Grid Computing

Mihaela JUGANARU-MATHIEU
mathieu@emse.fr

École Nationale Supérieure des Mines de St Etienne

2014-2015

Bibliographie (livres et revues) :

- Frédéric Magoulès, Jie Pan, Kiat-An, Tan Abhinit Kumar *"Introduction to Grid Computing"*, CRC Press, 2009
- Barry Wilkinson *"Grid Computing. Techniques and Applications"*, CRC Press, 2010

- F.T. Leighton *"Introduction aux Algorithmes et Architectures Parallèles"*, Morgan Kaufmann, 1992
- George S. Almasi, Allan Gottlieb *"Highly Parallel Computing"*, 2nd edition, The Benjamin/Cummings, 1994

- Christian Lavault *"Evaluation des algorithmes distribués : Analyse, complexité, méthodes"*, Hermès, 1995

- journal en ligne : *"International science grid this week"*,

Déroulement du cours :

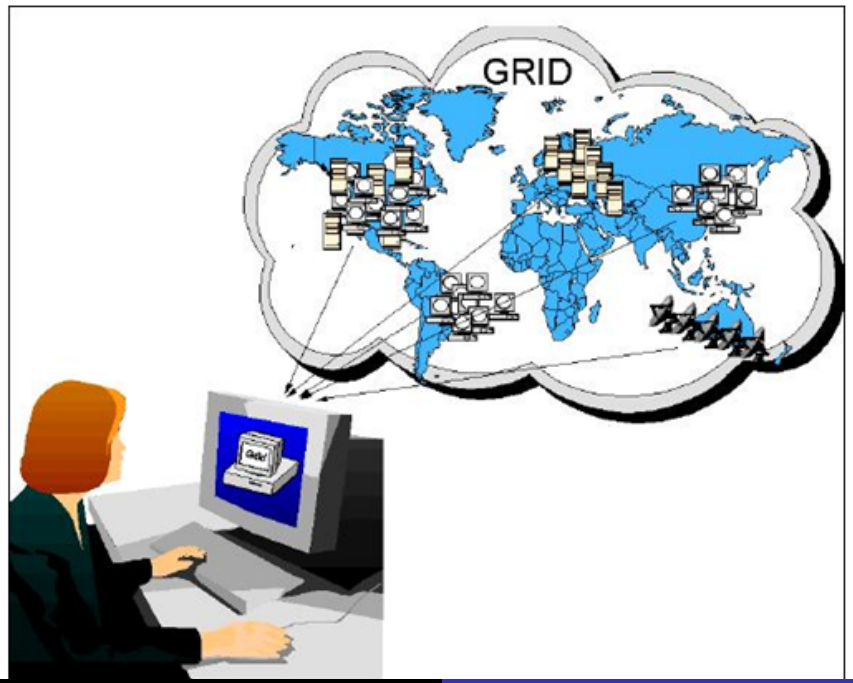
- cours magistral
- TD/TP : calcul parallèle - modèle PRAM, calcul distribué, MPI, OpenMP, Map Reduce et Hadoop

Plan

1. Introduction au Grid Computing
   - Concepts de base
   - Historique
   - Paradigmes du calcul parallèle
   - Paradigmes du calcul distribué
   - MPI
   - MPI
   - OpenMP
2. Problématique spécifique du Grid Computing
   - Management des données
   - Allocation de charge
   - Monitoring
   - Sécurité et confiance
3. Grid Middleware

Concepts de base

Grid Computing (Grille Informatique) désigne l'usage d'une infrastructure distribuée et ouverte qui interconnecte diverses ressources informatiques et qui permet à faire du calcul et / ou du partage de données ou des ressources physiques.



Une ressources informatique désigne : un ordinateur mono ou multi processeur, téléphones mobiles, espaces de stockage ...

Les équipements appartiennent à des organisations (entreprises, universités, ...) parfois géographiquement distantes qui se partagent les ressources. On parle d'**organisation virtuelle** pour désigner l'ensemble des ressources à disposition.

La **virtualisation** pour un utilisateur désigne que sa tâche se réalise de manière transparente dans le réseau qui forme la Grid, sur un seul équipement ou sur plusieurs. La tâche demandée peut, à son tour, être traitée comme un ensemble de services réalisés de manière parallèle et / ou distribué.

Selon les 3 critères du Grid Computing selon Ian Foster :

- les ressources ne sont pas l'object d'un contrôle centralisé
- des protocoles et interfaces standard et ouverts sont utilisés
- doit délivrer une qualité de service non-trivial en terme de débit (throughput), temps de réponse, disponibilité, sécurité et/ou co-allocation des ressources multiples.

Ian Foster "What is the Grid ? A Three Point Checklist" Web Published, 2002 - `http://dlib.cs.odu.edu/WhatIsTheGrid.pdf`

Grid vs. Superordinateur

Le calcul de haute performance (High Performance Computing **HPC**) se réalise sur le **superordinateurs** (au-delà de 10^3 processeurs contenus on parle de **massivement parallèle**).

On mesure en Mips (Millions of Instructions Per Second) ou MFlops (Million of Floating point operation per second).

Le contrôle y est centralisé et les processeurs sont reliés par un bus (ou autre type de réseau d'interconnexion).

Tendance actuelle : superordinateurs de TFlops et TBytes mémoire *(exemple : 65 TFlops pour le NEC SX-8 en 2004, 839 TFlops pour le NEC SX-9 en 2011, jusqu'à 141 TFlops par noeud pour NEC SX ACE en 2014)*.

Grid vs. Superordinateur

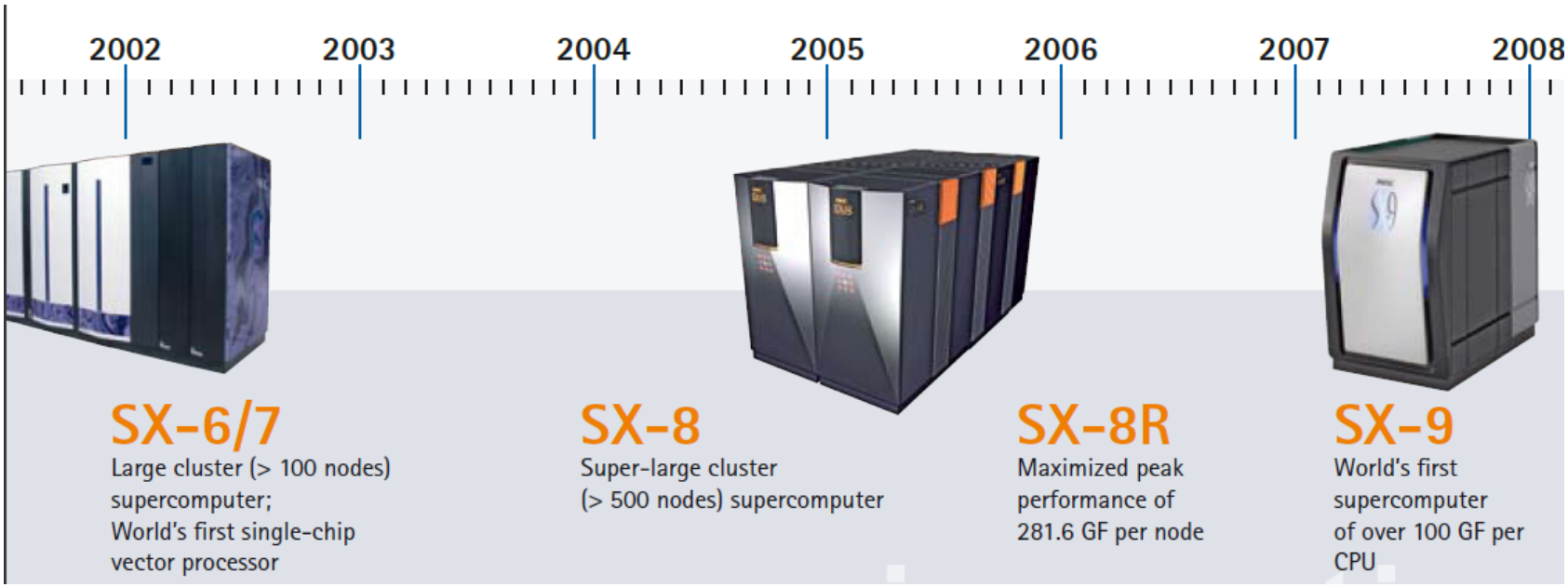


FIGURE: La gamme NEC (Japon)

Actuellement (décembre 2012) : 839 TFlops.

Grid vs. Superordinateur

Breakthrough Titan Performance

Jaguar Specs (2011)	
Compute Nodes	18,688
Login & I/O Nodes	256
Memory per node	16 GB
# of Opteron cores	224,256
# of NVIDIA K20 "Kepler" accelerators (2013)	N/A
Total System Memory	300 TB
Total System Peak Performance	2.3 Petaflops

Titan Specs (2012)	
Compute Nodes	18,688
Login & I/O Nodes	512
Memory per node	32 GB + 6 GB
# of Opteron cores	299,008
# of NVIDIA K20 "Kepler" accelerators (2013)	18,688
Total System Memory	710 TB
Total System Peak Performance	20+ Petaflops

FIGURE: Superordinateur TITAN (US) *chiffres novembre 2012*

Chaque noeud de calcul dispose de 12 cœurs CPU et 12 cœurs GPU. La machine a aussi une baie de stockage de quelques 10PetaBytes.
Localisation : Oak Ridge National Laboratory.

Grid vs. Superordinateur

Le calcul haut débit (High Throughput Computing **HTC**) est plutôt la caractéristique des Grids.

Une Grid (Grille) est utilisée pour des calculs intensifs et/ou pour le traitement et stockage des volumes importants de données.

