

fridolin katubayemwo

April 2023

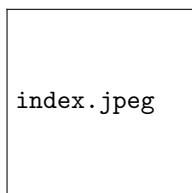
## 1 calcul Intensif

[12pt,a4paper]report [utf8]inputenc [T1]fontenc amsmath [french]babel amssymb  
makeidx graphicx caption here listings xcolor geometry epsfig tabularx hmar-  
gin=2cm,vmargin=2.5cm **cours de calculs intensifs**

**draft du syllabus** Par Dr David NIYUKURI

MasterI Mathématique et Physique Fondamentale et Appliquée

Année académique 2022-2023 April 17, 2023



UNIVERSITE DU BURUNDI  
FACULTE DES SCIENCES



# Contents

<b>1</b>	<b>calcul Intensif</b>	<b>1</b>
<b>2</b>	<b>objectifs du cours</b>	<b>3</b>
<b>3</b>	<b>Les enjeux des sciences du numérique et du calcul haute performance</b>	<b>5</b>
<b>4</b>	<b>L'infrastructure de calcul</b>	<b>9</b>
4.1	Les technologies des composants . . . . .	11
4.1.1	Le transistor MOS . . . . .	12
4.1.2	Les portes logiques CMOS . . . . .	13
4.1.3	La gravure de plus en plus fine avec les nouveaux transistors <b>FinFET</b> et <b>FD-SOI</b> . . . . .	13
4.1.4	L'intégration verticale, la nouvelle dimension pour augmenter la densité. . . . .	14
4.1.5	La technologie et la course à l'efficacité énergétique . . . . .	15
4.2	Les composants au cœur du HPC . . . . .	16
4.2.1	Les éléments de base . . . . .	16
4.3	L'intergiciel ou middleware . . . . .	21
4.3.1		22
<b>5</b>	<b>Le logiciel</b>	<b>25</b>
5.1	Le processus de développement des logiciels . . . . .	25
5.2	Les modèles de programmation . . . . .	26
5.2.1	Remarque . . . . .	28
5.3	Le débogage de code . . . . .	29
5.4	Le cycle de vie des données . . . . .	30
<b>6</b>	<b>Introduction</b>	<b>31</b>
<b>7</b>	<b>Les commandes de bases</b>	<b>32</b>
7.1	Opérations sur les fichiers et répertoire . . . . .	32
7.2	Gestion des droits . . . . .	37
7.3	Options de recherche . . . . .	40
7.4	Archivage et compression . . . . .	41
<b>8</b>	<b>Introduction</b>	<b>43</b>
<b>9</b>	<b>Installation de l'environnement Eclipse</b>	<b>44</b>
9.1	Introduction . . . . .	44
9.2	Installation . . . . .	45
<b>10</b>	<b>Structure de Projet: programmation modulaire</b>	<b>45</b>

<b>11 Compilation et Structure du fichier MakeFile (relation entre structure du projet et makefile)</b>	<b>46</b>
11.1 Téléchargement de l'outil Makefile . . . . .	47
11.2 Exemple d'application . . . . .	47
11.3 Rédaction d'un Makefile . . . . .	48
11.4 Quelques règles et variables spécifiques . . . . .	49
<b>12 But et domaines cibles</b>	<b>49</b>
<b>13 Type de parallélisme</b>	<b>50</b>
<b>14 Programmation avec openMP</b>	<b>53</b>
14.1 Algorithme séquentiel vs. Algorithme parallèle . . . . .	54
14.1.1 Parallélisme léger . . . . .	54
14.1.2 Parallélisme lourd . . . . .	55
14.1.3 Parallélisme léger + parallélisme lourd . . . . .	55
14.1.4 Quand/Comment/Pourquoi paralléliser un algorithme ? . . . . .	55
14.2 OpenMP . . . . .	56
14.2.1 Principe de fonctionnement . . . . .	56
14.2.2 Installation . . . . .	57
14.2.3 Quelques fonctions OpenMP bien pratiques . . . . .	58
14.2.4 Quelques schémas de construction bien pratiques . . . . .	59
<b>15 MPI</b>	<b>61</b>
15.1 Open MPI . . . . .	61
15.1.1 Installation . . . . .	62
15.2 OpenMP ou Open MPI ? . . . . .	68

#### Présentation du cours

Dans pratiquement tous les domaines, chercheurs, ingénieurs et entrepreneurs constatent que le calcul intensif numérique prend une place essentielle s'ils ambitionnent de demeurer dans la compétition mondiale de la recherche et de l'industrie.

La recherche scientifique s'est développée grâce à l'association de travaux théoriques et expérimentaux. L'émergence des très grands calculateurs a ouvert une nouvelle approche, celle de la simulation numérique. Tout en traitant, sous un angle nouveau, une grande partie des questions de recherche et d'ingénierie classiques, la simulation permet d'approcher des phénomènes complexes. Entre les machines mécaniques de Pascal du XVIIe siècle (la pascaline), celle de Babbage au XIXe siècle et les ordinateurs actuels fondés sur l'architecture de Von Neumann (1945) et exploitant les possibilités des semi-conducteurs (1956), on a pu quantifier le coût d'une opération arithmétique typique de base. Ce coût a décri d'un facteur 1015 en passant de la première machine aux ordinateurs actuels qui ont des puissances intrinsèques exprimées en téraflo, soit 1012 opérations par seconde.

Au milieu des années 1970, la société Cray développa une architecture permettant d'envisager un coût d'investissement pour un million de dollars pour un million d'opérations par seconde. Trente ans plus tard, avec les progrès technologiques des circuits électroniques et des sciences de la simulation, le nombre d'opérations a été multiplié par un million pour un coût équivalent. La plus puissante machine développe une puissance de calcul de 280 téraflops.

## 2 objectifs du cours

Ce cours de calcul intensif aura comme objectif de doter aux étudiants de master 1 Mathématique et Physique une brève introduction aux notions de calcul intensif et ses applications qui pourront les aider dans l'accomplissement de leurs projets de recherche. c'est ainsi que les étudiants apprendront essentiellement:

- le travail en lignes de commandes,
- l'installation d'un environnement de travail complet à partir d'outils libres et performants,
- l'utilisation plus avancée du compilateur gfortran (compilateur via un Makefile, optimisation, debugger, profiling),
- les techniques de programmation en parallèles: notions générales de parallélisme, parallélisation avec OpenMP 3, parallélisation avec MPI,
- Méthodes par différences finies et applications (dérivées numériques, électrostatique, équation de Schrödinger, équations de Maxwell, stabilité numérique),
- Méthodes d'optimisation par algorithmes génétiques,
- Notions de Unix et initiation à l'utilisation d'un cluster informatique.

Introduction générale aux simulations et aux calculs de haute performance La simulation numérique, outil essentiel en sciences fondamentales et appliquées, ambitionne de prédire le comportement d'objets complexes ou inaccessibles parce que trop petits, trop grands, ou trop lointains dans l'espace ou le temps. Comment appréhender le déploiement du prion, les instabilités de certaines supernova, la formation de l'univers primordial, l'évolution du climat sans recours à la simulation ? La simulation numérique est ainsi devenue omniprésente dans la recherche en astrophysique, climatologie, biologie, pour ne citer que quelques exemples.

La simulation est également indispensable pour concevoir et optimiser des objets complexes, ou des produits interagissant avec des êtres vivants. Les expériences en vraie grandeur ou en conditions réelles sont bien souvent très coûteuses, voire impossibles. Nous pouvons penser, bien sûr, aux armes nucléaires, mais aussi aux cosmétiques dont l'absence de toxicité doit désormais être démontrée sans recours à l'expérimentation sur des animaux. La première phase de la

démarche de simulation est de construire un modèle physique le plus complet possible. Il faut ensuite le traduire en équations mathématiques, souvent des équations aux dérivées partielles. Enfin, ces équations sont résolues numériquement sur ordinateur à l'aide d'algorithmes performants.

La démarche de conception et d'analyse est ainsi menée en s'appuyant sur des simulations associées à des expériences élémentaires ou partielles. La simulation permet aussi de réaliser à peu de frais des études paramétriques et d'optimiser des plans d'expériences. Elle permet ainsi d'étudier l'influence de différents paramètres tant physiques que technologiques et d'en déterminer les valeurs optimales. Elle est ainsi devenue l'un des moteurs essentiels de la réactivité et de la compétitivité de notre industrie.

La simulation numérique s'appuie comme évoqué plus haut sur le triptyque *modélisation physique – sciences du numérique – calcul haute performance*. Chacune de ces disciplines évolue et progresse selon sa propre échelle de temps. La plus récente, qui est la plus visible pour le grand public, est le calcul haute performance (ou calculs intensifs). L'accroissement spectaculaire de la puissance de calcul permise par la célèbre loi de Moore (doublement tous les dix-huit mois de la densité de transistors dans les processeurs...) a en effet radicalement modifié notre manière de vivre et de travailler. Cette avancée décisive des moyens de calcul a permis et permettra à la simulation d'aborder de nouveaux domaines de la physique et d'utiliser des modèles toujours plus fiables et plus prédictifs de la réalité. La modélisation physique, qui traduit en relations mathématiques les phénomènes physiques, est une discipline plus ancienne, dont certains des succès les plus spectaculaires, tels que les équations de Maxwell ou la théorie de la diffusion de la chaleur, remontent au XIXe siècle. Mais elle continue sa progression en sciences des matériaux, sciences du vivant et physique des hautes énergies, pour ne citer que quelques domaines particulièrement actifs.

Enfin, les sciences du numérique, qui imaginent des algorithmes permettant de simuler de manière fiable et précise les relations mathématiques représentant des phénomènes physiques, sont la branche la moins médiatisée de notre triptyque. Elles n'en constituent pas moins le lien indispensable entre la modélisation physique et les moyens de calcul. Même si quelques travaux précurseurs ont précédé l'apparition des ordinateurs, les sciences du numérique se sont principalement développées depuis les années 40. Domaine multidisciplinaire par excellence, elles progressent grâce aux apports conjoints des mathématiciens, ingénieurs, physiciens et informaticiens.

La confiance dans la simulation repose donc sur la validité des modèles physiques, la qualité des schémas numériques, et leur résolution à un niveau de précision suffisant. Elle est formalisée par des procédures de quantification des incertitudes qui permettent d'apporter une garantie, par exemple sur le fonctionnement et les performances d'un objet. Un élément essentiel dans ce processus de garantie est de disposer d'un ensemble suffisamment large de données expérimentales contraignantes. La reproduction fidèle des résultats expérimentaux permet de valider globalement la chaîne de simulation. C'est un complément nécessaire au travail théorique de conception des modèles physiques et numériques.

### 3 Les enjeux des sciences du numérique et du calcul haute performance

Depuis l'apparition des premiers ordinateurs à tubes, au milieu de la Seconde Guerre mondiale, les sciences du numérique et la simulation se sont développées de manière continue et extrêmement rapide. Cela à tel point que, depuis une quinzaine d'années, de nombreux scientifiques considèrent que la **simulation numérique** est devenue le troisième pilier de la science, aux côtés de l'**expérimentation** et de la **théorie**. Au-delà du domaine de la recherche scientifique, la simulation numérique est également devenue un outil indispensable pour l'industrie, l'innovation et la compétitivité des entreprises.

La simulation joue un rôle central dans de très nombreuses disciplines scientifiques. Elle permet d'étudier *in silico* des systèmes complexes hors de portée d'une approche purement analytique, parfois sans avoir recours à des expériences coûteuses et difficiles (voire impossibles) à mettre en place. La simulation permet ainsi d'examiner de très nombreux modèles, d'avoir accès aux paramètres les décrivant et, de ce fait, de prédire numériquement des comportements jamais observés. Cette démarche scientifique profondément renouvelée par la simulation est devenue usuelle dans de nombreux domaines scientifiques et techniques. Elle permet des gains de productivité importants dans l'industrie (conception des avions, crash test numériques pour les voitures, mise au point de produits cosmétiques, ...).

Un champ d'investigation bouleversé par la simulation est celui des systèmes pour lesquels une approche expérimentale globale est impossible. C'est par exemple le cas pour les sciences du climat, l'astrophysique ou même la simulation de certains accidents industriels. La simulation numérique est devenue un outil indispensable, voire constitutif de ces domaines. Par exemple, dès 1941, l'astronome Eric HOLMBERG construisit un calculateur analogique à base d'ampoules et de cellules photo-électriques pour simuler la force de gravité inversement proportionnelle au carré de la distance, et fut le premier à étudier théoriquement la collision de galaxies. Ces simulations pionnières ne comportaient que quelques dizaines d'objets mais poursuivaient le même dessein que les simulations les plus récentes qui en comptent plus de 500 milliards : modéliser des systèmes non accessibles par la seule observation. Les chapitres sur la dynamique des galaxies, le climat ou les séismes illustrent la place essentielle de la simulation numérique dans certains de ces domaines.

Un autre domaine où l'utilisation du calcul est un outil incontournable est celui du **traitement des données**. Qu'elle soit recueillie sur l'internet, issue des capteurs maintenant installés dans de nombreux objets dit connectés ou produite par de grands projets scientifiques comme le CERN (Organisation européenne pour la recherche nucléaire, <http://home.cern.fr/about>) ou le GENOSCOPE (Centre national de séquençage, <http://www.genoscope.cns.fr/>), la masse de données est en perpétuelle augmentation. Le CERN produit aujourd'hui environ 30 pétaoctets de données par an ; lorsqu'il sera mis en service dans quelques années le télescope SKA (<https://www.skatelescope.org>) en

Australie occidentale produira 160 téraoctets de données par seconde ; elles devront être analysées par un supercalculateur dédié. La valorisation scientifique (ou commerciale) de ces masses de données est un enjeu majeur des années à venir. L'acquisition par Total d'un super calculateur, PANGÉA (<http://www.top500.org/system/17807>) ; l'objectif de Pangéa est de gagner en temps et en précision dans la modélisation des sous-sols), parmi les plus puissants au monde en est un des exemples récents. Si, comme nous venons de le voir, la simulation s'applique à des domaines extrêmement variés, il existe néanmoins une sorte de trame commune à toutes les activités de simulation. Il faut tout d'abord une étape de modélisation qui vise à construire une représentation, nécessairement approchée, du système ou du phénomène physique à étudier. Ensuite, cette représentation doit être transcrite en équations par un modèle mathématique. Il faut ensuite traduire ce dernier en un algorithme interprétable par un ordinateur. Cela consiste souvent à passer d'une description continue de l'espace et du temps à une représentation discrète de l'espace et du temps. Il existe de grandes classes de méthodes numériques pour effectuer cette discrétisation et le développement de nouvelles méthodes constitue un domaine de recherche très actif en mathématiques appliquées. Ces méthodes numériques sont ensuite mises en œuvre dans des codes de calcul qui sont ainsi capables de résoudre le modèle mathématique initial. Certains aspects du développement de ces codes numériques (**software**, en anglais) sont présentés dans le point sur Le logiciel : les entrées/sorties, le génie logiciel. Une fois le code de calcul opérationnel, il convient de passer à une étape de **vérification** qui consiste à s'assurer que le code résout effectivement les équations initiales puis à une étape de **validation** des résultats, au sens de la physique. Ces deux étapes sont essentielles afin de s'assurer que malgré les approximations inévitables faites dans les phases de modélisation et de discrétisation, le code permet de reproduire fidèlement le comportement du système physique étudié. Enfin, il est appliqué un processus de **qualification** visant à maîtriser les incertitudes et à déterminer le domaine de validité du code. Les méthodes mises en œuvre pour la validation et la qualification des codes de calcul sont exposées dans le point sur les principes de validation d'un logiciel.

Les profonds bouleversements apportés par les sciences du numérique reposent en grande partie sur les progrès du **hardware** et l'accroissement continu et exponentiel de la puissance de calcul des ordinateurs depuis plusieurs décennies. La figure (2.1) issue du classement des 500 ordinateurs les plus puissants au monde (Voir [www.top500.org](http://www.top500.org)) montre l'évolution de la puissance de calcul (Cette puissance de calcul est estimée sur la base d'un calcul d'algèbre linéaire (test Linpack, voir <http://www.netlib.org/benchmark/hpl/index.html> pour plus de détails).) au cours des vingt-cinq dernières années. La puissance, mesurée en nombre d'opérations flottantes par seconde (Flop/s) est représentée en échelle logarithmique sur l'axe vertical : elle augmente de manière exponentielle et double environ tous les treize mois ! Comme illustré par la courbe rouge représentant la puissance de calcul des ordinateurs personnels, il faut seulement une quinzaine d'années pour que la puissance des supercalculateurs soit offerte au plus grand nombre par le biais des objets du quotidien, y compris



les téléphones et les tablettes.

Derrière cette progression régulière se cachent plusieurs révolutions tech-

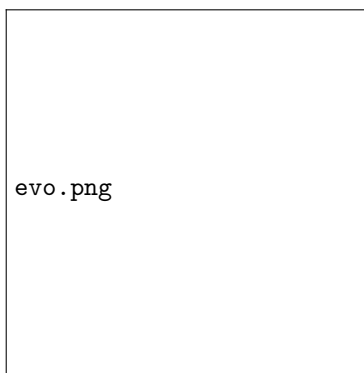


Figure 1: Évolution de la puissance de calcul au cours de ces vingt dernières années. (source [www.top500.org](http://www.top500.org), voir texte pour plus de précision)

nologiques. La première a eu lieu dans les années 70 avec l'apparition des processeurs **vectoriels** capables d'effectuer une même opération sur un grand nombre de variables en même temps. Les ordinateurs comportaient alors un seul **processeur** ayant un seul **cœur** de calcul. C'est l'augmentation de la puissance de calcul de ce processeur unique qui était responsable de l'augmentation des performances. Au début des années 90, la performance des processeurs a commencé à saturer, notamment en raison de la difficulté technologique à augmenter leur fréquence. Pour maintenir la croissance de la puissance de calcul des ordinateurs, ils ont donc été équipés de plusieurs processeurs. En 1993, 20 % des ordinateurs du top500 dont 10 des 50 premiers n'avaient qu'un seul processeur. Une vingtaine d'années plus tard, les ordinateurs les plus puissants ont des millions de cœurs de calcul et même les derniers ordinateurs du top 500 en ont près de dix mille. L'avènement de ce qui est appelé désormais le **parallélisme** a profondément bouleversé l'architecture des codes de calcul. Aujourd'hui, l'agrégation d'un très grand nombre de processeurs standards pour construire un supercalculateur est en train d'atteindre ses limites, notamment en raison de la très forte consommation électrique nécessaire (une dizaine de méga-Watts pour les machines les plus puissantes). Pour atteindre l'exaflop ( $10^{18}$  opérations par seconde) avec les technologies actuelles, il faudrait plusieurs centaines de méga-Watts pour assurer l'alimentation de la machine, ce qui n'est économiquement pas viable (Alimenter une machine d'un méga-Watt coûte environ un million d'euros par an.). De nouvelles architectures de calcul (technologie many-core ou accélérateurs, par exemple) sont en cours de développement afin de surmonter cette barrière énergétique. Ces architectures très novatrices équipent d'ores et déjà de nombreuses machines, dont les deux premières du top 500. Au vu des évolutions actuelles et quelles que soient les

voies technologiques qui se révéleront les plus pertinentes, il est très probable que les ordinateurs à venir auront plusieurs milliers de nœuds interconnectés comportant chacun plusieurs dizaines de milliers d'unités de calcul. Faire travailler ensemble ces dizaines de millions d'éléments représente un des grands défis des années à venir.

Au-delà des unités de calcul, les ordinateurs ont besoin de nombreux autres composants : mémoires, stockage, réseaux de communication... Les performances de ces composants ont également progressé de manière spectaculaire, toutefois à un rythme un peu inférieur à celui de celles des processeurs. Cela constitue donc aujourd'hui parfois un véritable goulot d'étranglement. À titre d'exemple, accéder à un nombre stocké dans la mémoire centrale peut prendre entre 10 et 100 fois plus de temps que de réaliser une multiplication ou une addition : il peut être bien plus efficace de refaire les calculs de ce nombre plutôt que de le stocker puis d'aller le lire. Pour surmonter ce mur de la mémoire (*memory wall*) et alimenter de manière efficace les unités de calcul en données, une gestion complexe de la mémoire a été mise en place au prix d'une complexité d'utilisation encore accrue.

Ces défis technologiques et les perspectives scientifiques ouvertes par l'accroissement des moyens de calcul ne concernent pas seulement la communauté du calcul haute performance (HPC). L'économie en énergie, au cœur de la conception des ordinateurs, est également une nécessité pour tous les systèmes embarqués, entre autres les téléphones et tablettes.

Les évolutions technologiques profondes des calculateurs ont des conséquences importantes sur la programmation, l'architecture des codes de calcul et les méthodes mathématiques qui sont utilisées. La programmation sert deux objectifs : la spécification des calculs à faire et l'affectation efficace des ressources matérielles. Avec la complexité des nouvelles architectures qui auront de nombreux nœuds de calcul avec un très fort degré de parallélisme interne (plusieurs dizaines milliers de cœurs) et une très forte hiérarchie mémoire, l'utilisation optimale des ressources de calcul devient un métier de spécialistes. Afin de rendre ces architectures plus accessibles et de permettre la portabilité d'un logiciel entre des machines ayant des technologies différentes, de nombreux logiciels médiateurs, ou **middleware** et **bibliothèques** sont développés, conçus pour permettre une programmation des calculs faisant autant que faire se peut abstraction de l'architecture sous-jacente. Ils reposent sur un compromis entre portabilité et facilité d'utilisation d'un côté et performances de l'autre. Le Graal de la parallélisation automatique ou d'une abstraction complète de l'architecture reste un horizon difficile à atteindre.

