique, et ils contiennent des savoirs précieux d'un POV scientifique
  - Objectifs des scientifiques: que leur programme termine (besoin de performance), éviter les problèmes logiques (risques de synchro), éviter les erreurs de calcul (stabilité numérique)
- Cette complexité des programmes scientifiques + complexité des machines => besoin de standardisation majeur
  - C'est la raison d'être du standard MPI : tout le monde l'utilise pour faire causer ses programmes, et les constructeurs se débrouillent pour faire des implémentations efficaces sur leur matériel.
  - Il existe des versions open-sources de MPI (MPICH et openMPI), pour ceux qui font des clusters de linux, mais les gros constructeurs ont leurs versions specifique, optimisée aux petits oignons
- Les fonctions des middleware MPI :
  - démarrer des programmes sur chaque machine impliquée (par exemple accès ssh + fork, mais le constructeur se débrouille)
    - centraliser les affichages stdout/stderr de tous les processus sur la machine appelante
    - attendre la fin de tous les processus
  - échanger des données efficacement entre les machines (c'est le coeur de l'API)
  - synchroniser les processus participants (très peu. à part les barrières, on fait

tout à la main)

- MPI ne se charge que des communications inter-noeuds. Ce n'est pas adapté au multithread.
  - On peut déployer un rank par noeud, mais les middleware ne sont pas optimisés, donc c'est une mauvaise idée.
  - Plusieurs solutions existent pour gérer la concurrence dans le noeud. On parle de MPI + X, avec différentes valeurs de X
  - OpenMP: pragmas ajoutées dans le code pour demander au compilateur de multithreader certaines boucles d'itération
  - CUDA/Vulkan: langages dédiés pour tirer partie des GPU et autres accélérateurs
  - diverses technos propriétaires ou expérimentales
- Tout le monde est d'accord pour dire que MPI est loin d'être optimal
  - Dur à utiliser, très dur à utiliser efficacement, très très dur à utiliser à très large échelle (exascale)
  - D'autres modèles sont proposés:
    * PGAS (partitioned global address space), modèle unifiant intra-noeud (espace privé) et inter-noeud.
      · Par exemple, OpenSHMEM où chaque processus a un tas privé et un tas partagé/symétrique (synchronisé automatiquement)
      · Mais au final ça reste très complexe à mettre en oeuvre. Ce n'est pas SPMD et il y aurait besoin d'outils formels pour la correction.
    * Graphe de tâches, bon modèle permettant de découpler l'intention des calculs de leur implémentation, avec liberté d'optim au runtime.
      · tâches indépendantes, DAG de tâches, workflow complexes, dataflows.
      · Faire un runtime efficace est un sujet de recherche actuel (StarPU l'un des meilleurs au monde), et décrire les calculs mathématiques sous cette forme est pas entièrement trivial. Il y a 3 niveaux de description: le calcul math, l'un des algorithmes qui font ce calcul, l'une des implems de l'algo choisi.
      · La correction est simple au niveau mathématique, et chaque brique est un kernel de calcul hautement optimisé. Le test logiciel classique peut suffire.
  - Mais ces modèles ont du mal à s'implanter: le milieu est extrêmement conservateur car les codes sont trop gros pour évoluer facilement, et les scientifiques veulent se concentrer sur leur science, pas apprendre une nouvelle techno s'ils n'y sont pas contraints.

- Au final, MPI a encore de beaux jours devant lui, surtout à échelle raisonnable (jusqu'à 4096 noeuds). Peut-être qu'au delà ils vont être obligés d'apprendre à utiliser les tâches.