Domaines d'application : calculs intensifs, bases de données, Machine Learning, Data Mining, IA ..., SIG, météo (-)
Domaine privilégié : animation graphique (3D) avec une Grid de processeurs graphiques (Grid of GPU).

Grid vs. Superordinateur

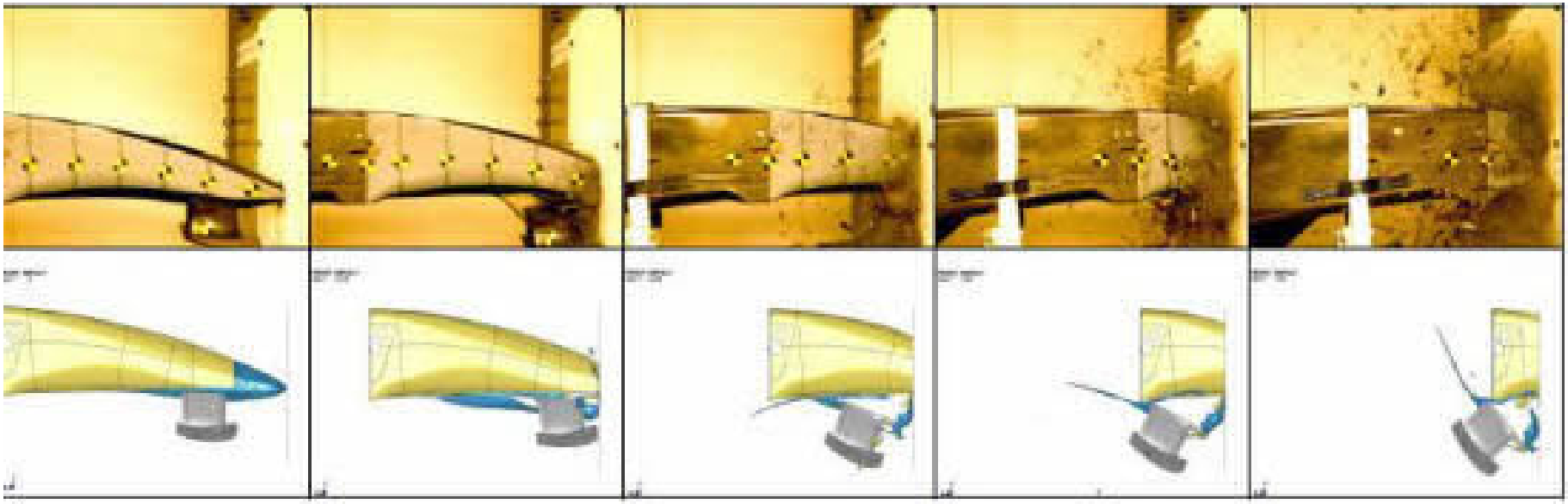


Figure 6: High speed image sequence of front impact crash test vs. simulation results

FIGURE: Images réelles et représentations obtenues par simulation d'un crash test d'impact (par analyse d'élément fini)

Grid vs. Superordinateur

L'architecture est souple, car pas centralisée, ce qui la rend tolérante aux pannes, disponible et robuste.

Le principe de base de fonctionnement est celui de la **répartition** - répartition à grande échelle.

La répartition est la mise à disposition d'un ensemble de ressources et de services connectés via un réseau pour tous les usagers possédant un droit d'accès en un point quelconque.

Répartition vs. parallélisme

Une Grid peut contenir comme noeuds : les postes de travail, des cluster, des serveurs puissants, des superordinateurs, des ressources variées.

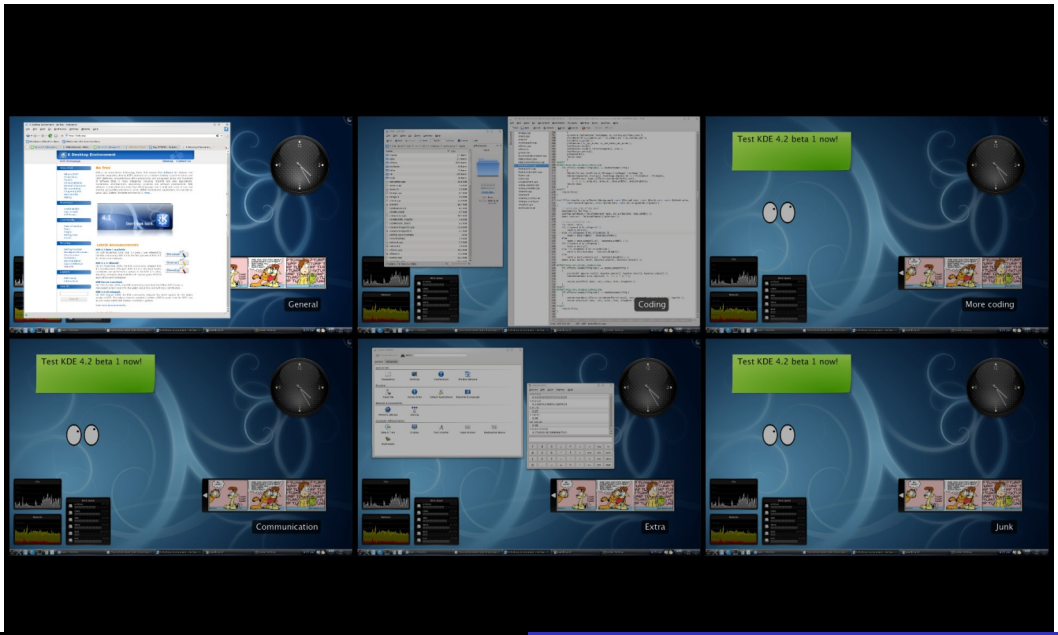
Cluster

Le cluster (ferme de serveurs ou ferme de calcul) est à mi-chemin entre la Grid et les superordinateurs.
Il s'agit d'un groupe de serveurs groupés physiquement et attaqués par des **nœuds frontaux**, autres nœuds composants le cluster : **nœuds de calcul** et **nœuds de stockage**.

Desktop Grid

Il s'agit d'une Grid formée par des PC (usuellement utilisés à seulement 10% de leur capacité).

Certains systèmes d'exploitation propose des outils de constitution de telles Grids.



Logiciel pour la Grid

Deux besoins clairs :

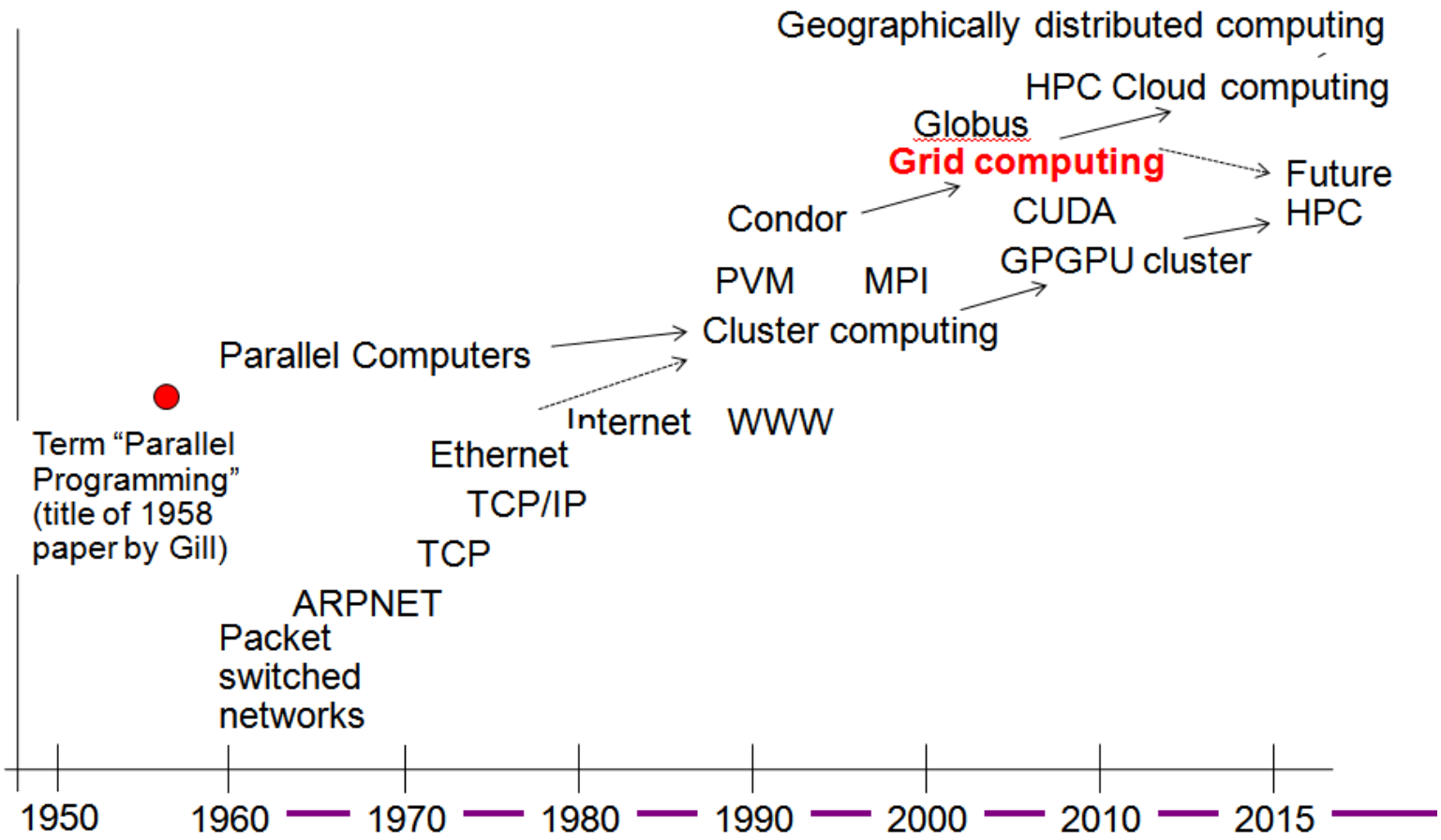
- un logiciel d'interface utilisateur qui permet d'indiquer le traitement à réaliser et les besoins en ressources
- le **middleware** le software spécifique qui permet de réaliser concrètement le partage des ressources de la Grid. *Exemples de middleware : Globus Toolkit, gLite et UNICORE.*

