Panorama des paradigmes de parallélisation pour les architectures exascales

P.-Fr. Lavallée – 23 mai 2019

Architectures exascales

Deux types d'architectures exascales sont annoncées :

Hybrides accélérées

- Perlmutter, NERSC
- Aurora A21, Argonne
- Frontier, Oak Ridge
- Sugon, Shuguang
- Tianhe-3, NSC Tianjin
- EuroHPC/EPI ?

Homogènes Manycores

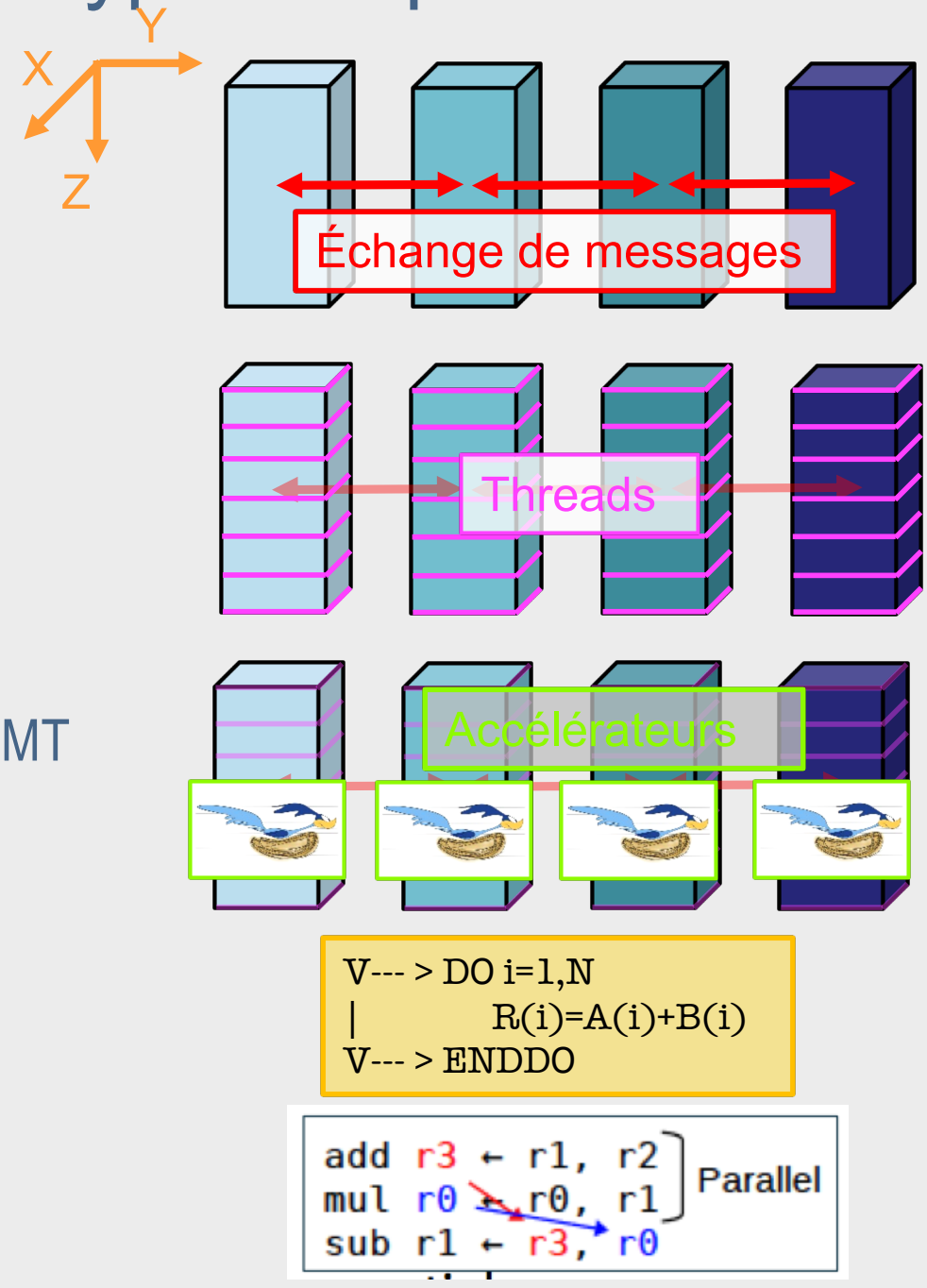
- Sunway, NSC Jinan
- Post-K, Riken

Contraintes communes pour une utilisation efficace :

- Parallélisme massif multi-niveaux
- Hiérarchie/Organisation mémoire complexe

Typologie des différents types de parallélisme

- ***Domain Parallelism***
  - Parallélisation inter-nœuds par échange de messages
- ***Shared Memory Parallelism***
  - Parallélisation intra-nœud mémoire partagée type threads légers
- ***Offload (if accelerated architecture)***
  - Parallélisation accélérateur de type SIMT
- ***Data Parallelism***
  - Vectorisation SIMD
- ***Instruction Level Parallelism***
  - Exécution concurrentes d'instructions indépendantes (*hardware*)