III.2) Programmation MPI

- Chaque processus est désigné par son rang (au sein d'un communicator si on veut faire des sous-groupes)
- Tous les programmes MPI doivent commencer par `MPI_Init(&argc, &argv)` et finir par `MPI_Finalize()` pour le fork/join des processus.
- Retrouver son identité:
  - `MPI_Comm_size( MPI_COMM_WORLD, &size )` -> trouver la taille du communicateur (= combien on est)
  - `MPI_Comm_rank( MPI_COMM_WORLD, &rank )` -> trouver mon rang dans l'ensemb

III.2.a) Partie pratique : Installation de SimGrid

- On va utiliser le simulateur SimGrid pour les projets. Il se trouve que je suis contributeur du projet, mais ce n'est pas la seule raison.
  - Manque d'accès à une ressource de cacul; plus reproductible; permet des études impossibles sur vraie platf
  - Et puis c'est le meilleur simulateur de MPI existant. Sérieusement.

III.2.b) Programmes MPI typiques

- Les programmes typiques sont tous plus ou moins construits sur le même modèle
  - Personne ne fait de code affreusement inventif en MPI. Par exemple, personne n'aurait l'idée de faire un runtime MPI pour le WAN, pour faire un système distribué pair-à-pair comme TOR ou bittorrent ou autre. MPI reste confiné au monde HPC, sur supercalculateurs, avec quelques formes de programmes classiques
  - Un article de 2009 donne les 7 nains, qui sont les 7 formes de programme classique. `https://dl.acm.org/doi/pdf/10.1145/1562764.1562783` A View of the Parallel Computing Landscape, du Par Lab (Berkeley Parallel Computing Lab).
    * La liste a été étendue à une douzaine de formes d'applications par ces auteurs
    * matrice dense, matrice creuse, FFT, programmation dynamique, backtrack et Branch&Bound, grille structurée, grille non-structurée, machine

à états finie (FSM), etc.

  - On a maintenant des Proxy Apps qui sont des modèles réduits d'applications majeures, utilisées pour tester les runtimes et le matériel.
    * 14 proxyapp majeures, et environ 75 ProxyApp au catalogue du dépot américain (exascaleproject.org).
    * Les européens poussent 6 programmes phares (pas miniapp, des grosses applis)

- Il convient donc de séparer ce qu'on peut faire en MPI, de la sous-partie de ce que les gens font habituellement en MPI.
  - Les programmes MPI en pratique sont souvent extrêmement réguliers, pour correspondre aux maths et aussi pour éviter les pbs de synchro
  - Calcul itératif par phase, alternant calcul et communication: Je calcule une phase, j'échange avec les autres et je passe à la phase suivante, jusqu'à atteindre le critère de fin.
  - produit de matrice par bloc: on échange les bloc d'opérandes pour calculer le résultat. A sur les lignes B sur les colonnes.
  - Ou bien map-reduce: application du même calcul à toutes les données, puis centralisation du résultat par une opération comme max/sum/average
  - Avant et après toutes ces phases de calcul, on a un scatter/gather qui suit le fork-join du programme. Initialement, les données sont sur rank-0 qui les distribue aux participants. En fin de programme, tout doit revenir sur rank-0.
- Il y a bien des fous qui utilisent MPI à pleine puissance. En haut du Top500, au DoE/DoD. Mais ce n'est pas le sujet du module.

III.3) Communications point-à-point MPI

- C'est le coeur de l'interface, il y a bcp de fonctions. La sémantique de ces opérations a été définie par des experts HPC: Tout pour la perf, même si c'est pas formel.
- Formes de communications en MPI
  - point-à-point: entre deux ranks. Send et receive.
  - collectives: broadcast, alltoall, alltoallv, scatter, gather, barrier, etc. On y revient dans un autre chapitre.
  - one-way: on écrit dans la mémoire de l'autre. Attention, c'est une partie du MPI peu commune (et glissante) depuis MPI 2.0 seulement (1998). Pas au programme du module.
- Variantes bloquante ou asynchrone pour la plupart des fonctions

