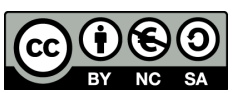
INSTITUT DU
DÉVELOPPEMENT ET DES
RESSOURCES EN
INFORMATIQUE
SCIENTIFIQUE

Programmation hybride MPI-OpenMP

Rémi Lacroix - Dimitri Lecas

CNRS — IDRIS

v4.0 9 septembre 2024



Plan I

Plan II

Introduction

Loi de Moore

Enoncé de la loi de Moore

La loi de Moore dit que le nombre de transistors que l'on peut mettre à un coût raisonnable sur un circuit imprimé double tous les 2 ans.

Consommation électrique

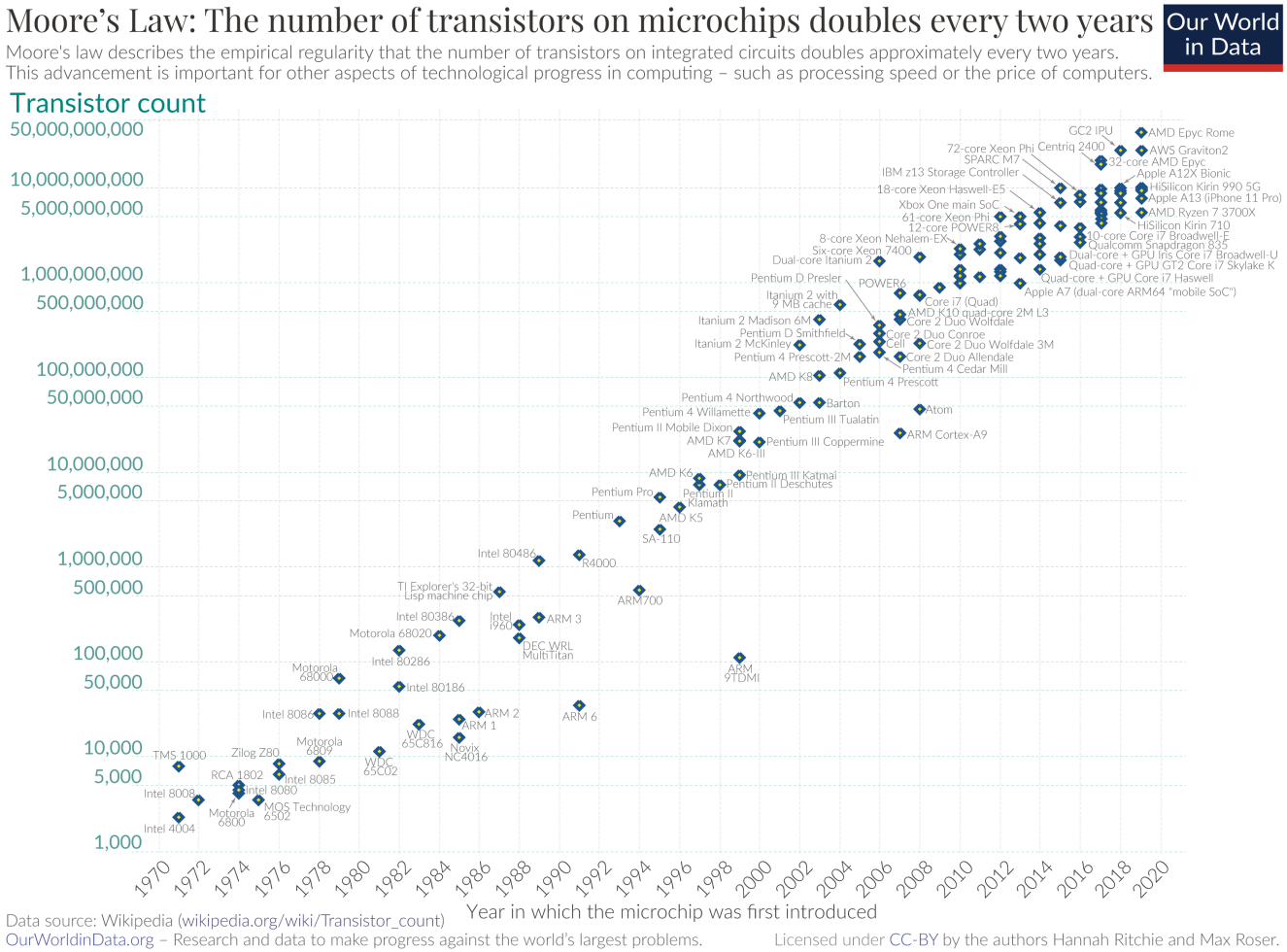
- Puissance électrique dissipée = $C \times frequence \times voltage^2$ ($\propto frequence^3$).
- Puissance dissipée par cm^2 limitée par le refroidissement.
- Coût de l'énergie.

Loi de Moore et consommation électrique

- La fréquence des processeurs n'augmente plus en raison de la consommation électrique prohibitive (fréquence maximale bloquée autour de 3 GHz depuis 2002-2004).
- Le nombre de transistors par puce continue à doubler tous les 2 ans.

=> le nombre de cœurs par puce augmente (les Intel Sapphire Rapids ont jusqu'à 60 cœurs, les AMD EPYC 96 cœurs.

Loi de Moore



By Max Roser, Hannah Ritchie - https://ourworldindata.org/uploads/2020/11/Transistor-Count-over-time.png, CC BY 4.0

Le *memory wall*

Causes

- Les débits vers la mémoire augmentent moins vite que la puissance de calcul des processeurs.
- Les latences (temps d'accès) de la mémoire diminuent très lentement.
- Le nombre de cœurs par barrette mémoire augmente.

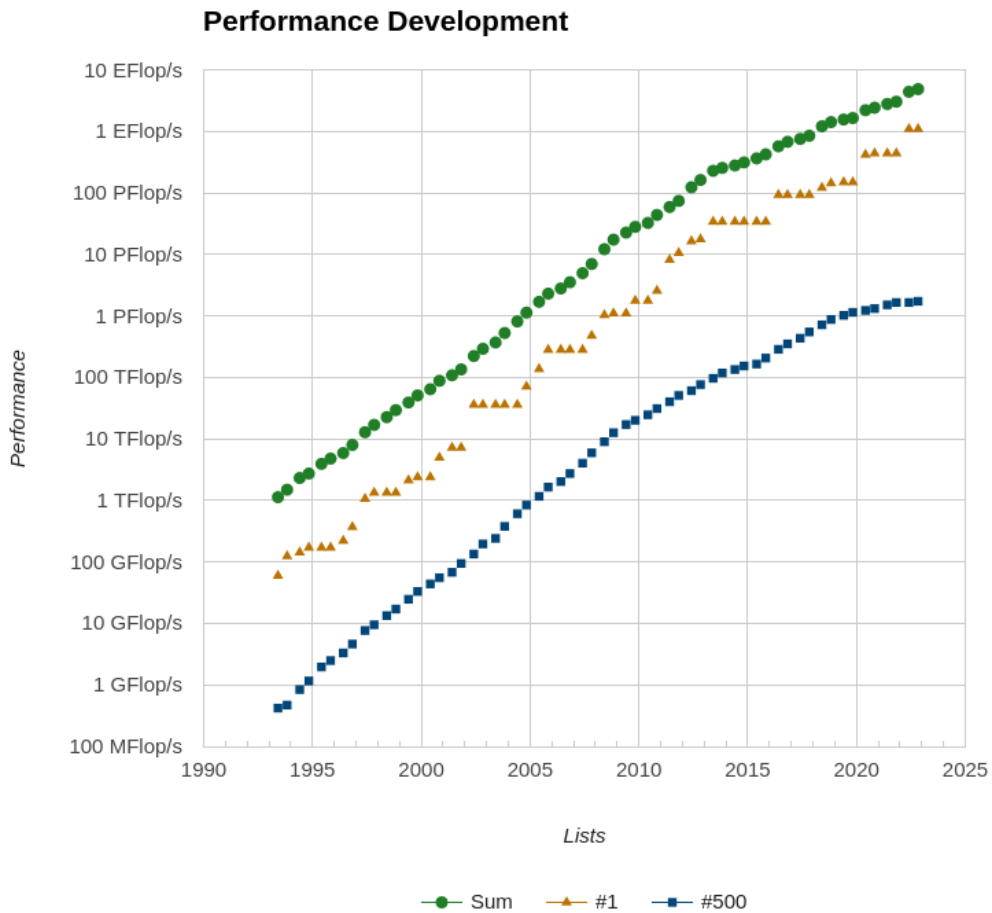
Conséquences

- L'écart entre les performances théoriques des cœurs et la mémoire se creuse.
- Les processeurs passent de plus en plus de cycles à attendre les données.
- Il est de plus en plus difficile d'exploiter la performance des processeurs.

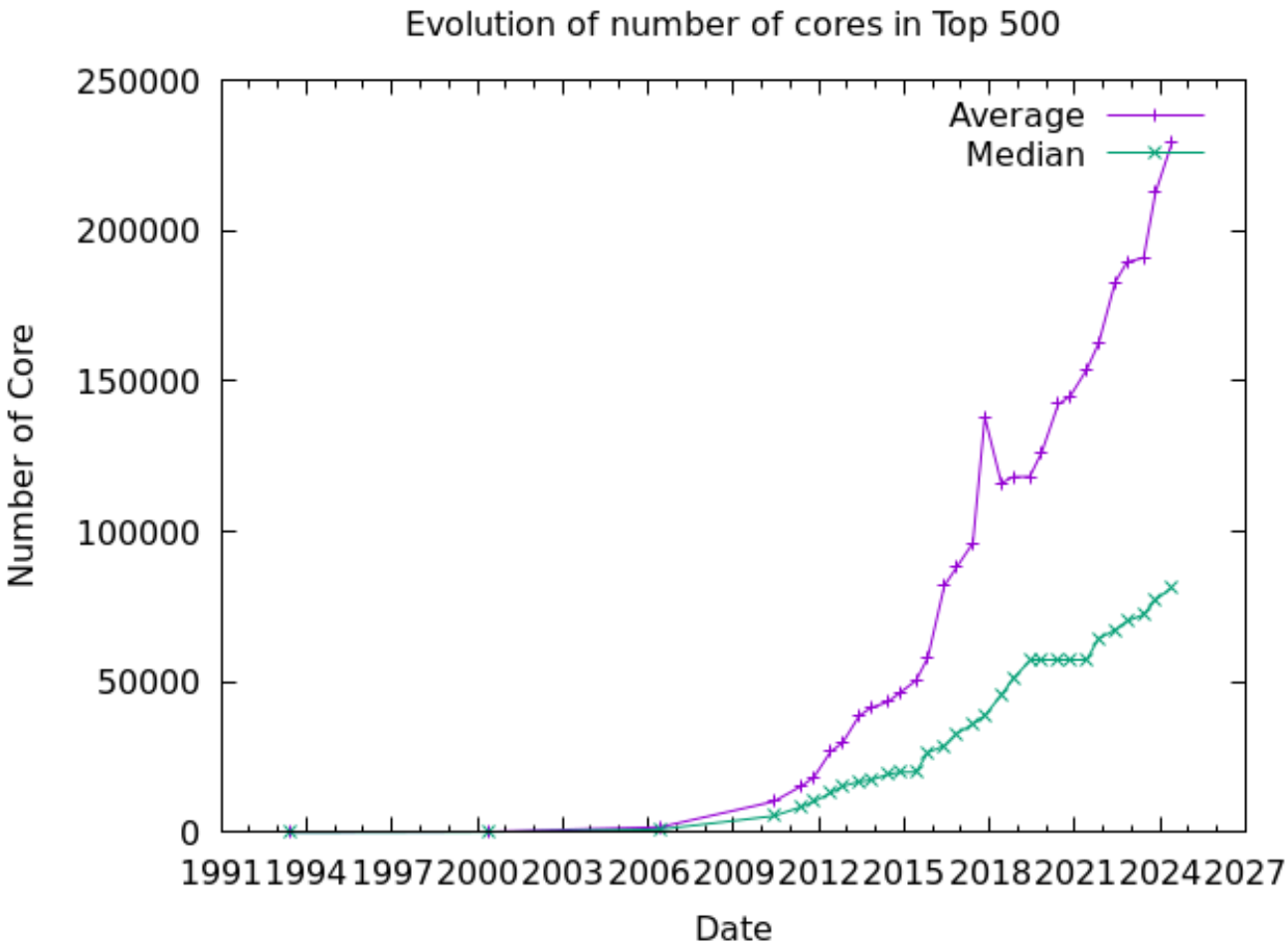
Solutions partielles

- L'ajout de mémoires caches est essentiel.
- Parallélisation des accès via plusieurs bancs mémoire comme sur les architectures vectorielles (Intel Skylake : 6 canaux, AMD EPYC : 8 canaux).
- Si la fréquence des cœurs stagne ou baisse, l'écart pourrait se réduire.