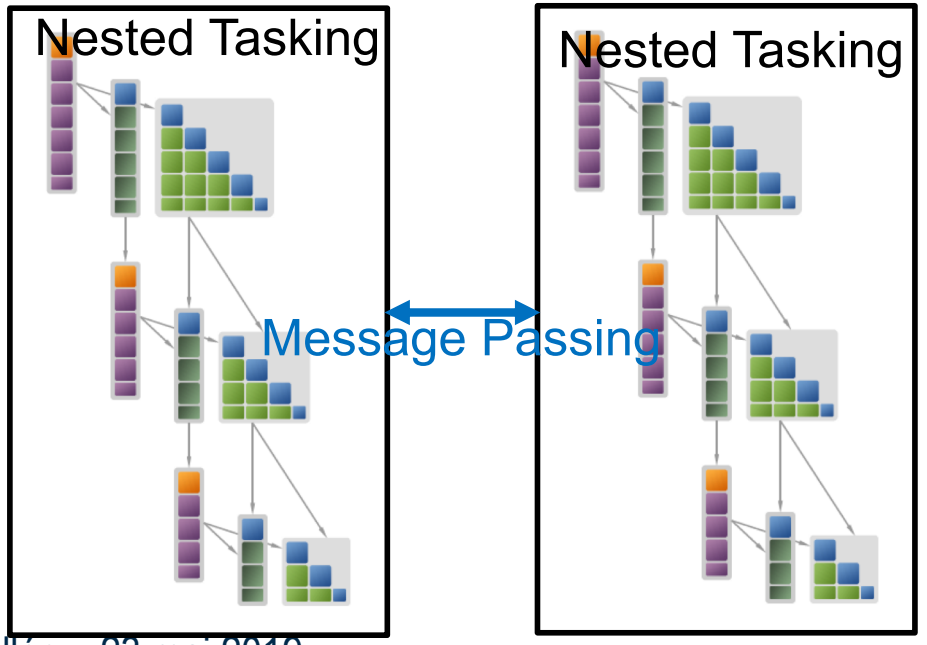


Le paradigme de parallélisation idéal

- Portable
  - Agnostique de l'architecture
  - Non propriétaire
- Performant
  - Gestion efficace du parallélisme à tous les niveaux
    - ✓ Intra-nœud (vectorisation, threads, tâches, NUMA, hiérarchie et affinité mémoire, accélérateurs)
    - ✓ Inter-nœud (asynchrone, extensible, malléable/souple)
- Optimiser la productivité (rapport perf. versus le temps de développement)
  - Accessible au non expert
  - Ecosystème associé (outils de débogage et analyse de perf. simple d'utilisation)
  - Peu intrusif dans le code
- Pérenne
  - Normalisé ou norme de facto
  - Base importantes d'utilisateurs et de développeurs
- Compatible avec les langages du HPC (Fortran, C, C++, Python)
- Robuste et stable