  - Collectives asynchrones depuis MPI 3.1 seulement (2015)
- La sémantique des communication MPI est troublante
  - C'est bloquant jusqu'à ce que la mémoire puisse être utilisée pour autre chose. Donc peut-être que la donnée n'est pas transmise, mais copiée dans les buffers de l'OS ou de la carte réseau
  - En particulier, si deux processus font `MPI_Send` l'un vers l'autre en même temps, ce n'est pas forcément bloquant. Ca passe si la donnée peut être bufferisée
  - Ce point est valide que l'on utilise `MPI_Send` ou `MPI_Isend + MPI_Wait`
  - Il existe `MPI_Ssend` pour Synchronous send, qui bloque jusqu'au début de la réception par le récepteur. Peut-être que la fin n'est pas encore là.
- Les performances des communications MPI sont sur-optimisées, au point où on évite les copies mémoire
  - Si les données sont grosses, mode rendez-vous. Le message n'est envoyé qu'une fois que le récepteur a indiqué le buffer où écrire (pour éviter la mise en buffer coté récepteur)
  - Si les données sont petites, mode eager. Le message est envoyé au plus vite, et mis en buffer coté récepteur au besoin

III.4) Julia en mémoire partagée

- Le TP de la semaine est le suivant :
  - `https://simgrid.github.io/SMPI_CourseWare/topic_basics_of_distribut memory_programming/julia_set/`
    - C'est guidé à l'américaine, ça peut être utile.
  - En fait, ce que vous avez écrit la semaine dernière est très proche de ce qu'il faut faire.
    - On avait utilisé des threads, ce seront des ranks MPI dans simgrid (c'est la même chose, sauf que c'est en mémoire partagée où chaque thread voit pas les autres. Mais c'est le même système de fichiers)
  - On va utiliser des communications pour synchroniser nos ranks au lieu de mutex. Y'a deux écoles
    - soit vous passez un baton de la parole pour savoir qui doit écrire à quel moment
    - Soit vous centralisez le résultat des écritures sur rank0 (qui a une grosse mémoire par rapport aux autres). Chacun fait les calculs dans son coin, puis envoie un message à rank0 qui range les données en mémoire puis écrit tout sur disque ensuite.

- Objectif pédagogique de cette activité : que chaque rank MPI accède à sa part du tableau sans marcher sur celle des autres
  - c'est donc la base du SIMD
- Code du 2D à rendre pour le vendredi 16 (si vous voulez valider le module)

IV) Performances de programmes parallèles

- Maintenant qu'on sait programmer MPI, il est temps de parler de performances.

IV.1) Speedup théorique

- Efficacité du parallélisme = accélération.
  - $S_p = \frac{T_1}{T_p}$ (T_1 = temps avec un seul processeur; T_p = à p procs)
  - speedup à 2 procs = temps séquentiel sur temps à deux procs. Si c'est 2, c'est top
- Accélération linéaire : parallélisme optimal. Quand on s'y met à 5, on va exactement 5 fois plus vite
- Accélération sur-linéaire : attention. Rarement possible. Effets de cache ?
- Accélération sub-linéaire : ralentissement dû au parallélisme