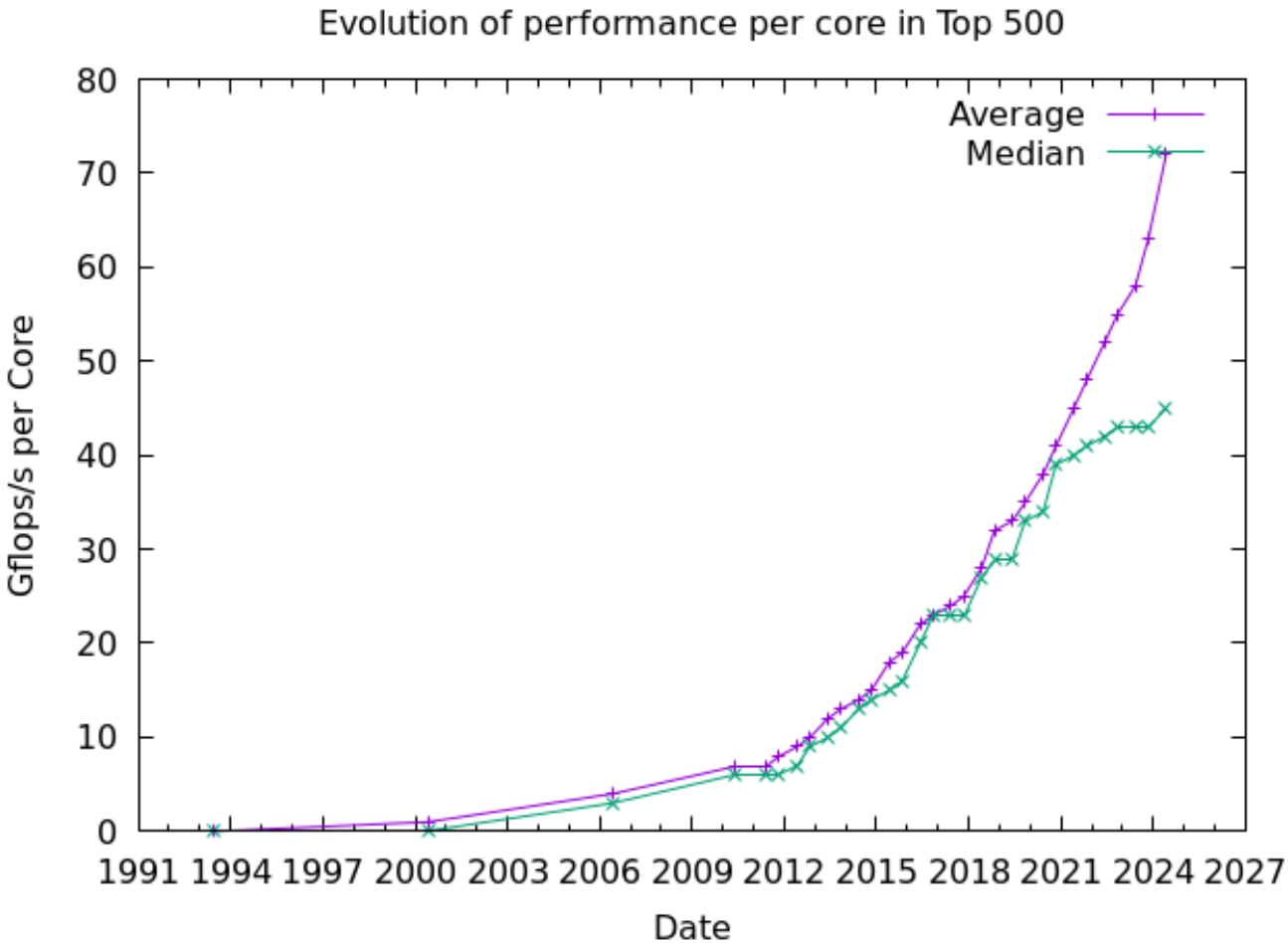
Top 500 - Évolution de la performance



Top 500


Evolution of number of cores in Top 500
Average
Median
Number of Core
Date
250000
200000
150000
100000
50000
0
1991
1994
1997
2000
2003
2006
2009
2012
2015
2018
2021
2024
2027

Top 500


Evolution of performance per core in Top 500
Average
Median
Gflops/s per Core
Date
0
10
20
30
40
50
60
70
80
1991 1994 1997 2000 2003 2006 2009 2012 2015 2018 2021 2024 2027

Du côté des supercalculateurs

Evolution technique

- La puissance de calcul des supercalculateurs augmente plus rapidement que la loi de Moore (mais la consommation électrique augmente également).
- Le nombre de cœurs augmente rapidement (architectures massivement parallèles et *many-cores*).
- Architectures hybrides de plus en plus présentes (GPUs).
- L'architecture des machines se complexifie à tous les niveaux (processeurs/cœurs, hierarchie mémoire, réseau et I/O).
- La mémoire par cœur stagne et commence à décroître.
- La performance par cœur stagne, voire décroit.
- Le débit vers les disques augmente moins vite que la puissance de calcul.

Loi d'Amdahl

Enoncé

La loi d'Amdahl prédit l'accélération théorique maximale obtenue en parallélisant idéalement un code, pour un problème donné et une taille de problème fixée.

$$Acc(P) = \frac{T_s}{T_{//}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}} < \frac{1}{\alpha} \quad (P \to \infty)$$

avec *Acc* le *speedup*, T_s la durée d'exécution du code séquentiel (monoprocesseur), $T_{//}(P)$ la durée d'exécution du code idéalement parallélisé sur P cœurs et α la partie non parallélisable de l'application.

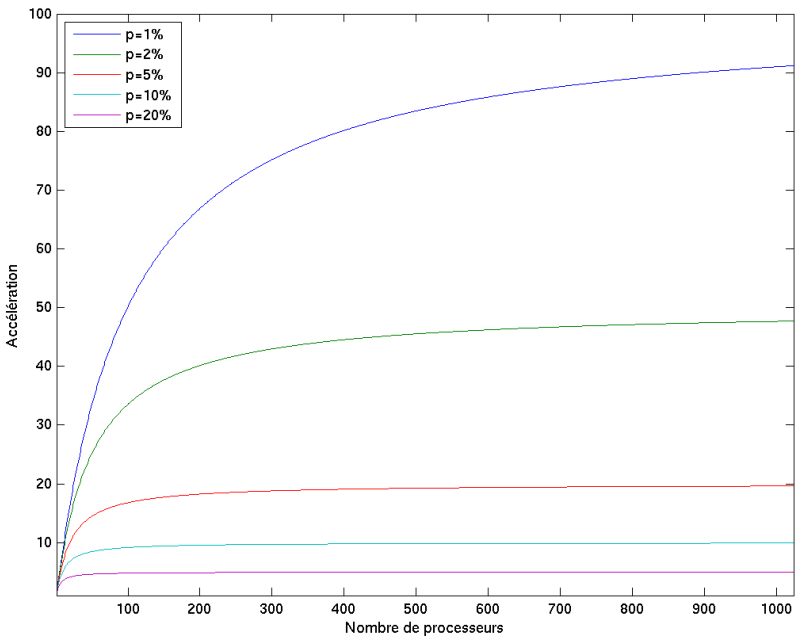
Interprétation

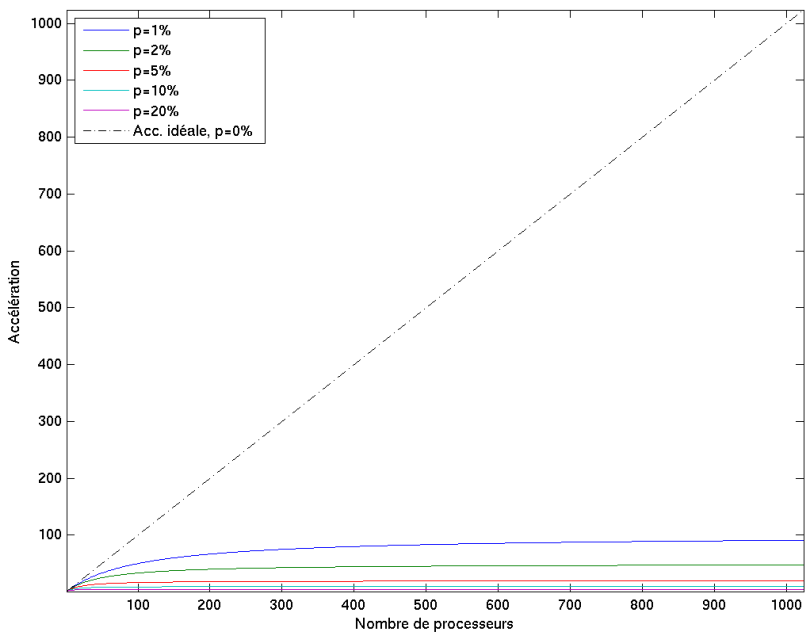
Quel que soit le nombre de cœurs, l'accélération est toujours inférieure à l'inverse du pourcentage que représente la partie purement séquentielle.

Exemple : si la partie purement séquentielle d'un code représente 20% de la durée du code séquentiel, alors quel que soit le nombre de cœurs, on aura : $Acc < \frac{1}{20\%} = 5$

Accélération théorique maximale

Cœurs	α (%)								
	0	0.01	0.1	1	2	5	10	25	50
10	10	9.99	9.91	9.17	8.47	6.90	5.26	3.08	1.82
100	100	99.0	91.0	50.2	33.6	16.8	9.17	3.88	1.98
1000	1000	909	500	91	47.7	19.6	9.91	3.99	1.998
10000	10000	5000	909	99.0	49.8	19.96	9.99	3.99	2
100000	100000	9091	990	99.9	49.9	19.99	10	4	2
∞	∞	10000	1000	100	50	20	10	4	2





Loi de Gustafson-Barsis

Enoncé

La loi de Gustafson-Barsis prédit l'accélération théorique maximale obtenue en parallélisant idéalement un code, pour un problème de taille constante par cœur et en supposant que la durée de la partie séquentielle n'augmente pas avec la taille globale du problème.

$$Acc\,(P) = P - \alpha\,(P - 1)$$

avec *Acc* le *speedup*, P le nombre de cœurs et α la partie non parallélisable de l'application.

Interprétation

Cette loi est plus optimiste que celle d'Amdahl car elle montre que l'accélération théorique croît avec la taille du problème étudié.

Conséquences pour les utilisateurs

Conséquences pour les applications

- Il faut exploiter un grand nombre de cœurs relativement lents.
- La mémoire par cœur tend à baisser, nécessité de la gérer rigoureusement.
- Besoin d'un niveau de parallélisme toujours plus important pour utiliser les architectures modernes (au point de vue puissance de calcul, mais aussi quantité de mémoire).
- Les entrées-sorties deviennent également un problème de plus en plus présent.

Conséquences pour les développeurs

- Fin de l'époque où il suffisait d'attendre pour gagner en performance (stagnation de la puissance de calcul par cœur).
- Besoins accrus de compréhension de l'architecture matérielle.
- De plus en plus compliqué de développer dans son coin (besoin d'experts en HPC et d'équipes multi-disciplinaires).

Evolution des méthodes de programmation

- MPI est toujours prédominant et le restera encore un certain temps (communauté d'utilisateurs très importante et majorité des applications actuelles).
- L'hybride MPI-OpenMP se développe : approche privilégiée pour les supercalculateurs ?
- La programmation sur GPU se développe, mais reste complexe et nécessite un niveau supplémentaire de parallélisme (MPI(+...)+GPU).
- De nouveaux langages de programmation parallèle apparaissent régulièrement (UPC, Coarray-Fortran, langages PGAS, X10, Chapel...), mais tendent à rester en phase expérimentale (à des niveaux d'avancement très variables).