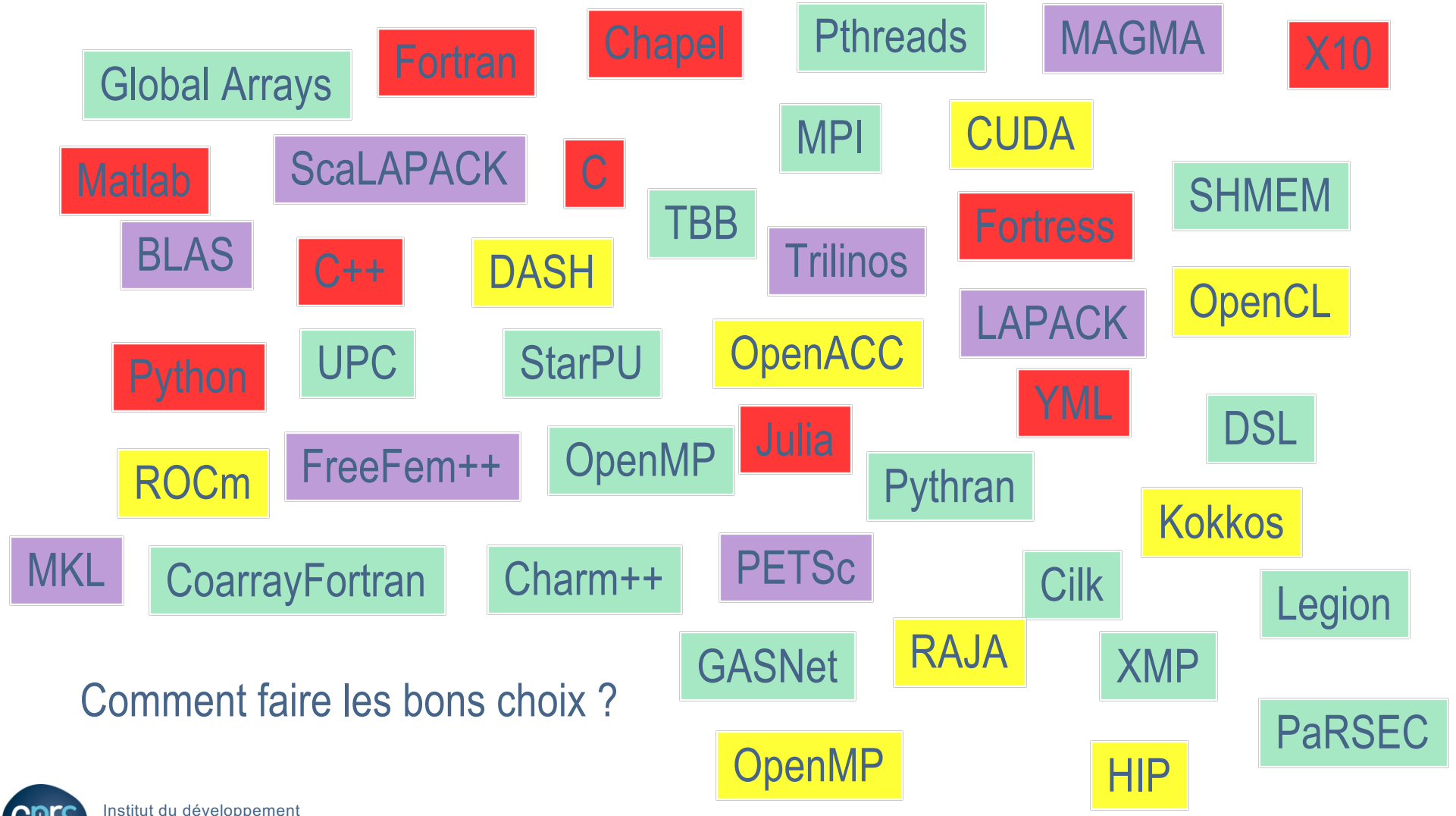
Le paradigme de parallélisation idéal

- Aujourd'hui, aucun paradigme de parallélisation ne répond à l'ensemble de ces critères. Il ne fait aucun doute qu'aucun nouveau paradigme n'y répondra d'ici à l'arrivée des premières architectures exascales.
- Si on dissocie les différents niveaux de parallélisme, alors pour chaque niveau, on peut trouver un ou plusieurs modèles de parallélisation qui respectent l'essentiel des ces contraintes.
- À horizon des supercalculateurs exaflopiques, la seule stratégie viable est donc de **mixer plusieurs paradigmes de parallélisation**, chacun s'adressant spécifiquement à un des niveaux de parallélisme décrit précédemment.
- C'est par exemple le choix fait dans le projet ECP (Exascale Computing Project - www.exascaleproject.org) pour SLATES (réécriture de LAPACK et ScaLAPACK pour les architectures exascales)



Langages et paradigmes de parallélisation…

L'écosystème de programmation et de parallélisation est riche, très riche, beaucoup trop riche…



Comment faire les bons choix ?

Disponibilité annoncée

	MPI	OpenMP	XMP	CAF	CUDA	OneAPI	UPC	HIP	OpenCL	OpenACC
Post-K	■	■	■	■						
Perlmutter	■	■			■					
Frontier	■	■		■			■	■		
Aurora	■	■				■				
Tianhe-3	■	■							■	
Sunway	■	■								■
ECP	■	■								

	Charm++	GASNet	Chapel	RAJA	Kokkos	Legion	PaRSEC	UPC++
Post-K								
Perlmutter								
Frontier	■	■	■					
Aurora								
Tianhe-3								
Sunway								
ECP		■		■	■	■	■	■

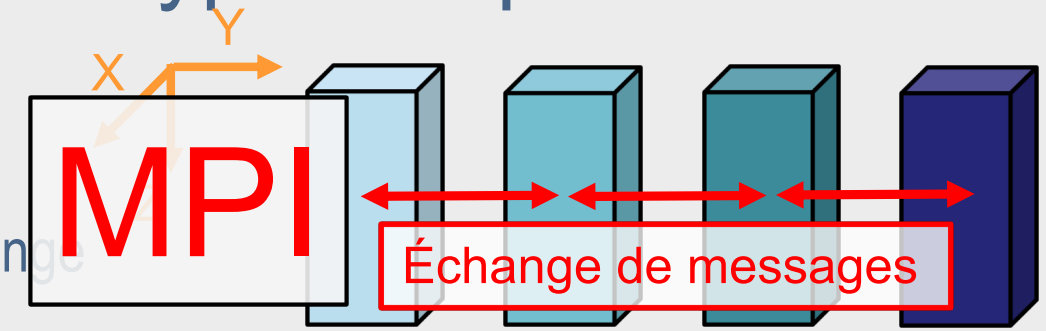
Portability Across DOE Office of Science HPC Facilities

- https://performanceportability.org/

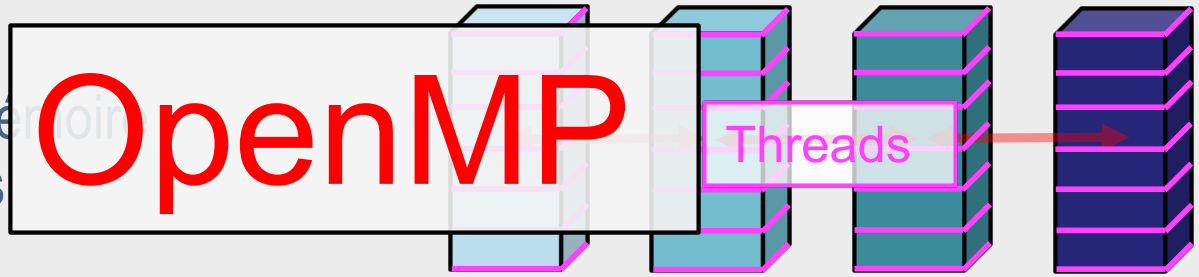
Approach	Benefits	Challenges
Libraries	Highly portable, not dependent on compiler implementations.	Many GPU libraries (e.g. CUFFT) are C only (requiring explicit interfaces to use in FORTRAN) and don't have common interfaces. May lock-in data layout. In many cases libraries don't exist for problem.
OpenMP 4.5	Standardized. Support for C, C++, FORTRAN and others. Simple to get started.	Limited expressibility (particularly on GPUs). Lacks "views". Reliant on quality of compiler implementation - which are generally immature on both GPU and CPU systems.
OpenACC	Standardized. Support for C, C++, FORTRAN.	Limited support in compilers, especially free compilers (e.g. GNU).
Kokkos	Allows significant expressibility (particularly on GPUs.)	Only supports C++. Vector parallelism often left-out on CPUs.
Raja	Allows incremental enhancements to codes. Many back-ends.	Only supports C++. Lacks data "views" for more advanced portability requirements.
DSLs	Highest expressibility for appropriate problems	Limited to only a small number of communities. Needs to be maintained and supported for new architectures.