L'utilisation efficace du calcul haute performance implique des équipes pluridisciplinaires réunissant des informaticiens, des mathématiciens appliqués et des spécialistes des domaines applicatifs concernés. Un dialogue approfondi et une compréhension mutuelle entre ces disciplines sont indispensables pour que les avancées majeures attendues se réalisent. Cet élan très important des sciences du numérique et du calcul haute performance est mondial et concerne aussi bien la recherche académique que l'industrie. Citons les immenses perspectives

dans le domaine des sciences de la vie (génomique, conception de médicaments, médecine personnalisée...), des sciences humaines (analyses de réseaux sociaux, étude du champ lexical de très grands corpus littéraires...) ou du traitement des images (reconnaissance digitale ou faciale, imagerie médicale...). L'association d'une gigantesque puissance de traitement de données et de nouvelles méthodes d'analyse a contribué à un renouveau profond de l'intelligence artificielle, y compris dans sa définition. Les illustrations les plus médiatisées sont actuellement dans le domaine des jeux (jeu de Go), mais le potentiel d'applications dans le domaine de la santé, de l'aide à la décision ou même du sport (analyse du mouvement du sportif) est immense

## 4 L'infrastructure de calcul

Pour que les ordinateurs puissent devenir ces outils si répandus et si puissants, de nombreux domaines ont dû faire des avancées significatives. L'analyse numérique s'est développée parallèlement au développement des possibilités des différentes générations d'ordinateurs, ouvrant la voie à la capacité prédictive des simulations. L'essor des langages de programmation, des systèmes d'exploitation et des outils associés a donné naissance à une nouvelle science, spécifique au monde des ordinateurs (*Computer Science* en anglais), qui a acquis ses lettres de noblesse depuis quelques décennies. De nombreux développements matériels ont vu le jour permettant de délivrer toujours plus de capacité de calcul avec des machines de taille raisonnable et consommant toujours moins (pour une performance donnée). Le monde de l'informatique haute performance est donc en perpétuelle évolution, associé à un recours grandissant à la simulation numérique. Si nous regardons les acquis de pratiquement soixante-dix ans d'utilisation des ordinateurs les plus puissants (on parle dans ce cas de calcul intensif, ou HPC pour High Performance Computing), on se rend compte qu'il faut désormais parler d'infrastructure de calcul et non plus simplement d'ordinateur.

Cette notion décrit mieux les capacités d'une organisation (industrielle ou de recherche) à conduire des simulations numériques avancées car elle englobe non seulement la machine (l'ordinateur) mais aussi tous les éléments indispensables pour mener à bien l'expérience numérique.

L'infrastructure de calcul repose sur un centre de calcul (bâtiments, servitudes diverses, etc.), des réseaux (internes et externes ouvrant l'accès au monde extérieur), des moyens de stockage des résultats produits, naturellement des ordinateurs de puissance mais aussi sur les équipes en charge d'assurer le bon fonctionnement de l'ensemble et de développer les composants logiciels indispensables tant à la machine qu'aux utilisateurs. Avec l'émergence du big data, le point de vue se déplace d'une vision centrée sur l'ordinateur<sup>1</sup> vers celle des flots de données et leurs traitements. Ce point vise à définir précisément cette infrastructure de calcul et de mettre l'accent sur sa complexité toujours

---

<sup>1</sup>Les ordinateurs sont les descendants des premiers calculateurs. Dans ce document, nous utiliserons donc indifféremment *calculateur* ou *ordinateur* pour désigner une machine électronique capable d'être programmée pour réaliser des algorithmes.

croissante, inhérente au calcul haute performance et de montrer les défis à venir (limiter la consommation électrique, prendre en compte les limitations imposées par la physique, la manipulation des données,...) et la nécessaire R&D pour apporter les meilleures solutions informatiques aux utilisateurs. Ces éléments peuvent être vus comme trois niveaux de *poupées russes* (fig. 2.2), chacun ayant un impact direct sur le développement des applications.

Au centre du schéma se placent les éléments matériels et logiciels indispensables pour assurer la fonction calcul et la prise en compte des flots de données. Nous y décrirons les éléments technologiques nécessaires à la constitution d'un centre de calcul en partant des composants les plus élémentaires, y seront abordés les aspects matériels présents et à venir (CPU, GPU, nœuds de calcul, clusters,...), mais aussi logiciels de bas niveau (système d'exploitation, langages de programmation,...).

Puis se trouve la couche représentant les logiciels indispensables aux simulations et les méthodes de développement mises en œuvre pour la réussite des grands projets logiciels. Il s'agit ici de mettre en perspective les divers outils utiles au monde du calcul haute performance. Cela couvrira la problématique du développement de codes, la gestion des flux de données et du stockage des résultats, ainsi que l'exploitation de ces derniers.

Enfin, dans la couche extérieure se trouvent les méthodes numériques et algorithmiques. Cette section expliquera les processus utilisés pour passer des équations de la physique au monde des programmes sur ordinateurs. Nous allons donc maintenant explorer les éléments réalisant la fonction calcul.

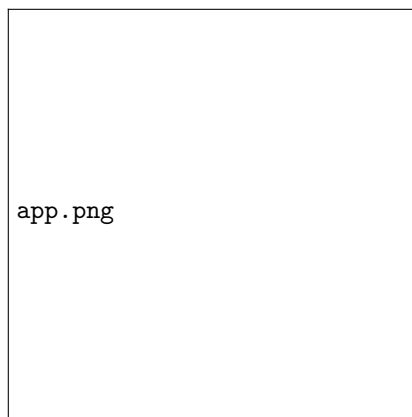


Figure 2: Les applications s'appuient sur une hiérarchie de domaines imbriqués les uns dans les autres comme des poupées russes

## 4.1 Les technologies des composants

Ce point, à forte coloration technologique, détaille les éléments principaux nécessaires pour réaliser un super ordinateur et son environnement. Nous allons donc suivre le chemin inverse des loupes successives de la figure (2.3) en partant de l'élément le plus petit, le transistor, qui servira à fabriquer un **processeur** et sa **mémoire**. Nous verrons ensuite ce que sont les **nœuds** de calcul, qui reliés ensemble, de-

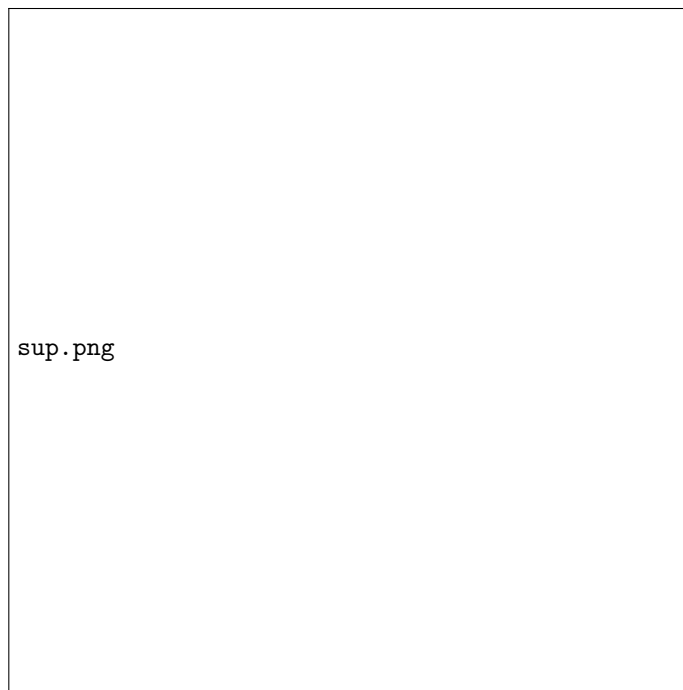


Figure 3: Cette suite de loupes explique l'empilement des technologies constituant un super ordinateur. Les échelles varient de quelques dizaines de nm pour un transistor (14nm en 2016) à plusieurs dizaines de mètres pour un super ordinateur.

viendront un **cluster** qui sera installé dans un centre de calcul. Nous concluons cette section par les logiciels nécessaires pour assurer le bon fonctionnement et le bon usage des ordinateurs. Nous donnons maintenant un éclairage sur les technologies nécessaires pour construire les divers composants présents au cœur des ordinateurs.

Si les premiers ordinateurs fonctionnaient avec des lampes, aujourd'hui l'électronique (pour le grand public tout comme pour les ordinateurs de puissance) repose uniquement sur l'utilisation de **transistors**, dont l'industrialisation a débuté en 1950. La maîtrise de la technologie des transistors est donc un enjeu majeur pour l'industrie de l'électronique comme pour le consommateur. Les fonctions

de calcul, réalisées par un processeur, ou de mémorisation de l'information sont toutes mises en œuvre par un assemblage astucieux de transistors. Une rapide explication du fonctionnement d'un transistor et de son utilisation pour réaliser des portes logiques est donnée dans les lignes qui suivent.

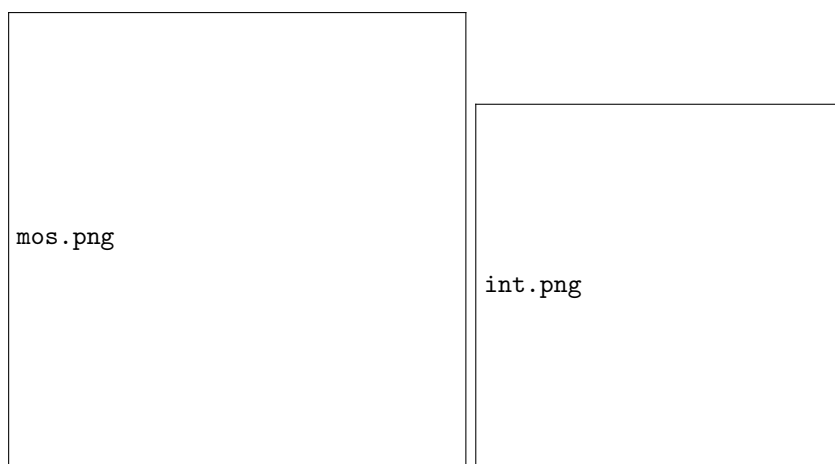


Figure 4: à gauche nous avons une vue en coupe d'un transistor nMOS et à droite un modèle interrupteur des transistors MOS.

#### 4.1.1 Le transistor MOS

La technologie **CMOS** (pour Complementary Metal Oxide Semiconductor ) est devenue la technologie dominante pour les circuits intégrés. Elle utilise le transistor MOS, un dispositif semi-conducteur à trois électrodes appelées source, drain et grille, qui permet de contrôler un courant entre la source et le drain en fonction de la tension appliquée sur sa grille.

Le transistor MOS de type N (nMOS représenté figure (2.4 (a))), se compose d'un barreau de semi-conducteur dopé P appelé corps (ou *body* , *well* , *bulk* ) dans lequel sont incrustées des bandes de silicium dopées N constituant les électrodes source et drain. L'espace entre la source et le drain est surmonté d'une bande très fine d'oxyde de silicium qui sépare le corps à l'électrode appelée grille. Le dopage sous la source et le drain, effectué par l'introduction d'impuretés (dopants), va permettre de changer les propriétés électriques du semi-conducteur sous le contrôle de la grille. Un dopage de type N provoque un excès d'électrons alors qu'un dopage de type P provoque un déficit d'électrons. On introduit de cette façon des porteurs de type électrons et trous respectivement. Le dopage du corps permet d'isoler le transistor d'avec ses voisins.

Ce transistor nMOS se comporte comme un interrupteur commandé. Sans aucune tension (ou différence de potentiel) entre la grille et la source, aucun courant ne circule entre source et drain, le transistor est bloqué et se comporte

alors comme un interrupteur ouvert (fig. 5). Lorsqu'une tension est appliquée entre la grille et la source, les électrons en excès dans le silicium dopé N se propagent puis s'accumulent sous la grille jusqu'à former un canal N. Lorsque la tension appliquée est supérieure à la tension dite de seuil, le canal devient conducteur par déplacement d'électrons. Un courant électrique circule alors entre la source et le drain, le transistor est passant et se comporte comme un interrupteur fermé (fig. 2.4 (b)). La technologie CMOS ( Complementary Metal Oxide Semiconductor ) utilise une combinaison complémentaire de transistors de type N avec des transistors de type P dont la composition est duale de celle du nMOS (dopants de type N dans le corps et de type P sous la source et le drain), et qui se comporte d'une façon symétrique : quand la tension diminue entre la grille et la source, les porteurs en excès (trous) dans le silicium dopé P sous la source se propagent puis s'accumulent sous la grille jusqu'à former un canal P qui devient conducteur par déplacement de trous, lorsque la tension devient inférieure à la tension de seuil (fig. 5) ; le transistor pMOS est alors passant.

#### 4.1.2 Les portes logiques CMOS

En disposant de manière symétrique des couples de transistor nMOS et pMOS qui ont un fonctionnement dual, un transistor est passant alors que l'autre est bloquant, on forme des portes logiques qui répondent à des fonctions combinatoires élémentaires (NON, ET, OU...). La mise en série ou en parallèle des transistors nMOS et pMOS permet la réalisation de portes logiques plus complexe où l'état bas en sortie est obtenu par un réseau série ou parallèle de transistor nMOS passants et l'état haut obtenu par le réseau dual de transistors pMOS passants.

L'assemblage de ces portes logiques permet de construire des fonctions plus complexes comme un additionneur 1 bit et le chaînage de ces fonctions réalise les opérations de base du calcul: addition, multiplication, etc.

L'évolution très rapide des performances des calculateurs est décrite par ce que l'on appelle la loi de Moore, en d'autres termes, le doublement tous les deux ans du nombre de transistors qui peuvent être intégrés dans une puce, et par voie de conséquence, la réduction de coût des composants électroniques semi-conducteurs.

#### 4.1.3 La gravure de plus en plus fine avec les nouveaux transistors FinFET et FD-SOI

Avec la réduction des dimensions du transistor en technologie planaire, le canal sous la grille devient de plus en plus profond avec une zone trop éloignée pour être contrôlée correctement. Cela conduit à des phénomènes parasites tels que la variabilité ou encore des courants de fuite importants dans une zone très fine et donc une consommation électrique bien trop élevée à l'échelle de la puce. La solution trouvée par les spécialistes de la technologie silicium consiste à faire un canal plus fin au niveau de la grille. Dans le transistor FinFET, le canal devient

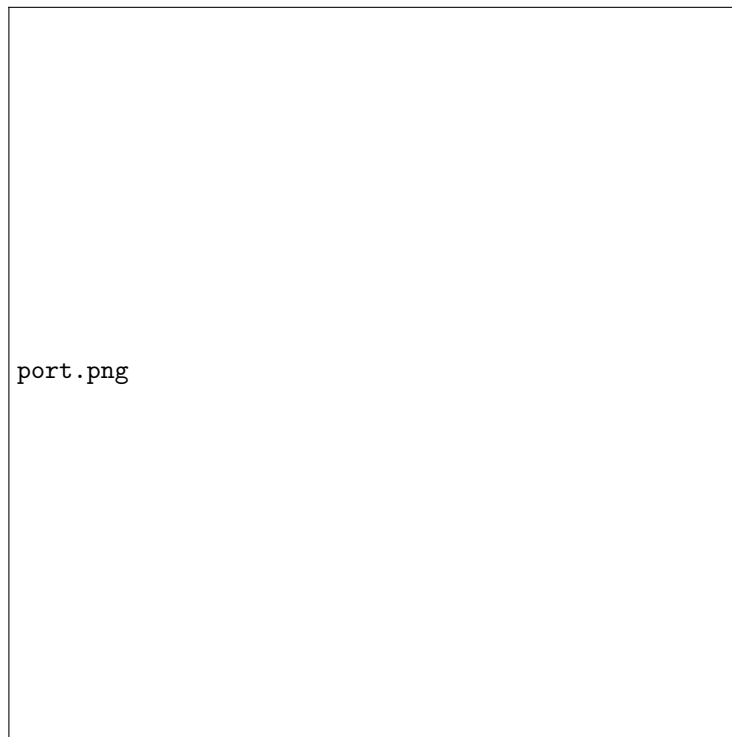


Figure 5: Portes logiques élémentaires : NON-ET, NON-OU, additionneur 1 bit.

vertical et la grille l'entoure comme l'indique la figure (2.6 (a)). Pour le FD-SOI, le canal reste horizontal, mais il est isolé par une couche d'oxyde comme le montre la figure (2.6(b)). Les deux solutions ont leur propre complexité de fabrication, la croissance verticale du canal pour le FinFET, l'uniformité de la couche de silicium au-dessus de l'isolant pour le FD-SOI. Néanmoins, la fabrication de puces FinFET reste plus onéreuse qu'avec le FD-SOI, mais ce dernier ne pourra pas indéfiniment être réduit en taille. La course à la densité ne s'arrêtera pas avec ces innovations. Les technologues étudient déjà d'autres solutions de transistors verticaux à base d'un canal de type nanotube entouré par sa grille.

#### 4.1.4 L'intégration verticale, la nouvelle dimension pour augmenter la densité.

Une autre solution pour gagner en densité consiste à empiler les couches de dispositifs à la verticale. Cela est d'autant plus intéressant pour des composants à motif régulier comme les mémoires. Aujourd'hui, Samsung développe de nouvelles structures à 32 couches pour les mémoires **FLASH** (V-NAND 3D). Tou-



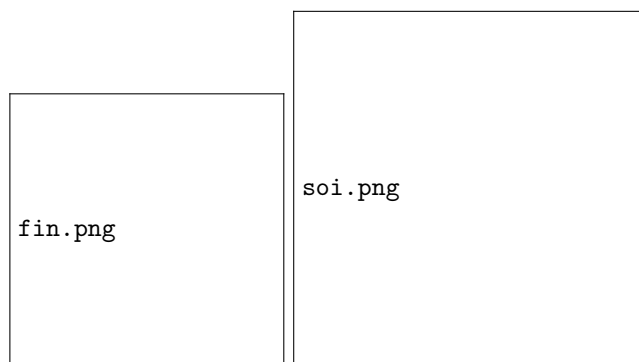


Figure 6: A gauche, transistor FinFET et à droite, transistor FD-SOI.

jours dans ce domaine des mémoires, Intel et Micron développent en commun un nouveau dispositif de point mémoire non volatile et sans transistor appelé 3D XPoint . L'architecture consiste en un damier en trois dimensions, où les points mémoire se situent à l'intersection des lignes de mots et de bits, permettant aux cellules d'être adressées individuellement. Ce motif a été développé pour être facilement extensible en vertical et ainsi augmenter la capacité mémoire en gardant les processus de photolithographie classiques. Ces nouvelles mémoires qui apparaissent sur le marché pour remplacer les disques durs magnétiques par des **SSD** (Solid-State-Drive) vont également à terme remplacer les mémoires dynamiques **DRAM** trop consommatrices en énergie électrique. Cette mutation modifiera toute la hiérarchie mémoire des processeurs, mais n'interviendra qu'avec l'amélioration de l'endurance des mémoires non volatiles et la conception de contrôleurs mémoires intelligents.

#### 4.1.5 La technologie et la course à l'efficacité énergétique

La loi de Moore fut longtemps accompagnée de la loi de Dennard, c'est-à-dire le doublement de la fréquence de fonctionnement des processeurs à chaque génération s'accompagnant aussi d'une réduction de la tension d'alimentation résultant ainsi en une densité d'énergie constante. Les architectes de circuits intégrés ont donc vécu quelques dizaines d'années d' happy scaling où la performance était automatiquement améliorée avec l'augmentation de la fréquence sans impacter la consommation électrique. Dans les années 2000, la fréquence s'est stabilisée et les architectures multi-cœurs se sont développées conformément à la loi de Moore. Mais la tension d'alimentation reste maintenant pratiquement constante au travers des générations technologiques, ce qui accroît par conséquent la densité d'énergie en mode actif. De plus, la finesse des transistors augmente drastiquement les courants de fuites qui deviennent prédominants. Aujourd'hui, nous avons atteint les limites de l'évacuation de l'énergie sur du silicium. C'est pourquoi le principal défi est de réduire la consommation

électrique dans les circuits intégrés, ce qui aura aussi pour effet de stabiliser le coût d'exploitation des centres de calcul au sens large (coût de l'électricité).

## 4.2 Les composants au cœur du HPC

### 4.2.1 Les éléments de base

Les principes de base des ordinateurs ont moins de 70 ans, sans remonter aux machines mécaniques. Beaucoup considèrent que l'architecture type d'un ordinateur numérique moderne a été vulgarisée par la note *First draft of a Report on the EDVAC* écrite en 1945 par le mathématicien John VON NEUMANN (fig. 2.7). Il considère qu'un ordinateur est composé de quatre parties :

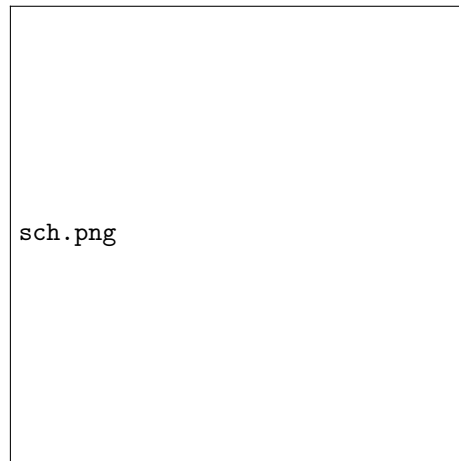


Figure 7: Schématisation de l'architecture de von Neumann. (d'après [https://fr.wikipedia.org/wiki/Architecture de von Neumann](https://fr.wikipedia.org/wiki/Architecture_de_von_Neumann)).

- Une unité de calcul comprenant une unité arithmétique et logique et des registres de mémorisation ;
- une unité de contrôle, qui gère la séquence des instructions et qui comporte un compteur ordinal (Program Counter en anglais ou PC) qui indique la séquence (l'ordre) des instructions à effectuer;
- la mémoire qui stocke à la fois les données (traitées ou à traiter) et la séquence des instructions ou programme. Le fait que les instructions et les données soient dans une même mémoire (ou dans un même espace mémoire) est caractéristique de l'architecture dite "de Von Neumann"
- les **entrées-sorties** qui permettent d'échanger des données avec le monde extérieur.