Introduction

Jean Zay

Partition

- Partition scalaire
  - 720 nœuds de calcul scalaires
  - 2 processeurs Intel Cascade Lake 6248 (20 cœurs à 2,5 GHz), soit 40 cœurs par nœud
  - 192 Go de mémoire par nœud
- Partition accelérée
  - 396 nœuds de calcul accélérés quadri-GPU
    - 2 processeurs Intel Cascade Lake 6248 (20 cœurs à 2,5 GHz), soit 40 cœurs par nœud
    - 192 Go de mémoire par nœud
    - 126 nœuds avec 4 GPU Nvidia Tesla V100 SXM2 16 Go
    - 270 nœuds avec 4 GPU Nvidia Tesla V100 SXM2 32 Go
  - 31 nœuds de calcul accélérés octo-GPU
    - 2 processeurs Intel Cascade Lake 6226 (12 cœurs à 2,7 GHz), soit 24 cœurs par nœud
    - 20 nœuds à 384 Go de mémoire
    - 11 nœuds à 768 Go de mémoire
    - 8 GPU Nvidia Tesla V100 SXM2 32Go
  - 52 nœuds de calcul accélérés octo-GPU
    - 2 processeurs AMD Milan EPYC 7543 (32 cœurs à 2,80 GHz), soit 64 cœurs par nœud
    - 512 Go de mémoire par nœud
    - 8 GPU Nvidia A100 SXM4 80 Go
  - 364 nœuds de calcul accélérés quadri-GPU
    - 2 processeurs Intel Sapphire Rapids 8468 (48 cœurs à 2,10 Ghz), soit 96 cœurs par nœuds
    - 512 Go de mémoire par nœuds
    - 4 GPU Nvidia H100 SXM5 80 Go

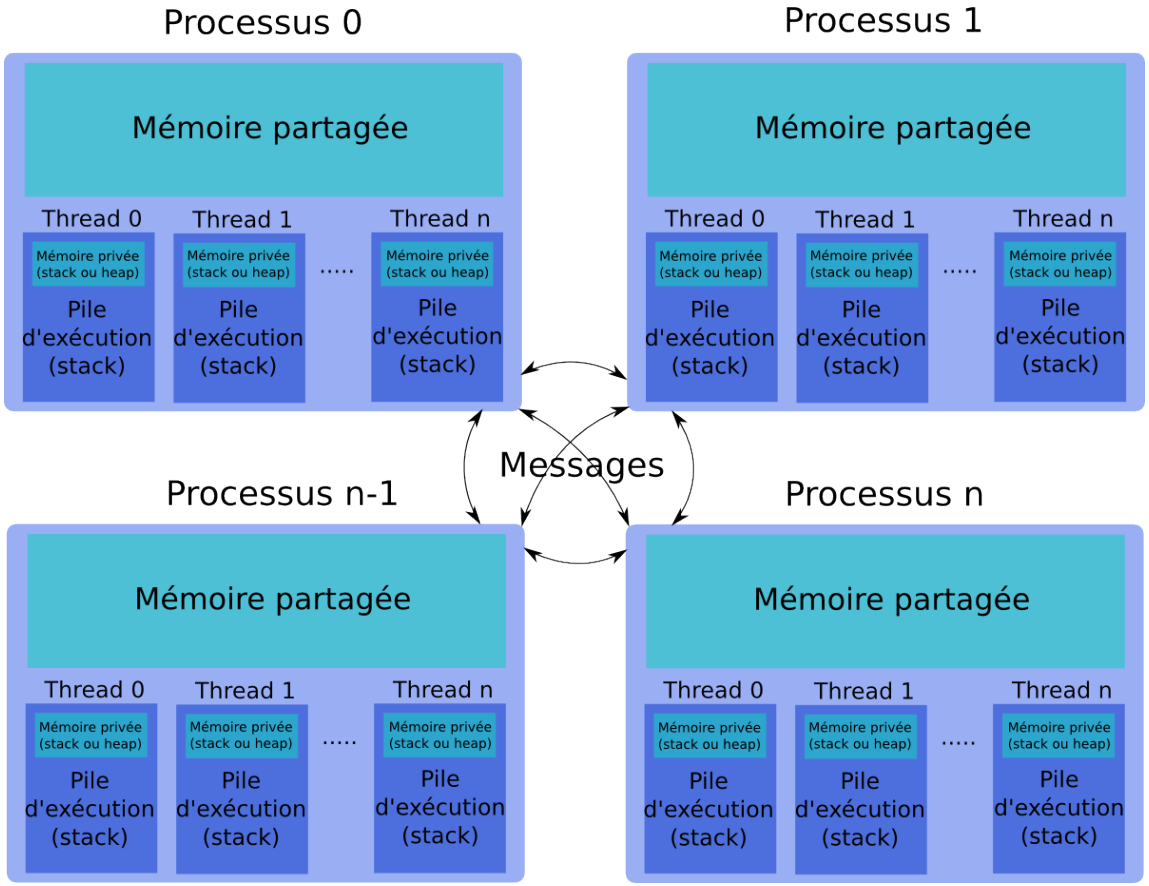
Programmation hybride

Définitions

- La programmation hybride parallèle consiste à mélanger plusieurs paradigmes de programmation parallèle dans le but de tirer parti des avantages des différentes approches.
- Généralement, MPI est utilisé au niveau des processus et un autre paradigme (OpenMP, OpenACC, pthreads, Cuda, langages PGAS, UPC...) à l'intérieur de chaque processus.
- Dans cette formation, nous traitons exclusivement de l'utilisation d'OpenMP avec MPI.

Programmation hybride

Schéma de principe



MPI et le *multithreading*

Support des *threads* dans MPI

La norme MPI prévoit un sous-programme particulier pour remplacer `MPI_Init` lorsque l'application MPI est *multithreadée*. Il s'agit de `MPI_Init_thread`.

- La norme n'impose aucun niveau minimum de support des *threads*. Certaines architectures et/ou implémentations peuvent donc n'avoir aucun support pour les applications *multithreadées*.
- Les rangs identifient uniquement les processus, pas les *threads* qui ne peuvent être précisés dans les communications.
- N'importe quel *thread* peut faire des appels MPI (dépend du niveau de support).
- N'importe quel *thread* d'un processus MPI donné peut recevoir un message envoyé à ce processus (dépend du niveau de support).
- Les appels bloquants ne bloquent que le *thread* concerné.
- L'appel à `MPI_Finalize` doit être fait par le *thread* qui a appelé `MPI_Init_thread` et lorsque l'ensemble des *threads* du processus ont fini leurs appels MPI.

MPI_Init_thread

MPI_Init_thread

```
int MPI_Init_thread(int *argc, char *((*argv)[]),
                    int required, int *provided)
```

Le niveau de support demandé est fourni dans la variable *required*. Le niveau effectivement obtenu (qui peut être moindre ou meilleur que demandé) est récupéré dans *provided*. Les différents niveaux de support sont :

- `MPI_THREAD_SINGLE` : Un seul *thread*. Identique à `MPI_Init`.
- `MPI_THREAD_FUNNELED` : Les processus peuvent lancer plusieurs *threads*, mais seul le *thread* principal 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_Init_thread

Exemple d'initialisation

```
#include <mpi.h>
#include <stdio.h>

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

  MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&provided);
  if (provided < MPI_THREAD_MULTIPLE)
    MPI_Abort(MPI_COMM_WORLD,42);
  MPI_Finalize();
}
```

MPI_Query_thread

MPI_Query_thread retourne le niveau de support du processus appellant :

```
  int MPI_Query_thread(int *provided)
```

MPI_THREAD_SINGLE

MPI_THREAD_SINGLE

`MPI_THREAD_SINGLE` : seul un *thread* par processus peut s'exécuter. Il n'est pas possible de faire de l'hybride. OpenMP ne peut pas être utilisé. C'est le niveau si on utilise `MPI_Init`.

MPI_THREAD_FUNNELED

MPI_THREAD_FUNNELED

- 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_Is_thread_main indique si le *thread* appellant est le *thread* principal ou pas

```
int MPI_Is_thread_main(int *flag)
```

MPI_THREAD_FUNNELED

Les appels MPI doivent être faits en dehors des régions parallèles OpenMP ou dans les régions OpenMP *master/masked* ou dans des zones protégées par un appel à `MPI_Is_thread_main`

```
MPI_Is_thread_main(&flag);
if (flag) {
  MPI_Send(buffer,n,MPI_INT,dest,tag,MPI_COMM_WORLD);
}
```

```
#pragma omp master /* ou #pragma omp masked */
{
  MPI_Send(buffer,n,MPI_INT,dest,tag,MPI_COMM_WORLD);
}
```

Avantages

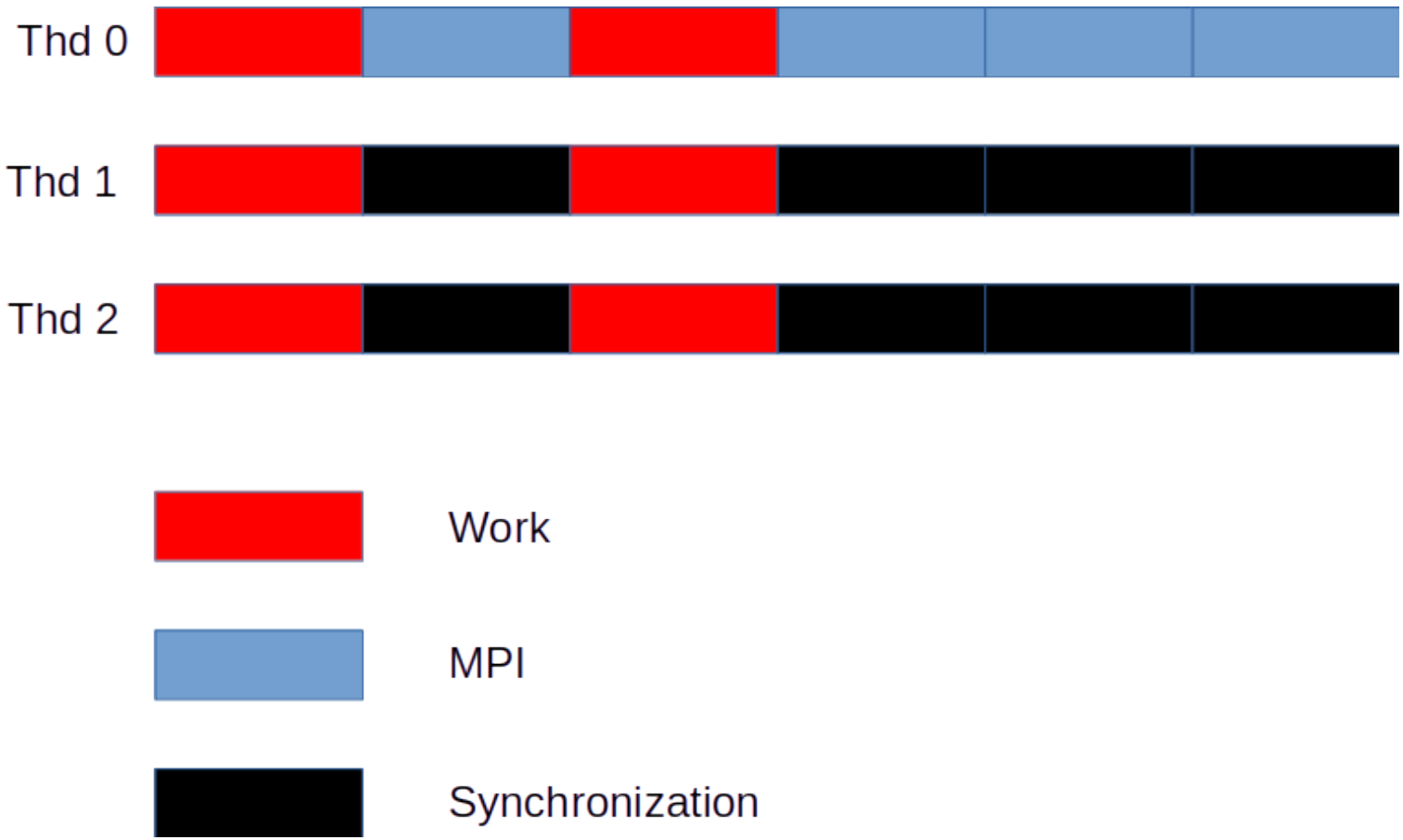
- Simple à mettre en place.
- Mais génère un déséquilibre entre les threads.
- Et une mauvaise affinité mémoire.

Attention après une région OpenMP *master/masked* il n'y a pas de synchronisation, il faut souvent en faire une explicitement avec une barrière OpenMP.

MPI_THREAD_FUNNELED

Schéma de principe

Imaginez une situation où l'on fait une communication collective et des communications point à point entrecoupées par des calculs.