Typologie des différents types de parallélisme

- ***Domain Parallelism***
  - Parallélisation inter-nœud par échange de messages



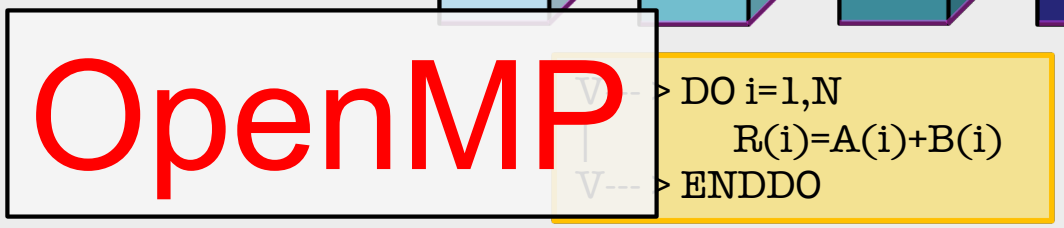
- ***Shared Memory Parallelism***
  - Parallélisation intra-nœud mémoire partagée type threads légers



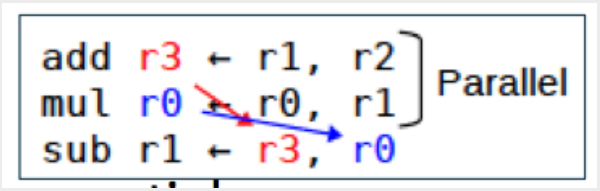
- ***Offload (if accelerated architecture)***
  - Parallélisation accélérateur de type SIMT

OpenMP-OpenACC-Kokkos

- ***Data Parallelism***
  - Vectorisation SIMD



- ***Instruction Level Parallelism***
  - Exécution concurrentes d'instructions indépendantes (*hardware*)



Les bibliothèques

De nombreuses bibliothèques sont disponibles dans des domaines très variés (systèmes linéaires, problèmes aux valeurs propres, optimisation, FFT, EDP, AMR, etc.). Lorsque cela est possible, c'est **LA voie à privilégier** :

- des optimisations de bas niveau et des performances excellentes impossibles à obtenir à la main… ;
- gestion implicite du parallélisme et des accélérateurs si la bibliothèque a implémenté ces fonctionalités ;
- portabilité d'une machine à une autre si la bibliothèque n'est pas propriétaire et est disponible ;
- optimisation de l'investissement, plus rapide à implémenter, à maintenir ;
- généralement simple à utiliser, même si on est parfois contraint d'utiliser les structures associées ;
- robustesse, stabilité numérique, MAJ régulière avec des méthodes / algorithmes récents, correction de bogues, etc. ;
- disponibilité de documentations, forums d'aide, etc. ;
- interface avec les langages de calcul scientifique standards (Fortran/C/C++).

MPI

- MPI est une bibliothèque standardisée d'échange de messages, librement disponible, qui vise performance et portabilité sur une grande variété d'architectures (à mémoire distribuée, à mémoire partagée, cluster de SMP, etc.)
- Le standard MPI est disponible : www.mpi-forum.org (868 pages !)
- Régulièrement enrichi avec de nouvelles fonctionnalités au fil des versions
- Une application parallélisée avec MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communicant via des appels à des sous-programmes de la bibliothèque MPI :
  - gestion de l'environnement MPI ;
  - communications point à point ;
  - communications collectives ;
  - communications RMA ou OSC ;
  - mémoire partagée au sein d'un nœud ;
  - topologies et types dérivés ;
  - entrées-sorties parallèles avec MPI-IO.

MPI – Environnement

```
program MonCodeMPI
use  mpi
call MPI_INIT(code)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code) ! # de processus MPI
call MP                                                          cessus
...
call MP
end pro
```

Attenti
pour le
()

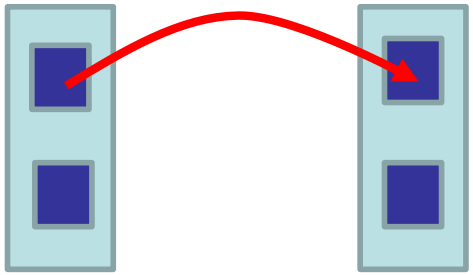
MPI_Init_thread

```
int MPI_Init_thread(int *argc, char *((*argv)[]),
                    int required, int *provided)
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
```

Le niveau de support demandé est fourni dans la variable *required*. Le niveau effectivement obtenu (et qui peut être moindre que demandé) est récupéré dans *provided*.

- `MPI_THREAD_SINGLE` : seul un *thread* par processus peut s'exécuter
- `MPI_THREAD_FUNNELED` : l'application peut lancer plusieurs *threads* par processus, mais seul le *thread* principal (celui qui a fait l'appel à `MPI_Init_thread`) peut faire des appels MPI
- `MPI_THREAD_SERIALIZED` : tous les *threads* peuvent faire des appels MPI, mais un seul à la fois
- `MPI_THREAD_MULTIPLE` : entièrement *multithreadé* sans restrictions

MPI – Communications point à point (P2P)

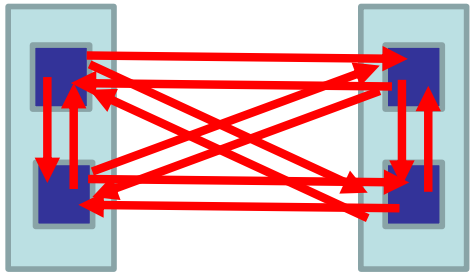


Mode	*Bloquant*	*Non bloquant*
Envoi standard	`MPI_SEND()`	`MPI_ISEND()`
Envoi synchrone	`MPI_SSEND()`	`MPI_ISSEND()`
Envoi *bufferisé*	`MPI_BSEND()`	`MPI_IBSEND()`
Réception	`MPI_RECV()`	`MPI_IRECV()`

