Programmation hybride MPI-OpenMP

Pierre-Francois.Lavallee@idris.fr
Philippe.Wautelet@aero.obs-mip.fr

CNRS — IDRIS / LA

Version 3.0.1 — 1 décembre 2017

Disponibilité et mise à jour

Ce document est amené à être remis à jour régulièrement.
La version la plus récente est disponible sur le serveur web de l'IDRIS, rubrique *Supports de cours* :

https://cours.idris.fr

IDRIS - Institut du développement et des ressources en informatique scientifique
Rue John Von Neumann
Bâtiment 506
BP 167
91403 ORSAY CEDEX
France
http://www.idris.fr

Sommaire I

Sommaire II

Préambule

Présentation de la formation

L'objet de cette formation est de faire une présentation de la programmation hybride, ici MPI+OpenMP, ainsi qu'un retour d'expériences d'implémentations effectives d'un tel modèle de parallélisation sur plusieurs codes applicatifs.

- Le chapitre *Introduction* a pour but de montrer, au travers des évolutions technologiques des architectures et des contraintes du parallélisme, que le passage à la parallélisation hybride est indispensable si l'on veut tirer parti de la puissance des machines massivement parallèles de dernière génération.
- Or, un code hybride ne pourra être performant que si les modes parallèles MPI et OpenMP ont déjà été optimisés.
- Le chapitre *Programmation hybride* est entièrement dédié à l'approche hybride MPI+OpenMP. Les avantages liés à la programmation hybride sont nombreux :
  - gains mémoire,
  - meilleures performances,
  - meilleur équilibrage de charge,
  - granularité plus importante, d'où une extensibilité améliorée,
  - meilleure adéquation du code aux spécificités matérielles de l'architecture cible.

  Cependant, comme vous pourrez le constater dans les TPs, l'implémentation sur une application réelle nécessite un investissement important et une maîtrise approfondie de MPI et d'OpenMP.

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 = $frequence^3$ (pour une technologie donnée).
- Puissance dissipée par cm^2 limitée par le refroidissement.
- Coût de l'énergie.