MPI_THREAD_SERIALIZED

`MPI_THREAD_SERIALIZED` : tous les *threads* peuvent faire des appels MPI, mais un seul à la fois. Dans les régions parallèles OpenMP, les appels MPI doivent être réalisés dans des sections *critical* (si nécessaire pour garantir un seul appel MPI simultané)

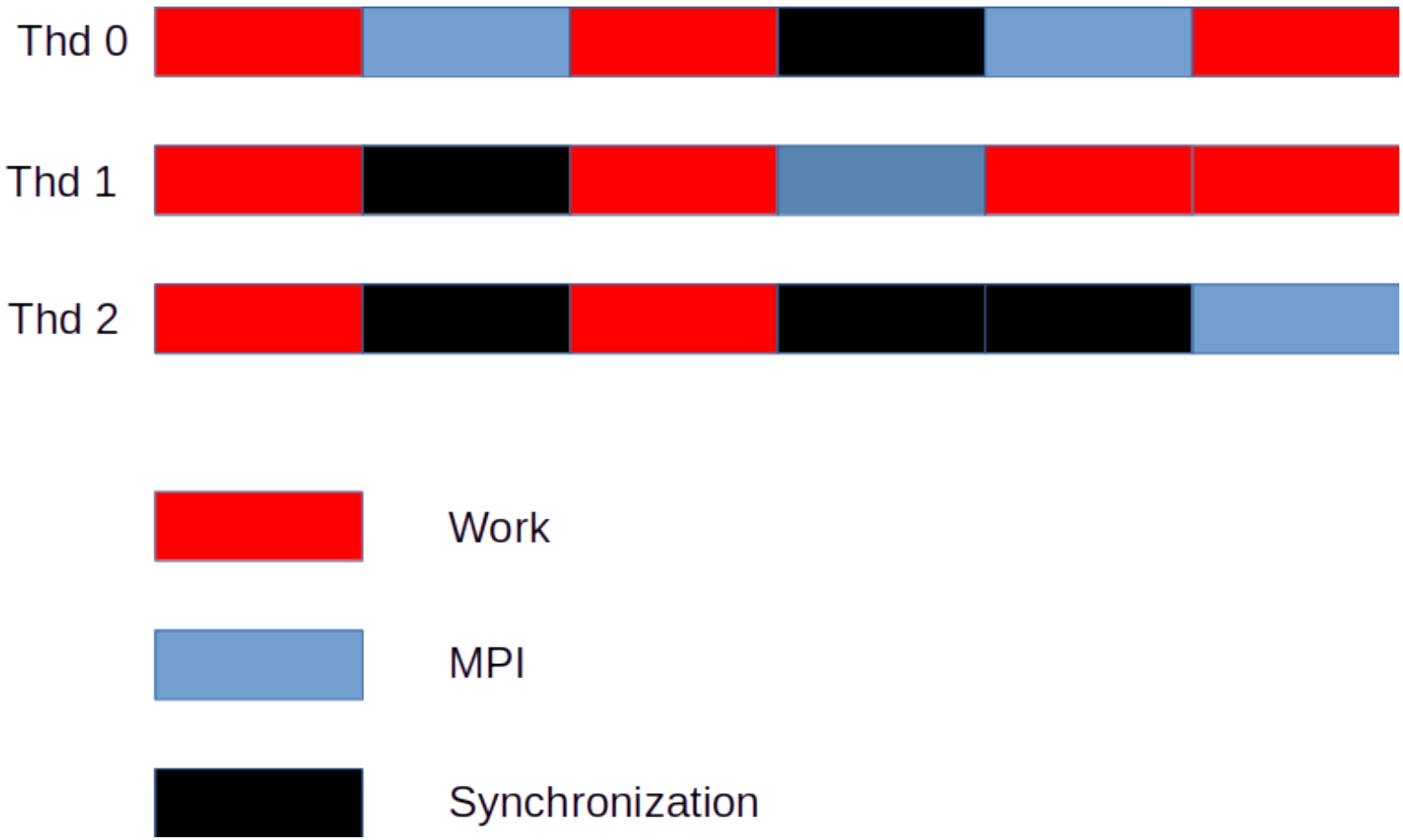
```
#pragma omp critical (mpi)
{
  MPI_Send(buffer,n,MPI_INT,dest,tag,MPI_COMM_WORLD);
}
```

Avantages

- Bonne affinité mémoire.
- Mais génère des attentes de threads

MPI_THREAD_SERIALIZED

Schéma de principe



MPI_THREAD_MULTIPLE

`MPI_THREAD_MULTIPLE` : entièrement *multithreadé* sans restrictions.

Avantages

- Bonne affinité mémoire.
- Moindre surcout en synchronisation.
- Mais plus délicat à écrire et à maintenir.

MPI_THREAD_MULTIPLE

Restrictions sur les appels MPI collectifs

En mode `MPI_THREAD_MULTIPLE`, l'utilisateur doit s'assurer que les opérations collectives sur le même communicateur, fenêtre mémoire ou descripteur de fichier sont correctement ordonnées entre les différents *threads*.

- Cela implique qu'il est interdit d'avoir plusieurs *threads* par processus faisant des appels collectifs avec le même communicateur sans s'assurer que tous les processus les font dans le même ordre.
- On ne peut donc pas avoir à un instant donné 2 *threads* qui font chacun un appel collectif avec le même communicateur (que les appels soient différents ou pas).
- Par exemple, si plusieurs *threads* font un appel à `MPI_Barrier` avec `MPI_COMM_WORLD`, l'application peut se bloquer.
- 2 *threads* appelant chacun un `MPI_Allreduce` (avec la même opération de réduction ou pas) peuvent obtenir des résultats faux.
- 2 appels collectifs différents ne peuvent pas non plus être utilisés (un `MPI_Reduce` et `MPI_Bcast` par exemple).

MPI_THREAD_MULTIPLE

Restrictions sur les appels MPI collectifs

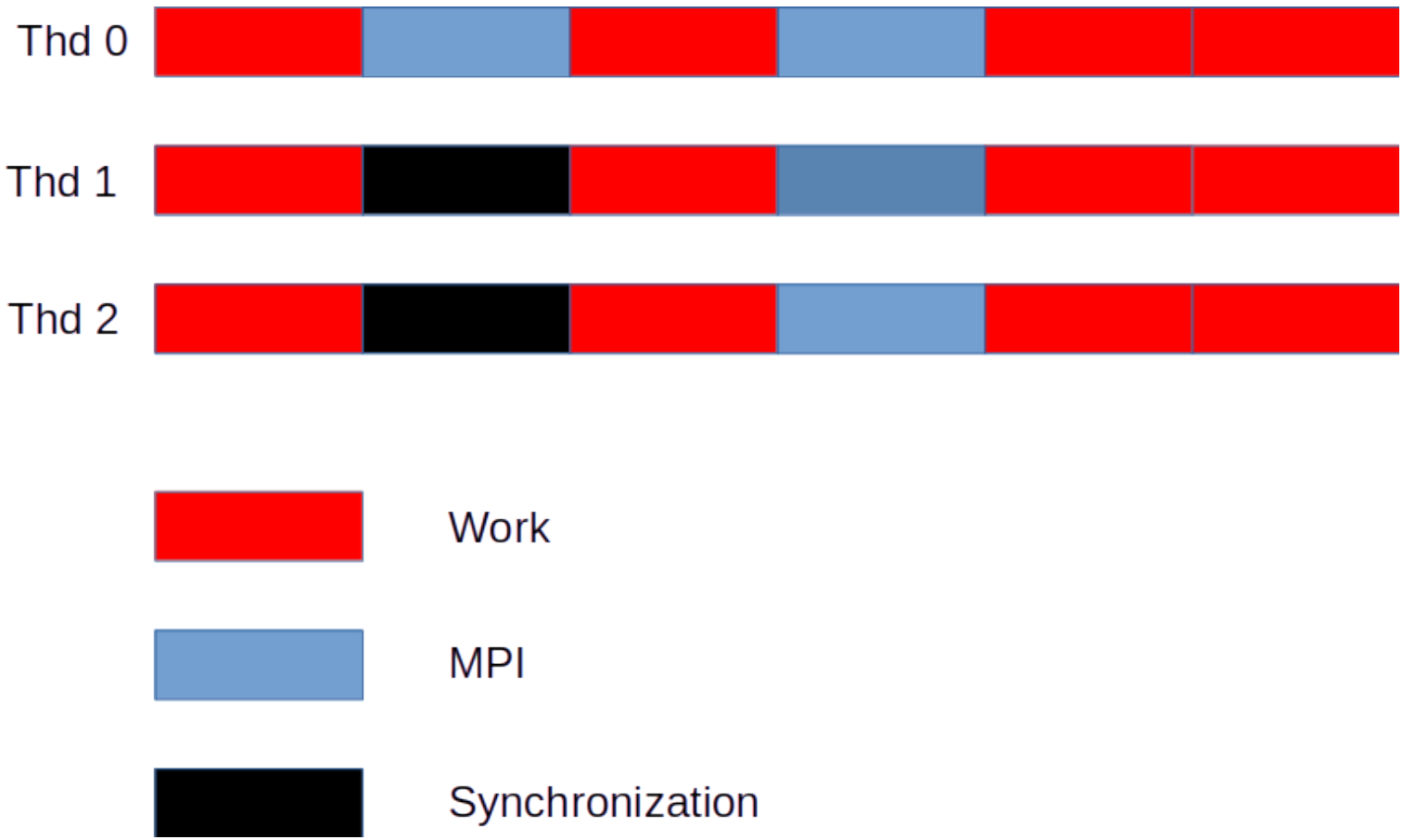
Pour éviter ce genre de difficultés, il existe plusieurs possibilités :

- Imposer l'ordre des appels en synchronisant les différents *threads* à l'intérieur de chaque processus MPI,
- Utiliser des communicateurs différents pour chaque appel collectif,
- Ne faire des appels collectifs que sur un seul *thread* par processus.

Remarque : en mode `MPI_THREAD_SERIALIZED`, le problème ne devrait pas exister car l'utilisateur doit obligatoirement s'assurer qu'à un instant donné au maximum seul un *thread* par processus est impliqué dans un appel MPI (collectif ou pas). Attention, l'ordre des appels doit néanmoins être respecté.

MPI_THREAD_MULTIPLE

Schéma de principe



MPI_Probe

MPI_Probe

En mode MPI_THREAD_MULTIPLE l'utilisation de MPI_Probe (ou MPI_IProbe) et la réception du message avec MPI_Recv (ou MPI_IRecv) n'est pas *thread-safe*.

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &statut);
MPI_Get_count(&statut, MPI_INT, &msgsize);
buf = (int *) malloc(msgsize*sizeof(int));
MPI_Recv(buf, msgsize, MPI_INT, statut.MPI_SOURCE,
         statut.MPI_TAG, comm, MPI_STATUS_IGNORE);
```

Si plusieurs threads exécutent cet exemple, rien n'assure que le MPI_Recv correspondra bien au message détecté par MPI_Probe.
MPI 3 introduit MPI_Mprobe (ou MPI_Improbe) et MPI_Mrecv (ou MPI_Imrecv) pour résoudre ce problème.

```
MPI_Mprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &message, &statut);
MPI_Get_count(&statut, MPI_INT, &msgsize);
buf = (int *) malloc(msgsize*sizeof(int));
MPI_Mrecv(buf, msgsize, MPI_INT, &message, &statut);
```

Piège

Piège

- Voici un exemple avec un bogue.
- Cet exemple tourne sur 2 processus MPI.
- Chaque processus utilise 2 threads.
- Les envois ne sont pas en mode synchrone.