Outils accessoires : surveillance de la Grid (charge, communication), I/O (récupération des résultats)

Grid Computing - "vieille technologie"

Le Grid Computing est une technologie arrivée à maturité. Elle est basée sur :

- la philosophie du **calcul parallèle** et **calcul distribué**
- **réseaux**
- formats et protocoles ouverts


Geographically distributed computing
HPC Cloud computing
Globus
Grid computing
Future
HPC
Condor
CUDA
GPGPU cluster
PVM
MPI
Cluster computing
Parallel Computers
Internet
WWW
Term "Parallel Programming" (title of 1958 paper by Gill)
Ethernet
TCP/IP
TCP
ARPNET
Packet switched networks
1950
1960
1970
1980
1990
2000
2010
2015

Parallélisme

Deux traitements se réalisent **en parallèle** quand ils s'exécutent en même temps sur des CPU différentes.
Une **application** est dite **parallèle** quand elle se compose de traitements parallèles et s'exécute sur une architecture adéquate.
On parle d'algorithme parallèle quand on a besoin de plusieurs CPU pour le réaliser et d'architecture parallèle quand on dispose de plusieurs CPU travaillent de manière indépendante, synchronisée ou non.
Les traitements qui composent une application parallèle ne sont pas (toujours) indépendants : on partage des données $\rightarrow$ où met-on les données partagées ?

Classification de Flynn

Selon Michel Flynn en 1966 :

Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Flynn, M. (1972). *"Some Computer Organizations and Their Effectiveness"*. IEEE Transactions on Computers, Vol. C-21, No 948

Classification de Flynn

S.I.S.D. : Single Instruction Single Data stream

- architecture monoprocesseur
- calculateur von Neumann conventionnel
- *exemples : un PC mono-core, un serveur SUN, etc ...*

S.I.M.D. : Single Instruction Multiple Data stream

- les processeurs exécutent de façon synchrone la même instruction
- sur des données différentes (e.g. éléments d'un vecteur, d'une matrice, d'une image)
- une unité de contrôle diffuse les instructions

Classification de Flynn

M.I.S.D. : Multiple Instruction Single Data stream

- ça n'existe pas

M.I.M.D. : Multiple Instruction Multiple Data stream

- plusieurs processeurs (à la rigueur hétérogènes) exécutent de façon asynchrone des instructions (même code ou code différent)
- sur des données différentes
- chaque processeur a sa propre unité de contrôle sa propre mémoire (registres, RAM)
- il y a ou pas une mémoire commune
  - mémoire partagée (si il y a une mémoire commune)
  - mémoire distribuée si aucun zone de mémoire n'est commune

Programmation SIMD, MIMD

Avantages du SIMD :

- Facilité de programmation et de débogage
- Processeurs synchronisés ! coûts de synchronisation minimaux
- Une seule copie du programme
- Décodage des instructions simple

Avantages du MIMD :

- Plus flexible, beaucoup plus général
- Exemples :
  - mémoire partagée : OpenMP, threads POSIX
  - mémoire distribuée : PVM, MPI (depuis C/C++/Fortran)

Programmation des architectures parallèles

On utilise le plus souvent le paradigme **SPMD** : Single Program Multiple Data - un même code fonctionne sur plusieurs unités de calcul.
On utilise également (plus difficile à programmer) le mode MPMD : Multiple program Multiple Data, néanmoins il n'y a jamais un code différent sur chaque unité, les programmes sont dupliqués.

Speed-up

Soient les notations suivantes :

- T_1 le temps d'exécution en mode séquentiel sur un seul processeur
- T_n le temps d'exécution sur une architecture parallèle avec n processeurs

Le speed-up se défini comme :

$$speedup = \frac{T_1}{T_n}$$

Bien évidement $speedup < n$.

Très peu d'applications ont le rapport $\frac{speedup}{n}$ proche de 1 (quelques application de traitement d'image on ce ratio autour de 95%).

Loi d'Amdahl

Soient encore les notations :

- f la fraction du programme séquentiel qu'on peut paralléliser
- S_n le speed-up pour n processeurs
- O l'overhead = temps de communication entre processeurs, de synchronisation, autres ...

Dans une approche purement théorique : $O = 0$ et

$$T_n = (1 - f)T_1 + \frac{fT_1}{n}$$

$$S_n = \frac{n}{1 + (1 - f)n}$$

Loi d'Amdahl

On déduit :

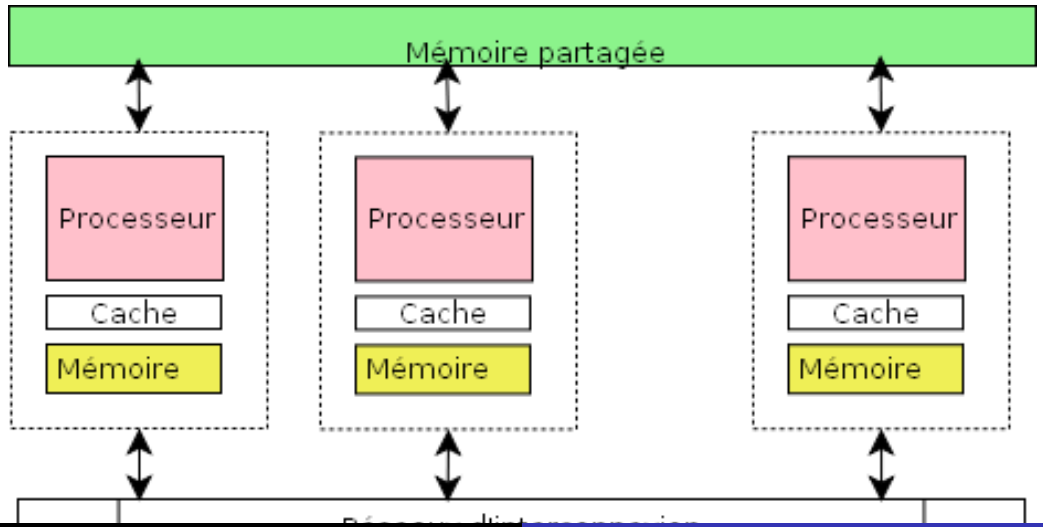
$$S_\infty < \frac{1}{1-f}$$

En réalité $O \neq 0$

$$S_n = \frac{T_1}{(1-f)T_1 + \frac{fT_1}{p} + O}$$

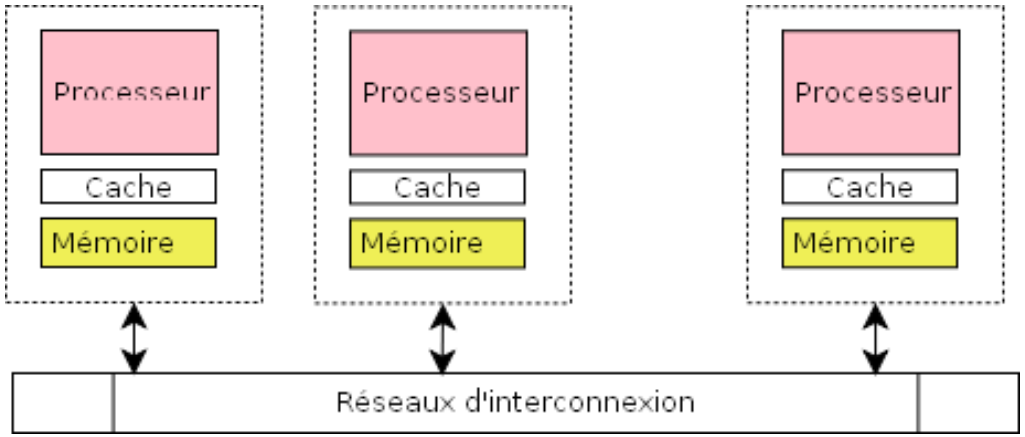
Mémoire partagée

- la communication entre processeurs se fait par ce biais
- besoin de protocole/méthode d'accès à la mémoire partagée en cas de consultation simultanée (lecture/écriture)
- s'il y a réplication de l'information partagée dans les caches → problème de cohérence
- espace d'adressage grand



Mémoire distribuée

- les processeurs communiquent exclusivement par envoi de messages
- synchronisation plus difficile



Fonctionnement similaire aux architectures distribuées.