Cette organisation se retrouve dans les ordinateurs actuels, avec souvent l'unité de calcul et l'unité de contrôle regroupées dans le processeur, la mémoire composée de différents types de mémoires (la hiérarchie mémoire) et les liens de communication. La suite de cette partie va donc être organisée suivant la structure processeur, hiérarchie mémoire et communication.

## 1. Processeur

Le processeur ou **CPU** en anglais (Central Processing Unit) est la partie où les instructions sont exécutées. Son évolution est fortement liée à l'évolution de la technologie. Les premiers ordinateurs étaient réalisés en composants discrets (tubes à vide, ensuite transistors, enfin circuits intégrés). Les microprocesseurs ont finalement prévalu dans les supercalculateurs, mais il a fallu du temps pour que la technologie permette d'intégrer assez de fonctions pour qu'ils puissent avoir des fonctionnalités utilisables dans les grands ordinateurs. Le premier microprocesseur (circuit intégré comportant toutes les parties d'un processeur dans un seul circuit intégré) a été produit par Intel en 1971. Le processeur Intel 4004, originellement conçu pour être le cœur d'une calculatrice, s'est montré bien plus universel et se trouve à l'origine du succès de la société Intel sur ce marché. Il comportait 2 300 transistors et travaillait sur des données de 4 **bits** à une fréquence d'horloge <sup>2</sup> d'environ 740 kHz. Il a été suivi par le 8008 qui, avec 3 500 transistors pouvait travailler sur des mots de 8 bits puis, en 1974, le 8080, le premier système à bus d'adresse<sup>3</sup> et de donnée séparé. L'évolution de la technologie des semi-conducteurs, illustrés par la loi de Moore, a permis une rapide amélioration des performances. Le nombre de transistors pouvant être économiquement intégrés dans une puce croissant, les unités de calculs se sont complexifiées pour traiter des mots de plus en plus larges (16 bits, puis 32 et 64 bits). Cette taille des mots a aussi un effet sur l'espace d'**adressage** : les microprocesseurs 8 bits ont dès le début un espace d'adressage sur 16 bits soit 65 536 positions (64 K<sup>4</sup> mots), donc les opérations sur les adresses mémoire sont soit simplifiées (opérations relatives à un registre d'adresse de base), soit réalisées en plusieurs opérations élémentaires. Les processeurs 16 bits sont plus orthogonaux dans ce sens, mais on s'est vite rendu compte que 64 K mots était une limitation pour accéder à la fois aux données et aux instructions dans les programmes. D'où l'idée de différentes pages de 64 K pouvant être changées en modifiant des données dans un registre. Cela s'est reflété dans les différents modèles mémoire des premiers ordinateurs du type PC fondés sur l'architecture Intel 8086 (en fait, les premiers PCs étaient équipés du circuit Intel 8088, qui avait un bus externe 8 bits pour réduire le nombre de broches d'entrées-sorties, et donc le coût du

---

<sup>2</sup>La fréquence d'horloge est la fréquence du signal qui séquence toutes les opérations dans le microprocesseur. Plus elle est élevée, plus le microprocesseur est rapide et performant.

<sup>3</sup>Les informations stockées dans la mémoire sont accédées par une adresse unique à l'instar de l'adresse d'une maison dans une rue.

<sup>4</sup>En informatique, K représente 1024, donc 64 K = 65 536.

circuit). Le passage en mots de 32 bits a vraiment permis d'utiliser les microprocesseurs pour le calcul scientifique, à la fois grâce au large espace d'adressage ( $2^{32}$  soit 4 294 967 295 positions mémoire) et au fait qu'un mot de 32 bits permet de mémoriser directement une représentation d'un nombre flottant en simple précision (type `float` en C, souvent suivant la norme IEEE 754). Le coût décroissant des mémoires a engendré, dès les années 90, des besoins d'adressage en mémoire virtuelle approchant les 4 Go, ce qui poussa le développement d'architectures de microprocesseurs 64 bits (MIPS R4000 en 1991, DEC Alpha en 1992, jeu d'instruction X86-64 en 1999, etc.). En fait, dès les années 60, certaines machines utilisaient des mots de 64 bits (comme l'IBM 7030), les microprocesseurs 64 bits ne sont arrivés que près de quinze ans après leur utilisation par Cray Research dans le Cray-1. Les microprocesseurs actuels haut de gamme ont une architecture interne en 64 bits et ont donc un espace d'adressage théorique de 16 exbioctet ( $Eio$ ) =  $2^{64}$  octets = 18 446 744 073 709 551 616 octets. Le problème de l'espace d'adressage limité par rapport à la mémoire physique pouvant être rattaché à la machine a été totalement résolu par les architectures 32 et surtout 64 bits. Les solutions d'adressage plus ou moins indirectes des données (comme les systèmes de fichiers) peuvent donc être remises en question par ces machines à large espace d'adressage qui peuvent adresser directement un nombre gigantesque d'objets. Un besoin pour le calcul scientifique est d'être capable de représenter des entités réelles qui ne sont pas simplement des entiers en puissance de deux. L'Institut IEEE a standardisé la représentation machine (binaire) des nombres flottants dans son standard IEEE 754. Contrairement au calcul en entiers, le calcul en flottants produit des approximations (tout réel ne peut être représenté par un nombre en virgule flottante) qui peuvent faire diverger des calculs itératifs. La représentation IEEE 754 comporte certaines valeurs particulières, comme le zéro signé, la représentation des infinis, et les NaN (Not a Number). La représentation simple précision est représentée sur 32 bits, la double précision sur 64 bits et la quadruple précision sur 128 bits (et la précision étendue des architectures x86 sur 80 bits, 16 bits pour l'exposant et 64 bits pour la mantisse). Le tableau à la figure (2.8) détaille les précisions possibles de chacune des représentations.

## 2. Les GPU

**GPU** est l'acronyme anglo-saxon de Graphics Processing Units, il désigne les coprocesseurs apparus au milieu des années 90 afin de soulager les processeurs des PC (CPU) des tâches graphiques devenues de plus en plus gourmandes en ressources calcul. Sous l'influence du marché du jeu vidéo, ces architectures ont évolué à un rythme sans précédent entre 1995 et 2007, date de l'introduction de l'architecture **CUDA** par la société Nvidia, au point où les GPU deviennent concurrentiels avec les architectures classiques pour le calcul haute performance.

Rappelons enfin que d'un point de vue matériel, l'architecture des GPU

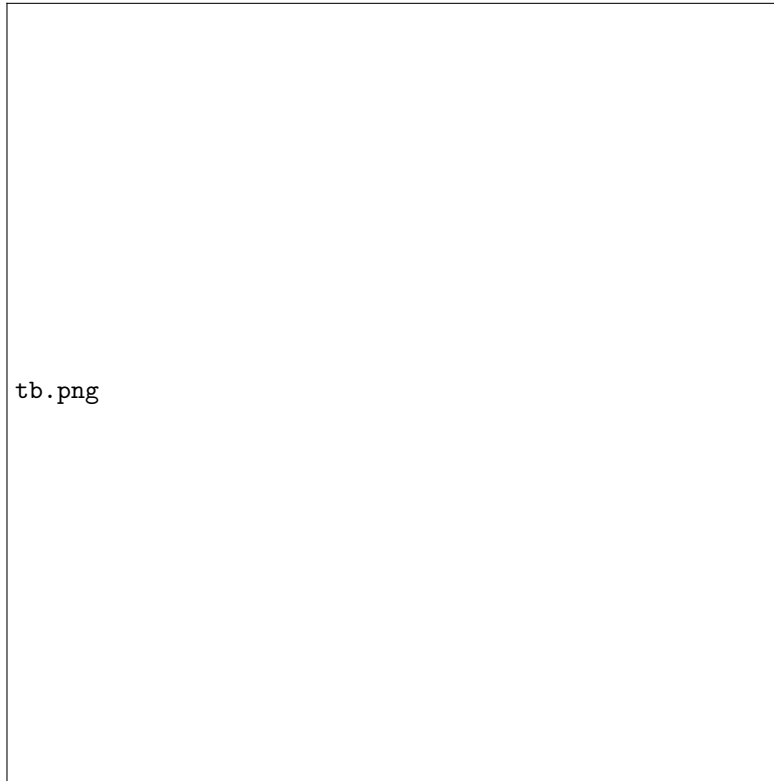


Figure 8: Précision de la représentation IEEE754 (d'après [https://fr.wikipedia.org/wiki/Virgule\\_flottante](https://fr.wikipedia.org/wiki/Virgule_flottante))

est conçue pour optimiser le traitement de tâches graphiques, dont la spécificité est de répéter les opérations pour un grand nombre de pixels; on utilise parfois le terme anglo-saxon de *stream processor* pour renforcer le lien fort avec l'implantation du pipeline graphique dans lequel les données s'écoulent comme dans une canalisation dans un seul sens (des données brutes vers le tableau matriciel de pixels). À la fin des années 90, les GPU prennent en charge une partie de plus en plus grande des fonctionnalités du pipeline graphique ; cependant, les différentes normes (OpenGL / DirectX) ayant évolué, les algorithmes de rendu graphique sont devenus de plus en plus complexes et leur mise en œuvre devient difficile à porter d'une architecture à une autre, voire d'une famille à une autre pour un même vendeur. Ainsi, au début des années 2000, apparaissent les premiers GPU programmables. L'avantage majeur de la programmabilité des GPU est évidemment le gain en souplesse d'implantation des nouveaux algorithmes complexes, et cela en logiciel plutôt que matériel. Rappelons encore que par conception les GPU sont constitués d'une col-

lection d'unités matérielles de calcul, appelés *shaders processors* de deux types : les *vertex processors* (architecture **MIMD**) et les *fragment processors* (architecture **SIMD**) fonctionnant en parallèle. Au milieu des années 2000, on réalise rapidement que ces architectures peuvent offrir une puissance de calcul plus importante que les CPU et ainsi ouvrir une nouvelle voie au calcul haute performance.

### 3. Les nœuds de calcul

La brique de base composant un supercalculateur est appelée un nœud de calcul. Ce nœud rassemble les éléments décrits plus haut au sein d'une tour de type station de travail ou lame ( *blade* en anglais) pour une intégration plus compacte pour les machines de très grande taille (fig. 2.9(a)). Les divers composants (processeurs, mémoire...) sont assemblés sur une carte mère qui sert à la fois au support des composants et à la transmission des signaux électriques entre les composants (un **PCB**). La qualité de conception d'une carte mère impacte les performances de la machine (durée de vie, vitesse de calcul, capacité de suivi de fonctionnement [ *monitoring* ]...). La carte mère peut héberger un ou plusieurs processeurs. L'usage est de compter en *socket* <sup>5</sup>. Une carte mère pour le HPC est souvent de type *bi-socket* (deux processeurs) ou *quadri-socket* comme illustré sur la figure 2.9(b). Sur un nœud s'exécute une instance du système d'exploitation (Linux dans le monde HPC).

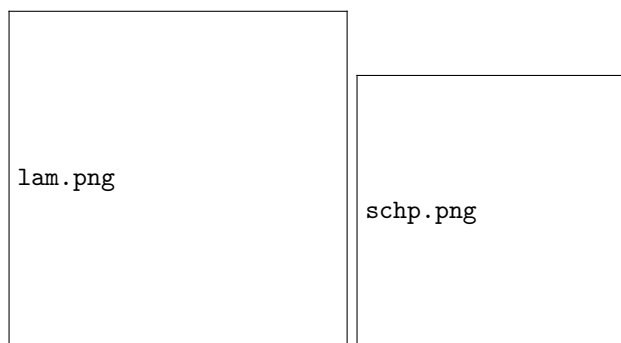


Figure 9: A gauche nous avons une lame de calcul de Tera1000. La dénomination lame et à droite Schéma de principe d'une carte mère quadri-socket. Les liens rouges sont des connexions ultra rapides entre les processeurs, les liens noirs les connexions des processeurs à leurs mémoires (de type DDR4). Les liens verts et violets représentent les liens de la machine vers le monde extérieur : en violet vers un autre nœud de calcul, en vert vers les unités de stockage. est due à la forme en lame de hachoir de l'ensemble des éléments de structuration du nœud.

### 4. Les systèmes de calcul

<sup>5</sup>Un socket est un support matériel permettant d'enficher un processeur sur une carte mère.

Si l'on n'utilise qu'un seul nœud de calcul, il est aisé de voir que la puissance de cette (petite) machine n'est pas à la hauteur des attentes des utilisateurs (un bi-socket Haswell fera dans les 600 GFlops). Pour accéder à de plus grandes puissances de calcul, la solution retenue depuis près de deux décennies est de réaliser des grappes denses (clusters en anglais) de machines (fig. 2.10) reliées entre elles par des réseaux performants (dans le cas du HPC de l'Infiniband par exemple). Pour une utilisation de type Grille, Nuage (Cloud) ou analyse de grosse masse de données (big data) des réseaux moins performants (Gigabit Ethernet) suffisent en général. Pour gagner de la place, les supercalculateurs ne sont pas un empilement de stations de travail (on parle, dans ce cas, de ferme de calcul) mais plutôt un ensemble de baies (racks en anglais) contenant plusieurs dizaines de nœuds de calcul. La densité de la machine introduit de nombreuses complexités qu'il faut prendre en compte dès la phase de conception : emplacement et nature des alimentations, type de refroidissement, nombre de périphériques possibles, moyens de surveillance du matériel, nature des connexions entre les nœuds d'une baie, position des **switchs**, etc

La figure 2.11 illustre deux topologies possibles pour le réseau. Chaque trait reliant une bulle numérotée est un câble physique qui peut être soit en cuivre (pas cher, mais utilisable seulement sur de courtes distances [1,5 m maximum à 100 Gbit/s]) soit être une fibre optique (chère, mais ouvrant l'accès à de longues distances).

## 5. Les centres de calcul

De par sa taille, un supercalculateur (et son système de stockage) ne rentrera pas sous un bureau. Toute une infrastructure est nécessaire pour l'accueillir et le faire fonctionner. Un bâtiment (ou une partie de bâtiment) va héberger des salles machines (calculateur, stockage, réseaux) spécialement conçues pour supporter le poids des différentes baies de calcul et pour délivrer le froid utile au refroidissement des équipements (climatisation à des valeurs précises de température et d'hygrométrie, eau chaude ou froide) comme illustré sur la photo en bas à droite de la figure 2.13. Des équipes de spécialistes conçoivent et exploitent ces salles pour maximiser l'efficacité énergétique de l'ensemble, en suivant un ensemble de bonnes pratiques.

## 4.3 L'intergiciel ou middleware

Si à l'origine les programmes étaient codés au plus près du matériel (en assembleur), il est très vite apparu qu'un ensemble d'outils était requis pour développer confortablement et efficacement des logiciels. Le système d'exploitation donnant une vue plus abstraite de la machine, les bibliothèques d'aide au calcul ou les langages de programmation et leurs supports d'exécution ou exécutifs

figure=baie.png,width=

Figure 10: Un cluster de calcul est une agrégation de machines (nœuds) reliées entre elles par un réseau rapide. La performance de la machine ainsi créée est fonction du nombre de nœuds agrégés, de la performance unitaire de chacun, mais aussi des performances de la topologie du réseau.

Figure 11: Différentes topologies réseau. Les bulles bleues sont des nœuds de calcul. Les bulles rouges et vertes sont des commutateurs réseau (**switch**) permettant de relier les différents nœuds entre eux. Le cercle orange indique où peut se faire un élagage pour économiser sur le nombre de câbles, au détriment de la performance.)

figure=nd.png,width=

Figure 12: Vision d'un centre de calcul centrée sur les données.

figure=super.png,width=

Figure 13: Photographies de quelques-uns des éléments indispensables au fonctionnement d'un supercalculateur. Ce sont les servitudes du centre de calcul.

associés sont souvent regroupés sous le vocable d'intergiciels (ou **middleware** en anglais).

#### 4.3.1

Le système d'exploitation et les logiciels bas niveau

De la création d'un programme à son exploitation par un utilisateur, un grand nombre d'étapes doivent être parcourues, elles peuvent être résumées dans le schéma de la figure 2.14.

Dans ce schéma, les étapes sont indiquées dans les cases violettes, les intervenants sont indiqués en noir et les logiciels dans les cases vertes. L'idée initiale d'un programme vient d'un utilisateur ingénieur ou chercheur, le programmeur est responsable de sa transformation dans un langage de programmation. Ces deux premières phases sont réalisées par des humains, les phases suivantes sont réalisées par des logiciels.



Dans le schéma temporel précédent le compilateur est responsable de la transformation du code source vers le code exécutable et de la création des liens vers les bibliothèques nécessaires au programme. Le système d'exploitation est responsable de la gestion des ressources de la machines pendant l'exécution du programme, du démarrage du programme (chargement mémoire) à sa fin (désallocation de ressources), ou durant son exécution avec les allocations de ressources (temps de calcul, mémoire, bande passante réseau, disque, etc.).

Nous trouvons plus souvent une description en couches comme dans la figure

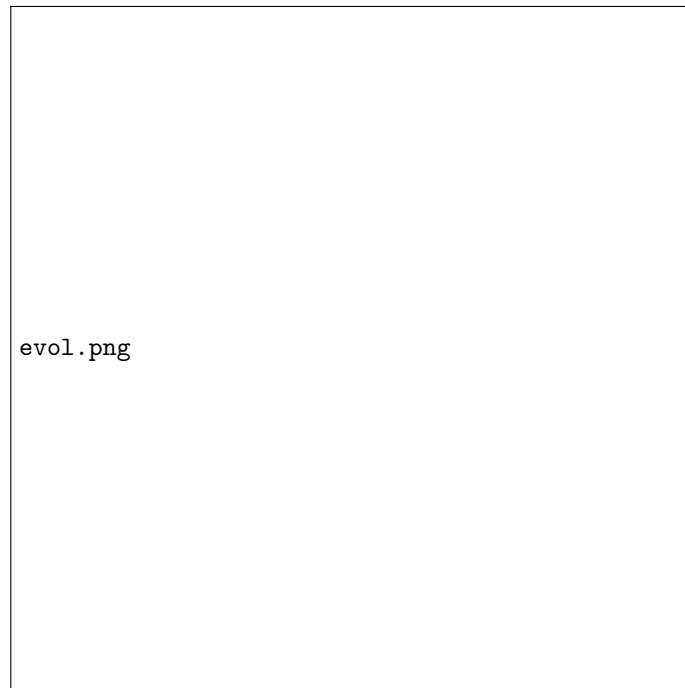


Figure 14: Évolution d'une application de sa conception à son exécution.

2.15. Le système d'exploitation étant alors vu comme une virtualisation des périphériques physiques sur lesquelles les applications et les bibliothèques vont s'appuyer. Cette description en couche permet, de la même façon que les couches ISO en réseau, une certaine indépendance entre les couches les plus hautes et les couches physiques. Nous pouvons parcourir les différentes couches en partant du bas :

- Le processeur ou, le plus souvent, un groupe de processeurs sont regroupés sur un circuit électronique. Il existe plusieurs familles de processeurs définis par leur jeu d'instructions (**ISA**), principalement Intel, ARM, MIPS, POWER, SPARC, etc. ;
- pour le **système d'exploitation**, en particulier dans le cadre de ma-

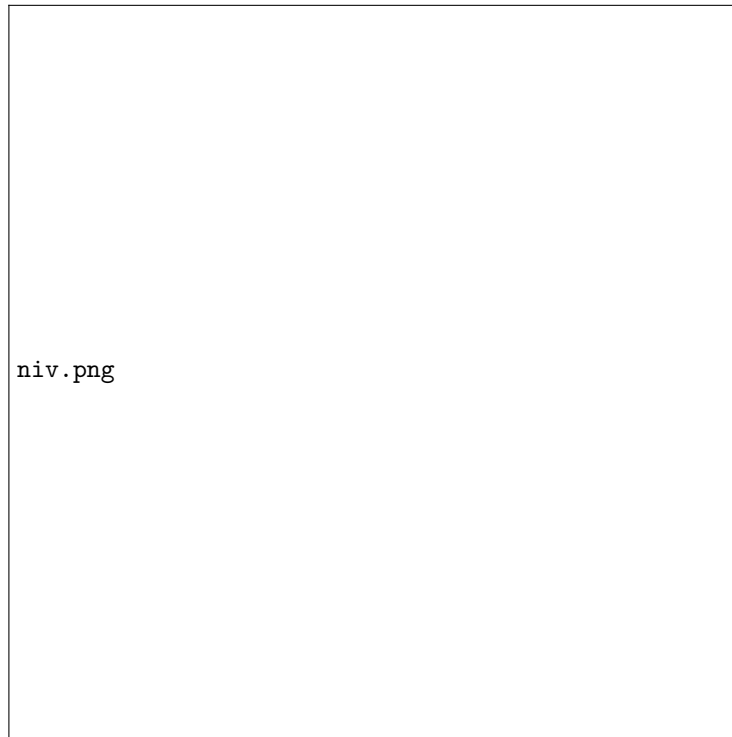


Figure 15: Niveaux de virtualisation. Du processeur à l'application.

chines parallèles permettant d'exécuter des simulations à grande échelle, plusieurs approches sont possibles. Une machine parallèle sera constituée d'un grand nombre de processeurs. Plusieurs situations seront possibles : chaque processeur utilisera une instance d'un système d'exploitation. (système mono-processeur/cœur) ou des groupes de processeurs (des chiplets) seront regroupés, et le système d'exploitation sera chargé de répartir les différentes applications sur ces processeurs ;

- les bibliothèques et exécutifs sont des parties de programmes sur lesquelles un programmeur pourra s'appuyer afin de réaliser son application. Elles sont, en général, fournies par le constructeur de l'ordinateur et reposent sur une interface plus ou moins normalisée, ce qui permet de rendre le programme indépendant d'une plateforme matérielle (processeurs, mémoire, réseau...). Les bibliothèques incluent, entre autres, des fonctions permettant d'effectuer des calculs ou opérations de manière simplifiée et/ou optimisée, tandis que les exécutifs regroupent des fonctionnalités nécessaires pour interagir avec l'environnement système et matériel. Ainsi les supports exécutifs comportent les implémentations de modèles de programmation

parallèles comme MPI ou OpenMP (voir le chapitre sur l'introduction à la programmation parallèle). Suivant cette nomenclature, un exemple de bibliothèque est BLAS (Basic Linear Algebra Subprograms) : la bibliothèque de référence permettant d'effectuer des calculs d'algèbre linéaire (opérations sur vecteur et des matrices) et un exemple d'exécutif est **MPI** (Message Passing Interface) : le modèle de référence pour la communication des données entre différents processeurs ;

- les applications sont écrites dans un langage de programmation (C, Fortran, Java, Python, R, etc). Ces langages sont soit compilés (traduits en code machine), soit interprétés. Pour les langages compilés (C, Fortran), le compilateur se chargera de transformer les programmes source en langage exécutable pour le processeur (ISA), pour les langages interprétés, le programme est interprété par une machine virtuelle.