```
for (j = 0; j < 2; j++) {
  if (rank == 1) {
    for (i = 0; i < 2; i++)
      MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    for (i = 0; i < 2; i++)
      MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
  }
  else { /* rank == 0 */
    for (i = 0; i < 2; i++)
      MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
    for (i = 0; i < 2; i++)
      MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  }
}
```

Piège

Rank 0		Rank 1	
Thd 0	Thd 1	Thd 0	Th 1
MPI_Recv		MPI_Send	
MPI_Recv		MPI_Send	
MPI_Send		MPI_Recv	
MPI_Send			
MPI_Recv			MPI_Send
	MPI_Recv		MPI_Send
			MPI_Recv
MPI_Recv	MPI_Recv	MPI_Recv	MPI_Recv

Ordre possible des communications

- Il est préférable d'utiliser des étiquettes différentes sur chaque thread.
- Ou d'utiliser un communicateur différent sur chaque thread.

Gain mémoire

Programmation hybride, l'aspect gain mémoire

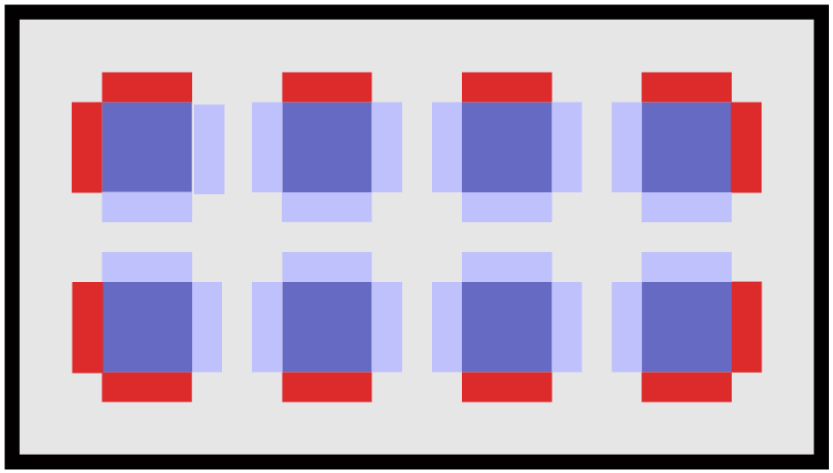
Pourquoi un gain mémoire ?

- La programmation hybride permet d'optimiser l'adéquation du code à l'architecture cible. Cette dernière est généralement constituée de nœuds à mémoire partagée (SMP) reliés entre eux par un réseau d'interconnexion. L'intérêt de la mémoire partagée au sein d'un nœud est qu'il n'est pas nécessaire de dupliquer des données pour se les échanger. Chaque *thread* a visibilité sur les données *SHARED*.
- Les mailles fantômes ou halo, introduites pour simplifier la programmation de codes MPI utilisant une décomposition de domaine, n'ont plus lieu d'être à l'intérieur du nœud SMP. Seules les mailles fantômes associées aux communications inter-nœuds sont nécessaires.
- Réduction de l'empreinte mémoire lorsque la taille de certaines structures de données dépend directement du nombre de processus MPI.

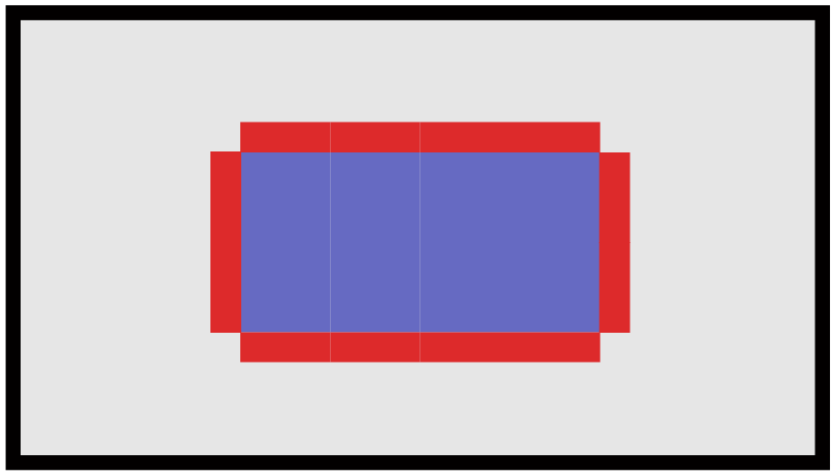
Programmation hybride, l'aspect gain mémoire

Exemple domaine 2D, décomposition suivant les 2 directions

Noeud SMP à 8 coeurs, décomposition de domaine flat MPI



Noeud SMP à 8 coeurs, décomposition de domaine hybride



Mailles fantômes intra-noeud

Mailles fantômes inter-noeud

Sous-domaine associé à un processus MPI

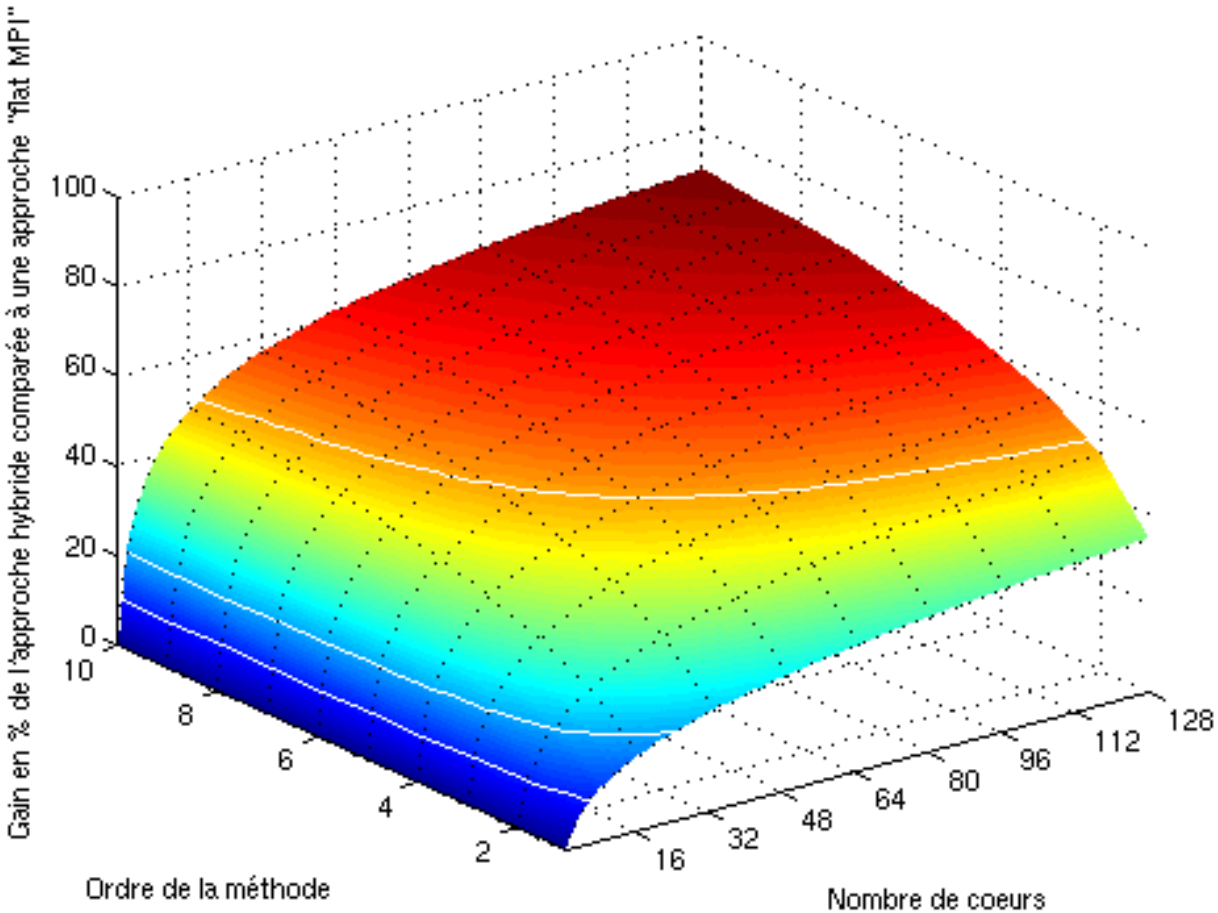
Programmation hybride, l'aspect gain mémoire

Extrapolation sur un domaine 3D

- Essayons de calculer, en fonction de l'ordre de la méthode numérique (h) et du nombre de cœurs du nœud SMP (c), le gain mémoire relatif obtenu en utilisant une version hybride au lieu d'une version *flat* MPI d'un code 3D parallélisé par une technique de décomposition de domaine suivant ses trois dimensions.
- On prendra les hypothèses suivantes :
  - On fait varier l'ordre de la méthode numérique h de 1 à 10.
  - On fait varier le nombre de cœurs c du nœud SMP de 1 à 128.
  - Pour dimensionner le problème, on suppose qu'on a accès à 64 Go de mémoire partagée sur le nœud.
- Le résultat de la simulation est présenté dans le transparent suivant. Les isovaleurs 10%, 20% et 50% sont représentées par des lignes blanches sur la surface résultat.

Programmation hybride, l'aspect gain mémoire

Extrapolation sur un domaine 3D



Programmation hybride, l'aspect gain mémoire

Gain mémoire effectif sur quelques codes applicatifs

- Source : « *Mixed Mode Programming on HECToR* », *Anastasios Stathopoulos, August 22, 2010, MSc in High Performance Computing, EPCC*
- Machine cible : HECToR CRAY XT6.
  1856 *Compute Nodes (CN)*, chacun composé de deux processeurs AMD 2.1 GHz à 12 cœurs se partageant 32 Go de mémoire, pour un total de 44544 cœurs, 58 To de mémoire et une performance crête de 373 Tflop/s.
- Résultats, la mémoire par *node* est exprimée en Mo :

Code	Version pure MPI		Version hybride		Gain mémoire
	Nbre MPI	Mém./*Node*	MPI x *threads*	Mém./*Node*	
CPMD	1152	2400	48 x 24	500	4.8
BQCD	3072	3500	128 x 24	1500	2.3
SP-MZ	4608	2800	192 x 24	1200	2.3
IRS	2592	2600	108 x 24	900	2.9
Jacobi	2304	3850	96 x 24	2100	1.8

Programmation hybride, l'aspect gain mémoire

Gain mémoire effectif sur quelques codes applicatifs

- Source : « *Performance evaluations of gyrokinetic Eulerian code GT5D on massively parallel multi-core platforms* », *Yasuhiro Idomura et Sébastien Jolliet, SC11*
- Exécutions sur 4096 cœurs
- Machine utilisée : Fujitsu BX900 avec des processeurs Nehalem-EP à 2,93 GHz (8 cœurs et 24 Gio par nœud)
- Toutes les tailles sont données en Tio

Système	MPI pur	4 threads/prc		8 threads/prc	
	Total (code+sys)	Total (code+sys)	Gain	Total (code+sys)	Gain
BX900	5.40 (3.40+2.00)	2.83 (2.39+0.44)	1.9	2.32 (2.16+0.16)	2.3

Programmation hybride, l'aspect gain mémoire

Conclusion sur les aspects gains mémoire

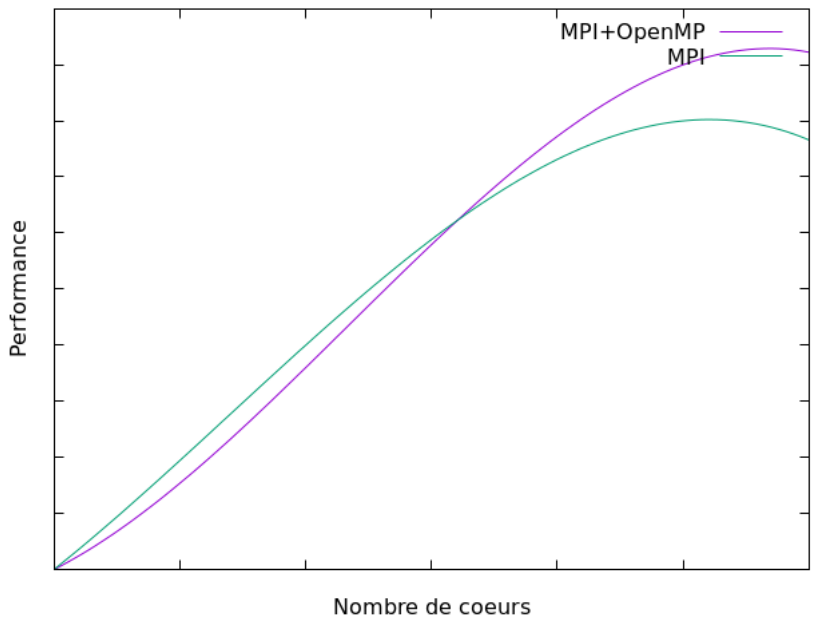
- Trop souvent, cet aspect est oublié lorsqu'on parle de la programmation hybride.
- Pourtant, les gains potentiels sont très importants et pourraient être mis à profit pour augmenter la taille des problèmes à simuler !
- Plusieurs raisons font que le différentiel (MPI vs. Hybride) va s'amplifier de plus en plus rapidement pour les machines de prochaine génération :