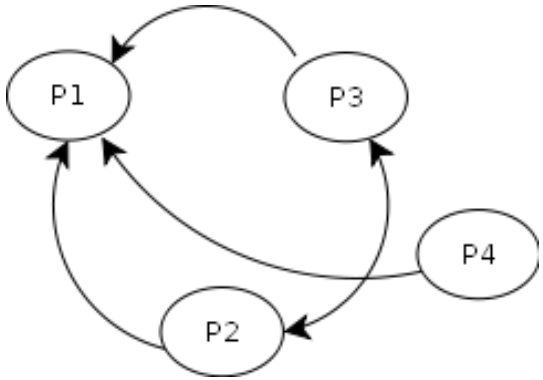
Paradigmes du calcul distribué

Une application distribuée se définit comme une application qui a besoin pour être réalisée des ressources physiquement distinctes.
Une architecture distribuée est une entité de traitement composée des éléments de calcul, de stockage et réseau physiquement distincts.
Trois types de besoins générant applications distribuées :

- travail collaboratif
- calcul
- volume très important des données *(exemple - base de données distribuée)*

Application distribuée

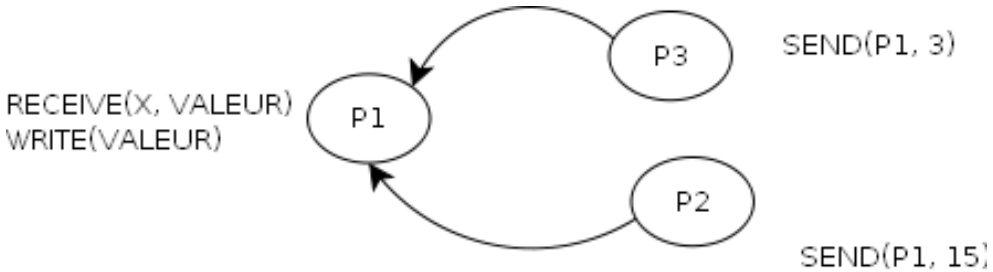
Une application distribuée se compose d'unité de traitement communicantes.



Les flèches indiquent la direction des communication entre les unités de traitement, communément appelés **processus**.

Application distribuée

Le fonctionnement d'un programme séquentiel est déterministe, mais pour une application distribuées ce n'est pas le cas.



L'ordre de délivrance des messages n'est pas garanti en général, des mécanismes d'estampillage (ou autres), doivent être mis en place.

La complexité d'un algorithme séquentiel s'exprime en fonction du temps et de l'espace mémoire utilisé, la complexité d'une algorithme distribué s'exprime en temps, espace et en **nombre de**

Algorithmique distribuée

Les problèmes spécifique à ce mode de fonctionnement :

- Temps et état dans un système réparti : datation, observation, mise au point, synchronisation.
- Algorithmes de base des systèmes répartis (élection, terminaison, etc) : gestion de groupes, calcul réparti.
- Tolérance aux fautes, diffusion fiable, gestion de groupes : serveurs fiables, gestion de copies multiples.
- Consensus et validation : service de consensus, transactions réparties.
- Gestion répartie de l'information : cohérence de caches, objets répartis

MPI

MPI - Message Passing Interface
C'est une API standard apparue en 1994, qui permet de réaliser dans des langages de haut niveau (FORTRAN, C, C++, Python, Java) des programmes parallèles basés sur l'**échange des messages**.
(exemples - libraires mpi.h ou mpif.h)

MPI offre donc la possibilité d'écrire du code dans le langage de son choix et faire appel aux fonctions de l'API pour réaliser la partie communication. **Il n'y a pas de mémoire globale.**

MPI

MPI est utilisé autant pour la programmation parallèle en mode SPMD, que en mode MIMD (MPMD - Multiple Program Multiple Data), que pour l'écriture des applications distribuées.
Il y a des implémentations gratuites de MPI : OpenMPI, LAM/MPI et MPICH (historique), la plupart des fabricants offrent des implémentations plus performantes payantes.
C'est devenue de facto un standard.
Depuis 1997 la version MPI-2 qui permet en plus entre autres des opération d'entrée/sortie et la gestion dynamique des processus.

MPI

Un **message** est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s).
En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :

- l'identificateur du processus émetteur
- l'identificateur du processus récepteur
- le type de la donnée
- sa longueur

Une application est formée par un ensemble de processus communicants, chaque processus est capable d'envoyer, recevoir et traiter les messages.

MPI - initialisations

En MPI un programme correspond à un ensemble de processus capable de communiquer ensemble - cet ensemble forme un **communicateur**.
Chaque processus a un numéro d'ordre unique au sein du communicateur qui est son identifiant.

Initialement chaque processus fait appel à une fonction d'initialisation de l'environnement MPI, puis détecter son communicateur et son identifiant.

- `MPI_INIT(&argc, &argv)` et `MPI_FINALIZE()` permettent d'initialiser/arrêter l'environnement MPI.
- La variable globale `MPI_COMM_WORLD` indique le communicateur.

MPI - initialisations

Les fonctions :

- `MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)` fournit le nombre de processus dans le communicateur
- `MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)` fournit de rang du processus

Ces deux fonctions apparaissent toute de suite après le `MPI_INIT`.

MPI - type de données

On peut utiliser les types standard (entier, virgule flottante, etc) ou définir ses propres types de données : vecteur, composé, etc. Néanmoins il est préférable d'utiliser les types redéfinis :

Table 5.1 Predefined data types for MPI

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing

MPI

Les communications point à point se réalisent avec `MPI_SEND`, `MPI_RECV`, `MPI_SENDRECV`.
Les communications collectives :

- `MPI_BCAST` pour la diffusion
- `MPI_GATH` pour le "gathering" (opération inverse de la diffusion)
- pour de sommes, produits et autres opérations avec une opérande venant de chaque processus : `MPI_SUM`, `MPI_PROD`, `MPI_MAX` ...

La synchronisation : `MPI_BARRIER`(*no_communicateur*).

Exemple de programme FORTRAN /MPI - source IDRIS

```
1  program anneau
2    use mpi
3    implicit none
4    integer, dimension(MPI_STATUS_SIZE) :: statut
5    integer, parameter                  :: etiquette=100
6    integer                             :: nb_procs,rang,valeur, &
7                                           num_proc_precedent,num_proc_suivant,code
8    call MPI_INIT(code)
9    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
10   call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
11
12   num_proc_suivant=mod(rang+1,nb_procs)
13   num_proc_precedent=mod(nb_procs+rang-1,nb_procs)
14
15   if (rang == 0) then
16      call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
17                    MPI_COMM_WORLD,code)
18      call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
19                    MPI_COMM_WORLD,statut,code)
20   else
21      call MPI_RECV(valeur,1,MPI_INTEGER,num_proc_precedent,etiquette, &
22                    MPI_COMM_WORLD,statut,code)
23      call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc_suivant,etiquette, &
24                    MPI_COMM_WORLD,code)
25   end if
26
27   print *,'Moi, proc. ',rang,', j''ai reçu ',valeur,' du proc. ',num_proc_precedent
28
29   call MPI_FINALIZE(code)
30 end program anneau
```

Programme C-MPI qui réalise un anneau de processus communicants

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

# define TAG 9090
int main (argc, argv)
    int argc;
    char *argv[];
{
  int rank, size;
  int next, prev, message, message_recu;

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

  next = (rank + 1) % size;
  prev = (rank + size - 1) % size;
```

Programme C-MPI qui réalise un anneau de processus communicants (suite)

```
  if (rank ==0)
  {
      message = rank + 1000;
      MPI_Send(&message, 1, MPI_INT, next, TAG, MPI_COMM_WORLD);
      MPI_Recv(&message, 1, MPI_INT, prev, TAG, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
      message_recu = message;
  }
  else
  {
     MPI_Recv(&message, 1, MPI_INT, prev, TAG, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
     message_recu = message;
     message = message + rank;
     MPI_Send(&message, 1, MPI_INT, next, TAG, MPI_COMM_WORLD);
  }

  printf(" Je suis le processus %d et j'ai recu la valeur %d du process %d\n", rank, message_recu, prev);

  MPI_Finalize();
  return 0;
}
```

Attention à l'ordre d'écriture des opération de lecture-écriture (envoi/réception) !

Il faut veiller à éviter d'écrire du code bloquant !
Par exemple un circuit d'attente se forme entre les processus de l'application. On parle de l'**interblocage**.

Exemple : si on inverse l'ordre des opérations sur le processus 0, on arrive dans un interblocage, car tous les processus attendent de leur voisin une valeur qui n'arrivera jamais.

Les communications sous MPI sont de type point à point ou de type global et toujours asynchrone. On peut choisir entre :

- communications bloquantes : `MPI_Send` et `MPI_Recv` - tant que le message envoyé n'est pas reçu la fonction `MPI_Send` n'est pas terminé.
- communication non-bloquantes : `MPI_ISend` et `MPI_IRecv`

Les communications globales sont toujours bloquantes.
Parmi les fonctions classiques : `MPI_Bcast` (la diffusion *one to all*), `MPI_Gather` (la concentration - *all to one*) et `MPI_Allgather` (*all to all*)

Les fonctions "globales" de type communications - calcul :

Global communication operation	MPI function
Broadcast operation	`MPI_Bcast()`
Accumulation operation	`MPI_Reduce()`
Gather operation	`MPI_Gather()`
Scatter operation	`MPI_Scatter()`
Multi-broadcast operation	`MPI_Allgather()`
Multi-accumulation operation	`MPI_Allreduce()`
Total exchange	`MPI_Alltoall()`

Syntaxe fonctions de communication

```
int MPI_Send(void *smessage, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm)
```

`smessage` est l'adresse du buffer mémoire à envoyer, ce buffer contient `count` valeurs de type `datatype`.
L'envoi est à destination du processus de numéro `dest`. Cette communication est faite dans le communicateur `comm`.
La valeur `tag` permet de distinguer (différencier) plusieurs messages entre deux mêmes processus.

```
int MPI_Recv(void *rmessage, int count,
            MPI_Datatype datatype, int source,
            int tag, MPI_Comm comm, MPI_Status *status)
```

`status` est une structure avec les information de synthèse sur le déroulement de l'envoi

Syntaxe fonctions de communication (2)

```
int MPI_Sendrecv (void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount,
        MPI Datatype_recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status)
```

C'est une fonction qui combine l'envoi et la réception de message entre 2 processus (chacun est émetteur et récepteur).

Syntaxe fonctions de communication (3)

```
int MPI_Bcast (void *message, int count,
               MPI_Datatype type, int root,
               MPI_Comm comm)
```

La fonction de diffusion est réalisée par le processus de numéro `root`. Le contenu du buffer `message` sera connu par tous les autres processus.

Syntaxe fonctions de communication (4)

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype, int root,
               MPI_Comm comm)
```

Cette opération dite de concentration (inverse de la diffusion) consiste dans l'envoi du contenu du buffer `sendbuf` par tous les processus sauf `root` dans la position adéquate du buffer `recvbuf`.