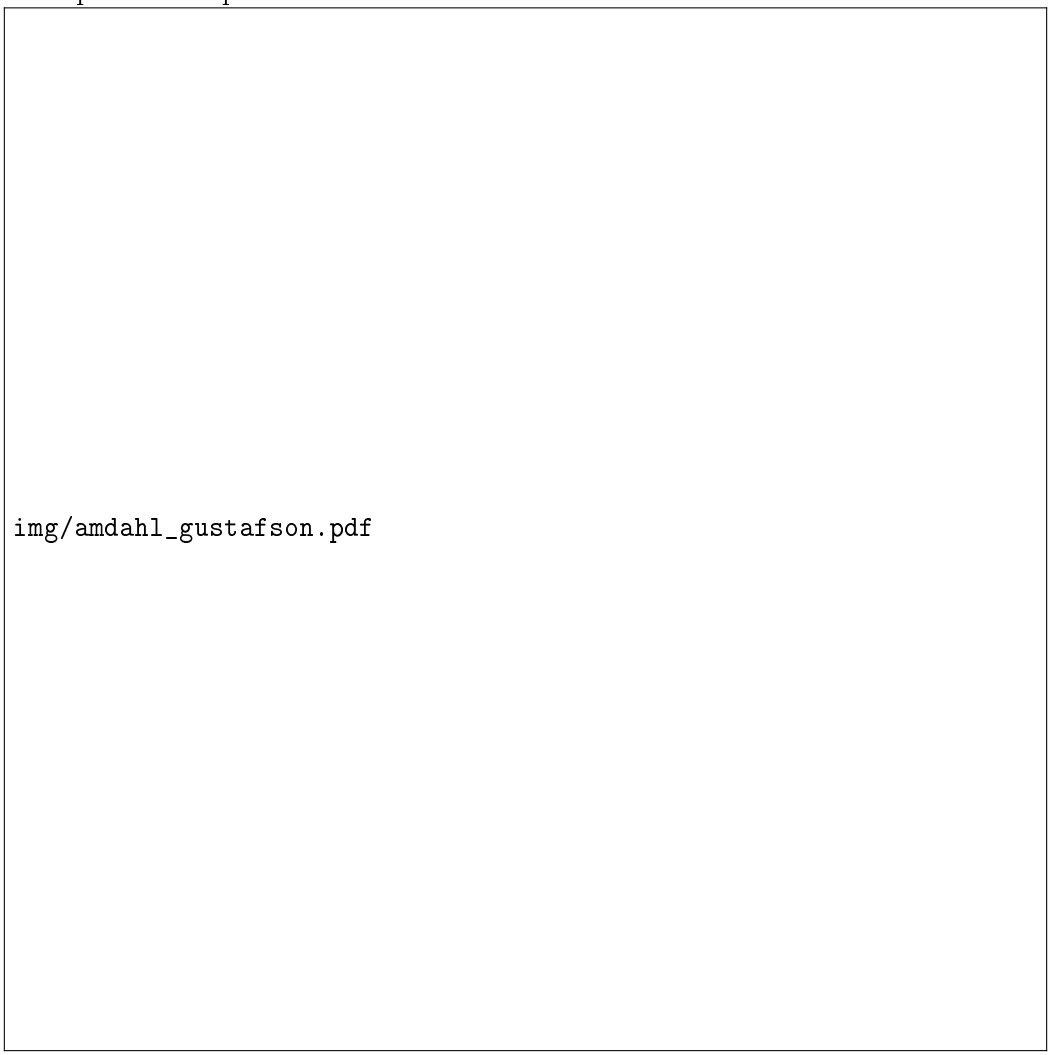
IV.1.a) Speedup optimal: loi d'Amdahl

- On suppose que mon programme a une partie purement séquentielle et le reste parallélisable. On note γ la proportion parallélisable.
  - $T_{seq} = 1$ On normalise en posant que le temps séquentiel vaut 1
  - $T_p = (1-\gamma) + \frac{\gamma}{p}$ Le temps parallèle à p processeur: la partie séquentielle ne bouge pas, le reste est accéléré
  - On obtient: $\frac{T_1}{T_p} = \frac{1}{(1-\gamma)+\frac{\gamma}{p}} = S_p$
- Cette façon de calculer le speedup est la loi d'amdahl.
  - On remarque que si $p \to \infty$: $S = \frac{1}{(1-\gamma)}$: l'accélération est limitée par la partie séquentielle
  - Si $(1-\gamma) \to 0$, on a $S_p \sim p$: l'accélération est linéaire
- Amdahl travaille à quantité de travail (taille de pb) fixe, et c'est un peu pessimiste : ça se passe rarement bien.
  - Calculons pour $\gamma = 10\%$ (c'est peu) et $p = 24$

$$S_{24} = \frac{1}{(1-0.10) + \frac{0.10}{24}} = \frac{1}{0.90416666} = 1.106 = 10\% \text{ de speedup, c'est nul}$$