  1. La multiplication du nombre total de cœurs,
  2. La multiplication rapide du nombre de cœurs disponibles au sein d'un nœud ainsi que la généralisation de l'*hyperthreading* ou du SMT (possibilité d'exécuter simultanément plusieurs *threads* sur un seul cœur),
  3. La généralisation de méthodes numériques d'ordre élevé (le coût du calcul brut étant de moins en moins élevé grâce notamment aux accélérateurs matériels)
- Certains pensent que cela pourrait rendre quasi obligatoire le passage à la programmation hybride.

Performance

Hybride – Performance

Introduction

- OpenMP est généralement moins performant que MPI à cause de synchronisations plus fréquentes.
- Lorsque MPI scale bien, la version hybride ne doit pas s'attendre à surpasser la version pure MPI.



Hybride – Équilibrage de charge

Équilibrage de charge

- L'équilibrage de charge dynamique est compliqué avec MPI.
- Plus le nombre de processus MPI augmente, plus l'équilibrage est difficile.
- OpenMP a des instructions intégrées qui peuvent faire de l'équilibrage de charge dynamique sur les boucles (`schedule dynamic` et `omp task`).
- La programmation hybride est plus adaptée pour l'équilibrage dynamique qu'une approche pure MPI.

Présentation du *benchmark*

Description du *Multi-Zone NAS Parallel Benchmark*

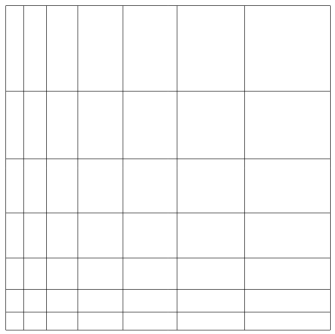
- Le *Multi-Zone NAS Parallel Benchmark* est un ensemble de programmes de tests de performances pour machines parallèles développé par la NASA.
- Ces codes utilisent des algorithmes proches de certains codes de CFD.
- La version multi-zones simule un domaine complexe constitué de plusieurs maillages.
- Trois applications différentes avec 8 tailles de problème différentes sont
- Les sources sont disponibles à l'adresse : `https://www.nas.nasa.gov/software/npb.html`.

Présentation du *benchmark*

Application choisie : BT-MZ

BT-MZ : méthode de résolution tridiagonale par blocs.

- La taille des zones est très variable. Mauvais équilibrage de charge.
- Une heuristique est utilisée pour équilibrer la charge des processus.
- L'approche hybride devrait améliorer la situation.



Application choisie : SP-MZ

SP-MZ : méthode de résolution pentadiagonale scalaire.

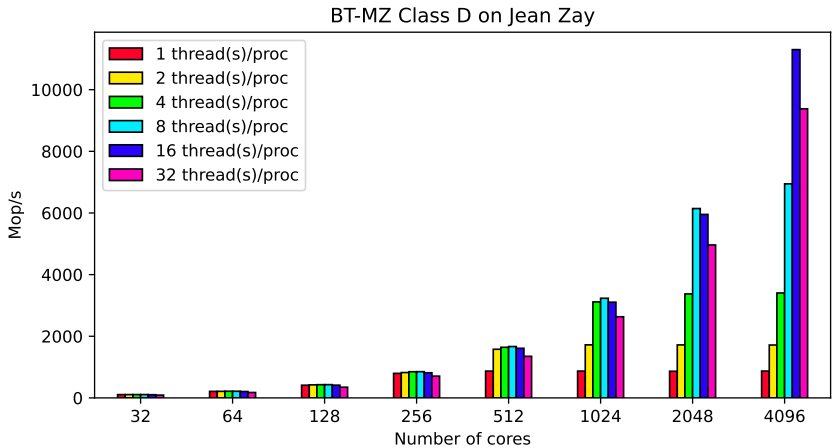
- Toutes les tailles de zones sont égales. Parfait équilibrage de charge.
- L'approche hybride ne devrait rien apporter sur ce point.

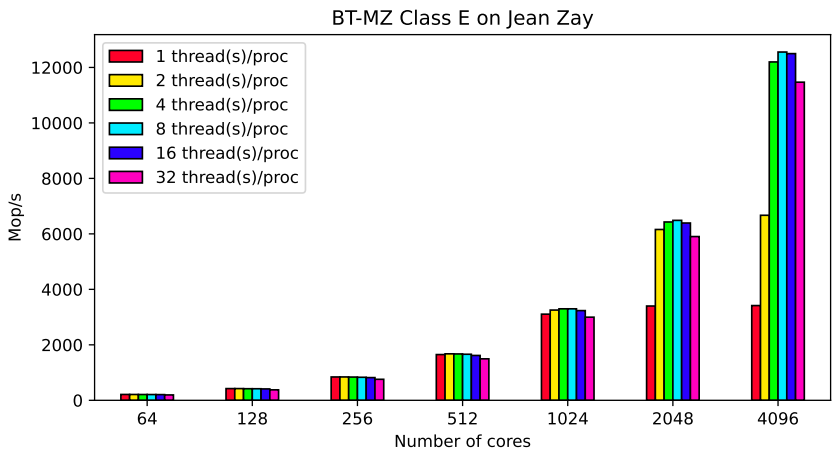
Tailles de problème sélectionnées

- Classe D : 1024 zones (et donc limité à 1024 processus MPI), 1632 x 1216 x 34 points de maillage (13 Gio)
- Classe E : 4096 zones (et donc limité à 4096 processus MPI), 4224 x 3456 x 92 points de maillage (250 Gio)

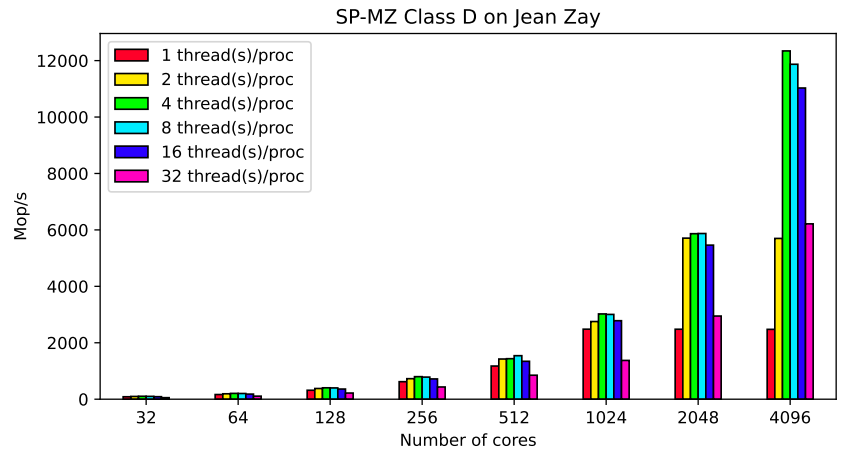
NPB-MZ sur Jean Zay

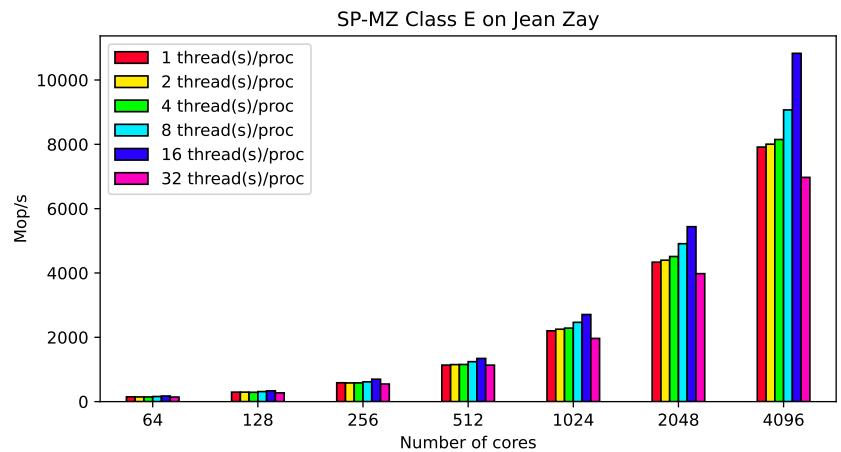
BT-MZ





SP-MZ





Analyse des résultats

Analyse des résultats : BT-MZ

- La version hybride est équivalente à la version MPI pour un nombre de processus pas trop grand.
- Lorsque le déséquilibrage de charge apparaît en MPI pur (à partir de 512 processus pour la classe D et de 2048 pour la classe E), la version hybride permet de garder une très bonne extensibilité en réduisant le nombre de processus.
- La limite de 1024 zones en classe D et de 4096 en classe E limite à respectivement 1024 et 4096 processus MPI, mais l'ajout d'OpenMP permet d'aller bien plus loin en nombre de cœurs utilisés tout en obtenant une excellente extensibilité.

Analyse des résultats

Analyse des résultats : SP-MZ

- Bien que n'ayant pas de déséquilibrage de charge, ce *benchmark* profite dans certains cas du caractère hybride de l'application.
- La limite de 1024 zones en classe D et de 4096 en classe E limite à respectivement 1024 et 4096 processus MPI, mais l'ajout d'OpenMP permet d'aller bien plus loin en nombre de cœurs utilisés tout en obtenant une excellente extensibilité.

Effets architecture non uniforme

Architecture non uniforme

La plupart des machines de calcul modernes ont une architecture non uniforme :

- Accès mémoires non uniformes (NUMA, Non Uniform Memory Access) avec les composants (barettes) mémoire attachés à des sockets différents à l'intérieur d'un même nœud.
- Caches mémoire partagés ou pas entre cœurs ou groupes de cœurs.
- Cartes réseaux connectées à certains sockets.
- Réseau non uniforme (par exemple plusieurs niveaux de switchs réseaux).

Effets

- Complexifie la compréhension des problèmes de performance et l'optimisation des codes.
- Importance du placement des processus à l'intérieur d'un nœud et entre les nœuds.
- Impact limité sur les performances des codes MPI (communications uniquement).
- Impact potentiellement très important sur les codes OpenMP (attention au First Touch).

Utilisation optimale du réseau d'interconnexion

Impact de la programmation hybride

- Moins de messages mais des messages plus gros
- Moins de "parallélisme" dans les communications