## 5 Le logiciel

### 5.1 Le processus de développement des logiciels

Le deuxième niveau de la poupée russe présentée en introduction est composé de la partie logicielle. Plus exactement de l'architecture logicielle qui va être le squelette de l'application. Cette architecture aura besoin de répondre à différents besoins et s'adapter à différentes contraintes. Tout d'abord, elle devra permettre d'atteindre les objectifs de l'application qui utilisera cette architecture : performances, robustesse, pérennité, portabilité... Elle devra autoriser l'utilisation d'un ou plusieurs modèles de programmation.

Très souvent une application a pour but de modéliser des phénomènes complexes et multiphysiques. Il est donc nécessaire que les applications élémentaires se couplent entre elles afin d'offrir des capacités accrues de modélisation. Tout grand logiciel doit suivre un processus rigoureux de développement, étape clef dans le cycle de vie du logiciel, le processus de vérification/validation (V&V) doit permettre de s'assurer que le logiciel développé répond de manière correcte aux exigences initiales. Afin de définir le processus lui-même, il est nécessaire de définir les concepts et la différence entre vérifier et valider :

- Vérifier consiste à s'assurer que les choses sont faites conformément à ce qui avait été défini ;
- valider consiste à s'assurer que le résultat est bien atteint.

Cette mise en œuvre passe par la mise en place de tests qui vont avoir chacun leur rôle respectif. Il existe de nombreux cas tests différents, parmi les principaux, nous pouvons citer :

- **Les tests unitaires**

Les tests unitaires consistent à tester individuellement les composants de l'application. On pourra ainsi valider la qualité du code et les performances d'un module.

- **Les tests fonctionnels** Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils reposent donc sur les spécifications fonctionnelles et techniques. Tests unitaires et tests fonctionnels vont être utilisés conjointement afin de réaliser le processus de V&V. La mise en place de tous ces tests n'a de sens que s'ils sont associés à des métriques. En effet, rien ne sert de multiplier les cas tests s'il n'est pas possible de vérifier quels modules ou quelles fonctionnalités sont effectivement vérifiés et validés. Pour ce faire, deux métriques sont principalement utilisées :

- **Le taux de couverture logicielle de la base de cas tests** : La couverture logicielle du code (ou code coverage) permet d'évaluer la qualité d'un jeu de test en vérifiant quelles sont les parties du code qui sont appelées lors des tests. Si du code est appelé pendant un test, cette portion de code est considérée comme couverte ; a contrario, tout le code non appelé est considéré comme non couvert. La couverture s'exprime donc sous forme d'un pourcentage représentant la proportion de code couverte sur la quantité totale de code.
- **Le taux de couverture fonctionnelle de la base de cas tests** : Il s'agit ici de mesurer quelles sont les fonctionnalités (au sens physique du terme) couvertes par la base de cas tests. Ces fonctionnalités physiques à couvrir sont spécifiées via une approche de type PIRT (Physical Identification Ranking Table) et des outils permettent de vérifier automatiquement que ces fonctionnalités sont effectivement vérifiées.

- **Les tests d'intégration**

Ces tests sont exécutés pour valider l'intégration des différents modules entre eux et dans leur environnement d'exploitation définitif. Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.

- **Les tests de non-régression**

Les tests de non-régression permettent de vérifier que des modifications n'ont pas altéré le fonctionnement de l'application.

## 5.2 Les modèles de programmation

On utilise un langage de programmation pour créer des applications car l'ordinateur n'est d'aucune utilité sans programmes. Chaque langage a des caractéristiques

différentes et permet d'utiliser des paradigmes de programmation<sup>6</sup> plus ou moins complexe. L'étude de ces paradigmes pourrait faire l'objet d'un livre entier, nous nous contenterons de résumer les caractéristiques des langages de programmation les plus souvent utilisés. Les langages de programmation que l'on rencontre le plus souvent sont :

- **Fortran**, c'est un langage quia eu beaucoup de succès dans le domaine scientifique car relativement éloigné des spécificités des machines. Beaucoup de bibliothèques scientifiques actuellement disponibles ont été écrites en Fortran ;
- **C**, développé à partir des années 70, ou **C++** sont des langages impératifs (C++ est en plus orienté objet) et compilés plus proche de l'architecture des machines. C'est un inconvénient du point de vue de l'expressivité pour le scientifique, mais c'est un avantage pour la personne responsable de l'optimisation machine ;
- **Python** est un langage orienté objet interprété conçu dans les années 90. Les performances que l'on peut obtenir sont assez faibles, mais sa versatilité d'interfacer des programmes C/C++ permet d'utiliser facilement des bibliothèques, d'enchaîner plusieurs programmes ou d'utiliser des accélérateurs matériels ;
- **Matlab**, Scilab sont des langages interprétés spécialisés dans le calcul numérique. Ils permettent d'utiliser facilement de nombreuses bibliothèques spécialisées ;
- **Java** est un langage orienté objet, compilé en deux étapes, dont le développement a démarré dans les années 90. Une première étape permet de produire un programme exécutable indépendant d'une machine, une machine virtuelle est ensuite chargé d'interpréter le code puis de le compiler peu à peu au cours de l'exécution (compilation **JIT**<sup>7</sup>).  
Historiquement, ce langage n'était pas apprécié pour le développement d'applications scientifiques, mais de récentes extensions montrent un regain d'activité dans le domaine du calcul haute performance :
  - X10 est un langage de programmation orienté objet spécialisé pour la programmation d'applications parallèles développé par IBM ;
  - Scala est un langage de programmation orienté objet également dédié aux applications parallèles développé à l'EPFL ;
- **CUDA** (Compute Unified Device Architecture) et **OpenCL** (Open Computing Language) sont des langages dédiés à la programmation d'accélérateurs de calculs. Pour effectuer des calculs importants (grande taille mémoire

---

<sup>6</sup>[https://en.wikipedia.org/wiki/Programming\\_paradigm](https://en.wikipedia.org/wiki/Programming_paradigm)

<sup>7</sup>[https://fr.wikipedia.org/wiki/Compilation\\_à\\_la\\_volée](https://fr.wikipedia.org/wiki/Compilation_à_la_volée)

et/ou temps de calcul très important), il faut utiliser une machine parallèle. Dans ce cas, le programmeur doit s'appuyer sur trois niveaux de parallélisme :

- Un parallélisme d'instruction avec les architectures super scalaires et les instructions vectorielles ;
- un parallélisme en mémoire partagée au sein d'un nœud où plusieurs flux d'exécution (threads) indépendants ont accès au même espace mémoire ;
- un parallélisme en mémoire distribuée entre les nœuds d'une grappe de calcul où des processus ont accès à des espaces mémoire séparés et communiquent via le réseau.

Pour se faire le programmeur doit :

- \* Paralléliser son application manuellement, c'est-à-dire préparer un programme pour tous les nœuds de la machine parallèle et utiliser des bibliothèques de communication comme MPI ;
- \* paralléliser son application automatiquement en utilisant soit :
  - un compilateur qui va réaliser automatiquement ou semi automatiquement la démarche précédente. Les compilateurs parallélisants n'utilisent que le langage source pour extraire le parallélisme et produire le code binaire parallèle. Les compilateurs semi-automatiques sont guidés par des directives fournies par le programmeur pour produire du code parallèle (par exemple, OpenMP ou OpenACC).
  - un langage spécialisé pour les accélérateurs parallèles.

Voici deux exemples d'outils permettant une parallélisation automatique :

#### 1. **OpenMP<sup>8</sup> (Open Multi-Processing)**

C'est un modèle de programmation pour la parallélisation d'application sur des machines à mémoire partagés. Son utilisation se fait via C/C++ ou FORTRAN dans lesquels des directives spécifiques aident le compilateur à préparer le code pour une utilisation sur une machine parallèle ;

#### 2. **OpenACC<sup>9</sup> (Open ACCelerators)**

C'est un modèle de programmation pour la parallélisation d'application pour des machines hétérogènes utilisant des accélérateurs de calcul (GPU). Son utilisation est similaire à OpenMP, il faut ajouter des directives à un programme pour indiquer au compilateur.

### 5.2.1 Remarque

Concernant le parallélisme en mémoire distribuée, le paradigme de passage de message synchrone porté par la bibliothèque **MPI** (Message Passing Interface)

---

<sup>8</sup><http://openmp.org>

<sup>9</sup><http://www.openacc.org>

et proche du formalisme CSP (Communicating Sequential Processes) connaît une domination écrasante. Il s'agit d'assigner un numéro (le rang) unique à chaque processus et à leur permettre de s'échanger des messages. L'échange est synchrone<sup>10</sup> en ce qu'il nécessite non seulement que le processus envoyant les données spécifie la donnée à envoyer, mais aussi que le processus recevant spécifie l'adresse mémoire où stocker cette donnée. Au-dessus de ces communications point-à-point sont construites des communications dites collectives qui font intervenir plus de deux processus.

Un modèle comme OpenMP ou une application séquentielle a la même comportement une fois OpenMP mis en place et où les zones parallèles peuvent être introduites de manière incrémentale, l'introduction de MPI implique que tout le code sera par défaut exécuté par l'ensemble des processus et la sémantique est donc changée par rapport à la version séquentielle.

### 5.3 Le débogage de code

Lorsqu'un code ne s'exécute pas convenablement, il convient de trouver la (ou les) source(s) du problème. Elles peuvent être de différente nature ; liées à la compilation des sources – par exemple un problème de bibliothèques utilisées à l'exécution ou la compilation, ou plus fréquemment à une erreur dans l'algorithme implémenté ou les données louées pour le faire fonctionner. La commande `ldd` lancée dans le même environnement d'exécution aide à comprendre le travail du loader qui charge les bibliothèques à l'exécution. Dans le second cas, on est souvent aidé par l'affichage de la pile d'exécution dans la sortie d'erreur. Si le programme a été compilé avec sa table des symboles (ensemble des variables humainement lisibles), on aura parfois la bonne ligne de code, mais c'est souvent plus en amont que l'erreur a été commise. On utilise aussi différents outils de débogage en fonction de la nature du programme.

Si le programme est séquentiel, le débogueur natif des systèmes gnu-linux est `gdb`. Il permet de charger un exécutable simple et après un arrêt posé judicieusement un peu avant la zone supposée de l'erreur de faire du pas à pas dans l'exécution du programme, tout en suivant les valeurs des variables, voire en modifiant certaines. Il est en ligne de commande, mais de nombreuses interfaces graphiques lui sont dédiées, telle que `ddd`. Ce débogueur est lié aux compilateurs `gnu`, `gcc`, `g++`, `gfortran`... mais d'autres compilateurs peuvent être analysés. Toutefois, les compilateurs propriétaires, comme INTEL `icc` viennent souvent avec un compilateur dédié qui interprète souvent mieux leur table de symboles. Les codes parallèles peuvent être analysés avec `gdb` s'ils s'exécutent sur des petits cas, sur un seul nœud de calcul. Lorsque l'on utilise un grand nombre de nœuds, il est plus adéquat d'utiliser les débogueurs propriétaires tels que Totalview ou Alinéa DDT. Ces débogueurs prétendent à une très bonne scalabilité pour un

---

<sup>10</sup>Dans les dernières versions de la norme MPI, il existe néanmoins la possibilité d'utiliser une version asynchrone des primitives MPI, permettant de recouvrir la communications et les calculs, voire des primitives de communication asymétriques (one sided).

impact minime sur le calcul. Ils permettent en tout cas de visualiser des variables réparties sur un grand nombre de processeurs.

## 5.4 Le cycle de vie des données

Les données sont de plus en plus au cœur de la recherche et à la source de découvertes et d'innovations. Elles sont devenues la richesse des communautés et de l'industrie au même titre que les brevets, le savoir-faire et les biens matériels. Ainsi, il devient crucial de penser leur génération et leur gestion au cours du temps. Cette démarche est décrite dans le cycle de vie des données (voir fig. 2.16). Il apparaît donc que les données sont bien au cœur de la recherche et de

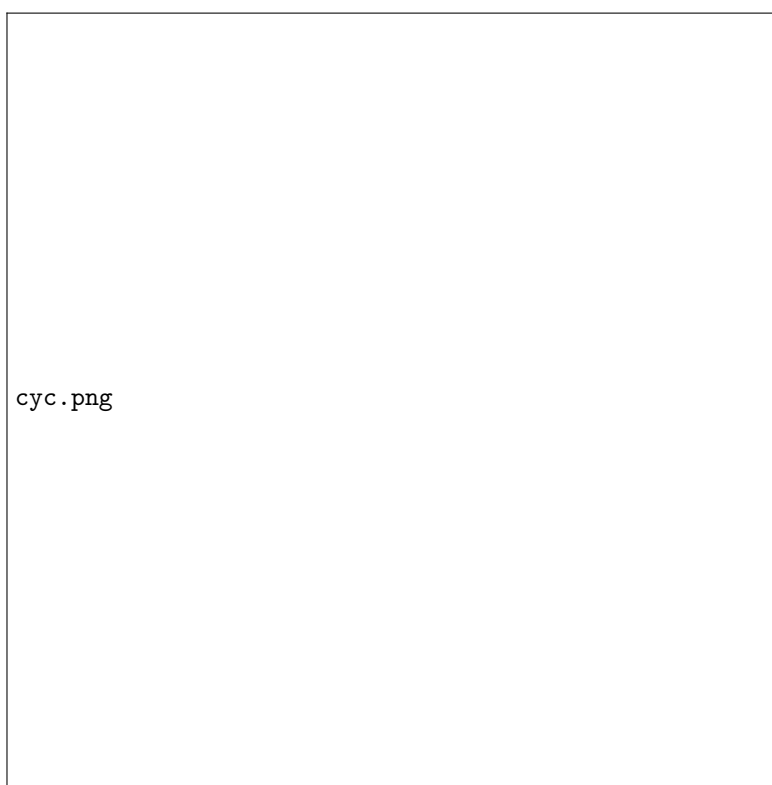


Figure 16: Le cycle de vie des données.

l'innovation et que l'explosion des volumes fait que seuls les systèmes comme les calculateurs hautes performances peuvent les traiter. Les entrées/sorties parallèles sont devenues un composant critique du calcul haute performance moderne. Les simulations sur les machines Petafloquiques produisent quotidiennement des jeux de données allant de plusieurs téraoctets aux pétaoctets.



L'analyse de ces données et leur visualisation nécessitent des capacités de lectures et de traitement particulièrement efficaces. Travail en ligne de commandes dans l'environnement Linux

## 6 Introduction

Unix est un système d'exploitation multi-tâches et multi-utilisateurs. Unix met ses fonctionnalités à la disposition des programmes d'application sous forme d'appels système.

L'Unix est un système hiérarchisé. Chaque fichier est enregistré dans un répertoire. Un répertoire être de ou contenir des fichiers, il peut aussi contenir d'autres répertoires que nous appellerons sous-répertoire. Unix a trois types de fichiers :

- **fichiers simples:** sont des fichiers qui peuvent contenir des textes, codes sources, fichier exécutable, etc.
- **Fichiers spéciaux:** Ceux-ci représentent des périphériques physiques tels que des terminaux et des lecteurs de disque. Les "pilotes de périphériques" traduira toute référence à ces fichiers dans les instructions matérielles nécessaires pour effectuer les tâches.
- **Repertoires:** ils contiennent des "pointeurs" pour des fichiers simples, les fichiers spéciaux et d'autres répertoires.

LINUX correspond à proprement parler au noyau d'un système d'exploitation de type UNIX (noyau MONOLITHIQUE). Ceci signifie qu'il s'agit des fondations du système définissant le type de système de fichiers ainsi que la gestion des périphériques. A ce noyau se rattache certains modules. Le noyau et ses modules constituent le système d'exploitation proprement dit. Une distribution LINUX est un système d'exploitation comportant le couple noyau + modules sur lequel ont été greffés des utilitaires afin de constituer un système complet et convivial.

LINUX est libre (ce qui n'est pas forcément synonyme de gratuit). Cela signifie que les codes sources sont accessibles et modifiables par qui veut. Vous pouvez par exemple récupérer le noyau de LINUX par FTP ( <ftp://ftp.kernel.org/pub/> ) et vous créer votre propre distribution.

Comme la plupart des systèmes d'exploitation modernes, Linux fournit deux interfaces pour la saisie utilisateur. Tous les réglages que vous effectuez via l'interface graphique ( *Graphical User Interface*, GUI) peuvent être en effet également effectués sous forme de lignes de commande via ce que l'on appelle le shell. Le shell est un programme qui agit comme interface entre le système et l'utilisateur. On parle également d'interface système. Il comprend un interpréteur en ligne de commande qui accepte l'entrée de l'utilisateur via le clavier, l'évalue, met en route les programmes en conséquence et renvoie le résultat à l'utilisateur sous forme d'une sortie texte. Chaque shell a par ailleurs son propre langage de programmation permettant d'écrire des scripts shell, par

exemple, pour appeler des programmes ou simplifier des tâches administratives.

Chaque shell fonctionne dans un terminal. Au début de l'ère informatique, ce sont des appareils indépendants, les terminaux *hardcopy* (imprimante ou écran et clavier), qui étaient utilisés. Ceux-ci ont été remplacés sur les ordinateurs modernes par des consoles virtuelles (ou émulateurs de terminaux), programmes qui fournissent une fenêtre graphique permettant aux utilisateurs d'interagir avec le shell.

Un appel de programme via le terminal est effectué grâce au nom du programme. La plupart du temps, il est possible d'utiliser certaines fonctions supplémentaires grâce à des options. Si un programme attend des arguments (par exemple sous forme de fichiers ou de chemins d'accès à un répertoire), ceux-ci sont généralement spécifiés après les options choisies. Voici une vue d'ensemble des commandes Linux les plus courantes et des programmes de ligne de commande associés.

## 7 Les commandes de bases

### 7.1 Opérations sur les fichiers et répertoire

#### 1. *man* - Accéder aux pages du manuel en ligne

la commande *man* ouvre les pages de manuel de votre distribution Linux directement dans le terminal.

Utilisez le schéma suivant pour accéder à une page de manuel : *man* [OPTION] SUJET

Exemple: *man octave*

il vous donnera, toutes les descriptions du logiciel octave. pour quitter et revenir au prompt, taper *q*. Les pages *man* Linux sont divisées en 10 domaines : Appel système, Fonctions du langage de programmation C, Voir les packs spécifiques, Fichiers et Formats de fichier, Jeux, Divers, Commandes pour l'administration du système, Fonctions cœur (noyau Linux), Nouvelles commandes.

#### 2. *pwd* - Imprimer le répertoire de travail

Utilisez *pwd* (abréviation de print working directory) pour faire afficher le nom du répertoire de travail actuel. Voici la syntaxe de la commande : *pwd* [OPTIONS]

A part le nom du répertoire dans lequel vous travaillez, cette commande vous permet aussi à vous donner le chemin pour y arriver.

#### 3. *cd* - Changer de répertoire

Navigation dans l'arborescence des répertoires. La commande *cd* (pour change directory) est utilisée pour la navigation dans l'arborescence des répertoires.

Voici la syntaxe de la commande : *cd* [OPTION] REPERTOIRE

Si aucun répertoire cible n'est spécifié, *cd* passe automatiquement au répertoire home de l'utilisateur. Si *cd* est utilisé avec un signe moins (-), il passe au répertoire précédent.

#### 4. ***ls* - Lister le contenu des répertoires**

Liste du contenu du répertoire. La commande *ls* (pour liste) sert à afficher le contenu d'un répertoire (les noms de tous les fichiers et dossiers du répertoire spécifié). La syntaxe de la commande est la suivante : *ls* [OPTIONS] REPERTOIRE

Si *ls* est utilisé sans spécification de répertoire, la commande liste le contenu du répertoire courant. A l'aide des options supplémentaires, vous pourrez définir quelles informations à afficher et comment.

#### 5. ***mkdir* - Création de répertoires**

La commande *mkdir* signifie make directory. Il permet aux utilisateurs de Linux d'établir de nouveaux répertoires. Utilisez la syntaxe suivante pour créer un nouveau répertoire dans le répertoire courant : *mkdir* [OPTION] NOMREPERTOIRE. par exemple:

```
jean@kaherero: /Bureau$ mkdir HPC/
```

cela signifie que, le répertoire HPC a été créé au Bureau de la machine jean@kaherero.

#### 6. ***rmdir* - Suppression de répertoires**

Si vous voulez supprimer un répertoire spécifique, utilisez la commande *rmdir* selon la syntaxe suivante : *rmdir* [OPTION] REPERTOIRE  
Exemple:

```
jean@kaherero: /Bureau$ rmdir HPC/
```

Ici nous constatons que le répertoire HPC vient d'être supprimé dans son emplacement.

Avec *rmdir*, vous ne pouvez supprimer que les répertoires vides. Pour supprimer un répertoire comprenant des fichiers et sous-dossiers, utilisez la commande *rm* (remove) avec l'option -r. Attention : *rmdir* n'a pas

besoin de confirmation pour procéder à la suppression. Les répertoires sélectionnés sont supprimés immédiatement, sans possibilité de revenir en arrière.