Machine	*Niveau*
Blue Gene/Q, PAMID_THREAD_MULTIPLE=0	32%
Blue Gene/Q, PAMID_THREAD_MULTIPLE=1	100%
Ada+POE	37%
Ada+POE MP_CSS_INTERRUPT=yes	85%
Ada+IntelMPI I_MPI_ASYNC_PROGRESS=no	4%
Ada+IntelMPI I_MPI_ASYNC_PROGRESS=yes	94%

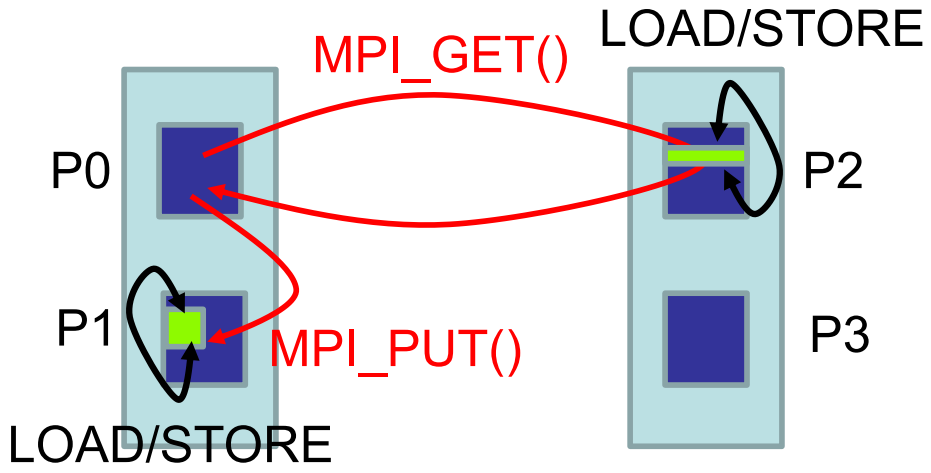
MPI – Communications collectives

- Les communications collectives permettent de faire en une seule opération une série de communications point à point
- Une communication collective concerne toujours tous les processus du communicateur indiqué. Attention aux risques de *deadlock* (si il manque un processus à l'appel) !



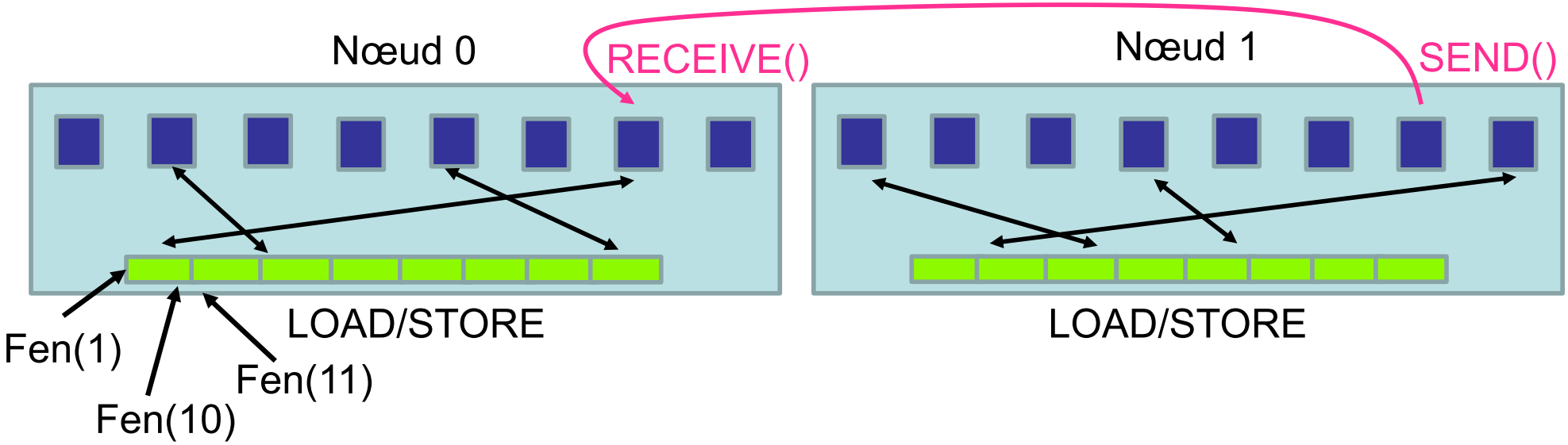
- Depuis MPI-3, il existe des versions non-bloquantes des primitives de communications collectives.
- Préfixée par I (*immediate*) : MPI_IREDUCE(), MPI_IBCAST(), MPI_IBARRIER(), etc.
- Ces primitives non bloquantes permettent d'implémenter un recouvrement calculs / communications
- Attente avec les appels MPI_WAIT(), MPI_TEST() et leurs variantes
- Attention, suivant la machine, l'implémentation MPI et l'environnement d'exécution, cette fonctionnalité peut être implémentée de façon plus ou moins efficace !