- Impact du nombre de flux de communications sur l'usage de la bande passante réseau ?

SBPR

- Développement IDRIS d'un petit benchmark de Saturation Bande Passante Réseau
- Ping-pong MPI en parallèle
- Tests avec un nombre de flux en parallèle et une taille des messages variables

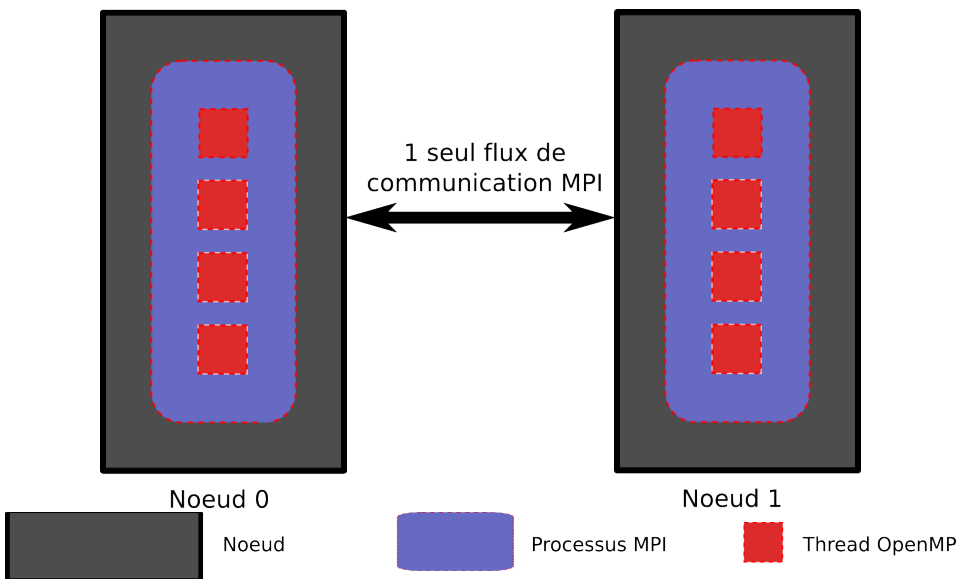
Utilisation optimale du réseau d'interconnexion

SBPR version `MPI_THREAD_FUNNELED`

Approche `MPI_THREAD_FUNNELED` :

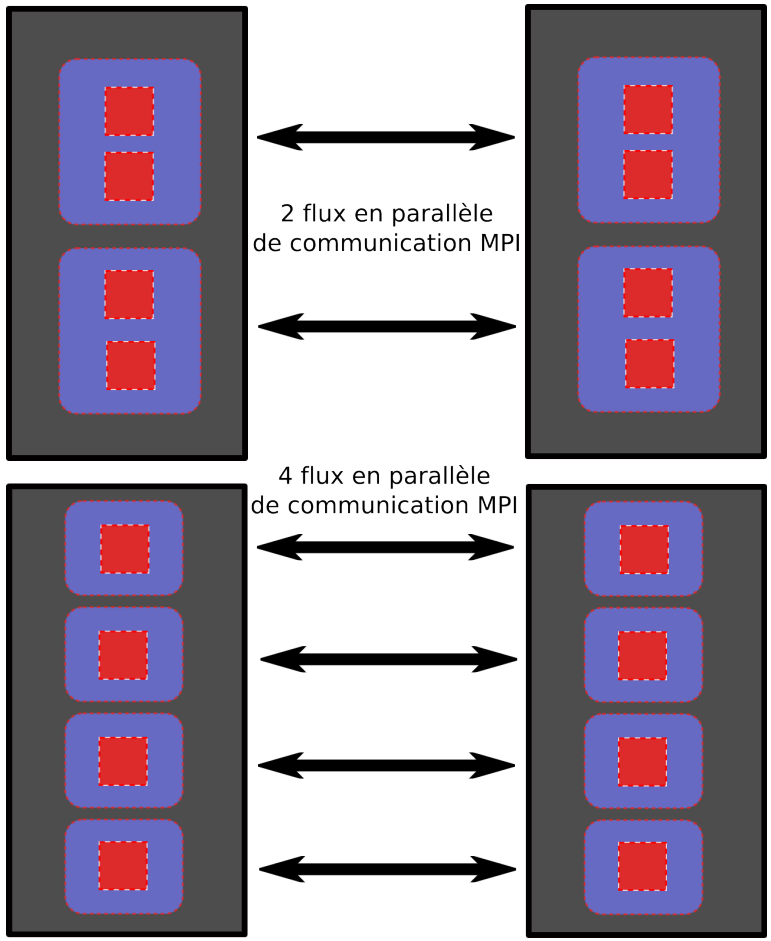
- Nombre variable de processus MPI par nœud (et donc autant de flux de communications en parallèle).
- Seul le *thread* OpenMP maître communique.

SBPR `MPI_THREAD_FUNNELED` : exemple sur un nœud SMP à 4 cœurs



Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_FUNNELED` : exemple sur un nœud SMP à 4 cœurs



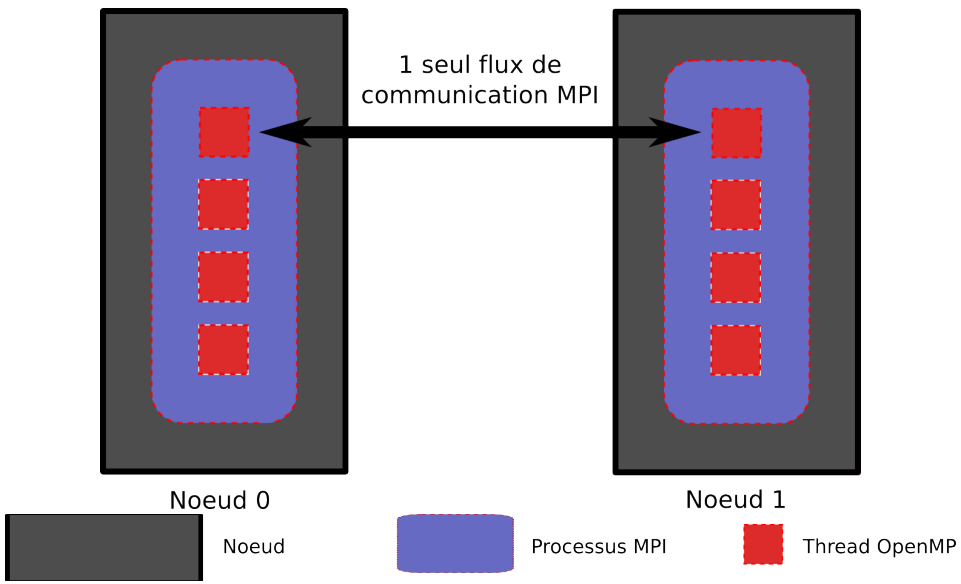
Utilisation optimale du réseau d'interconnexion

SBPR version `MPI_THREAD_MULTIPLE`

Approche `MPI_THREAD_MULTIPLE` :

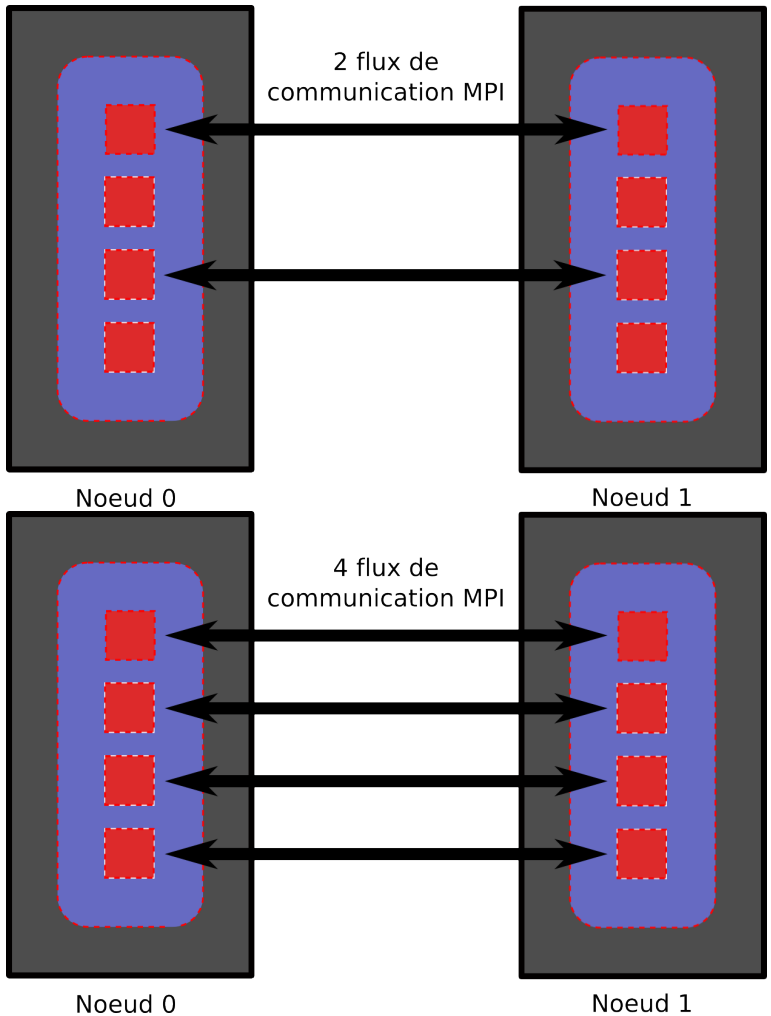
- Unique processus MPI par nœud.
- Nombre variable de threads OpenMP participant aux communications.

SBPR `MPI_THREAD_MULTIPLE` : exemple sur un nœud SMP à 4 cœurs



Utilisation optimale du réseau d'interconnexion

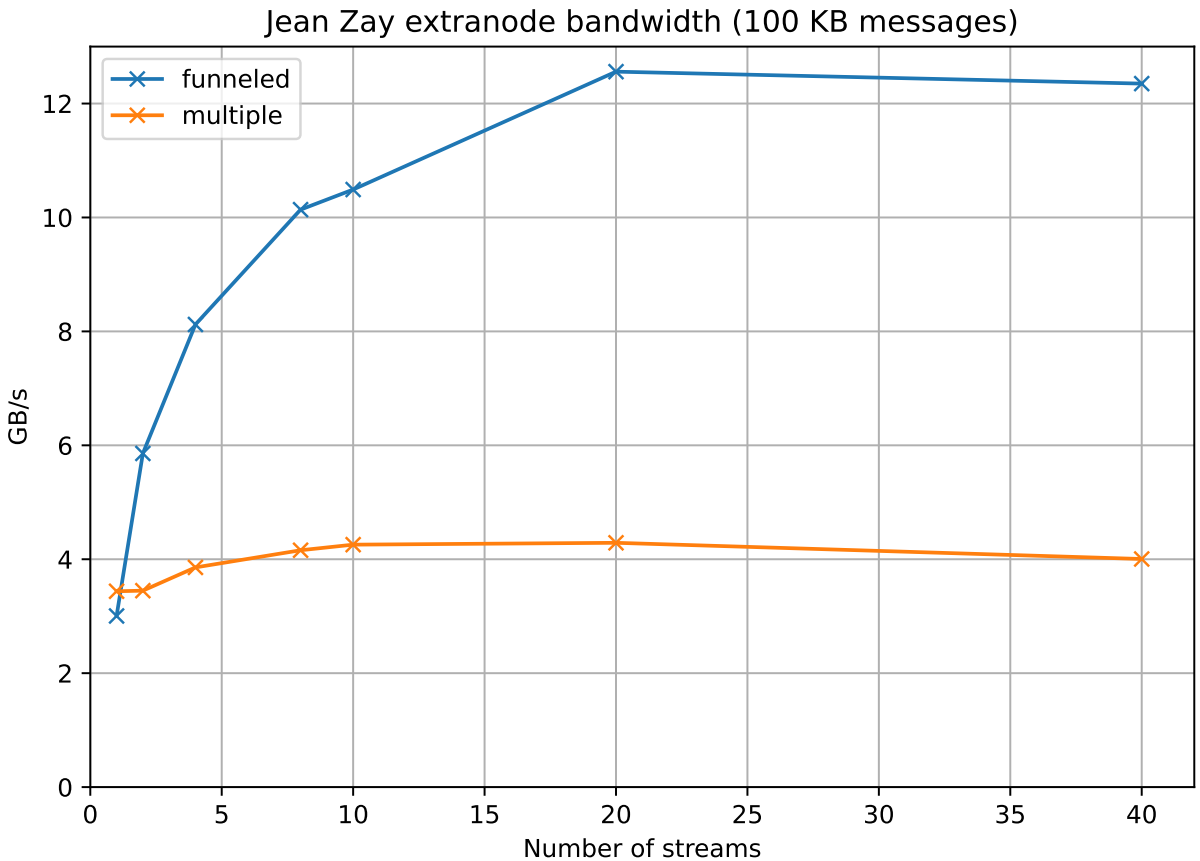
SBPR `MPI_THREAD_MULTIPLE` : exemple sur un nœud SMP à 4 cœurs



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Jean Zay

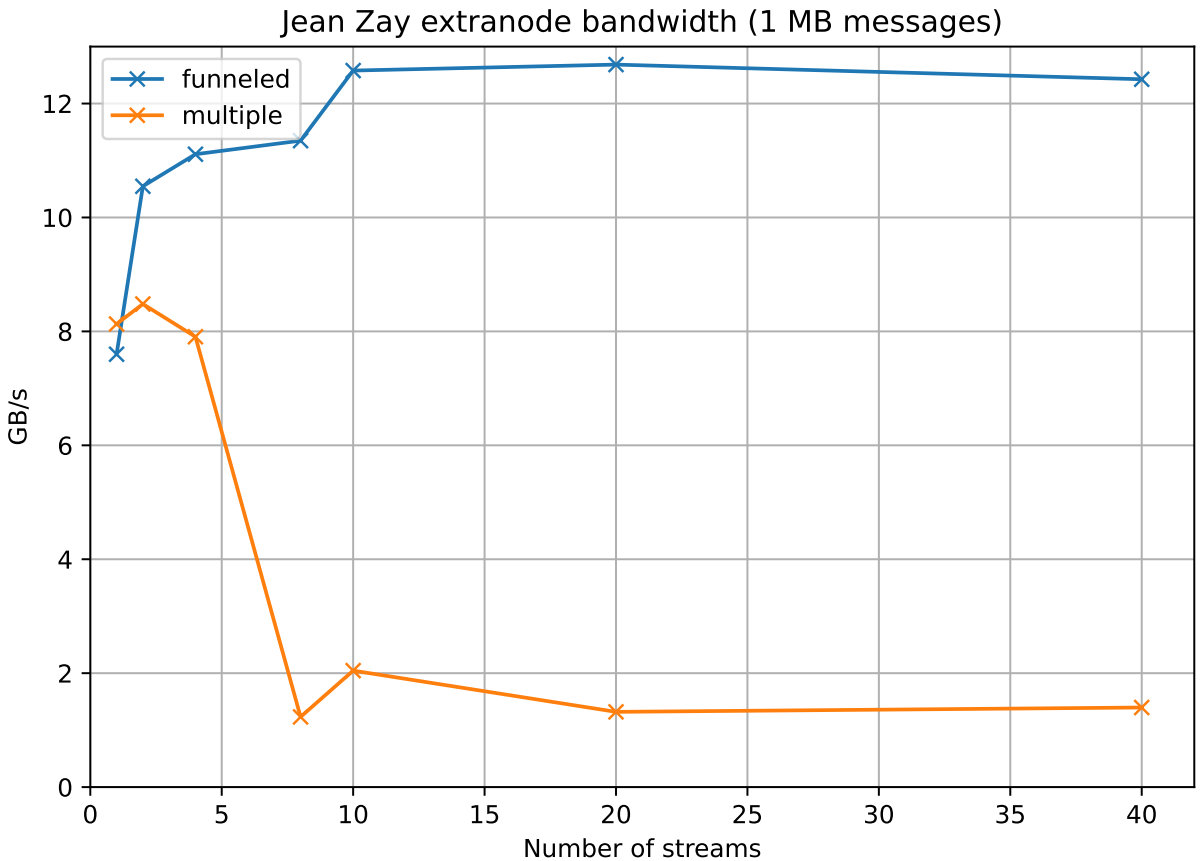
1 lien OmniPath, débit crête 100 Gbps, Intel MPI 2021.9



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Jean Zay

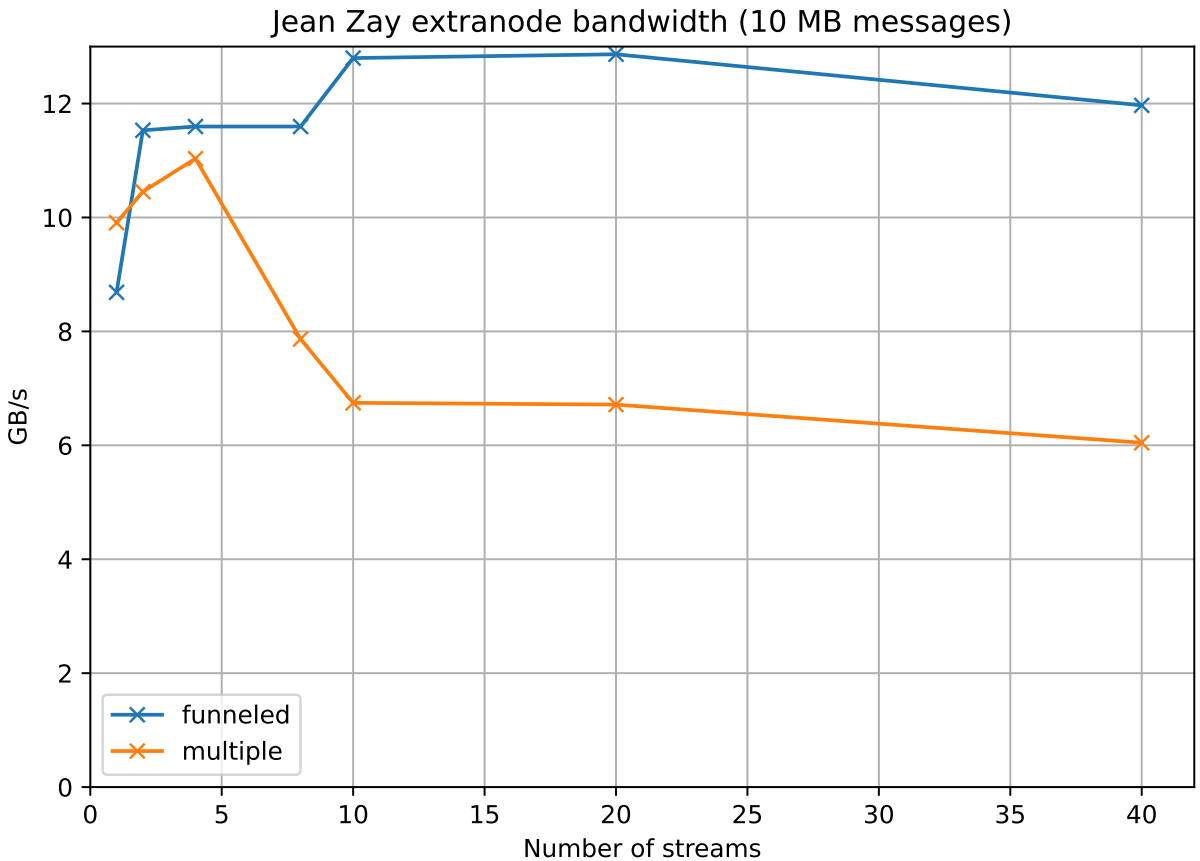
1 lien OmniPath, débit crête 100 Gbps, Intel MPI 2021.9



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Jean Zay

1 lien OmniPath, débit crête 100 Gbps, Intel MPI 2021.9



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Jean Zay

- Plus les messages sont gros, plus les débits tendent à être élevés.
- Mode `MPI_THREAD_FUNNELED` :
  - Augmentation du débit avec le nombre flux en parallèle.
  - Assez facile d'atteindre le débit crête.
- Mode `MPI_THREAD_MULTIPLE` :
  - Surcoût non négligable de la gestion des threads.
  - Mauvaise implémentation ?

Conclusion

- Pas forcément pertinant d'utiliser autant de threads OpenMP que de cœurs au sein d'un nœud et autant de processus MPI que de nœuds.
- Equilibre à trouver entre parallélisme/granularité des communications et granularité des calculs.
- Importance à relativiser en fonction du poids des communications.

Travaux pratiques

TP1 — Barrière de synchronisation hybride

Objectif

Synchroniser l'ensemble des *threads* OpenMP situés sur différents processus MPI.

Enoncé

On vous demande de compléter le fichier *barrier_hybride.c* afin que tous les *threads* OpenMP situés sur les différents processus MPI soient synchronisés lors d'un appel au sous-programme *barrierMPIOMP*.

TP2 — PingPong hybride parallèle

Objectif

Calculer la bande passante réseau soutenue entre deux nœuds.

Enoncé

On vous demande d'écrire un code hybride de PingPong en parallèle, permettant de déterminer la bande passante réseau entre deux nœuds.

- Dans une première version, on utilisera le niveau de support des threads `MPI_THREAD_FUNNELED` (i.e. 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). Dans ce cas, le nombre de flux de communication en parallèle sera égal au nombre de processus MPI par nœud.
- Dans une deuxième version, on utilisera le niveau de support des threads `MPI_THREAD_MULTIPLE` (i.e. entièrement *multithreadé* sans restrictions) et l'exécution se fera avec un processus MPI par nœud. Dans ce cas, le nombre de flux de communication en parallèle sera égal au nombre de threads actifs exécutant le PingPong en parallèle.

TP3 — Heat3D, de la version MPI à l'hybride

Objectif

Hybridiser une application parallèle MPI en rajoutant un niveau de parallélisme OpenMP.

Enoncé

1. Intégrer des directives OpenMP à la version parallèle MPI du code Heat3D en utilisant le niveau de support des threads `MPI_THREAD_FUNNELED`.
2. Transformer la version `MPI_THREAD_FUNNELED` en version `MPI_THREAD_MULTIPLE` de sorte à pouvoir paralléliser les communications MPI.
3. Comparer les performances obtenues avec les différentes versions (pur MPI, hybride `MPI_THREAD_FUNNELED`, hybride `MPI_THREAD_MULTIPLE`).

TP4 — HYDRO, de la version MPI à l'hybride

Objectif

Hybridiser une application parallèle MPI en rajoutant un niveau de parallélisme OpenMP.

Enoncé

1. Intégrer des directives OpenMP à la version parallèle MPI du code HYDRO.
2. Comparer les performances obtenues avec les différentes versions (pur MPI, hybride). L'extensibilité est-elle bonne ?
3. Quelles améliorations peuvent être apportées pour obtenir de meilleures performances ? Faites des essais et comparez.

Raisons pour faire de la programmation hybride

Raisons pour faire de la programmation hybride

Avantages de la programmation hybride (1)

- Optimisation de la consommation de mémoire totale