## 7. *cp* Copier un fichier

La commande *cp* (pour copy) est utilisée pour copier des fichiers et des répertoires. La syntaxe de base de la commande est : *cp* [OPTIONS] *SOURCE CIBLE*

Sous la *SOURCE*, on trouvera l'élément à copier. Comme *CIBLE* du processus de copie, on définira un fichier ou un répertoire. Si vous définissez un fichier existant comme fichier cible, son contenu sera écrasé par celui du fichier source. Autrement, vous pouvez créer un nouveau fichier pour la cible portant le nom de votre choix. Si plusieurs fichiers sont copiés, la cible doit être un répertoire. Il en va de même si un répertoire est copié. Pour copier un fichier source vers un fichier de destination dans le répertoire courant : *cp* [OPTIONS] FICHIERSOURCE FICHIERCIBLE  
Exemple :

```
cp fichier.txt fichiercopie.txt
```

Pour copier un fichier source du répertoire courant dans un répertoire cible : *cp* [OPTIONS] FICHIERSOURCE REPERTOIRECIBLE Exemple :

```
cp fichier.txt home/user /documents/202211
```

Pour copier plusieurs fichiers sources dans un répertoire cible : *cp* [OPTIONS] FICHIERSOURCE1 FICHIERSOURCE2 REPERTOIRECIBLE  
Exemple :

```
cp fichier.txt fichier.odt home/user /documents/2022
```

Pour copier un répertoire source du répertoire courant dans un répertoire cible : *cp* REPERTOIRESOURCE REPERTOIRECIBLE Exemple :

```
cp repertoire1 home/user/documents/2018
```

Si vous souhaitez copier des répertoires avec tous leurs contenus, les sous-répertoires doivent être associés dans le processus de copie en utilisant l'option *-r*.

---

<sup>11</sup>2022 est le nom du répertoire auquel on veut appliquer l'action, rien avoir avec l'année. la même conception est valide dans ce chapitre

## 8. *cat* - Impression de fichiers sur l'écran

Le programme en ligne de commande *cat* (abréviation de concatenate) a été développé comme un outil de fusion de contenu contenu dans le terminal. Utilisez la syntaxe suivante pour appeler la commande *cat* dans le terminal pour lire un fichier et le sortir sur *stdout* (sortie standard) : *cat* OPTIONS FICHIER Si vous avez plusieurs fichiers, il sera nécessaire de les séparer par des espaces : *cat* OPTIONS FICHIER1 FICHIER2 Les contenus des fichiers se lient à l'aide des signes de direction (*;*, *|* et *—*). Par exemple, utilisez le chevron supérieur à (*;*) pour fusionner le contenu de deux fichiers dans un troisième fichier :

```
cat fichier1.txt fichier2.txt ; fichier3.txt
```

## 9. *more* - Impression de fichiers un écran à la fois

La commande *more* imprime le contenu des fichiers nommés, un écran plein à la fois. La forme du *more* commande est: *more filename(1) filename(2)... filename(n)*

## 10. *mv* - Déplacer et renommer des fichiers

La commande *mv* modifie le nom ou l'emplacement d'un fichier ou d'un répertoire. Les formats de la commande *mv* sont

```
mv ancienfichier nouveaufichier
```

Déplacer un fichier dans un autre répertoire : *mv* [OPTIONS] FICHIER SOURCE REPERTOIRECIBLE.

Par exemple : *mv* fichier1.txt home/user /documents/2022

Déplacer plusieurs fichiers sources vers un répertoire cible : *mv* [OPTIONS] FICHIERSOURCE1 FICHIERSOURCE2 REPERTOIRECIBLE

Par exemple : *mv* fichier1.txt fichier2.txt home/user/documents/2022

Déplacer un sous-répertoire du répertoire courant vers un répertoire cible : *mv* [OPTIONS] NOMREPERTOIREOLD NOMREPERTOIRENEW

Par exemple : *mv* repertoire1 home/user /documents/2022

## 11. *rm* - Suppression de fichiers et de répertoires

La commande *rm* supprime les fichiers et les répertoires. Attention : Il n'y a aucun moyen d'inverser ce processus (bien que voir la section sur la sauvegarde). Le format de la commande *rm* est

```
rm [-i] [-r] fd(1) fd(2)... fd(n)
```

où fd(1..n) sont des fichiers ou des répertoires.

- -i Se renseigner avant de supprimer un fichier ( y pour supprimer, si non taper une autre lettre).
- -r Supprime récursivement un répertoire et tout son contenu et sous-répertoires (à utiliser avec une extrême prudence).

## 12. *diff* - Recherche des différences entre deux fichiers

La commande diff compare le contenu de deux fichiers. Le format de diff est *diff fichier1 fichier2*. Par exemple,

```
alph% diff dataA dataB
```

La commande diff compare les fichiers ligne par ligne des fichiers dataA et dataB.

## 13. *grep* - Recherche de chaînes dans les fichiers

La commande grep analyse un fichier pour l'occurrence d'un mot ou d'une chaîne et imprime toute ligne dans laquelle il est trouvé. Le format de grep est

```
grep [-i] 'string' filename(1) filename(2)... filename(n)
```

## 14. *file*

Emission du type de fichier. La commande *file* peut être utilisée pour fournir des informations sur le type d'un fichier. L'appel repose sur le schéma suivant :

```
file [OPTIONS] FICHIER
```

## 15. *stat*

Émission des dates et marques de temps. La commande *stat* (statut) peut être utilisée pour afficher les dates des accès et modifications des fichiers et répertoires sélectionnés. La syntaxe générale de la commande est :

```
stat [OPTIONS] FICHIER
```

Le format de sortie peut être ajusté à l'aide d'options.

## 16. *uniq*

Éliminer les doublons des listes de fichiers et sorties de programmes. La commande *uniq* est généralement utilisée avec *sort* pour se débarrasser des lignes en double des fichiers triés. Dans l'exemple suivant, la commande *sort* est reliée à *uniq* par le pipe (—) pour trier tout d'abord le fichier et ensuite éliminer les doublons :

sort fichier.txt — uniq

## 7.2 Gestion des droits

Sous Linux, les droits d'accès et de propriété des fichiers et des répertoires peuvent être facilement paramétrés via le terminal. Les lignes de commande les plus importantes pour la gestion des droits sont `chown` et `chmod`. L'appartenance à un groupe est gérée à l'aide de la commande `chgrp`.

### 1. *chmod* - Modification des autorisations d'accès

*chmod* en anglais "Changing Access Permissions" est la commande qui modifie les "autorisations" sur un fichier ou un répertoire. Il donne ou supprime l'accès pour un autre utilisateur ou groupe d'utilisateurs de lire, modifier ou exécuter l'un des fichiers dont vous êtes propriétaire.

Les utilisateurs du système se répartissent en trois catégories : vous qui êtes l'utilisateur, le groupe, toute personne dans la même classe que vous, comme *pg 1999*, *staff* ou *postdoc*. et enfin d'autres, donc toute personne qui utilise les ordinateurs de l'Institut ou laboratoire (parfois appelé monde).

Le format de la commande `chmod` est

```
chmod ugo+-rwX fd(1) fd(2)... fd(n)
```

où `fd(1..n)` peut être un fichier ou un répertoire. où *ugo* Spécifiez *u* (utilisateur), *g* (groupe) ou *o* (autre). ~~+~~ = Spécifiez + (addition), ~~-~~ (soustraction) ou = (ensemble). *rwX* Spécifiez *r* (lecture), *w* (écriture) ou *X* (exécution).

Pour les fichiers, l'autorisation de lecture signifie que vous pouvez lire le contenu d'un fichier, l'autorisation d'écriture signifie que vous pouvez modifier le fichier et l'autorisation d'exécution signifient que vous pouvez exécuter le fichier (s'il est exécutable ou s'il s'agit d'un script shell). Pour les répertoires, l'autorisation de lecture signifie que vous pouvez voir quels fichiers se trouvent dans le répertoire, l'autorisation d'écriture signifie que vous pouvez ajouter et supprimer des fichiers du répertoire et exécuter signifie que vous pouvez accéder aux fichiers dans ce annuaire. Notez que pour accéder à un fichier, vous devez avoir l'autorisation d'exécution sur tous les répertoires au-dessus de celui du fichier système (y compris celui dans lequel il réside). De plus, vous devez disposer de l'accès approprié autorisations pour ce fichier. Les permissions sur un fichier peuvent être affichées en utilisant la commande

```
alph% ls -lgF
... -rw-r--r-- 1 newuser pg 1999 1451 Jan 18 11:02 phone
drwxr-xr-x 2 newuser pg 1999 512 Jan 22 12:37 thesis/
```

...  
alph%

Il y a 10 champs au début de l'entrée. La première lettre indique si l'entrée est un fichier (-), un répertoire (d), ou autre chose (comme s ou l). Les lettres suivantes doivent être considérées par groupes de trois. Le premier groupe de trois fait référence à l'utilisateur. Le deuxième groupe de trois fait référence aux membres du groupe "pg 1999". Le dernier groupe de trois fait référence au monde. À l'intérieur de chaque groupe de trois, les trois entrées font référence à lire (r), écrire (w) et exécuter (x). Un trait d'union (-) indique que quelque chose n'est pas défini. Dans l'exemple ci-dessus, tous les utilisateurs du système peuvent lire et changer dans le répertoire appelé thesis et tous les utilisateurs peuvent lire le fichier phone, et en plus le propriétaire peut écrire dans à la fois le répertoire de la thèse et peut changer le téléphone du fichier. Si, par exemple, le matériel contenu dans le dossier téléphonique était personnel, il est possible de s'assurer que personne d'autre ne peut lire le fichier en tapant

alph% chmod g-r o-r phone

et la nouvelle sortie de ls -agl serait

```
alph%ls -agl
...
-rw----- 1 newuser pg 1999 1451 Jan 18 11:02 phone
drwxr-xr-x 2 newuser pg 1999 512 Jan 22 12:37 thesis
...
alph%
```

dans ce cas, *g-r* fait référence à "group remove read" et *o-r* "others remove read". Vous pouvez également utiliser `alpha% chmod g+r` pour permettre à n'importe qui dans le groupe "pg 1999" de lire le fichier.

## 2. *chattr*

Gérer les attributs des fichiers. Le programme en ligne de commande *chattr* (abréviation de change attribute) permet d'octroyer des attributs aux fichiers ou répertoires. Une adaptation des attributs de fichiers est prise en charge par divers systèmes de fichiers (par exemple ext2, ext3, ext4, XFS, ReiserFS, JFS, JFS et OCFS2). Utilisez *chattr* selon la syntaxe suivante pour définir un attribut :

chattr [OPTIONS] +ATTRIBUT FICHIER

Les attributs définis peuvent être supprimés selon le même schéma :



chattr [OPTIONS] -ATTRIBUT FICHIER

Par exemple, définissez l'attribut `-i` pour empêcher les changements (suppressions ou modifications) dans un fichier ou un répertoire :

chattr +i datei.txt

D'autres attributs et options possibles peuvent être trouvés dans la page d'aide du programme *chattr*.

### 3. *chgrp*

Gérer les groupes pour des fichiers ou répertoires. La commande *chgrp* est l'abréviation de change group. Elle est utilisée pour gérer les appartenances d'un groupe à des fichiers et répertoires. Pour appliquer *chgrp* à un fichier ou répertoire sélectionné, vous devez avoir des droits propriétaire ou *root*. En outre, seuls les groupes auxquels vous appartenez vous-même sont disponibles pour la sélection. *chgrp* est utilisé selon la syntaxe suivante :

chgrp [OPTIONS] GROUPE FICHIER

ou :

chgrp [OPTIONS] GROUPE REPERTOIRE

L'option `-R` inclut les sous-dossiers et les fichiers contenus dans un répertoire.

### 4. *chown*

Gérer les droits de propriété. Habituellement, le créateur d'un fichier ou d'un répertoire est automatiquement son propriétaire (owner). La commande *chown* signifie change owner et permet donc de modifier les paramètres propriétaires. La commande est utilisée selon le schéma suivant :

chown [OPTIONS] [UTILISATEUR][:[GROUPE]] FICHIER

ou :

chown [OPTIONS] [UTILISATEUR][:[GROUPE]] REPERTOIRE

Il y a quatre combinaisons possibles pour définir les droits propriétaire pour les utilisateurs ou les groupes.

Il est possible de réinitialiser le propriétaire et le groupe selon les spécifications suivantes :

*chown [OPTIONS] proprietairenom:groupe\$nom fichier.txt*. Pour réinitialiser le groupe (l'utilisateur reste inchangé), on suivra :

*chown [OPTIONS] :groupesnom fichier.txt*. Pour réinitialiser le propriétaire (le groupe reste inchangé), on suivra :

*chown [OPTIONS] propriétairenom fichier.txt*

Enfin ci-dessous, l'utilisateur est réinitialisé en fonction de la spécification. Le groupe est défini sur le groupe par défaut de l'utilisateur connecté :

*chown [OPTIONS] propriétairenom: fichier.txt*

Les modifications peuvent être étendues aux sous-répertoires en utilisant l'option -R. Afficher les attributs de fichier

#### 5. **lsattr**

Si vous voulez afficher les attributs qui ont été définis pour un fichier ou un répertoire, utilisez la commande lsattr (abréviation de list attributes) selon le schéma suivant :

lsattr [OPTIONS] FICHIER/REPERTOIRE

## 7.3 Options de recherche

Linux offre diverses commandes avec lesquelles il est possible d'effectuer des recherches directement depuis le terminal.

#### 1. **find**

Effectuer une recherche dans le système de fichier. *find* est un programme en ligne de commande utilisé pour rechercher des fichiers. L'appel de programme est basé sur la syntaxe suivante :

find [OPTIONS] [REPERTOIRE]  
[CONDITIONDERECHERCHE][ACTIONS]

Le répertoire spécifié est le répertoire de départ de la recherche. La recherche est alors effectuée dans ce répertoire ainsi que ses sous-répertoires. Si vous ne spécifiez pas de répertoire, *find* lance la recherche à partir de votre répertoire courant.

Les options vous permettent de définir des critères de recherche et des actions. L'action par défaut est -print. Les critères de recherche courants sont le nom du fichier (-name NOMFICHIER[SUFFIXE]), un nom d'utilisateur (-user NOMUTILISATEUR), la taille du fichier (-size n[cwbkMG]), le jour de l'accès au fichier (-atime [+ -]n) ou le jour des modifications effectuées (-mtime [+ -]n).

Les méta-caractères et caractères de remplacement peuvent être utilisés lors de la recherche de noms de fichiers. Mettez-les entre guillemets pour empêcher qu'ils ne soient interprétés par le Shell.

Exemple :

find /tmp -name "\*.odt" -mtime -3 -size +20k

Le répertoire de départ est `/tmp`. Le programme en ligne de commande `find` émet sur la sortie standard tous les fichiers avec l'extension `.odt` qui sont plus grands que 20k et qui ont été modifiés pour la dernière fois il y a moins de 3 jours. Pour plus d'options sur la commande `find`, reportez-vous à la page d'aide du programme.

## 2. *locate*

Recherche dans l'index des fichiers. Le programme en ligne de commande *locate* permet également de rechercher des fichiers via le terminal. Contrairement à *find*, cependant, ce n'est pas le répertoire de fichiers qui est fouillé, mais plutôt une base de données dédiée à cette fin et régulièrement mise à jour. Par conséquent, *locate* permet d'obtenir des résultats de recherche beaucoup plus rapidement que *find*.

Pour rechercher un fichier spécifique dans la base de données, *locate* est appelé selon le schéma suivant :

```
locate MODELEDERECHERCHE
```

Le modèle de recherche peut contenir des méta-caractères tels que des caractères de remplacement (\*). Mettez-les entre guillemets pour empêcher une interprétation par le Shell. Dans l'exemple suivant, *locate* émet tous les fichiers avec l'extension *.png*.

```
locate "*.png"
```

La commande *locate* est sensible à la casse. Pour ignorer les différences de majuscules et minuscules dans la recherche, utilisez l'option `-i`. Le fichier `/var/lib/locatedb` sert de base de données pour l'indexation des fichiers. Cette dernière contient une liste de tous les fichiers du système de fichiers à un moment donné et doit être mis à jour régulièrement. Pour ce faire, utilisez la commande *updatedb*.

3. ***updatedb*** Mettre à jour l'index des fichiers. Une recherche via *locate* ne fonctionnera de manière fiable que si le fichier `/var/lib/locatedb` est continuellement tenu à jour. La commande *updatedb* permet donc de mettre à jour la base de données manuellement. Notez que vous avez besoin pour ce faire de disposer des droits root: *updatedb*

## 7.4 Archivage et compression

Linux offre diverses technologies avec lesquelles les fichiers peuvent être compressés et envoyés dans des archives. Veuillez noter que chaque archivage n'implique pas toujours une compression. Ainsi, le programme d'archivage de fichiers *tar* est généralement utilisé avec des programmes de compression tels que *gzip*, *bzip2* ou *xz*.

## 1. *tar*

Écrire et extraire des fichiers dans les archives *tar*. La commande *tar* signifie tape archiver, un programme développé à l'origine pour sauvegarder les données sur des lecteurs de bandes. Aujourd'hui encore, *tar* est l'un des programmes les plus populaires pour l'archivage de données sous Linux.

Le programme permet d'écrire séquentiellement différents fichiers et répertoires dans un fichier *tar* et de les restaurer si nécessaire. Contrairement au format zip couramment utilisé sous Windows, tous les droits d'utilisateur du fichier archivé sont conservés même après que ce dernier est décompressé.

Le programme en ligne de commande *tar* est appelé selon la syntaxe suivante :

```
tar [OPTIONS] FICHIERS
```

Si vous voulez créer une nouvelle archive, utilisez *tar* avec les options *-c* (create new archive) et *-f* (write archive or read from specified file). Dans l'exemple suivant, fichier1.txt et fichier2.txt sont écrits dans l'archive exemple.tar qui vient d'être créée.

```
tar -cf exemple.tar fichier1.txt fichier2.txt
```

Pour visualiser le contenu d'une archive, utilisez *tar* avec les options *-t* (afficher le contenu d'une archive), *-v* (sortie détaillée) et *-f* (voir ci-dessus).

```
tar -tvf exemple.tar
```

Si les fichiers archivés doivent être décompressés dans le dossier courant, *tar* est utilisé avec les options *-x* (extraire les fichiers de l'archive) et *-f* (voir ci-dessus).

```
tar -xf exemple.tar
```

Avec *-j* (bzip2), *-J* (xz), *-z* (gzip) et *-Z* (compress), *tar* offre également des options qui permettent de compresser ou décompresser des archives en appelant un autre programme pendant le processus.

Dans l'exemple suivant, fichier1.txt et fichier2.txt sont archivés dans exemple.tar.gz et compressés avec gzip.

```
tar -czf exemple.tar.gz fichier1.txt fichier2.txt
```

La commande suivante extrait et décompresse tous les fichiers archivés dans exemple.tar.gz.

```
tar -xzf exemple.tar.gz
```

## 2. *gzip* / *gunzip*

Compresser ou dézipper des fichiers avec *gzip*. *gzip* (abréviation de GNU zip) est un programme qui facilite la compression et la décompression des fichiers. La syntaxe générale qui s'applique est :

```
gzip [OPTIONS] FICHIER(S)
```

Par exemple, utilisez *gzip* comme suit pour convertir le fichier *exemple.txt* au format compressé *exemple.txt.gz* :

```
gzip exemple.txt
```

Notez que *gzip* supprime le fichier d'application original dans le cadre du processus de dézippage par défaut. Ceci peut être modifié avec l'option *-k*.

```
gzip -k exemple.txt
```

Si nécessaire, le programme peut être appliqué à plusieurs fichiers en même temps. Chaque fichier source est converti en un fichier *gz* indépendant.

La commande

```
gzip exemple1.txt exemple2.txt exemple3.txt
```

génère les fichiers *exemple1.txt.gz*, *exemple2.txt.gz* et *exemple3.txt.gz*. Si vous voulez écrire plusieurs fichiers dans une archive compressée commune, utilisez *gzip* avec le programme d'archivage *tar*. Pour décompresser un fichier *gz*, utilisez la commande *gzip* avec l'option *-d*. Autrement, vous pouvez utiliser la commande *gunzip*.

```
gzip -d exemple.txt.gz gunzip exemple.txt.gz
```

Les fichiers *gz* décompressés sont également supprimés par défaut. Si vous voulez conserver le fichier *gz* en plus du fichier extrait, utilisez l'option *-k*.

La compression *gzip* est basée sur l'algorithme Deflate (une combinaison d'encodage Huffman et LZ77). Par rapport à d'autres méthodes de compression, *gzip* se caractérise par sa rapidité. Cependant, le niveau de compression est relativement faible.

Préparation de l'environnement de travail de HPC

## 8 Introduction

Un environnement de développement intégré, ou IDE, est un logiciel de création d'applications, qui rassemble des outils de développement fréquemment utilisés dans une seule interface utilisateur graphique (GUI). Un IDE se compose habituellement des éléments suivants :

- Editeur de code source: un éditeur de texte qui aide à la rédaction du code logiciel, avec des fonctions telles que la coloration syntaxique avec repères visuels, la saisie automatique en fonction du langage et la vérification de bogues dans le code pendant la rédaction,
- Utilitaire d'automatisation de version locale: des utilitaires qui permettent d'automatiser des tâches simples et reproductibles lors de la création d'une version locale du logiciel à destination du développeur lui-même, par exemple la compilation du code source en code binaire, la mise en paquet du code binaire et l'exécution de tests automatisés,
- Débogueur: un programme qui permet de tester d'autres programmes en affichant l'emplacement des bogues dans le code d'origine.