Exemple d'usage de broadcast et gathering

On diffuse à tous les processus la base b d'un logarithme et on concentre au niveau du processus 0 la valeur du logarithme en base b du numéro de chaque processus :

```
int main(int argc, char *argv[])
{
   int base;
   float tt;
   int rank, size;
   int i;
   float *buf_tt;

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

   printf(" processus %d parmi %d\n", rank, size);
   MPI_Barrier(MPI_COMM_WORLD);

   if (rank == 0)
   {
      printf(" Indiquez la base du logarithme :\n");
      scanf("%d", &base);
      buf_tt = (float *) malloc(size * sizeof(float));
      if (buf_tt == NULL)
         printf(" Problème d'allocation !\n");
```

```
    MPI_Bcast(&base, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank != 0)
       tt = log(rank)/log(base);

    MPI_Gather(&tt, 1, MPI_FLOAT, buf_tt, 1, MPI_FLOAT, 0, MPI_CO

     if (rank == 0)
       for (i = 1; i < size; i++)
          printf("de la part du P%d on a recu %f\n", i, buf_tt[i]

    MPI_Finalize();
    return 0;
}
```

Exécution du code MPI

On lance les processus avec l'une des commandes (équivalentes) :

```
mpiexec -n nb-proc program-name program-arguments
mpirun -np nb-proc program-name program-arguments
```

Si le système à disposition est parallèle (plusieurs coeurs), on lance les processus sur des unités de calcul différentes.

Exécution code MPI (2)

Sur une architecture distribuée on peut lancer des processus sur des noeuds différents :

- `mpirun -H nodeX,nodeY... executable` les noeuds peuvent se répéter, on crée autant de processus que des noeuds
- `mpirun -H nodeX,nodeY... -np N executable` on crée N processus distribués par OpenMPI sur les noeuds de la liste.
- il est possible et souhaitable d'indiquer un fichier de configuration : `myhost`

Exécution code MPI (3)

Exemple :

```
>more myhostfile
node01 slots=2
node02 slots=4
```

Usage :

```
mpi -hostfile myhost executable

mpi -hostfile myhost -np N executable

mpi -hostfile myhost -np N -loadbalance executable
```

on crée soit N processus, soit le nombre de slots indiqué (la somme) dans le fichier de configuration. La distribution peut aussi se faire de manière "équilibrée"

Programmation hybride

Il est possible de mélanger les deux styles de programmation et ainsi au niveau d'un processus MPI on peut générer des threads (OpenMP ou POSIX).

OpenMP

- OpenMP (Open Multiprocessing) est une API qui permet de travailler en mode mémoire partagée en utilisant des threads et/ou en exploitant le parallélisme à grain fin
- Proposé en 1997 est toujours et toujours actualisé (OpenMP Architecture Review Board)
- Répandu en milieu industriel
- Contient des : directives, librairies runtime et variables d'environnement
- Code facile à lire et portage du code séquentiel $\rightarrow$ code parallèle très rapide

OpenMP - concepts

Un programme est un seul processus (maà®tre) qui peut engendrer des threads (tâches) qui s'exécutent en parallèle.

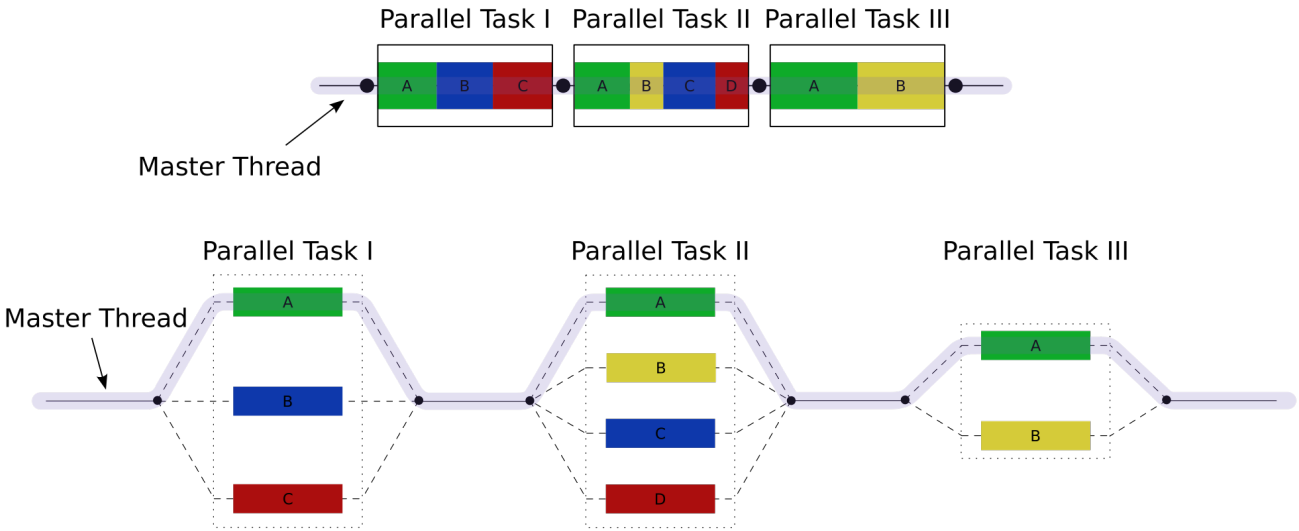


FIGURE: Source :
https://computing.llnl.gov/tutorials/openMP/

OpenMP - concepts

Chaque thread se voir nommer par un numéro d'ordre. L'ordre des threads est aléatoire, le processus maà®tre est toujours le premier thread.
Dans une région avec les threads parallèles les variables devient : partagées (shared) ou internes (private) selon des déclarations explicites. On peut définir des priorités pour la lecture/écriture concurrente.
On implémente aussi des notions de priorité, attente, flush, synchronisation, ...

OpenMP - implémentation

OpenMP est implémenté dans beaucoup de languages de programmation (Fortran, C, C++).
Les versions relativement récentes de `gcc` l'implémente. La compilation se réalise avec :

```
gcc -fopenmp prog.c -o .....
```

Les directives MPI sont de forme : `# pragma omp ...`

OpenMP - parallélisation à grain fin

Avec la directive `parallel` :

```
int main(int argc, char *argv[])
{
    const int N = 100000;
    int i, a[N];
#pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

OpenMP - programme Hello, world !

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

int main (int argc, char *argv[])
{
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and disband */
}
```

OpenMP - programme Hello, world ! - version avec une variable partagée

```
int main (int argc, char *argv[])
{
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel default(shared) private(tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

#pragma omp barrier
  /* All threads access this information */
  printf("I'm thread=%d and I see num_threads=%d\n", tid, nthreads);
  }  /* All threads join master thread and disband */
}
```

OpenMP - On forme un anneau de thread et on transmet une valeur à son voisin

```
int main (int argc, char *argv[])
{
  int nthreads, tid, value;
  int n= 10, a[10];
  int next, prev;
  int N;
  N = atoi(argv[1]);
/* Variable number of threads */
#pragma omp parallel shared(nthreads, n,a) private(next, prev, value, tid) num_threads(N)
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  nthreads = omp_get_num_threads();

#pragma omp barrier
  /* Only master thread does this */
  if (tid == 0)
  {
     printf("Number of threads = %d\n", nthreads);
     if (nthreads > n)
     {
       printf(" Nb Thread = %d greater than %d nb of shared memory cells.\n", nthreads, n);
       exit(0);
```

OpenMP - On forme un anneau de thread et on transmet une valeur à son voisin

```
#pragma omp barrier
  /* Generate random value */
  value = rand() % 25;
  printf("value from % d = %d\n", tid, value);
  prev = (tid + nthreads - 1) % nthreads;
  next = (tid + 1) %nthreads;
  a[next] = value;

#pragma omp barrier
  printf(" proc %d received from %d value = %d\n", tid, prev, a[tid]);

  }  /* All threads join master thread and disband */

}
```

La directive `barrier` indique un point de synchronisation (join pour les threads). En absence des ces points de synchronisations les valeurs obtenues seraient incohérentes et l'exécution sera non-déterministe.

1. Introduction au Grid Computing
   - Concepts de base
   - Historique
   - Paradigmes du calcul parallèle
   - Paradigmes du calcul distribué
   - MPI
   - MPI
   - OpenMP
2. Problématique spécifique du Grid Computing
   - Management des données
   - Allocation de charge
   - Monitoring
   - Sécurité et confiance
3. Grid Middleware
4. Projets Grid Computing

Problématique :

- types de données à gérer
- aspect transactionnel
- accès concurrent

Solutions :

- protocoles adaptés, algorithmes spécifiques (distribués)

Management des données

Les Grids sont typiquement de deux sortes :

- grilles de calcul : dédiées à faire des calculs intensifs
- grilles de données : dédiées plus au stockage et au traitement de données

Le management de données concerne plutôt les grilles de données.

Les données à gérer sont **hétérogènes** (format, taille, usage) et **dynamiques** pour la plupart.