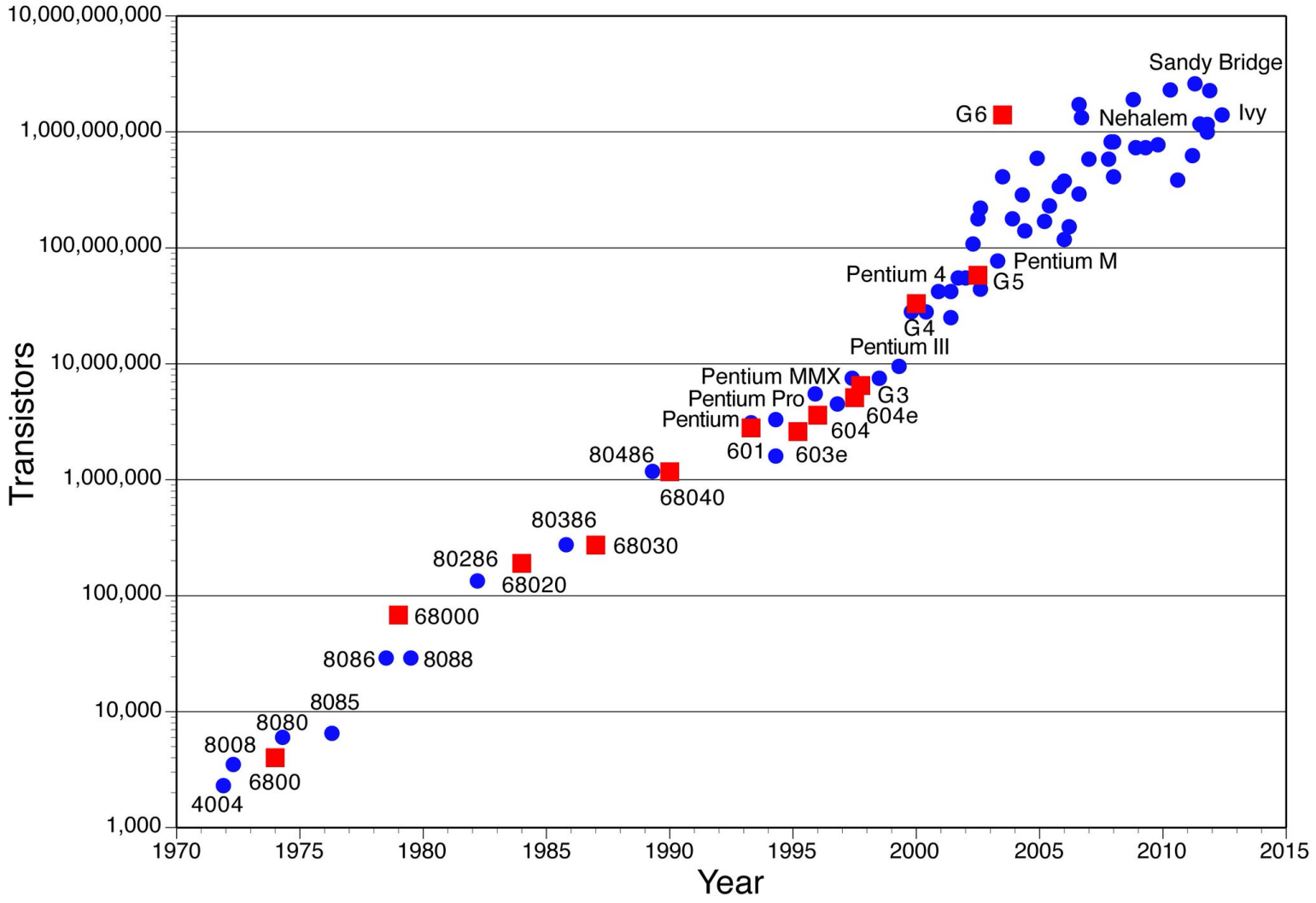
Loi de Moore et consommation électrique

- 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 Skylake ont jusqu'à 28 cœurs et supportent 56 *threads* simultanément, les AMD EPYC 32 cœurs et supportent 64 *threads* simultanément).

=> certaines architectures privilégient les cœurs à basse fréquence, mais en très grand nombre (IBM Blue Gene, Intel Xeon Phi)

Loi de Moore



http://en.wikipedia.org/wiki/Moore%27s_law

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

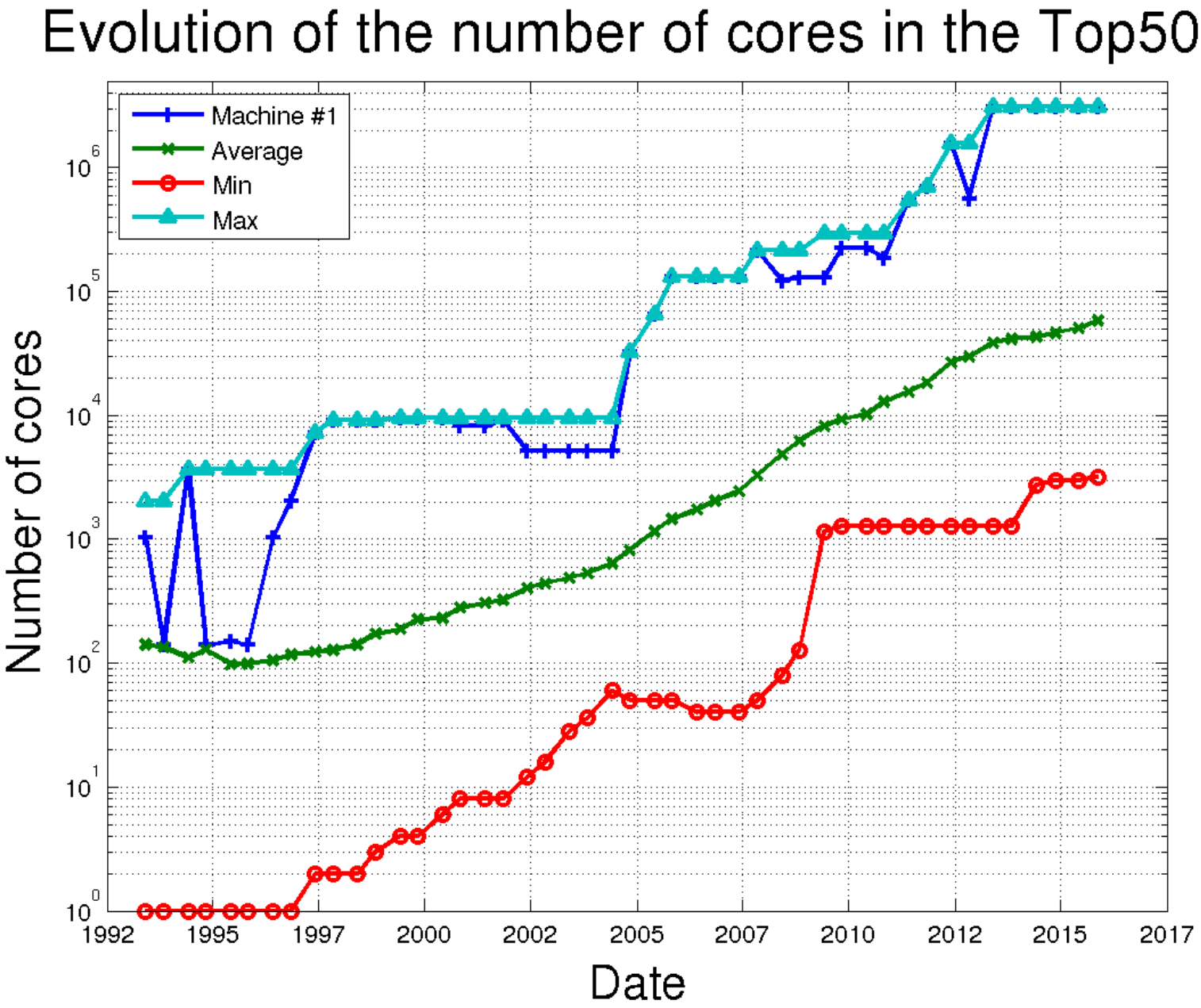
- L'écart entre les performances théoriques des cœurs et la mémoire se creuse.
- Les processeurs passent de plus en plus de cycles à attendre les données.
- Il est de plus en plus difficile d'exploiter la performance des processeurs.