Dans un environnement de développement intégré, les divers utilitaires ne nécessitent en général aucun réglage ni aucune intégration lors du processus de configuration. Les développeurs peuvent ainsi lancer rapidement le développement de nouvelles applications. Ils ne perdent pas non plus de temps à prendre en main chaque outil individuellement, car tous les utilitaires sont réunis dans une interface unique. Il s'agit d'un avantage non négligeable pour les nouveaux développeurs qui peuvent s'appuyer sur un IDE pour apprendre à utiliser les outils et les workflows standard de l'équipe qu'ils viennent d'intégrer. La plupart des fonctions d'un IDE font en fait gagner du temps, à l'instar de la saisie intelligente du code et de la génération automatique du code, qui permettent d'éviter la saisie manuelle de chaînes complètes de caractères. D'autres fonctions fréquemment disponibles dans les IDE aident les développeurs à organiser leur workflow et à résoudre les problèmes

## 9 Installation de l'environnement Eclipse

### 9.1 Introduction

Eclipse IDE est un environnement de développement intégré libre (le terme Eclipse désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse IDE est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT, d'IBM), et ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire des extensions.

Eclipse (logiciel) - Définition Source: Wikipédia sous licence CC-BY-SA 3.0. La liste des auteurs est disponible ici. AddThis Sharing Buttons Share to Facebook Share to TwitterShare to WhatsAppShare to SkypeShare to PinterestShare to Email AppShare to Plus d'options... Eclipse Développeur Eclipse Foundation Dernière version 3.3.1.1 - Europa (le 23 octobre 2007) Version avancée 3.4 M3 - Ganymede (le 1er novembre 2007) Environnement Multiplate-forme Langue Multilingue Type IDE Licence EPL Site Web [www.eclipse.org](http://www.eclipse.org)

Eclipse IDE est un environnement de développement intégré libre (le terme Eclipse désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse IDE est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT, d'IBM), et ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire des extensions.

La spécificité d'Eclipse IDE vient du fait de son architecture totalement développée autour de la notion de plug-in (en conformité avec la norme OSGi) : toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in. Plusieurs logiciels commerciaux sont basés sur ce logiciel libre, comme par exemple IBM Lotus Notes 8, IBM Symphony ou Websphere Studio Application Developer.

## 9.2 Installation

Pour commencer, il faut taper dans le terminal ***sudo apt-get install openjdk-7-jdk*** et passer à son installation. Une fois que cette installation est réussie, rendez-vous dans votre navigateur et entrer dans le site ***www.eclipse.org***. Une fois dans le site d'éclipse, cliquer sur l'onglet downloads et télécharger la version d'éclipse correspondant à votre système. Dès que le téléchargement est terminé, ouvre votre dossier téléchargement et décompresser le fichier. Après cette opération, rentrer encore dans le terminal, conduisez-vous dans le téléchargement par ***cd Téléchargement*** (qui est le dossier dans lequel se trouve le dossier eclipse dezipper), écrivez ensuite ***sudo mv eclipse/usr/lib***, qui permettra de déplacer eclipse dans le dossier application. Taper ensuite ***cd/usr/local/bin*** et par après ***sudo ln -s/usr/lib/eclipse/eclipse***, ces deux commandes vont permettre de créer un lien vers eclipse. Pour lancer eclipse il suffira maintenant d'écrire ***eclipse*** dans votre terminal.

Une fois que eclipse est lancé, clique sur la case utiliser ce répertoire par défaut dans la boîte de dialogue qui apparaîtra et l'éclipse va se lancer.

Introduction au projet de programmation de HPC

## 10 Structure de Projet: programmation modulaire

The main idea about project structure is that you have at least 2 folders include and src. Folders purpose is:

- include - PUBLIC header files (.h files).
- src - PRIVATE source files (.h and .m files).
- test - tests files if you write tests (indefinitely you should).
- libs - third party or your own libraries you depend on.

Headers files in include should be under folder named after your library domain. Reason behind this is that when you expose public header files you expose only include directory and when you *#include* files from library you do this *#include < HMM/Algorithm.h >* instead of *#include "Algorithm.h"* if it was in root of include.

The main idea about project structure is that you have at least 2 folders include and src. Folders purpose is:

- include - PUBLIC header files (.h files).
- src - PRIVATE source files (.h and .m files).
- test - tests files if you write tests (indefinitely you should).
- libs - third party or your own libraries you depend on.

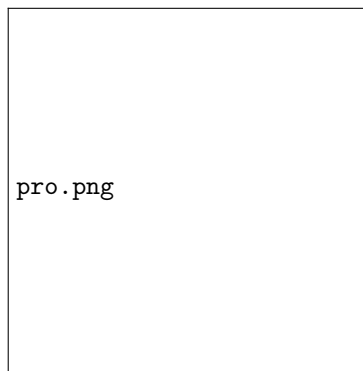


Figure 17: La structure d'un projet

## 11 Compilation et Structure du fichier Make-File (relation entre structure du projet et makefile)

Dans cette partie nous allons essayer de voir comment réaliser et utiliser un Make file.

Un Make file est un fichier qui sert aux gens qui utilisent les langages compilé comme C ou C++, Fortran, etc. de le faciliter au développement de leurs projets, notamment des projets où ils organisent le tout sur plusieurs fichiers et font de la programmation modulaire.

Au cours de cette partie, nous vous fournirons un apprentissage avec C, où nous aurons à travailler avec trois fichiers (dont deux fichiers sources et un



fichier d'en tête) et un outil Make comme nous l'avions déjà dit dans la partie introductive de ce chapitre. Pour y arriver, premièrement, nous allons montrer comment télécharger et l'installer l'outil make sur votre ordinateur et en suite nous passerons à une application.

## 11.1 Téléchargement de l'outil Makefile

L'outil Makefile c'est un produit de GNU, pour ce qui sont dans MacOS ou Linux, logiquement sont de la famille Os, ont normalement un outil Makefile intégré par défaut dans leur système et vous pouvez le vérifier en tapant dans la ligne de commande **make**. S'il est installé; il y aura un petit message vous disant: **\*\*\*pas de cibles spécifiques et aucun makefile n'a été trouvé. Arrêt**", et par défaut de ce message donc il n'est pas installé.

Pour installer l'outil Makefile, vous pouvez taper dans la ligne de commande **sudo apt install make**. Mais pour ce qui sont sur Windows peuvent le télécharger à partir de: <http://gnuwin32.sourceforge.net/packages/make.htm>. à partir de ce lien, vous allez récupérer le setup program et qui va directement vous proposer le téléchargement.

## 11.2 Exemple d'application

Pour commencer, on procède par la création d'un fichier Make (*c'est un fichier sans extension*) dans le même répertoire où se trouvent les fichiers sources, et vous le renommez **Makefile**. Ce fichier qui peut être ouvrable par n'importe quel éditeur de texte, il va se charger de la génération du ou des vos exécutables et également de tout ce qui est comme compilation. Voici l'exemple de nos deux fichiers sources que et du fichier d'en tête:

- Premier fichier source: `player.c`

```
include "player.h"
include <stdio.h>
include <string.h>

Player create(void)
{
    Player p;
    strcpy(p.name, "Unknown");
    p.level = 1;
    return p;
}

void say(Player p, char *message)
{
    printf("

```

- Deuxième fichier source: `main.c`

```
include "player.h"
include <stdio.h>

int main(void)
{
    Player p1 = create();
    say(p1, "Bonjour");
    return 0;
}

```

- Fichier d'en tête: `player.h`

```
#ifndef _PLAYER_H_
#define _PLAYER_H_

typedef struct Player
{
    char name[26];
    int level;
} Player;

Player create(void);
void say(Player p, char *message);

#endif

```

Sans Makefile, pour compiler ces programmes, on devrait utiliser la commande **`gcc *.c -o prog`** (avec prog, le nom de l'objet exécutable trouvé après compilation) et avec la commande **`./prog`** on parviendrait à lancer l'exécutable. Supposons que nous voulons modifier le fichier source main.c, pour avoir l'exécutable qui est actualisé, il nous est demandé de recompiler encore tout le projet, tandis que la mise à jour n'a concerné qu'un seul fichier; ce qui consomme beaucoup de temps en terme de compilation.

L'avantage de Makefile, est qu'on doit pouvoir indiquer toutes ces instructions des compilations étape par étape dans les détails et ainsi de gérer les dépendances c'est-à-dire que le Makefile sera capable simplement de régénérer les fichiers qui ont été mis en jours et de recréer l'exécutable final. Par cette opération, nous aurons à gagner en temps et en compilation. C'est pourquoi lorsqu'on utilise un Makefile, on n'utilise plus les même syntaxes montre ci-haut, on aura à passer par un fichier, est celui qui aura à compiler par une simple commande à taper et à gérer toutes les autres étapes intermédiaires de la compilation.

### 11.3 Rédaction d'un Makefile

Un Makefile est un fichier qui est constitué par:

- **une ou plusieurs dépendances:** sont des fichiers sources,
- **une ou plusieurs cibles:** qui sont des fichiers objets, obtenus après compilation des fichiers sources,
- **une commande:** qui met en relation la cible à sa dépendance et vice versa. Elle représente la syntaxe de compilation pour avoir le fichier objet.

Pour compiler le fichier Makefile, vous ouvrez le terminal dans le repertoire où se trouve votre fichier et en suite dans la ligne des commandes vous tapez **`make`** et il va assurer sa tâche. Après compilation, vous trouverez que vous obtiendrez tous les objets de vos fichiers ainsi que l'exécutable de votre projet.    prog : main.o player.o gcc -o prog main.o player.o

main.o : main.c gcc -o main.o -c main.c

player.o : player.c gcc -o player.o -c player.c    Ceci est le premier pas dans Makefile: relation entre dépendances, cibles et commandes. Prog ici est le nom de notre exécutable qui tient compte des fichiers sources.

Une fois que vous avez déjà votre fichier Makefile, si vous voulez actualiser ou modifier l'un de vos fichier d'en tête ou sources, vous trouverez lors de la compilation que seul le fichier modifier qui sera compiler pour apporter la modification au fichier exécutable.

L'exemple à la figure 5.3 n'est qu'une base d'un fichier Makefile, imaginez un projet avec plusieurs fichiers sources, le fait de correspondre chaque cible à son objet causera une perte de temps et c'est ce que Makefile veut corriger. Pour palier à ce problème, nous aurons à introduire dans le fichier Makefile ci-haut

des variables.

Les variables seront en majuscules et les valeurs en minuscules, ce la vous permettra qu'un jour si vous voulez changer l'une de valeur de ne plus parcourir tout le projet pour le faire. Dans le Makefile la variable sera noté par `$(CC)`  
`CC= gcc EXEC= prog SRC= (wildcard *.c)OBJ=(SRC: .c=.o)`

all: `(EXEC)(CC) -o @ -c (EXEC) : (OBJ) (CC) -o@` Ceci est le code du fichier Makefile dynamique en remplaçant les valeurs par les variables et en suivant quelques règles du make

## 11.4 Quelques règles et variables spécifiques

- **All**: dépend de l'exécutable, il devient comme un pointeur sur l'exécutable.
- **clean**: elle sert à nettoyer les objets et les exécutables du projet pour permettre le partage du projet avec des fichiers sources, d'en tête et le Makefile, car ces derniers pèsent moins. La variable *clean* n'a pas de valeur, mais elle est accompagnée par une commande, indiquant les éléments à supprimer, par exemple: `rm -ff *.o`
- **mrproper**: c'est une dépendance qui a comme cible *clean*. Sa commande est `rm -rf prog`. Avec prog le nom de l'exécutable. Une fois qu'on a utiliser cette commande dans le Makefile, la compilation utilisera le syntaxe **make mrproper**.
- **zip**: si vous voulez avoir un fichier zipper à la fin de votre projet.
- **\$@**: c'est une variable spécifique qui prend le nom de la cible (source),
- **\$<**: qui prend en compte les dépendances,
- **\$:** liste de dépendance
- **\$\***: il prend le nom du fichier

Le symbole pourcent point(%) devant o ou c par exemple, permet de prendre en charge tous les fichiers .o et tous les fichiers .c dans votre dossier. Pour les sources cette commande peut être remplacée par la commande `SRC = $(wildcard *.c)`.  
Introduction aux techniques de programmation parallèle Le parallélisme consiste à mettre en œuvre des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci.

## 12 But et domaines cibles

Le parallélisme ou les techniques y relatif, consiste à réaliser le plus grand nombre d'opérations en un temps le plus petit possible.

Le parallélisme a été conçu à des fins scientifiques. certains types de calculs se prêtent particulièrement bien à la parallélisation,comme:

- Dynamique des fluides,
- les Prédictiones météorologiques,
- la modélisation et simulation de problème de dimensions plus grandes,
- le traitement de l'informations,...

## 13 Type de parallélisme

Avant de parler de type de parallélisme, parlons un peu de la taxonomie de Flynn. La taxonomie de Flynn est une classification des architectures d'ordinateur, proposée par Michael Flynn en 1966. Les quatre catégories définies par Flynn sont classées selon le type d'organisation du **flux de données** et du **flux d'instructions**. la taxonomie de Flynn classe les machines en quatre catégories suivant leurs architectures;

### 1. Les machines à systèmes séquentiels:

Il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme, tant au niveau des instructions qu'au niveau de la mémoire. Cette catégorie correspond à l'architecture de von Neumann.

L'architecture dite architecture de von Neumann est un modèle pour un ordinateur qui utilise une structure de stockage unique pour conserver à la fois les instructions et les données demandées ou produites par le calcul. De telles machines sont aussi connues sous le nom d'ordinateur à programme enregistré. La séparation entre le stockage et le processeur est implicite dans ce modèle. ces machines utilisent une architecture **SISD** (Single Instruction,Single Data);

### 2. les machines traitant une grande quantité de données de manière uniforme:

Il s'agit d'un ordinateur qui utilise le parallélisme au niveau de la mémoire, par exemple le processeur vectoriel. Un processeur vectoriel est un processeur possédant diverses fonctionnalités architecturales lui permettant d'améliorer l'exécution de programmes utilisant massivement des tableaux, des matrices, et qui permet de profiter du parallélisme inhérent à l'usage de ces derniers.

Développé pour des applications scientifiques et exploité par les machines Cray et les supercalculateurs qui lui feront suite, ce type d'architecture a rapidement montré ses avantages pour des applications grand public (on peut citer la manipulation d'images). Elle est implémentée en partie dans les processeurs grand public par des instructions **SIMD** (Single Instruction,Multiple Data), soit grâce à une unité de calcul vectoriel dédiée (Altivec), soit simulée par des instructions bas niveau de type vectoriel (MMX/SSE). Contrairement au SIMD de type MMX, où il faut charger les vecteurs dans les registres en plusieurs opérations puis exécuter

une opération supplémentaire sur ses registres, dans une unité vectorielle on charge l'adresse d'une table de vecteurs, la largeur des vecteurs ou du pas et la longueur de la table à traiter par instruction dans un registre, l'instruction vectorielle enchaîne ensuite son calcul sur l'ensemble des vecteurs de cette table.

**3. Les machines utilisant plusieurs processeurs ou un processeur multi-cœur:**

Sont des machines utilisant une architecture MIMD (Multiple Instruction, Multiple Data) Dans ce cas, plusieurs unités de calcul traitent des données différentes, car chacune d'elles possède une mémoire distincte. Il s'agit de l'architecture parallèle la plus utilisée, dont les deux principales variantes rencontrées sont les suivantes :

- **MIMD à mémoire partagée**

Les unités de calcul ont accès à la mémoire comme un espace d'adressage global. Tout changement dans une case mémoire est vu par les autres unités de calcul. La communication entre les unités de calcul est effectuée via la mémoire globale.

- **MIMD à mémoire distribuée**

Chaque unité de calcul possède sa propre mémoire et son propre système d'exploitation. Ce second cas de figure nécessite un middleware pour la synchronisation et la communication.

Un système MIMD hybride est l'architecture la plus utilisée par les superordinateurs. Ces systèmes hybrides possèdent l'avantage d'être très extensibles, performants et à faible coût.

**4. des machines à réseaux systoliques**

Sont des machines à architectures MISD (Multiple Instruction, Single Data). Il s'agit d'un ordinateur dans lequel une même donnée est traitée par plusieurs unités de calcul en parallèle. Il existe peu d'implémentations en pratique. Cette catégorie peut être utilisée dans le filtrage numérique et la vérification de redondance dans les systèmes critiques.

Selon David Patterson et John Hennessy, "certaines machines sont des hybrides de ces catégories, bien sûr, mais cette classification traditionnelle a survécu parce qu'elle est simple, facile à comprendre et donne une bonne approximation. Avec cette approche, nous pouvons dire qu'il existe deux types de parallélisme:

1. **le parallélisme par flot d'instructions, également nommé, parallélisme de traitement ou de contrôle:**

c'est un parallélisme qui correspond au MIMD

2. **le parallélisme de données, où les mêmes opérations sont répétées sur des données différentes, le SIMD**

Des langages de programmation concurrente, des interfaces de programmation spécialisées et des algorithmes modèles ont été créés pour faciliter l'écriture

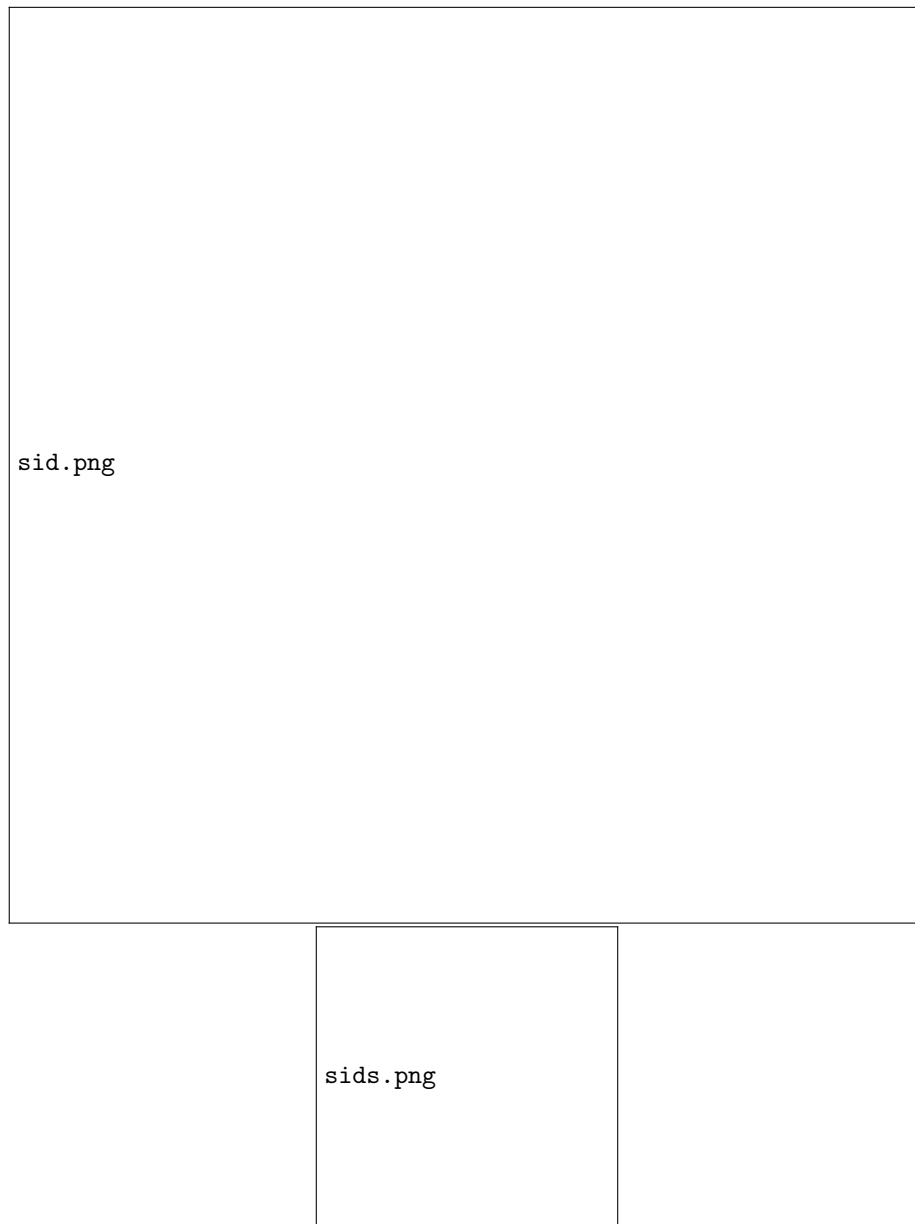


Figure 18: Catégories de la taxonomie de Flynn.  
*PU* : unité de calcul (d'un processeur unicœur ou multicœur). *Instruction Pool*  
: flux d'instructions. *Data Pool* : flux de données

de logiciels parallèles et résoudre le problème de traiter des informations sur des processeurs indépendants. La plupart des problèmes de concurrence sont des variantes des problèmes rencontrés dans les modèles appelés producteur-consommateur, lecteur-rédacteur ou Dîner des philosophes. Le choix du mécanisme d'échange de données dépend aussi du type d'architecture à laquelle le langage, l'interface ou l'algorithme est destiné.

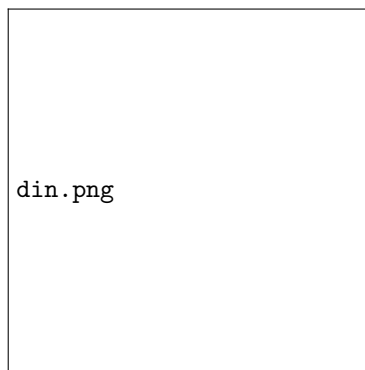


Figure 19: Illustration au problème dû au dîner de philosophes

Un programme destiné à s'exécuter sur une architecture à mémoire partagée ne serait pas identique à celui destiné à une architecture à mémoire distribuée ou à une architecture mixte. Les programmes s'exécutant sur les architectures partagées vont pouvoir simplement partager des zones mémoires, alors que les systèmes à mémoire distribuée vont devoir échanger des messages via un bus ou un réseau. Vu la nécessité de faire la programmation avec openMP ou MPI.