Grid – Base de données distribuées

Une grille de données aura une problématique encore plus lourde à gérer qu'un système de gestion de base de données distribué (SGBD D), parce que :

- les SGBD D ont un contrôle complet sur toutes les données existantes, le contrôle n'est que partiel en Grid
- les données sont en format unitaire (unique) et de taille prévisible en BD D, mais en Grid les données sont complètement hétérogènes
- opération de même type dans la BD D : `insert`, `delete`, `update`
- le SGBD D peuvent s'implémenter sur une Grid

Données statiques - données dynamiques

Les données d'une Data Grid peuvent être :

- **statiques** - étant générées puis gardées dans la Grid sans (ou avec très peu de) changements
  exemples : données de type ADN, archives numériques d'un journal, collections de test mis en partage
- **dynamiques** - subissant des modifications (et mises de jour) fréquentes
  exemple : les données d'une plateforme de e-business

Les opérations sont relativement simples sur les données statiques et deviennent beaucoup plus complexes sur données dynamiques.

Données statiques - données dynamiques

Les opérations sur les données statiques sont simples :

- une seule écriture (lors de la création)
- lecture depuis plusieurs endroits

Les opérations sur les données dynamiques sont (liste non-exhaustive) :

- mises à jour (update)
- lectures/écritures concurrentes
- transaction
- intégration avec un système extérieur
- synchronisation car possibles répliques dans le Grid

Problème d'adressage

Les données d'une Grid sont normalement disséminées dans divers systèmes de gestion de fichier ou SGBD. Il est fort possible de réaliser des copies (semblable au système de cache) pour placer de copies **répliques** de données afin de minimiser le temps d'accès et le trafic réseau.

Autant pour les données statiques que dynamiques un problème est commun et crucial : l'**adressage** - trouver où la données est gardée (l'endroit le plus convenable pour y accéder s'il y a des répliques).

De toute manière les applications/utilisateurs doivent pouvoir accéder aux données indépendamment de leur format ou taille et indépendamment de l'implémentation des ressources. Il s'agit d'assurer l'**accès unitaire.**

Fonctionnalités du système du système de gestion de données

- gestion des répliques
- gestion des méta-données
- transfert des données
- service de stogkage et accès
- traitement correct des transactions
- intégration

Gestion des répliques

Le service de gestion des répliques (Replication Management Service - RMS) doit réaliser :

- Créer au moins une réplique de l'ensemble des données ou, au moins, d'une partie
- Gestionner les répliques : ajouter, effacer ou modifier
- Enregistrer toute nouvelle réplique
- Garder le catalogue des répliques enregistrées de manière à permettre aux utilisateurs d'interroger puis d'accéder aux répliques
- Sélectionner la réplique optimale pour un utilisateur (optimalité jugée selon l'exécution de l'application de l'utilisateur)
- Assurer automatiquement la consistance des répliques pour le même ensemble des données quand celui-ci change

Gestion des méta-données

Les **méta-données** (les informations descriptives sur les données "catalogue") concernent les informations générales de toute données de la grille, mais aussi sur la grille elle-même,le utilisateurs, etc...

Trois types principaux de méta-données :

- information système = information sur la Grid elle même : capacités de stockage de chaque ressource, modalités d'accès depuis internet, l'état de ressources (actives/en attente), politique d'usage
- de réplication = information sur les répliques avec le mapping entre les données logiques et leur copies physiques
- information d'application = information définie par la communauté sur le contenu et la structure de données, des détails sémantiques sur les éléments qui la composent, les circonstances d'obtention

Gestion des méta-données

Autre classification des méta-données selon l'objet concerné :

- **méta-données sur les données** - il s'agit de :
  - informations physiques : localisation, taille, propriétaire, format, type de ficher
  - information de réplication
  - information de domaine : attribut des données utilisés dans un domaine (d'application ou utilisateur) et aussi les relations entre domaine
- **méta-données sur les ressources** - informations sur les ressources physiques et logiques ; seront utilisées lors de la création, la sauvegarde et le transfert des données

Gestion des méta-données

- **méta-données sur les utilisateurs** (un utilisateur est celui qui crée, modifie et efface les données et les méta-données) - données classiques d'identification (nom, adresse, password, droits d'accès et d'usage) et aussi des attributs `domaine` pour identifier l'organisation à laquelle il appartient et `groupe`.
- **méta-données sur les applications** - informations sur le contenu des données générées par les application, type des annotations éventuelles, procédures et traitements appliqués en entrée/sortie (utilisées surtout pour les applications composites)

Gestion des méta-données

Les méta-données sont très importantes pour : retrouver, localiser, accéder et gestionner la donnée désirée.
Les méta-données peuvent concerner les méta-données elles-mêmes et sont aussi sujet à la réplication.

Plus en détail : les méta-données intéressent autant l'utilisateur, que les autres service de gestion de données, que les application.

Elles sont, généralement, implémentées sous la forme d'une base de données relationnelle ou de documents XML. Le format XML demeure plus souple vis-à-vis de l'interfaçage éventuel avec d'autre application et services (y compris des Web Services) et a le statut de format ouvert.

Gestion des méta-données

Deux composants importants du service de gestion des méta-données :

- service de stockage : qui assure, en plus du stockage effectif des méta-données, la gestion des ses propres répliques
- service de publication et découverte (publication and discovery service)
  - publication : permet l'accélération de l'association entre un élément (des éléments) descriptif de la données et ses autres caractéristiques
  - découverte : permet l'identification de la donnée qui intéresse
  - le service assure les deux fonctionnalités, il est possible d'obtenir schémas (graphiques) généraux et statistiques
  - permet aussi d'ajouter des annotations, de les instancier et d'utiliser ses annotations

Transfert des données

Les données et les applications se trouvant et s'exécutant dans un environnement distribués et faiblement couplé il est impératif d'assurer dans les meilleurs conditions le **transfert** des fichiers.

Les caractéristiques du transfert dans une Grid :

- transfert à haute vitesse
- découpage en paquets (stripped data transfer)
- possibilité de transfert partiel
- transfert sous le contrôle d'un tiers (third-party control of transfer)
- transfert re-initialisable (restartable transfer)

Transfert des données

Techniquement le tranfert se réalise avec FTP ou avec GridFTP.

GridFTP est un protocole basé sur FTP et adapté aux réseaux avec une bande passante importante et une large étendue

- il intègre le GSI (Grid Security Infrastructure) et KERBEROS (cryptage et authentification)
- il offre
  - découpage des paquets, le contrôle d'un tiers
  - transferts à points de reprise et tolérants aux fautes
  - négociation préalable en TCP sur la taille des paquets

`https://it-dep-fio-ds.web.cern.ch/it-dep-fio-ds/Documentation/gridftp.asp`

Transfert des données

Modes de transfert adaptés au fonctionnement d'une Grid :

- **transfert en co-allocation** (Co-Allocation Data Transfer) - le message à transmettre, qui est souvent une donnée répliquée, est découpé en paquets et chaque paquet peut être envoyé depuis des sites de réplication distinct vers la destination. Deux mécanismes *stateless allocation / stateful allocation* qui prennent en compte ou pas l'état du réseau
- **Peer-to-Peer Data Sharing**

Traitement des transactions

Une **transaction** est un ensemble d'opérations de lecture/écriture sur un ensemble de données qui doit se réaliser comme une unique opération en laissant les données (la base de données) dans un état cohérent.
Propriétés d'une transaction (ACID) :

- atomicité : les opérations d'une transaction sont exécutées comme une unique opération
- consistance : la base reste dans un état cohérent à la fin de la transaction
- isolation : la transaction s'exécute indépendamment d'autres opérations sur la base
- durabilité : les mises à jour d'une transaction sont définitives

Traitement des transactions

Dans un environnement distribué et étendu comme une Grid les transactions sur des objets distants sont difficiles à mettre en œuvre car les accès ne sont pas centralisés. On rajoute à cela le problème de la synchronisation des répliques.

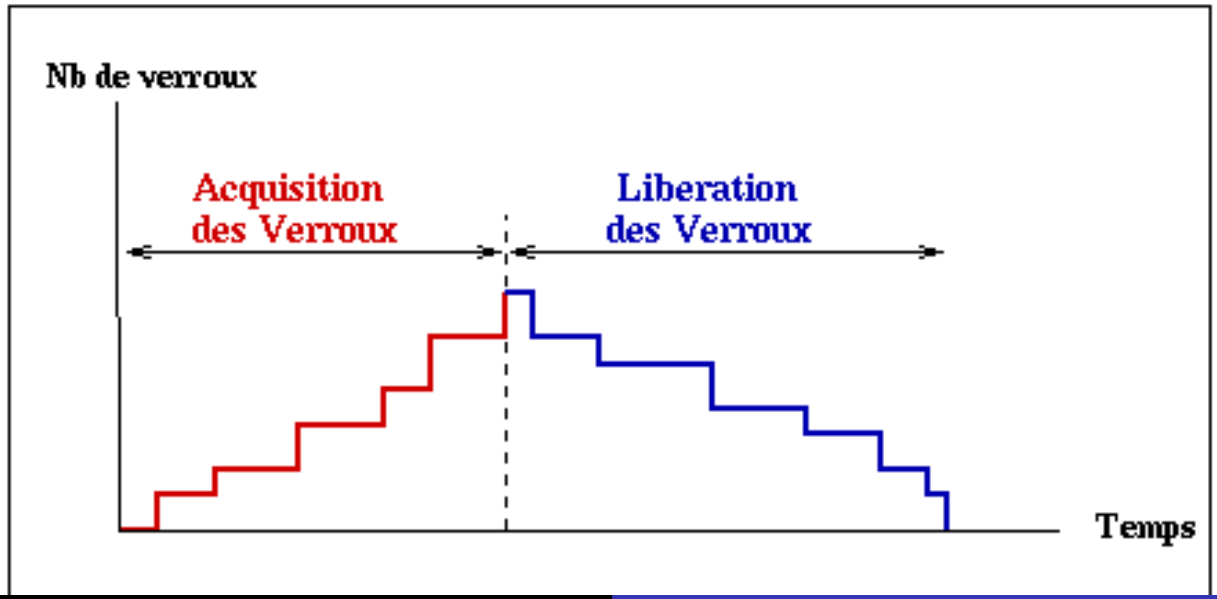
Dans une BD classique (centralisée) on a majoritairement des transactions à une seule phase avec un unique verrou sur les données concernées (tables entières, enregistrement ou champ) et le relâche en fin de transaction.