MPI – Communications mémoire à mémoire



- Les communications mémoire à mémoire (RMA ou OSC pour *Remote Memory Access* ou *One Sided Communications*) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement. Le processus cible n'intervient donc pas lors du transfert.
- Nécessite la création d'une fenêtre mémoire et d'un mécanisme spécifique de synchronisation
- Intérêt : optimisation des performances et simplification du codage de certains algorithmes

MPI – Utilisation de la mémoire partagée



- Depuis MPI-3, on peut créer une fenêtre en mémoire partagée pour les processus d'un même nœud
- Les données de la fenêtre mémoire partagée sont accessibles en lecture / écriture (via les mécanismes de *load / store* std) par tous les processus du nœud
- Intérêts : performances accrues, simplicité de programmation, réduction de l'empreinte mémoire…
- On peut ainsi implémenter une parallélisation hybride MPI-MPI, avec un premier niveau MPI intra-nœud utilisant la mémoire partagée et un deuxième niveau MPI (échange de messages) entre les nœuds

OpenMP

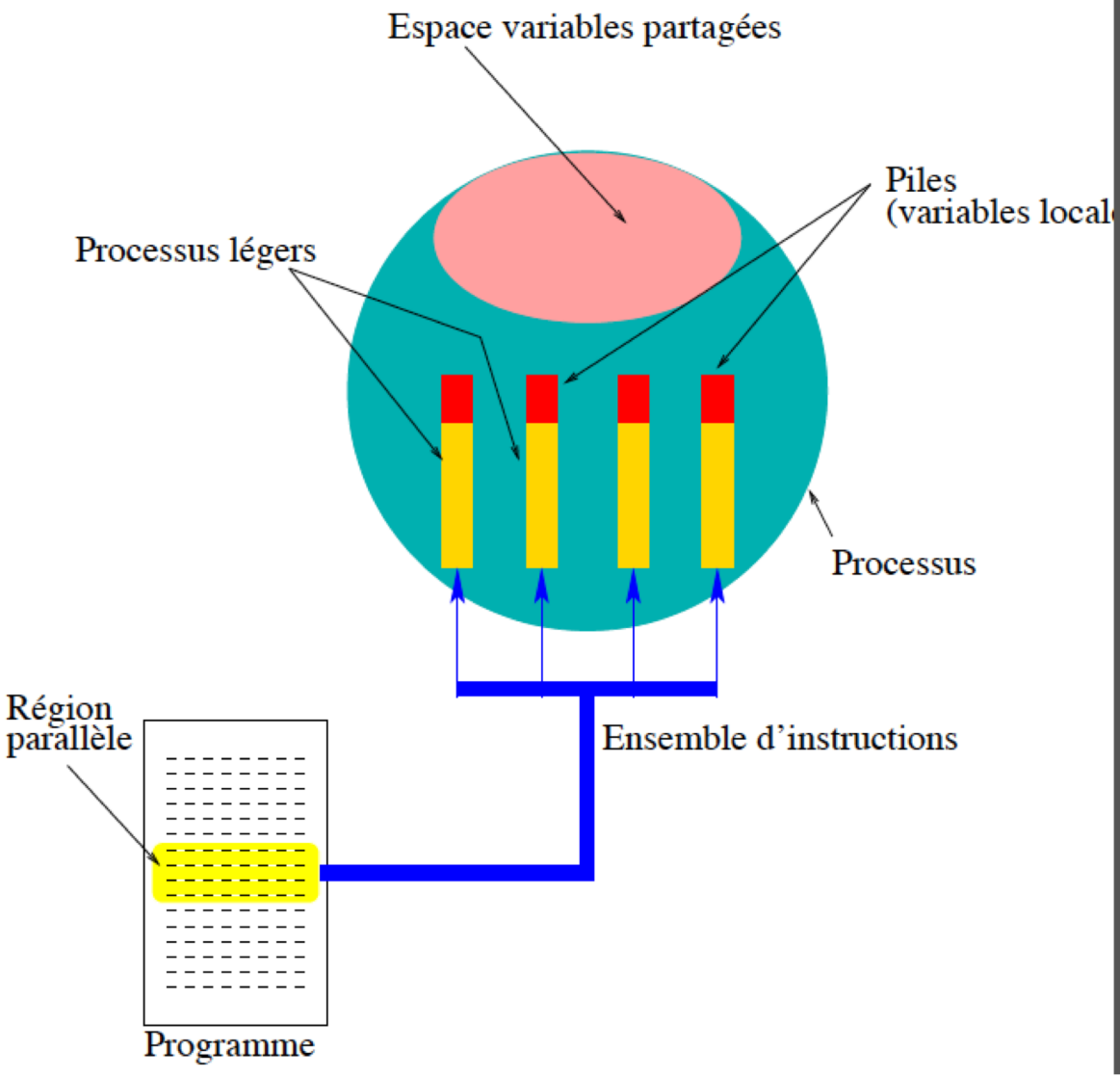
OpenMP est u
C++ ou Fortra

- OpenMP es
  de variable
- Spécificatio
  - création/
  - gestion c
  - partage
  - synchror
    (ATOMI
  - vectorisa
  - création
  - support

e de codes C,

de fonctions et
e
!)