IV.1.b) Loi de Gustafson

- Cette fois, on ne cherche plus à quel point on va plus vite pour faire une qte de travail fixe en parallèle, mais combien on perd si on n'a plus de parallélisme
- Avec Amdahl on connait la situation séquentiel (temps et taille) et on calcule le gain en parallèle.
  Avec Gustafson, c'est le contraire. On connait la situation parallèle et on calcule la perte en séquentiel



$$S_p = (1 - \gamma) + p \times \gamma = 1 + (p - 1)\gamma$$

- Gustafson donne des résultats bien plus optimiste puisqu'on arrête de gâcher à refaire la partie séquentielle sur chaque noeud.
  - Un processus calcule la partie non-parallélisable, les autres sse partagent le reste
  - Si $p \to \infty$ alors $S \to \infty$
- Et c'est avec ce genre de chose que les physiciens prennent une taille de grille plus grande que nécessaire et gâchent des ressources.
- On appelle *weak scaling* la mesure de Gustafson où la quantité totale de travail change (trop facile)
- On appelle *strong scaling* la mesure d'Amdahl où la quantité totale de travail est fixe (là c'est dur)
- Autre facon de l'écrire:
  - w_1 qte travail individuelle
  - Vitesse séquentielle: $\frac{w_1}{t_1}$; Vitesse parallèle : $\frac{p \times w_1}{t_p}$
  - Accélération:
$$\frac{\frac{p \times w_1}{t_p}}{\frac{w_1}{t_1}} = \frac{p \times w_1 \times t_1}{w_1 \times t_p} = p \times \frac{t_1}{t_p} = \dots$$

IV.2) Extensibilité

- Les lois précédentes sont des bornes théoriques. En pratique, il y a des facteurs limitants qui font qu'il est difficile de même atteindre ces bornes.
  - Synchronisation ajoutée pour la correction: les processus s'attendent
  - Goulet d'étranglement I/O: pas assez de données pour tout le monde. Mur mémoire du processeur
  - Algorithme pas assez extensible : deux mesures de complexité d'un algorithme: en calculs et en communication.
- Strong scalability: taille totale fixée, nb ranks augmente pour aller plus vite à taille constante
- Weak scalability: taille par rank fixée, nb ranks augmente pour calculer plus gros à temps constant

IV.3) Exercices

IV.3.a) Exercice 1: prédire l'accélération

- Un process passe 20% du temps dans un code séquentiel (I/O). Speedup pour 100 machines ?

- Amdahl:
  - $s_p = \frac{1}{(1-\gamma)+\frac{\gamma}{p}}$
  - $S_{100} = \frac{1}{(1-0.8)+\frac{0.8}{100}}$
  - $S_{100} \approx 4.81$
- Gustafson:
  - $S_p = 1 + (p-1)\gamma$
  - $S_{100} = 1 + (100-1) \times .8$
  - $S_{100} = 80.2$

IV.3.b) Exercice 2: dimensionnement

- programme parallèle passe 6% de son temps d'exécution dans les opérations d'entrée/sortie exécutées sur un seul processeur, quel est le nombre minimal de processeurs pour que le programme parallèle produise une accélération de 20 par rapport au programme séquentiel ?
- Gustafson:
  - $R = (1-\gamma) + p\gamma$
  - Ici on a $\gamma = 0.94$ et $R = 20$, donc on veut résoudre en p:
  - $R = (1-\gamma) + p\gamma$
  - $\Leftrightarrow p = \frac{R-(1-\gamma)}{\gamma}$
  - $\Leftrightarrow p \approx 21.21$
  - On doit donc utiliser **au moins 22 processeurs**
  - Rq: avec Amdahl pour 22 procs : $R = \frac{1}{(1-\gamma)+\frac{\gamma}{p}} \approx 9.65 < 20$
- Amdahl
  - $R = \frac{1}{(1-\gamma)+\frac{\gamma}{p}} = \frac{1}{0.06+\frac{0.94}{p}}$
  - $p \to \infty$, $S \to \frac{1}{0.06} \approx 16,666.$
  - Donc qque soit nb machines, le strong scaling n'atteindra jamais 20
- Et si on demande à Amdahl pour un speedup 4?