En environnement distribué on utilise diverses techniques : verrouillage à deux ou trois phases, estampillage, réalisation par agent, etc, mais on ne peux pas assurer toujours toutes les propriétés de la transaction ou parfois on peut générer un inter-blocage.

Traitement des transactions

Un **verrouillage à deux phases** peut assurer la cohérence mais il engendre des temps d'attente très (trop) longs. La principe :

- une phase où la transaction pose les verrous dont elle a besoin
- une phase où la transaction retire tous les verrous sans en poser de nouveaux



Synchronisation des données

Si les données sont uniquement en lecture le problème de la cohérence des répliques ne se pose pas (on lit la même donnée).

Les répliques servent en lecture et aussi en écriture. En cas d'écriture, il y a le risque d'écrire sur les répliques différentes d'une même donnée. Toute écriture devrait s'assurer de l'exclusivité de l'accès à la donnée.

Le consistance des répliques d'une même données peut s'évaluer sur 5 niveaux de -1 à 3 :

- -1 : possibilité des fichiers inconsistants (aucune synchronisation, même au niveau de la même ressource)
- 3 : toutes les répliques sont identiques pour toute donnée

La mise en œuvre dépend du middleware.

Gestion de stockage et d'accès aux données

La gestion de stockage et d'accès aux données est une une composante essentielle de la gestion globale des données dans une Grid.

Pratiquement elle est implémentée de manière très diverse d'un middleware à un autre.

La plupart des DFS (Distributed File System) sont basé sur le principe de la représentation arborescente de l'ensemble de fichiers de la Grid.

Intégration des données

L'**intégration** a pour but de collecter et combiner les données depuis plusieurs sources afin d'en offrir une vue uniforme.

L'intégration se compose d'une succession de pas : découverte, accès, transport (transfert), analyse et synthèse.
Elle se base autant sur le service de réplication, que sur le service de méta-données, que sur le service de stockage.

Il est fort probable que lors de la phase d'analyse les données s'avèrent ayant des formats différents.
L'analyse des données peut être faite au niveau statistique et au niveau sémantique.

Allocation de charge

L'**allocation de charge** est un module qui réalise une association entre les demandes de l'utilisateur (de traitement et de données) et des ressources présentes dans le Grid. On indique donc quelle ressource est associée à quelle demande.

L'allocation de charge peut se faire de manière :

- **statique** : lors de la phase d'initiation du travail : on décide du placement unique et définitif de chaque tâche
- **dynamique** : on décide lors de l'exécution soit d'un nouveau placement des tâches existantes, soit du placement des nouvelles tâches apparues (si application dynamique)

Termes utilisés : allocation de charge = placement = scheduling.

Allocation de charge

La **régulation de charge** est un module (optionnel) qui est capable de transmettre à d'autres ressources una partie de sa charge

Une allocation dynamique de charge est dite efficace si le temps pris par l'allocation (information, décision, transfert) est couvert par de gain de temps obtenu dans la réalisation de l'application.

$$T_{execution\_avec\_allocation\_dynamique} + T_{allocation\_dynamique} < T_{execution\_sans\_allocation\_dynamiqu}$$

Termes utilisés : régulation de charge = ré-équilibrage de charge = load balancing.
charge = load, tâche = task / job

Allocation de charge

La distribution de charge est un cas particulier de régulation qui vise à assurer qu'l n'y a pas de ressources non utilisées.
Difficultés de l'allocation de charge :

- allocation statique : problème difficile (NP-complet) pour des demandes non-homogènes et plus de 2 ressources demandés, plusieurs critères à optimiser (temps d'exécution, temps de transfert, temps de communication (throughput) pendant l'application)
- information précise et cohérente sur la charge des ressources cible pour le régulation de charge et l'allocation des application dynamique
- l'allocation dynamique / régulation se font sur la base d'une information partielle et indépendamment d'autre éventuelle demande de même type

Allocation de charge

Algorithmes pour la résolution de l'allocation statique :

- par partitionnement (bi-partitionnement)
- recherche itérative : permutation aléatoire jusqu'à minimum local
- application des algorithmes de type recuit simulé, recherche tabou, algorithme génétique

Pour l'allocation dynamique (et régulation de charge) on a :

- mécanismes de : initiation (le moins chargé / le plus chargé) , information (services de monitoring ou informations collectées à la demande) et décision (sur sa propre information de charge et celles collectées)
- stratégie : comment faire ?

Allocation de charge

Les stratégies possibles d'allocation dynamique et régulation :

- choix aléatoire de la destination (souvent pris comme référence de comparaison)
- choix de la cible avec les meilleurs possibilités affichées (conflits possibles)
- par collecte d'information élargie (sous-partie de Grid) et heuristiques spécifiques
- par diffusion avec les voisins directs

Allocation de charge avec des contraintes de données

Si les tâches font appel à des données avec réplication, la politique d'allocation doit en tenir compte.

Un services d'allocation dans un environnement faiblement couplé sera composé de :

- External Scheduler (ES) : chaque site contient un ES où les utilisateurs soumettent leurs tâches. On doit connaitre la charge sur les site distant et l'adresse des données
- Local Scheduler (LS) : pour chaque tâche envoyé depuis un site distant le LS lui affecte une ressource locale ; il a également une vision complète de son propre domaine
- Dataset Scheduler (DS) : garde des information sur la "popularité" des données locales et peut les transmettre au service de réplication en lui demandant aussi des nouvelles

Allocation de charge avec des contraintes de données

Les stratégies utilisées par le ES : JobRandom, JobLeastLoaded, JobDataPresent et JobLocal.

Les stratégies employées par le DS :

- DataDoNothing - aucune réplique demandée
- DataRandom - si le seuil d'accès est dépassé on demande une réplication sur un site aléatoire
- DataLeastLoaded - si besoin est de réplication celle-ci se fait sur le site le moins chargé

Monitoring

Le **service de monitoring** ou service d'information est censé collecter, traiter et mettre à disposition des informations physiques et logiques sur les ressources physiques de la Grid : unités de calcul, réseau, stockage, périphériques, etc...

Le service de monitoring a une tâche complexe car

- les informations traitées doivent être de qualité et celles-ci périment vite
- la collecte des informations doit se faire en respectant les règles d'authentification et d'autorisation
- les informations synthétiques fournies se font aussi en authentifiant le demandeur

les information traitées

Authentification, autorisation et contrôle d'accès

Dans une Grid l'authentification, l'autorisation des utilisateurs et le contrôle d'accès doivent prévenir l'utilisation abusive/malveillante de ressources et les problèmes de sécurité.

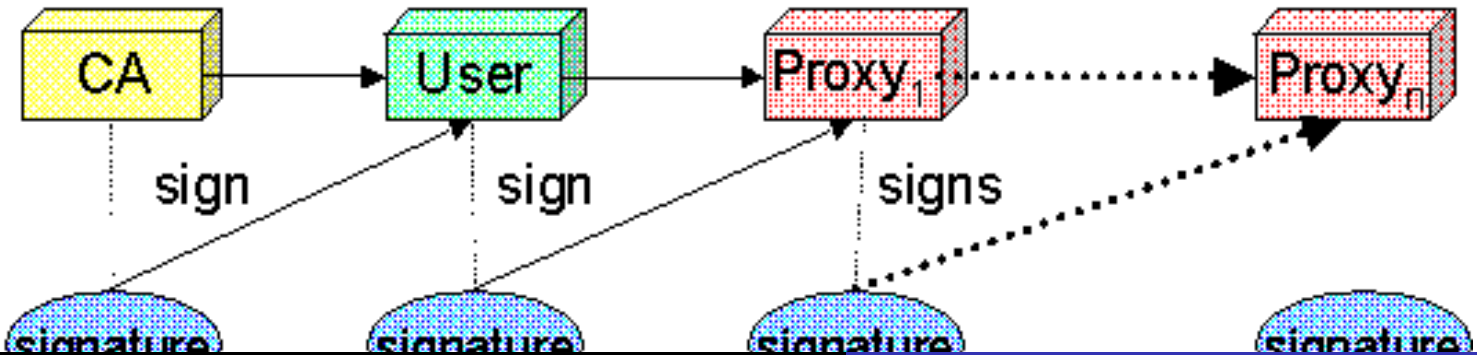
Authentification = on identifie un utilisateur comme étant bien quelqu'un qui a un droit dans la Grid.
Autorisation = on identifie si un utilisateur a un droit sur une ressource.
L'accès d'un utilisateur à une donnée lui ouvre automatiquement le droit d'accès à toutes les répliques. Pour cette raison les répliques sont disséminées dans des nœuds proches du nœud d'accès de l'utilisateur.

Dans une Grid on fait fonctionner le principe de la délégation

GSI

Grid Security Infrastructure (GSI) est une infrastructure basée sur la notion de clé privé / clé publique et celle de signature électronique.

La connexion se fait SSO (Sigle Sign-On) et un certificat est délivré. Ce certificat contient un clé publique et Une relation de confiance s'établit d'amblé avec les proxies de la Grid. Le certificat devient délégable surtout au niveau des proxies. Il est possible aussi de mettre en place des clés privées (délégables aussi).