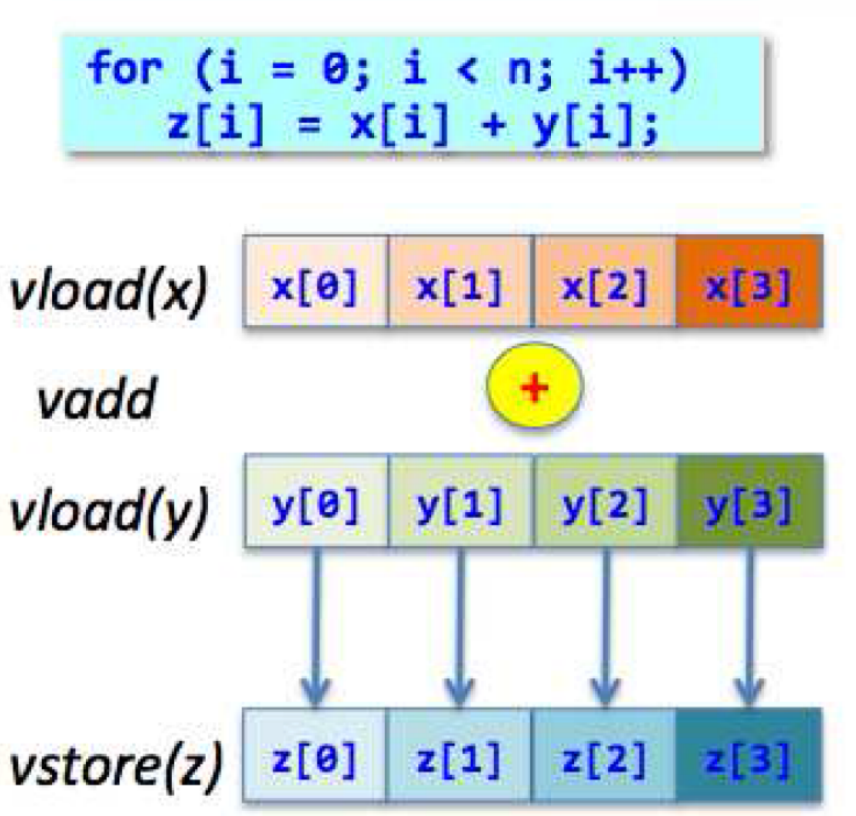
SECTION, …) ;
mutuelle



OpenMP – Vectorisation SIMD

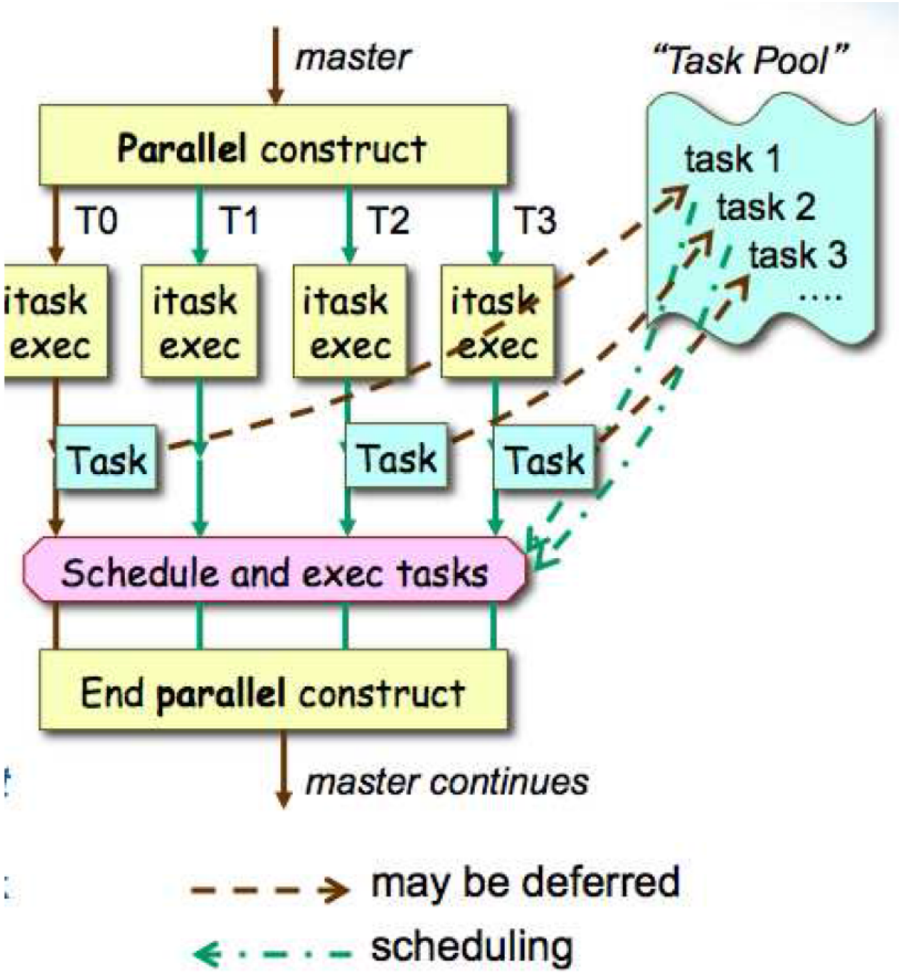
- SIMD = *Single Instruction Multiple Data*
- Une seule instruction / opération agit en parallèle sur plusieurs éléments
- Depuis la version 4.0 OpenMP offre la possibilité de gérer la vectorisation SIMD de façon portable et performante en utilisant les instructions vectorielles (AVX, AVX2, AVX512, SVE) disponibles sur l'architecture cible

```
somme=0
! $OMP SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
```