IV.3.c) Exercice 3: calculer γ

- Un programme présente une accélération de 9 sur 10 processeurs. Quelle est la fraction maximale du temps d'exécution séquentiel qui peut correspondre à des opérations purement séquentielles ?
- Si on parle de strong scalling, **loi de Gustafson**
  - Ici on a $S = 9$ et $p = 10$, on veut résoudre en γ :

    - $S = (1-\gamma) + p\gamma = 1 + \gamma \times (p-1)$
    - $\Leftrightarrow S - 1 = \gamma \times (p-1)$
    - $\Leftrightarrow \gamma = \frac{S-1}{p-1} = \frac{8}{9} \approx .89$
    - Donc **la partie purement séquentielle est environ 11%**
- Si on parle de weak scalling, **loi d'Amdahl**
  - Ici on a $R = 9$ et $p = 10$, on veut résoudre en γ :
  - $R = \frac{1}{(1-\gamma)+\frac{\gamma}{p}} \Leftrightarrow \frac{1}{R} = (1-\gamma) + \frac{\gamma}{p} = 1 + \frac{\gamma - p\gamma}{p}$
  - $\Leftrightarrow \frac{1}{R} - 1 = \frac{1-p}{p} \times \gamma$
  - $\Leftrightarrow \frac{1-R}{R} = \frac{1-p}{p} \times \gamma$
  - $\Leftrightarrow \gamma = \frac{p(1-R)}{R(1-p)} = \frac{-80}{-81} \approx 0.9876$
  - Donc **la partie purement séquentielle est environ 1.2** %
- Exercices repris de `https://www.cise.ufl.edu/~mssz/CompOrg/CDA3101-S16-Amd` `pdf`

IV.4) Mesure du temps en pratique

- Il y a le wall clock, system clock et user clock. On s'intéresse au wallclock pour simplifier
  - `gettimeofday()` retourne une structure mesurant les millisecondes. C'est pas assez précis
  - `clock_gettime()` donne des mesures à la nano (on obtient la durée par soustraction). C'est bien.
  - Si on veut aller plus loin, on utilise l'intrinsèque `rdtsc` qui lit la valeur sur l'horloge du processeur. Mais c'est hardcore car il faut savoir convertir la valeur retournée en durée. De même, on ne se pose pas ici le pb des horloges monotoniques et compagnie.
- Les processeurs ont des compteurs de performances accessibles avec des outils comme perf (Google entre autres), likwid (=like I knew what I'm doing – thèse allemagne 2019) ou PAPI (projet U. Tennesse très classique, les autres sont parfois des sur-couches)

```
$ sudo apt install papi-tools
$ papi_avail
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
```

```
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_BRU_IDL 0x80000010  No    No   Cycles branch units are idle
PAPI_FXU_IDL 0x80000011  No    No   Cycles integer units are idle
PAPI_FPU_IDL 0x80000012  No    No   Cycles floating point units are idle
PAPI_LSU_IDL 0x80000013  No    No   Cycles load/store units are idle
PAPI_TLB_DM  0x80000014  Yes   Yes  Data translation lookaside buffer mi
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside bu
PAPI_TLB_TL  0x80000016  No    No   Total translation lookaside buffer m
PAPI_L1_LDM  0x80000017  Yes   No   Level 1 load misses
PAPI_L1_STM  0x80000018  Yes   No   Level 1 store misses
PAPI_L2_LDM  0x80000019  Yes   No   Level 2 load misses
PAPI_L2_STM  0x8000001a  Yes   No   Level 2 store misses
```

- En conclusion, retenez que l'analyse des perfs de programmes parallèles est un sujet bien étudié, voir extrêmement fouillé.
  - Bcp d'outils très avancés existent, ils viennent de la recherche récente

IV.5) Le TP chaleur

- Il s'agit d'écrire un algo numérique itératif ultra classique : un stencil. Il s'agit de calculer les évolutions d'une donnée matricielle, sachant qu'à chaque étape, on a besoin des cases voisines pour caculer la case courante.
  - Ici, c'est un stencil à 5 points car on utilise 4 voisins, mais selon la méthode numérique, il peut y avoir besoin de plus de voisins. Voire probablement, d'itérations encore antérieurs des voisins (mais ça c'est rare)
- Ce sera l'occasion de mesurer le strong et weak scalings d'un programme dont le schéma de parallélisation est un découpage 2D du domaine.