- Meilleure extensibilité par une réduction du nombre de messages MPI, du nombre de processus impliqués dans des communications collectives (`MPI_Alltoall` n'est pas très extensible) et par un meilleur équilibrage de charge.
- Peut lever certaines limitations algorithmiques (découpage maximum dans une direction par exemple).
- Amélioration des performances de certains algorithmes en réduisant le nombre de processus MPI (moins de domaines = meilleur préconditionneur si on laisse tomber les contributions des autres domaines).

Raisons pour faire de la programmation hybride

Avantages de la programmation hybride (2)

- Moins d'accès simultanés en entrées-sorties et taille moyenne des accès plus grande. Cela entraîne moins de charge sur les serveurs de méta-données avec des requêtes de tailles plus adaptées. Les gains potentiels sur une application massivement parallèle peuvent être importants.
- Un code parallèle MPI est une succession de phases de calcul et de communication. La granularité d'un code est définie comme le rapport moyen entre deux phases successives de calcul et de communication. Plus la granularité d'un code est importante, plus il est extensible. Comparée à l'approche pure MPI, l'approche hybride augmente significativement la granularité et donc l'extensibilité des codes.

Inconvénients de la programmation hybride

- Complexité et niveau d'expertise accrus.
- Nécessité d'avoir de bonnes performances MPI ET OpenMP (la loi d'Amdahl s'applique séparément aux 2 approches).
- Gains en performances non garantis (surcoûts supplémentaires...).

Applications pouvant en tirer parti

- Codes limités par la quantité de mémoire et ayant de nombreuses données répliquées entre les processus ou ayant des structures de données dépendant du nombre de processus pour leur dimension.
- Codes nécessitant de l'équilibrage de charge dynamique.
- Codes ayant une extensibilité MPI limitée (par des `MPI_Alltoall` par exemple).
- Codes travaillant sur des problèmes à parallélisme à *grain fin* ou un mélange *grain fin* et *gros grain*.
- Codes limités par l'extensibilité de leurs algorithmes.

Conclusion

Conclusions sur l'approche hybride MPI/OpenMP

Conclusions

- Approche pérenne, basée sur des standards reconnus (MPI et OpenMP), c'est un investissement à long terme.
- Les avantages de l'approche hybride comparés à l'approche pure MPI sont nombreux :
  - Gain mémoire significatif
  - Gain potentiel en performance (à nombre fixe de cœurs d'exécution), grâce à une meilleure adaptation du code à l'architecture cible
  - Gain potentiel en terme d'extensibilité, permet en théorie de repousser la limite d'extensibilité d'un code d'un facteur égal au nombre de cœurs du nœud à mémoire partagée
- Ces différents gains sont proportionnels au nombre de cœurs du nœud à mémoire partagée, nombre qui augmentera significativement à court terme (généralisation des processeurs multi-cœurs)
- Seule solution viable permettant de tirer parti des architectures massivement parallèles à venir (exascale, ...) ?

MPI et GPU

Accélérateurs GPU

Introduction

- Développement rapide des GPU dans les supercalculateurs (9 dans le top 10 du TOP500)
- GPU = Monstre de puissance :
  - Intel Xeon Platinum 8468 (DP) : 2 TFLOP/s
  - NVIDIA H100 (DP) : 34 TFLOP/s, 67 TFLOP/s Tensor Core
- avec une mémoire rapide :
  - Intel Xeon Platinum 8468 (DDR5) : $\approx$ 300 Go/s
  - NVIDIA H100 (HBM3) : 3,35 To/s
- Mémoire max 80 Go : besoin d'utiliser plusieurs GPU

Accélérateurs GPU

Interconnexions

Divers interconnexions possibles entre CPU et GPU et entre GPU :

- PCI-Express : max 64 Go/s
- NVLink : 25 Go/s par lien (jusqu'à 18 liens pour un H100)

Espaces mémoires

- Généralement deux espaces mémoires distincts : CPU et GPU
- Support plus ou moins avancé de la mémoire unifiée avec cohérence de cache
- APU (Accelerated Processing Unit) : une seule mémoire mais semble en recul

MPI et GPU

Différents niveaux de support MPI

- CUDA-aware : accepte les messages stockés en mémoire GPU
- GPUDirect : transfert RDMA de GPU à GPU (nécessite du matériel compatible)
- Support généralement partiel :
  - communications collectives souvent non supportées
  - types dérivés et mémoire unifiée : attention aux performances

Exemple de Jean Zay

- OpenMPI CUDA-aware GPUDirect
- Réseau OmniPath : ordre d'initialisation crucial (binding GPU avant MPI_Init)
- Transfert via :
  - NVLink pour communication P2P entre GPU d'un nœud
  - PCI-Express vers carte réseau pour communication P2P entre GPU sur des nœuds différents

MPI et GPU

Exemple de communication

```
#if USE_CUDA_AWARE
    #pragma acc host_data use_device(inBuffer, outBuffer)
#else
    #pragma acc update host(outBuffer)
#endif
    MPI_Sendrecv(outBuffer, sizeOut, MPI_DOUBLE, procOut, tag,
                 inBuffer, sizeIn, MPI_DOUBLE, procIn, tag,
                 comm, MPI_STATUS_IGNORE)
#if USE_CUDA_AWARE
    #pragma acc end host_data
#else
    #pragma acc update device(inBuffer)
#endif
```