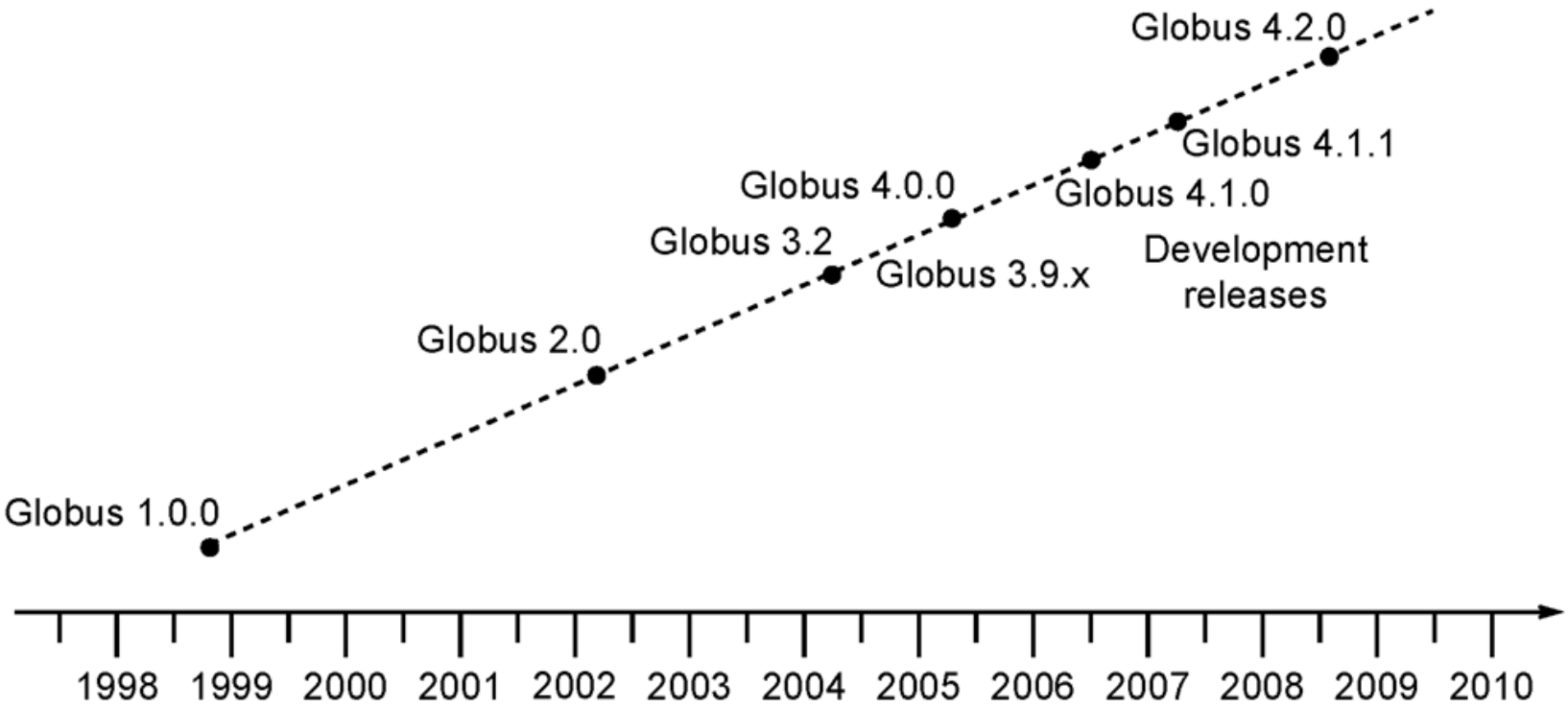


Grid Middleware

- **Globus**
- gLite
- UNICORE

Globus

Middleware réalisé au fil du temps suite à un projet de recherche académique aux Etats Unis, nouvelle récente : 02.15.2011 Announcing Release of GT 5.0.3.



Globus - ses standards

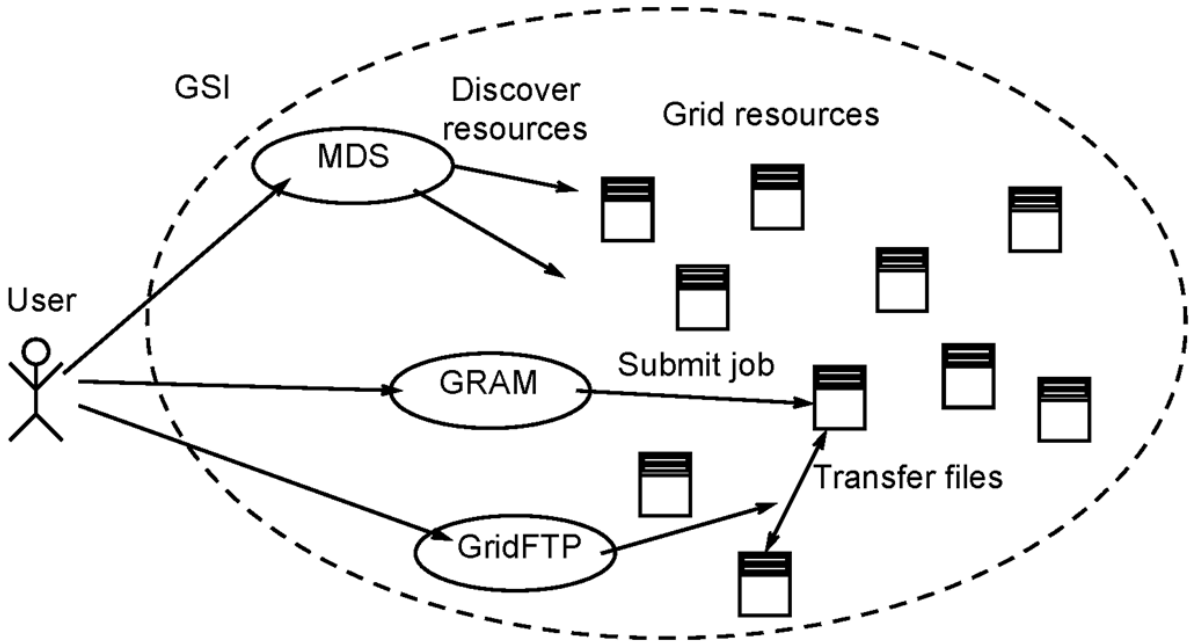
- Open Grid Services Architecture (OGSA)
- Open Grid Services Infrastructure (OGSI)
- Web Services Resource Framework (WSRF)
- Job Submission Description Language (JSDL)
- Distributed Resource Management Application API (DRMAA)
- WS-Management
- WS-BaseNotification
- SOAP
- WSDL
- Grid Security Infrastructure (GSI)

Globus

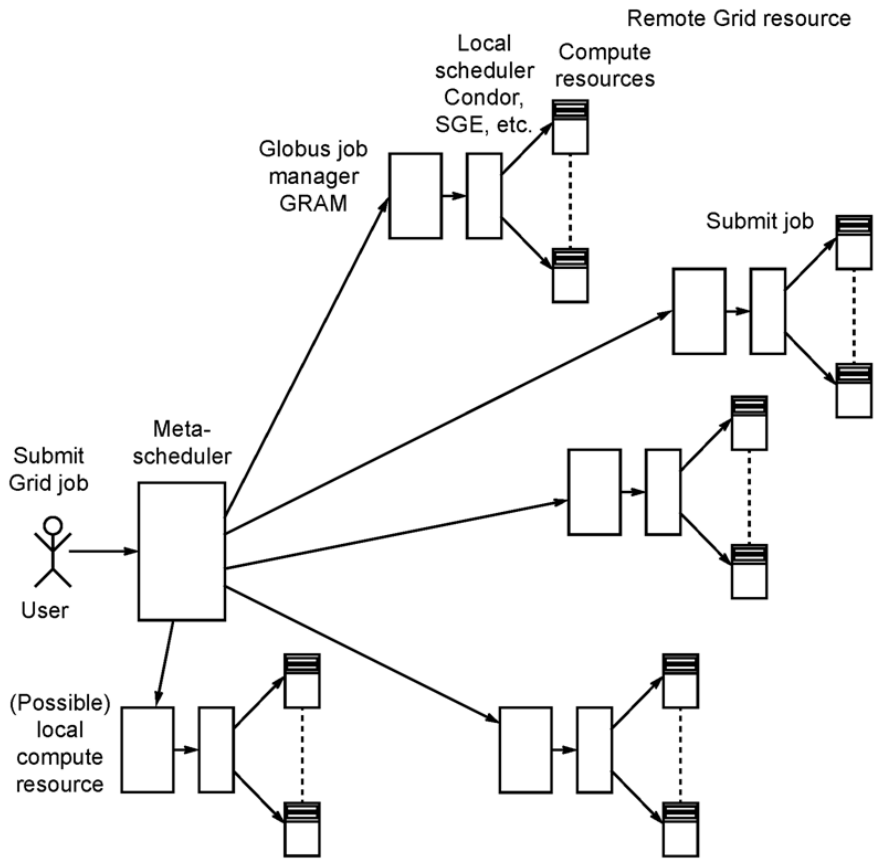
Les services essentiels qui le composent :

- gestion de ressources : Grid Resource Allocation & Management Protocol (GRAM)
- service de monitoring : Monitoring and Discovery Service (MDS)
- service de sécurité : Grid Security Infrastructure (GSI)
- service de gestion et mouvement de données : Global Access to Secondary Storage (GASS) and GridFTP

Globus


GSI
Discover resources
Grid resources
MDS
User
GRAM
Submit job
Transfer files
GridFTP

Globus - le scheduler



Projets Grid Computing

- projet SETI@home (1999 - 2001) - calcul distribué
- projet européen DataGrid (2002-2004) :
  `http://eu-datagrid.web.cern.ch/eu-datagrid/`
- projet européen GRID-TLSE (2002-2005) - algèbre des matrices creuses
- divers projets aux Etats Unis (cf. Conférence oct. 2010 de Barry Wilkinson)