Notes d'animation

Séance 1: Architecture et performance des ordinateurs

- Il faut se dépêcher pour avoir le temps de parler des performances en 1h30
- On pourrait vouloir donner le descriptif de mon ordi, Summit et Fugaku sur un doc distribué

- **Programme du jour:**
  - Présentation du module
  - Chap 1: Performance des ordinateurs et parallélisme

Séance 4: CM. Performance des programmes et MPI

- **Résumé de l'épisode précédent**
  - L'architecture des ordinateurs modernes est extrêmement complexe, et plutôt hiérarchique
  - Deux modèles principaux pour la perf: mémoire partagée (multithread), ou passage de messages
  - Dans les deux cas, le plus simple pour programmer ça est d'appliquer le même programme à différentes données en même temps (SPMD), cela évite la majeure partie des problèmes de synchro.
- **Retour TP**
  - Sur un Makefile,
    * les cibles doivent avoir comme nom le fichier produit
    * Il faut que les règles qui ne produisent pas un fichier (clean, run, dist, all) soient marquées PHONY pour que make le fasse meme si un fichier existe
    * Si on veut réduire les redites, on ne donne pas les rgèles de compil dans une variable, mais on fait une règle implicite %: %.c gcc $(CFLAGS) $^ -o $@ $(LDFLAGS)
      · Ensuite, donner les dépendances sans donner de règle fonctionne
    * Avoir une cible clean (+dist) est une bonne idée
  - On fait exit(1) directement pour les erreurs inrattrapables, car en C c'est pénible de ne pas oublier d'erreurs
    * c'est pas exit(0) en cas d'erreur. On est sensé documenter le code d'erreur de chaque type d'erreur.
  - Il est plus simple d'avoir des macros pour convertir (x,y) en position dans le tableau.
  - Erreur au plus tot pour la separation of concerns (argc != 2 => erreur, avant d'utiliser argc)
- **Programme du jour**
  - Chap 2: Programmer pour les performances
  - Chap 3: Mémoire partagée et programmation MPI
    * sections III.1 et III.2, jusqu'à l'installation de simgrid

Séance 5: CM. Communication point-à-point

- **Programme du jour**
  - Chap 3: Mémoire partagée et programmation MPI
    - III.2.b (programmes MPI typiques)
    - III.3 Communication P2P

Séance 6: CM

- **Résumé de l'épisode précédent**
  - Motivation du MPI: permettre aux scientifiques gros consommateurs de calcul de survivre à la complexité des ordinateurs
  - MPI pour la comm inter-noeuds, pas adapté au multithread. Très dur à utiliser car il faut gérer la mémoire à la main: pas d'accès à des buffers impliqués dans des opérations asynchrones
  - Il y a bien mieux dans les cartons (graphe de tâches, PGAS –partitioned global address space), mais reste dominant car les codes sont écrits, et ça marche jusqu'à 4096 procs
  - Il convient de séparer ce qu'on peut faire en MPI et ce qui est fait habituellement. Peu de formes de programmes, très peu d'interleavings
  - Communications point à point. Bloquant ou asynchrone. Il y a d'autres modes, mais pas au programme du module.
  - MPI est sur-optimisé. eager mode vs rdv pour éviter les copies mémoires. Implémentation sans cadre de pile.
- **Programme du jour**
  - Chap 4: performances de programmes parallèles
  - TP chaleur: communications point à point et performance des programmes //