## 14 Programmation avec openMP

OpenMP est depuis 2010 parmi les bibliothèques spécialisées les plus utilisées pour une exécution en mémoire partagée. Cette bibliothèque est supportée par plusieurs langages (C, C++ et Fortran) disponible sur plusieurs plate-formes (Linux, Windows, OSX,...); openMP regroupe des directives de compilation et des fonctions. Le compilateur gcc supporte openMP depuis la version 4.2, en ajoutant simplement une option sur la ligne de commande et en incluant le fichier d'en-têtes **omp.h**. La gestion des différentes fonctions est assurée par la librairie **libgomp**. Avant d'entrer dans le vif du sujet, je tiens à préciser que, hormis une connaissance des bases du C++, aucune connaissance de l'algorithmique distribuée n'est requise pour aborder cet article.

## 14.1 Algorithme séquentiel vs. Algorithme parallèle

Un algorithme est une méthode de résolution d'un problème par la description finie d'une séquence d'opérations (qui elle peut être infinie). Par exemple, le matin je me lève et je te bouscule, tu ne te réveilles pas (comme d'habitude). Lorsque la séquence est représentable par une suite d'états totalement ordonnée, alors l'algorithme est dit séquentiel. Lorsque la suite d'états n'est que partiellement ordonnée (au moins deux actions se déroulent en parallèle) alors on parlera d'algorithme parallèle. Par exemple, le matin le réveil se déclenche. D'un côté ma femme se lève puis me bouscule, d'un autre côté, le réveil diffuse une musique trop forte et indépendamment les volets s'ouvrent automatiquement éclairant la pièce d'une lumière agressive. Au final, je ne me réveille toujours pas, comme d'habitude.

En fine, un algorithme parallèle peut être vu comme plusieurs sous-algorithmes séquentiels qui se déroulent en parallèle. On distinguera cependant les communications entre les sous-algorithmes, selon qu'ils se partagent une même zone d'informations (et potentiellement y accèdent de manière concurrente) ou bien qu'ils utilisent un vecteur de communication dédié pour se partager certaines informations. Dans le premier cas, on parle de parallélisme léger (**multithreading** en anglais) tandis que dans le second cas de parallélisme lourd (**multiprocessing** en anglais).

**NB** : Tous les algorithmes ne sont pas toujours efficaces, ce n'est pas parce que l'algorithme est parallèle qu'il est nécessairement plus efficace.

### 14.1.1 Parallélisme léger

Le parallélisme léger traduit le fait que des instructions s'exécutent en parallèle tout en se partageant une même zone mémoire. Cela signifie que si plusieurs blocs d'instructions  $B_1, B_2, \dots, B_n$  s'exécutent en parallèle, rien n'empêche par défaut les blocs d'écrire ou lire l'état d'une variable simultanément, pouvant aboutir à une situation imprévisible et chaotique.

Ainsi sont apparues les fameux **MutEx**, variables particulières dont le rôle est d'obtenir un verrou exclusif (Mutual Exclusion) sur la mémoire partagée. Pour faire une analogie, imaginons plusieurs voitures (nos processus parallèles) circulant sur un réseau routier (zone de mémoire partagée). Tant que chaque voiture ne rencontre pas les autres, il n'y a pas de problème. Si par contre, deux voitures empruntent la même route, sur la même voie, les problèmes peuvent survenir assez rapidement (surtout si les véhicules roulent à contre sens l'un de l'autre). Dès lors, il est prudent avant de s'engager sur cette route que chacune des voitures puisse s'assurer un usage exclusif de celle-ci. Cela passe par une demande d'autorisation pour emprunter ladite route à un agent de la sécurité routière (le système d'exploitation). Si la route est libre, l'agent autorise la voiture qui a fait la demande et interdit à tout autre véhicule de l'emprunter. Une fois que la voiture qui s'est engagée a quitté la portion de route dangereuse, il avertit l'agent que la route est à nouveau libre. A contrario, si lorsqu'un



véhicule effectue la demande d'accès, la route est déjà utilisée par une autre voiture, alors l'agent met le demandeur en attente jusqu'à ce que la route soit libérée. Bien évidemment, la gestion des Mutex est un sujet important. En effet, il faut tout d'abord s'assurer qu'une fois la tâche effectuée, le Mutex est bien relâché. Dans le cas contraire, le risque est de bloquer le système, certains processus demeurant en attente d'obtention du Mutex. En outre, les Mutex créent des goulots d'étranglement, il faut donc les demander le plus tard possible et les relâcher dès que possible. Pour reprendre l'analogie avec les voitures, si je demande l'autorisation exclusive de circuler sur la route menant de mon travail à mon domicile (plusieurs km) alors que la zone de danger ne représente que quelques mètres, je bloque inutilement tout le monde (ce n'est pas très grave, au moins je suis tranquille sur ma route, par contre si quelqu'un faisait ça, je le gratifierai de plusieurs noms d'oiseaux). Bien évidemment, le fait de demander un Mutex a un coût, donc il est important de veiller à regrouper les instructions sensibles dans un même bloc lorsque cela est possible.

#### **14.1.2 Parallélisme lourd**

Le parallélisme lourd peut s'apparenter quant à lui au déroulement de plusieurs algorithmes distincts en simultané. Chacun des algorithmes ayant un objectif spécifique et plus ou moins indépendant des autres. Les algorithmes peuvent cependant avoir besoin d'échanger quelques informations, auquel cas un protocole de communication doit être mis en œuvre. Par exemple, pour modéliser un serveur web, il faut un algorithme principal qui écoute sur un port donné jusqu'à ce qu'un client se connecte. Quand un client se connecte un nouvel algorithme prend en charge le client et l'algorithme principal se remet tout simplement en écoute d'un nouveau client. Chaque prise en charge d'un client est indépendante.

Dans certains cas plus complexes, il faut tout de même que les différents processus communiquent un minimum sur leur état ou bien échangent ponctuellement des données. Cela peut s'opérer par l'utilisation de fichiers (type fifo) lorsque les processus s'exécutent sur la même machine (ou disposent d'un espace de stockage commun (e.g., montage NFS). Dans un cas plus général, cela peut passer par une communication réseau (e.g., protocoles UDP ou TCP).

#### **14.1.3 Parallélisme léger + parallélisme lourd**

Et non, parallélisme léger + parallélisme lourd, ça ne fait pas du parallélisme moyen, mais c'était bien tenté ! On parle dans ce cas-là de parallélisme hybride. L'idée est de tirer parti des deux paradigmes et c'est ce qu'on va faire dans la suite donc je n'en écris pas plus pour le moment.

#### **14.1.4 Quand/Comment/Pourquoi paralléliser un algorithme ?**

Dans certains cas, la conception d'algorithmes parallèles découle du bon sens et s'avère plus simple à mettre en œuvre qu'en séquentiel (e.g., mise en place d'un

serveur web). Dans d'autres cas, paralléliser les traitements n'est pas toujours intuitif. Cependant, avec l'arrivée, cette dernière décennie, d'un volume de données à traiter et analyser toujours plus important, cela devient une nécessité. Dans cette partie, nous présenterons deux standards permettant de faire l'un du parallélisme léger et l'autre du parallélisme lourd. Le code fourni est écrit en C++, tout d'abord parce que cela nous semble cohérent d'utiliser l'un des langages les plus performants pour faire des calculs intensifs et d'autre part parce que le C++ dispose (par rapport au C) des facilités offertes par la STL (qui n'existe plus stricto sensu puisqu'elle a été intégrée au standard C++).

## 14.2 OpenMP

Il est à noter que les implémentations d'OpenMP existent pour les langages C/C++ et Fortran. Dans la suite de cette partie, nous n'évoquerons que l'API C/C++ d'OpenMP.

### 14.2.1 Principe de fonctionnement

Lorsque l'on envisage de paralléliser tout ou partie d'un algorithme, il faut identifier les blocs d'instructions qui sont complètement indépendants entre eux, ceux qui ne le sont que partiellement et ceux qui ne le sont pas du tout. Un exemple simple d'algorithme permettant de calculer la moyenne d'une série de 20 valeurs : Entrée : T un tableau de 20 entiers Sortie : M un réel 1) Pour chaque valeur V du tableau T 2)  $M = M / 20$  3) Afficher le résultat L'état de cet algorithme paraît difficilement parallélisable, et pourtant l'étape 1 peut être clairement parallélisée en se rappelant que la moyenne de 20 valeurs peut être obtenue en calculant indépendamment d'un côté la somme S1 des 10 premières valeurs et d'un autre côté la somme S2 des 10 dernières valeurs, puis en sommant S1 et S2. La somme S1 peut également se décomposer en sommant les 5 premières valeurs d'un côté (somme S1a) et les 5 suivantes de l'autre (somme S1b), et ainsi de suite. Cependant, l'implémentation d'un tel algorithme (appelé réduction) demeure fastidieuse alors que cette décomposition reste assez courante et s'adapte à d'autres opérations (trouver le maximum parmi un ensemble de valeurs, calculer leur produit...).

L'idée d'OpenMP (en tout cas, l'idée que moi j'en ai), c'est de faire ce travail pour vous en expliquant juste au compilateur que cette partie doit être parallélisée en faisant une réduction.

Pour cela, OpenMP définit d'une part des directives et schémas de constructions et d'autre part des fonctions. Les directives sont des pragmas qui sont analysés par le préprocesseur et qui réécrivent les blocs d'instruction concernés selon le schéma de construction spécifié par la directive. Les fonctions permettent d'écrire des instructions spécifiques lorsqu'aucun schéma de construction ne correspond.

Ainsi, l'algorithme de calcul de la moyenne précédent peut se réécrire :

Entree : T un tableau de 20 entiers Sortie : M un reel Utiliser le schema de construction OpenMP "reduction de somme sur la variable M"

1/ Pour chaque valeur V du tableau T

a/  $M = M + V$

2/  $M = M / 20$

3/ Afficher le resultat

### 14.2.2 Installation

Il faut installer la librairie de développement OpenMP pour le compilateur gcc/g++. Pour les chanceux qui utilisent une distribution Debian/Ubuntu, rien de plus simple : `sudo apt install libgomp1` Pour les autres distributions, il faudra chercher sur les forums dédiés à vos distributions. Si vous envisagez d'utiliser un autre compilateur, alors il faudra également chercher sur les forums dédiés.

Tout d'abord, notre premier exemple a pour objet de vérifier que le compilateur (ou plus précisément l'éditeur de lien) résout correctement la liaison avec la librairie OpenMP.

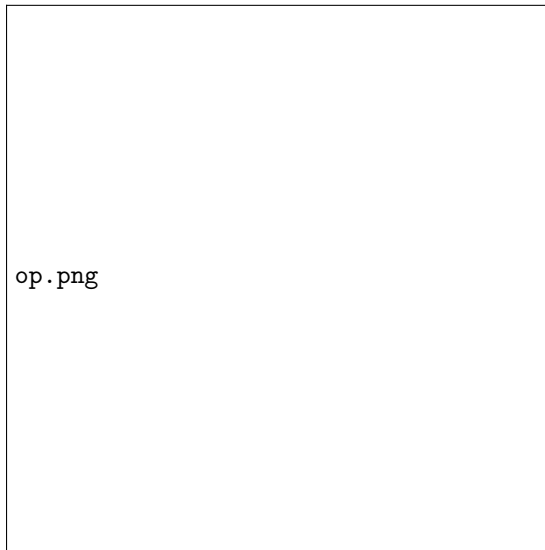
Commençons par créer un fichier `helloworld.cpp` assez rudimentaire :

```
include <iostream>
using namespace std;
int main(int argc, char** argv)
for (int i = 0; i < argc; ++i)
cout << "L'argument " << i << " a pour valeur"
<< " " << argv[i] << " " << endl;
return 0;
```

Compilons-le avec une ligne de commande tout aussi rudimentaire : `g++ -Wall -ansi -pedantic helloworld.cpp -fopenmp -o helloworld` Si tout se passe bien, vous n'avez aucun avertissement sur la sortie standard (Nous laissons le soin aux lecteurs dubitatifs d'aller regarder l'intérêt des options `-Wall`, `-ansi` et `-pedantic` qui ne sont pas l'objet de cette partie). Vous remarquerez l'option **-fopenmp** qui signifie tout simplement de gérer les directives OpenMP.

Exécutons ce programme avec quelques arguments : `./helloworld Premier test OpenMP pour "GNU/Linux Magazine"` et voyons ce que l'on obtient : Bon, c'était vraiment un exemple qui ne sert pas à grand-chose. Mais nous allons le modifier afin de paralléliser les itérations de la seule et unique boucle du programme. Après tout, chaque itération est indépendante et les zones mémoires accédées sont distinctes et peuvent donc être traitées en parallèle. Pour cela, on insère avant la boucle une directive OpenMP `pragma omp` indiquant d'utiliser le schéma de construction de boucle pour paralléliser (`parallel for`) :

```
include <iostream>
using namespace std;
int main(int argc, char** argv)
#ifdef OPENMP
cout << "Valeur de la macro OPENMP : " << OPENMP << " " <<
endl;
```



```
pragma omp parallel for
endif
for (int i = 0; i < argc; ++i)
#ifdef OPENMP
pragma omp critical
endif
cout << "L'argument " << i << " a pour valeur"
<< " " << argv[i] << " " << endl;
return 0;
```

Compilez ce code avec et sans l'option -fopenmp et vous verrez que vous n'aurez aucun avertissement.

### 14.2.3 Quelques fonctions OpenMP bien pratiques

Outre les directives, OpenMP propose également quelques fonctions assez utiles, telles que :

- `omp_get_max_threads()` qui renvoie le nombre maximum de processus légers qui pourront être déclenchés après l'appel de cette fonction ;
- `omp_get_num_procs()` qui renvoie le nombre de processeurs disponibles sur la machine au moment de l'appel ;
- `omp_get_num_threads()` qui renvoie le nombre de processus légers correspondants à un bloc d'instruction ;
- `omp_get_thread_num()` qui renvoie le numéro d'un processus léger dans un bloc parallèle.

Ces fonctions sont déclarées dans le fichier *omp.h* qu'il faudra bien évidemment inclure pour pouvoir les utiliser.

#### 14.2.4 Quelques schémas de construction bien pratiques

Nous allons revenir au premier algorithme permettant de calculer la moyenne de 20 valeurs, et bien évidemment ne pas commencer par se restreindre à exactement 20 valeurs.

Le programme prendra donc comme valeurs les arguments qui lui seront passés, les stockera dans un vecteur, calculera la moyenne puis l'affichera sur la sortie standard.

On commence par inclure les déclarations qui nous seront utiles :

```
include <iostream> include <cstdlib> // Pour atof() include <libgen.h> // Pour
basename() include <vector> include <omp.h> On déclare utiliser l'espace de
nommage standard (histoire de ne pas préfixer par std:: ce que l'on utilisera de
cet espace) : using namespace std; La fonction moyenne permet de prendre un
vecteur de réels double précision (paramètre passé en référence constante, car
on travaillera sur l'original plutôt qu'une copie et qu'on ne modifiera en aucun
cas les valeurs de ce vecteur). On récupère la taille du vecteur dans une variable
n, on initialise la somme à 0 dans la variable res, puis on lance le calcul de la
somme en appliquant le schéma de conception de boucle pour paralléliser avec
réduction de l'opérateur + sur la variable res. Dit autrement, chaque itération
sera traitée par un processus léger, qui travaillera sur une copie de la variable
res (qui vaut donc 0), puis les copies seront sommées par réduction (sommes
partielles et récurrences).
```

```
double moyenne(const vector<double> &v)
size_t n = v.size();
double res = 0;
#ifdef OPENMP
#pragma omp parallel for reduction(+: res)
#endif
for (size_t i = 0; i < n; ++i)
res += v[i];
return res / n;
```

Enfin, le programme principal s'assure dans un premier temps d'avoir au moins un argument (c'est toujours mieux pour calculer une moyenne). On crée un vecteur de réels **v** de la taille du nombre d'arguments passés en option (le **-1** provient du fait que le nom du programme n'est pas à prendre en compte dans le décompte).

La lecture des arguments est faite en parallèle, mais on limitera le nombre de processus simultanés à **3** (clause `num_threads(3)`). On en profite pour ajouter un peu d'infos de debuggage qui nous permettent d'utiliser deux des fonctions présentées précédemment.

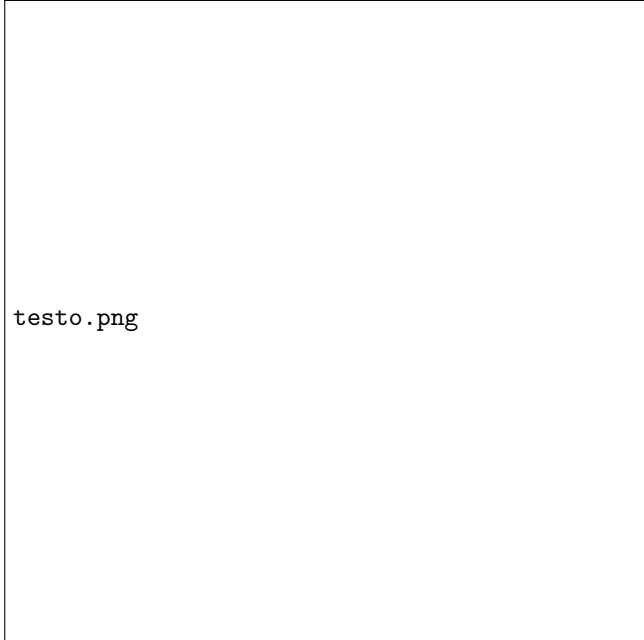
```
int main(int argc, char** argv)
if (argc < 2)
cerr << "usage: " << basename(argv[0]) << " v1 > ... < v_n > " << endl
```

```

    ii "Le programme calcule la moyenne des valeurs passees en argument"
    ii endl;
    return 1;
    vector<double> v(argc - 1);
    #ifdef OPENMP
    pragma omp parallel for num_threads(3)
    #endif
    for (int i = 1; i < argc; ++i)
        v[i - 1] = atof(argv[i]);
    #ifdef OPENMP
    pragma omp critical
    cout << "P[" << omp_get_thread_num()
    << " / " << omp_get_num_threads() << "]"
    << " argv[" << i << "]"
    << " = " << v[i - 1]
    << endl;
    #endif
    cout << "La moyenne des " << v.size()
    << " valeurs fournies en arguments est " << moyenne(v)
    << endl;
    return 0;

```

Pour changer, on compile et on teste avec quelques valeurs : C'est cohérent.



testo.png

Comme vous l'aurez compris, une directive OpenMP s'écrit sous forme d'un

pragma juste avant l’instruction concernée. Si on souhaite faire porter une directive sur un bloc d’instruction, il suffit d’encapsuler le bloc d’instruction entre accolades.

L’objet de cette partie n’est pas de détailler l’API complète d’OpenMP, mais d’illustrer son utilisation. Si vous avez compris le principe de fonctionnement, il ne vous reste qu’à lire la `summary card` disponible sur le site d’OpenMP pour avoir la liste exhaustive des directives, schémas de construction et fonctions disponibles.

## 15 MPI

Contrairement à la parallélisation légère, la parallélisation lourde implique surtout de prévoir un protocole de communication entre les différents processus. C’est ce que propose le standard MPI (Message Passing Interface)

### 15.1 Open MPI

Il existe plusieurs implémentations du standard MPI. Comme vous l’aurez compris, Open MPI est une implémentation libre et open source de ce protocole. Il en résulte qu’en théorie, nous parlerons tout autant de MPI que d’Open MPI (et que de toute autre implémentation) dans la suite. Écrire un programme avec Open MPI se résume à trois grandes étapes :

1. Initialisation de la communication entre les processus ;
2. Instructions à exécuter ;
3. Terminer la communication entre les processus.

Au sein d’un groupe de communication, certaines fonctions de MPI nécessitent que tous les processus du groupe communiquent ensemble. Ces routines sont dites collectives. A contrario, certaines fonctions peuvent s’exécuter indépendamment de tout ou partie des autres processus. Ces routines sont alors dites non collectives.

**NB** : Par défaut, MPI définit un groupe de communication universel regroupant tous les processus. Il est possible de créer d’autres communicateurs et ainsi de définir des sous-groupes de processus. C’est pourquoi MPI distingue les communications dites intracommunicantes (au sein d’un même groupe) et des communications dites intercommunicantes (entre des groupes distincts). Cette subtilité ne sera pas développée dans cette partie, c’était juste pour info.

Une des difficultés existantes que l’on rencontre fréquemment lorsque l’on établit des communications entre processus, c’est le blocage de la communication. Cela peut se produire si deux (ou plusieurs) processus parlent en même temps. MPI définit deux catégories de communications. Les communications synchrones (i.e., si Mickey parle à Dingo – Pluto c’est son chien – alors Dingo doit écouter

Mickey, et seulement ensuite Mickey peut s'occuper de Pluto) des communications asynchrones (i.e., si Mickey parle à Dingo, Dingo pourra écouter son message plus tard, cela n'empêche pas Mickey d'aller plus tôt s'occuper de Pluto).

### 15.1.1 Installation

Pour les moyennement chanceux qui utilisent une distribution Ubuntu, rien de plus simple pour avoir une version relativement obsolète : `sudo apt install libopenmpi-dev openmpi-bin openmpi-doc` Ceux qui sont sous Debian (sid) auront la chance d'avoir la dernière version. Ceci étant, à partir de la version 1.8.0 d'Open MPI, tout ce dont nous aurons besoin pour cet article est fonctionnel. Au moment de la rédaction de cet article, j'utilise la version 1.10.2 (package officiel de la distribution Ubuntu 16.04). Rien n'interdit bien évidemment de télécharger les sources, de les compiler et de les installer.

Pour les autres distributions, il faudra chercher sur les forums dédiés à vos distributions.