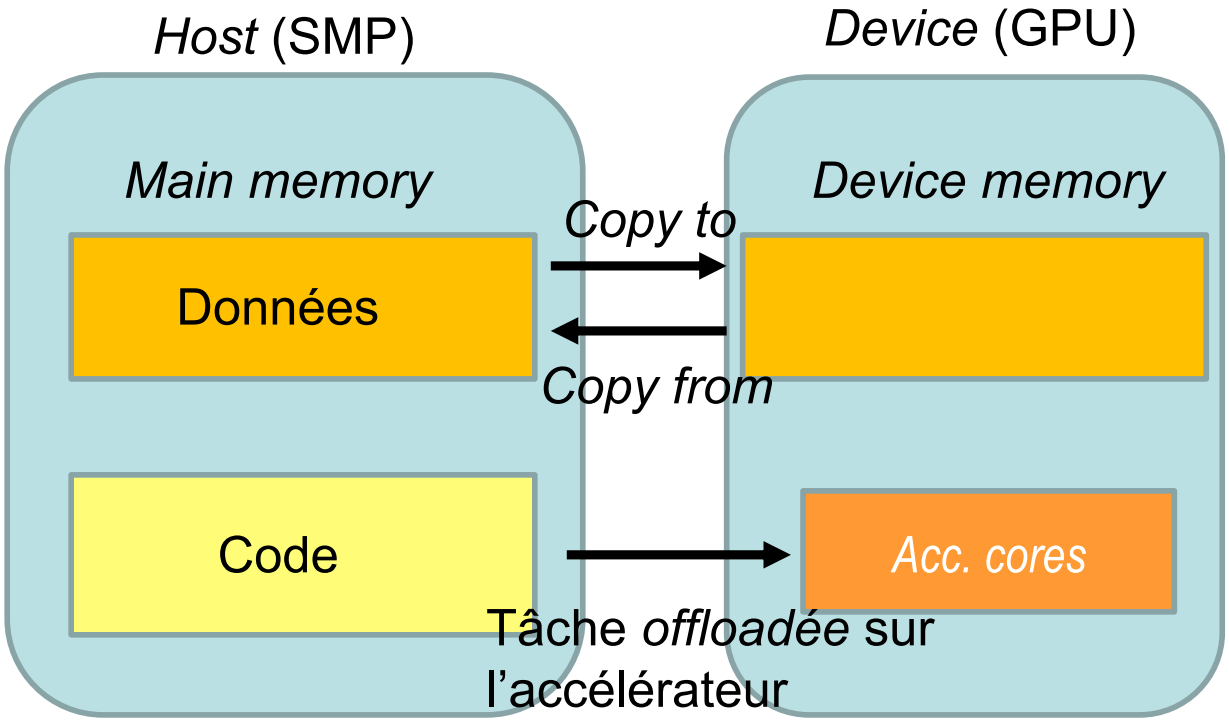
OpenMP – Tâches explicites

- Depuis la version 4.0 OpenMP permet de gérer des constructions de génération et de synchronisation de tâches explicites (avec ou sans dépendances).
- Une tâche OpenMP est constituée d'une instance de code exécutable et de ses données associées. Elle est exécutée par un thread.
- La clause DEPEND permet de gérer des dépendances entre des tâches explicites
- Plusieurs types de synchronisation sont disponibles via les directives TASKWAIT et TASKGROUP/END TASKGROUP



OpenMP – Offload accélérateurs

- Depuis la version 4.0 OpenMP propose un modèle d'exécution de type *offload* (*data*+code) pour les accélérateurs
- C'est à l'utilisateur de :
  - définir les parties de codes (*TARGET*) à exécuter sur l'accélérateur ;
  - gérer le transfert des données (*MAP*) depuis/vers (*TO/FROM*) la mémoire de l'accélérateur à défaut d'utiliser la mémoire managée (OpenMP 5.0)



OpenMP – *Offload* accélérateurs

```
void vadd_openmp(float *a, float *b, float *c, int size)
{
   #pragma omp target map(to:a[0:size],b[0:size],size) map(from: c[0:size])
   {
      int i;
      #pragma omp teams distribute parallel for
      for (i = 0; i < size; i++)
         c[i] = a[i] + b[i];
   }
}
```

- Les appels *offload* d'OpenMP peuvent être vus comme des tâches qui s'exécutent sur l'accélérateur et qui peuvent être asynchrones (clause NOWAIT) et/ou avec des dépendances optionnelles (clause DEPEND).
- Un soin particulier doit être porté aux transferts des données entre les mémoires de l'*host* et du *device,* sous peine d'obtenir des performances décevantes
- Dans la pratique, le but est de ne pas dépasser une dégradation de plus de 10 à 20 % par rapport à une approche bas niveau de type CUDA…

OpenACC – *Offload* accélérateurs

```
void vadd_openmp(float *a, float *b, float *c, int size)
{
   #pragma acc data copyin(a[0:size],b[[0:size]) copyout(c[0:size])
   {
      int i;
      #pragma acc parallel loop
      for (i = 0; i < size; i++)
         c[i] = a[i] + b[i];
   }
}
```

- Le principe est le même que l'*offload* OpenMP
- Spécifications complètes : https://www.openacc.org/
- Les compilateurs sont plus matures et stables que ceux d'OpenMP
- D'un point de vue utilisateur, on espère une fusion OpenACC / OpenMP *offload* pour ne plus avoir qu'un seul jeu de directives à gérer
- En attendant, OpenACC est une alternative temporaire lorsque les fonctionnalités ou les performances font défaut à l'approche *offload* OpenMP

Conclusions

- Les architectures exascales exhiberont une complexité importante (hiérarchie mémoire, hétérogénéité) et un parallélisme extrême (many-cœurs, SIMD, SIMT, etc.)
- Pour les utilisateurs, quelle que soit l'architecture cible, un travail conséquent d'adaptation des codes de calcul sera nécessaire pour obtenir des performances :
  - exposer plus de parallélisme intra-nœud dans les applications ;
  - augmenter le caractère vectoriel SIMD des applications ;
  - prendre en compte les hiérarchies et les aspects NUMA des différents niveaux de mémoire disponibles (affinité mémoire, affinité processeur, *binding*, …) ;
  - utiliser plusieurs niveaux de parallélisme, potentiellement avec des paradigmes différents, au sein des applications
- Les outils et les environnements de développement arrivent à maturité, mais il reste à adapter les applications de manière pérenne, un travail qui peut prendre plusieurs années…

« Good programming practices are more important than good programming models »

Programming for Exascale Computers

William Gropp Fellow, IEEE and Marc Snir Fellow, IEEE