Si vous envisagez d'utiliser un autre compilateur, alors il faudra également chercher sur les forums dédiés. À noter également que depuis la version 2.0.0, les numéros majeurs et mineurs de version d'Open MPI correspondent aux spécifications de MPI. Enfin, les liaisons syntaxiques du C++ (bindings) sont déconseillées depuis la version 2.2 de la spécification (c'est dommage, j'aime bien ce sucre syntaxique) et ne seront plus implémentées pour les nouvelles fonctionnalités. Dans cet article, les codes Open MPI fournis utiliseront la syntaxe C++ quand celle-ci est disponible. Si vous installez vous-même Open MPI à partir des sources (récentes), pensez à passer l'option `--enable-mpi-cxx` au script `./configure`. Ceux qui sont sous Debian (sid) auront la chance d'avoir la dernière version. Ceci étant, à partir de la version 1.8.0 d'Open MPI, tout ce dont nous aurons besoin pour cette partie est fonctionnel. Au moment de la rédaction de cet article, nous utilisons la version 1.10.2 (package officiel de la distribution Ubuntu 16.04). Rien n'interdit bien évidemment de télécharger les sources, de les compiler et de les installer. Pour les autres distributions, il faudra chercher sur les forums dédiés à vos distributions. Si vous envisagez d'utiliser un autre compilateur, alors il faudra également chercher sur les forums dédiés.

À noter également que depuis la version 2.0.0, les numéros majeurs et mineurs de version d'Open MPI correspondent aux spécifications de MPI. Enfin, les liaisons syntaxiques du C++ (bindings) sont déconseillées depuis la version 2.2 de la spécification (c'est dommage, j'aime bien ce sucre syntaxique) et ne seront plus implémentées pour les nouvelles fonctionnalités. Dans cet article, les codes Open MPI fournis utiliseront la syntaxe C++ quand celle-ci est disponible. Si vous installez vous-même Open MPI à partir des sources (récentes), pensez à passer l'option `--enable-mpi-cxx` au script `./configure`.

Un bel exemple qui ne sert à rien : lancer des processus qui disent simplement bonjour.

```
include <mpi.h>
include <iostream>
```



```

using namespace std;
int main(int argc, char** argv)
// Initialisation de la communication Open-MPI
MPI::Init();
// ou MPI::Init(argc, argv);
int rank = MPI::COMM_WORLD.Get_rank();
int nb_process = MPI::COMM_WORLD.Get_size();
char proc_name[MPI_MAX_PROCESSOR_NAME];
int proc_name_len;
MPI::Get_processor_name(proc_name, proc_name_len);
char lib_version[MPI_MAX_LIBRARY_VERSION_STRING];
int lib_version_len;
MPI::Get_library_version(lib_version, lib_version_len);
int major, minor;
MPI::Get_version(major, minor);
cout << "Le processus " << (rank + 1) << "/" << nb_process
<< " qui tourne sur la machine " << proc_name
<< " et utilise la librairie " << lib_version
<< " implementant le standard " << major << "." << minor
<< " vous dit BONJOUR !"
<< endl;
// Fin de la communication Open-MPI
MPI::Finalize();
return 0;

```

La routine `MPI::Init()` permet d'initialiser la communication entre les processus (c'est typiquement une routine collective). Cela permet notamment de créer un communicateur universel `MPI::COMM_WORLD` qui permettra à tous les processus de communiquer. Chaque processus peut récupérer son rang méthode `Get_rank()` au sein d'un communicateur (entre 0 et n-1 pour n processus), ainsi que le nombre total de processus de ce communicateur méthode `Get_size()`.

Chaque processus peut également récupérer le nom de la machine sur laquelle il est exécuté `MPI::Get_processor_name()`, la version de la librairie Open MPI installée `MPI::Get_library_version()` ou encore la version du standard MPI que cette librairie implémente `MPI::Get_version()`.

Enfin, la fin de la communication est actée par la routine `MPI::Finalize()`.

La compilation d'un programme utilisant Open MPI doit être effectuée via un compilateur dédié (en réalité, c'est une surcouche de votre compilateur préféré) : `mpic++` alias `mpicxx` (pour le C++) ou `mpicc` (pour le C). Ce compilateur exploite pleinement les variables d'environnement usuelles `CXX/CC` pour le compilateur à utiliser en interne.

Donc pour compiler notre programme : `mpic++ -Wall -ansi -pedantic -std=c++11 helloworld-mpi.cpp -o helloworld-mpi` Si on l'exécute (un peu bêtement), on obtient une instance qui affiche les informations demandées : `./helloworld-mpi`

Le processus 1/1 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2, package: Open MPI build@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan 21, 2016 implementant le standard 3.0 vous dit BONJOUR ! Eh oui, ma machine s'appelle spartacus (parce que c'est une bête de compét'), mais ce n'était pas ça la bonne question. La bonne question était : mais comment lance-t-on plusieurs processus ? Pour cela, il faut utiliser le programme mpirun, dont la syntaxe est mpirun [options mpirun] [programme] [args prog]. mpirun -np 2 ./helloworld-mpi

Le processus 2/2 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2, package: Open MPI build@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan 21, 2016 implementant le standard 3.0 vous dit BONJOUR !

Le processus 1/2 qui tourne sur la machine spartacus et utilise la librairie Open MPI v1.10.2, package: Open MPI build@lgw01-57 Distribution, ident: 1.10.2, repo rev: v1.10.1-145-g799148f, Jan 21, 2016 implementant le standard 3.0 vous dit BONJOUR ! L'option -np permet de spécifier le nombre de processus à lancer. Il est également possible de spécifier plusieurs hôtes possibles (machines devant être accessibles en réseau et disposer du même programme – typiquement sur un lecteur réseau) avec l'option -host, ou encore de fournir un fichier contenant ces informations, et bien d'autres possibilités de configuration encore.

C'est bien beau tout ça, mais si on faisait des processus qui communiquent, ce serait mieux...

Donc je propose d'attaquer un autre exemple qui ne sert à rien : calculer (inefficacement) la moyenne d'une série de notes !

Pour cela, détaillons l'écriture du fichier moyenne-mpi.cpp, qui commencera comme il se doit par quelques inclusions : 01: include <mpi.h> 02: include <iostream> 03: include <cstdlib> // Pour atof() et atexit() 04: include <libgen.h> // Pour basename() 05: include <vector> La première ligne permet d'inclure toute l'API Open MPI. La seconde nous permettra de faire des entrées/sorties (donc ici surtout des sorties). Les troisième et quatrième lignes serviront respectivement à transformer les arguments (chaînes de caractères) en réels, enregistrer des fonctions à exécuter à la fin du programme et à récupérer le nom du programme (un petit man 3 atof / man 3 atexit / man 3 basename devrait suffire à comprendre l'utilisation de ces fonctions). Enfin, nous allons stocker (choix plus que discutable) les valeurs passées en argument dans des vecteurs.

Afin de ne pas nous embêter à préfixer les objets/fonctions de l'espace de nom std, nous allons expliquer que nous utiliserons cet espace de nom : 07: using namespace std; Pour calculer la moyenne, il faut d'abord calculer la somme d'une série de valeurs : 09: double somme(const vector<double> &v) 11: if (!MPI::Is\_initialized()) 12: throw "MPI n'est pas initialisé"; 13: 14: size\_t n = v.size(); 15: double res = 0; 16: for(size\_t i = 0; i < n; ++i) 17: res += v[i]; 18: 19: return res; 20: La fonction **somme()** prend en entrée une référence constante sur un vecteur de réels double précision (ainsi le vecteur en entrée ne sera pas recopié et nous garantissons qu'il ne sera pas modifié).

Au passage, nous nous assurerons que la fonction **somme()** est appelée

après que l'environnement MPI a été initialisé (vérification à la ligne 11 ; si le test échoue – donc que l'environnement n'est pas initialisé – alors on lance une exception (ligne 12)). Le reste du code est somme toute assez classique.

Je profite de cet article pour présenter une astuce permettant de finaliser proprement les communications MPI quelle que soit la raison mettant fin aux instances. Pour cela, il suffit de définir une fonction ne prenant aucun paramètre et ne renvoyant rien, dont le corps contient le code qui devra être exécuté à la fin de l'exécution du programme : 22: void mpi\_ending() 23: //Findela communication Open – MPI 24: MPI :: Fin

Puis, après avoir initialisé la communication Open MPI, il suffit d'empiler cette fonction dans les fonctions à exécuter à la fin du programme. 27: int main(int

argc, char\*\* argv) 29: // Initialisation de la communication Open MPI 30: MPI::Init(); 31: atexit(mpi\_ending); Comme dans le précédent exemple, il est de

bon ton de récupérer le rang du processus courant et le nombre total de processus qui tournent en parallèle : 33: int rank = MPI::COMM\_WORLD.Get\_rank(); 34 :

int nb\_process = MPI::COMM\_WORLD.Get\_size(); Nous nous assurons que

le programme (ou plutôt les programmes) a été appelé avec au moins un argument, ce qui est toujours mieux pour calculer une moyenne. Dans le cas

contraire, seul le processus ayant le rang 0 (et celui-ci existe nécessairement) affichera un message d'usage sur la sortie d'erreur. Le programme renverra 1 au

système pour signifier un fonctionnement anormal et du fait de l'utilisation de la fonction atexit() à la ligne 31, nous avons la garantie que tous les processus

termineront correctement la session MPI. 36: if (argc < 2) 37: if (rank < 38: cerr << "usage: " << basename(argv[0]) << " v1 > ... < v\_n > " << endl 39: <<

"Le programme calcule la moyenne des valeurs passées en argument" 40: << endl; 41 :

return 1; 42 : 43 : Si le programme arrive jusqu'ici, c'est qu'au moins un argument a été saisi. Nous allons donc commencer à identifier quel processus tra-

vaillera sur quelles données... Nous savons que le programme a été lancé avec **argc** arguments, incluant le nom de l'exécutable. Ainsi le nombre de valeurs à

traiter est égal à (**argc** - 1). Nous savons également qu'il y a **nb\_process** en

cours d'exécution pour traiter ces valeurs. Nous allons donc répartir ces valeurs équitablement sur chacun des processus. Chaque processus aura au moins **q**

(= **argc** - 1) **nb\_process** valeurs à traiter. Il reste **r** = (**argc** - 1) % **nb\_process**

à répartir sur les processus (calculs des lignes 45 et 46). Arithmétiquement, **r**

est strictement inférieur à **nb\_process**. Aussi, chaque processus ayant un rang (**rank**)

strictement inférieur à **r** traitera **q** + 1 valeurs et chaque processus ayant un rang supérieur ou égal à **r** traitera **q** valeurs (calcul de la valeur **size** à

la ligne 47). Il reste à calculer l'indice de la première valeur à traiter par le processus au rang **rank**. Si **rank** est strictement inférieur à **r**, alors les processus

précédents traiteront (**q** + 1) \* **rank** valeurs. Si **rank** est supérieur ou égal à **r**, alors les processus précédents traiteront **q** \* **rank** + **r** valeurs. Ces

calculs se factorisent assez facilement à la ligne 48. Nous en profitons pour afficher une sortie de debugage indiquant l'intervalle des indices traités par le

processus en cours (lignes 50 à 53). 45: int q = (argc - 1) / nb\_process; 46 :

int r = (argc - 1) % nb\_process; 47 : size\_vec = q + (rank < r); 48 : size\_vec\_start =

(q \* rank) + ((rank < r) ? rank : r); 50 : cout << "P" << rank << " : " << size\_vec\_start << " << size\_vec - 1 << " << endl;

51 : 52 : 53 :

54 :

55 :

56 :

57 :

58 :

59 :

60 :

61 :

62 :

63 :

64 :

65 :

66 :

67 :

68 :

69 :

70 :

71 :

72 :

Maintenant que chaque processus connaît son nombre de valeurs à traiter ainsi que l'indice de la première valeur à traiter, nous construisons un vecteur de la bonne dimension (ligne 55) que nous initialisons en convertissant les arguments de la ligne de commandes (chaînes de caractère C) en réels (lignes 57 à 59). 55: `vector<double> v(vec_size);` 57 : `for(size_t i = 0; i < vec_size; ++i)` 58 : `v[i] = atof(argv[i + vec_start]);` 59 : Chaque processus est ainsi capable de calculer la somme des valeurs dont il assure le traitement dans la variable **m**. Nous aurons également besoin d'une variable **m2** pour calculer la réduction de la somme à travers tous les processus (l'initialisation à **0** n'est pas une nécessité dans l'absolu, mais cela fait partie de mes bonnes pratiques de programmation). 61: `double m = somme(v), m2 = 0;` La réduction passe simplement par l'appel (ligne 63) de la méthode `Reduce` de l'objet `MPI :: COMM_WORLD` (qui est le communicateur universel). Le premier argument est l'adresse de la zone mémoire contenant les valeurs initiales. Le second argument est l'adresse de la zone mémoire qui contiendra le résultat de la réduction. Le troisième argument définit le nombre de valeurs initiales fournies par le processus (donc ici, nous n'avons qu'une seule valeur) et le quatrième argument définit le type de valeurs à réduire (ici, le type double (`MPI_DOUBLE`) – voir encadré sur les types de données échangeables entre les processus MPI). Le cinquième argument définit la réduction à appliquer (ici, une somme (`MPI_SUM`) – voir encadré sur les types de réductions proposées entre les processus MPI). Enfin, le dernier argument spécifie le rang du processus qui récupérera le résultat de l'opération de réduction (ici le processus 0 – qui existe nécessairement).

**NB** :Open MPI permet d'échanger des données de n'importe quel type, y compris des structures composites. Les types primitifs sont pré-existants (`MPI_INT`, `MPI_FLOAT`, `MPI_UNSIGNED_SHORT`, `MPI_BOOL`...). Les types de données sont de type `MPI::Datatype` (et non les types natifs du langage) et Open MPI propose des fonctions (méthodes statiques de la classe `MPI :: Datatype` dans le cas de l'API C++) permettant de définir n'importe quel type de données. Il existe également des types correspondant à des paires de types primitifs tels que `MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_2INT`... servant notamment aux opérations de réduction `MPI_MAXLOC` et `MPI_MINLOC` (voir note sur les réductions proposées entre les processus MPI). Open MPI permet d'effectuer plusieurs opérations de réduction. Toutes ne sont pas valides selon le type des données échangées.

Nom MPI	Signification
<i>MPI_MAX</i>	Maximum
<i>MPI_MIN</i>	Minimum
<i>MPI_SUM</i>	Somme
<i>MPI_PROD</i>	Produit
<i>MPI LAND</i>	ET logique
<i>MPI_BAND</i>	Et binaire
<i>MPI_LOR</i>	OU logique
<i>MPI BOR</i>	OU binaire
<i>MPI_LXOR</i>	OU exclusif logique
<i>MPI_BXOR</i>	OU exclusif binaire
<i>MPI_MAXLOC</i>	Valeur maximale et position globale parmi tous les processus
<i>MPI_MINLOC</i>	Valeur minimale et position globale parmi tous les processus

Open MPI permet de définir nos propres opérations de réduction. Le cas échéant, ces opérations sont présumées associatives. Il est possible de spécifier si ces opérations sont également commutatives, dans le cas contraire, ces opérations sont appliquées par ordre croissant de processus. Enfin, toujours à des fins de debuggage, nous afficherons les valeurs calculées/récupérées dans les variables **m** et **m2** pour le processus courant. 63: `MPI::COMM_WORLD.Reduce(m, m2, 1, MPI_DDOUBLE, MPI_SUM, 0);` 64: `cout << "P" << rank << " : "66 :<< "sommelocale" << m <<` `" , "67 :<< "sommeglobale" << m268 :<< endl;` Il est à noter que chaque processus travaille à son rythme. Toutefois, les opérations collectives (Init, Reduce, Finalize...) imposent aux processus de s'attendre mutuellement. Toutefois, avant d'afficher le résultat final du calcul de la moyenne, il est préférable de s'assurer que tous les processus ont fini d'écrire les informations de debuggage (lignes 65 à 68). Pour cela, il suffit de poser une barrière (ligne 70), c'est-à-dire une opération collective qui ne fait rien de plus que d'empêcher un processus de poursuivre tant que tous les autres processus ne sont pas au même point du programme. 70: `MPI::COMM_WORLD.Barrier();` L'usage de barrières est pratique, puisqu'il permet de synchroniser l'exécution des processus, toutefois – et l'analogie avec la barrière est très pertinente – cela les ralentit également. Ainsi, mettre une barrière juste avant l'opération de réduction ne présente aucun intérêt puisque la réduction elle-même fait office de barrière. De même, si le code de debuggage (lignes 65 à 68) est à terme supprimé, alors la barrière de la ligne 70 ne présente plus aucune utilité non plus.

Revenons au calcul de la moyenne. Suite à l'opération de réduction, seul le processus de rang 0 connaît le résultat de la réduction (on aurait pu utiliser la méthode Allreduce qui distribue le résultat à tous les processus, mais dans le cas présent, cela n'aurait rien apporté). Seul le processus de rang 0 (ligne 72) va donc diviser la somme globale par le nombre de valeurs (ligne 73 ; je rappelle à toutes fins utiles que nous avons garanti que argc est supérieur ou égal à 2 – ligne 36 – et donc que (argc - 1) est non nul). Enfin, ce processus va afficher la moyenne sur la sortie standard. 72: `if (!rank)` 73: `m2 /= (argc - 1);` 74: `cout <<` `"Moyenne calculée : " << m2 << endl;` 75: 1 reste à remercier une fois de plus la fonction **atexit** de faire le travail de finalisation de la communication MPI tout

en renvoyant 0 au système d'exploitation pour lui signifier que le programme s'est correctement exécuté. 77: return 0; 78: Compilons maintenant ce programme : `mpic++ -Wall -ansi -pedantic -std=c++11 moyenne-mpi.cpp -o moyenne-mpi` Et exécutons-le d'abord sans MPI : `./moyenne-mpi 1 2 3 4 5 6 7 8`

P0: [0, 7] P0: somme locale 28, somme globale 28 Moyenne calculée : 3.5 Cela semble fonctionner (c'est heureux).

Essayons maintenant dans l'environnement MPI, avec un seul processus : `mpirun -np 1 ./moyenne-mpi 1 2 3 4 5 6 7 8`

P0: [0, 7] P0: somme locale 28, somme globale 28 Moyenne calculée : 3.5 On obtient le même résultat, ce qui est plutôt rassurant.

Augmentons le nombre de processus, si possible avec une valeur qui ne soit pas un diviseur de 8 pour éviter de tomber dans un cas trop particulier : `mpirun -np 3 ./moyenne-mpi 1 2 3 4 5 6 7 8`

P0: [0, 2] P1: [3, 5] P1: somme locale 12, somme globale 0 P2: [6, 7] P2: somme locale 13, somme globale 0 P0: somme locale 3, somme globale 28 Moyenne calculée : 3.5 On peut vérifier que nos 8 valeurs sont correctement réparties sur les 3 instances. On s'aperçoit également que seul le processus de rang 0 connaît la valeur de m2 (tous les autres n'ont pas modifié la valeur par défaut de m2 – quelle bonne habitude de programmation que de systématiquement initialiser ses variables ;-)).

La question que nous avons jusqu'alors passée sous silence est de savoir ce qu'il se passe si nous exécutons notre programme avec plus d'instances que de valeurs. Eh bien, théoriquement ça fonctionne, puisque dans ce cas, les premiers processus auront une seule valeur à traiter et les suivants n'en auront pas. Vérifions rapidement : `mpirun -np 10 ./moyenne-mpi 1 2 3 4 5 6 7 8`

P3: [3, 3] P4: [4, 4] P0: [0, 0] P1: [1, 1] P1: somme locale 1, somme globale 0 P2: [2, 2] P3: somme locale 3, somme globale 0 P5: [5, 5] P2: somme locale 2, somme globale 0 P5: somme locale 5, somme globale 0 P8: [8, 7] P9: [8, 7] P9: somme locale 0, somme globale 0 P8: somme locale 0, somme globale 0 P7: [7, 7] P7: somme locale 7, somme globale 0 P6: [6, 6] P6: somme locale 6, somme globale 0 P4: somme locale 4, somme globale 0 P0: somme locale 0, somme globale 28 Moyenne calculée : 3.5 Cela fonctionne encore. C'était un petit pas dans le calcul de la moyenne, mais un grand pas dans le calcul distribué.

## 15.2 OpenMP ou Open MPI ?

La question qui pourrait se poser maintenant serait de savoir s'il est préférable d'utiliser OpenMP plutôt qu'Open MPI, ou inversement. Nous pourrions faire une réponse de normand en expliquant que cela dépend du contexte, et c'est finalement la seule véritable bonne réponse, mais dans le cas présent, l'objectif est d'illustrer les deux paradigmes. Par conséquent, la réponse sera pour cette partie de ne pas les opposer, mais de les utiliser conjointement (le ou n'étant pas exclusif en français, le titre de cette section n'est donc finalement pas fallacieux). Bien évidemment, certains langages intègrent plus ou moins implicitement et naturellement les paradigmes de parallélisme, tels que NodeJS ou Go. Comme

toujours, le choix du langage doit dépendre des objectifs et non de notre champ de compétence ou de notre affinité. Ce n'est pas parce que je n'aime pas le Java que je m'interdis de programmer dans ce langage (bon, c'est un mauvais exemple parce qu'il y a toujours une alternative préférable Java :-p).

Ceci étant, si le problème que l'on souhaite résoudre est simple et ne nécessite que peu de ressources matérielles, le choix reste assez libre. Dès lors que le volume de données à traiter devient colossal (ère du Big Data) le besoin de performance devient primordial et le passage à des langages tels que le C++ voire le C (qui contrairement à ce que l'on peut lire parfois – y compris dans d'illustres revues comme GNU/Linux Mag, Linux Pratique HS... – est un langage de haut niveau) sont une nécessité et les implémentations des standards OpenMP et Open MPI sont clairement opportunes.

Méthode et applications génétiques