Solutions partielles

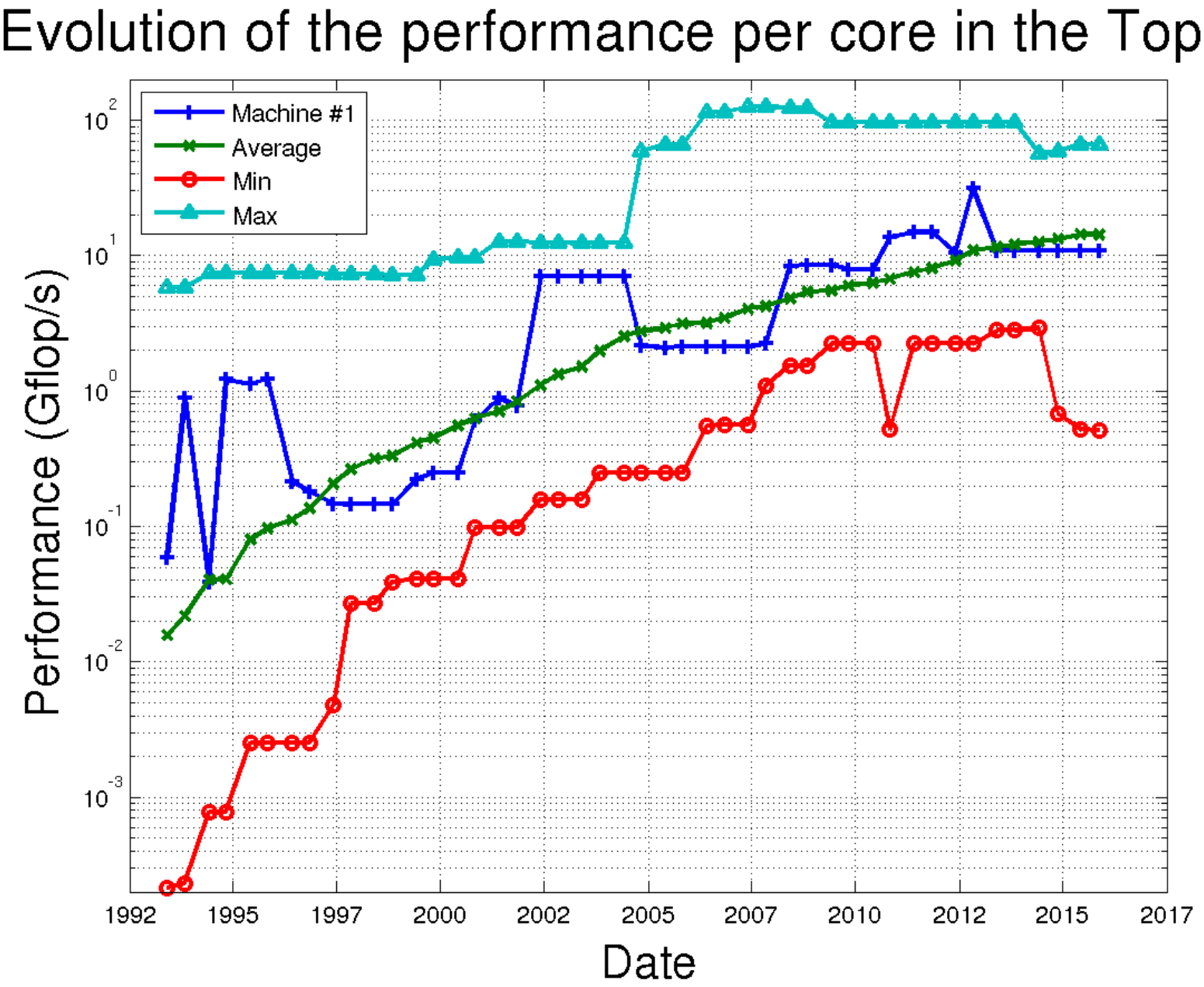
- L'ajout de mémoires caches est essentiel.
- Parallélisation des accès via plusieurs bancs mémoire comme sur les architectures vectorielles (Intel Skylake : 6 canaux, AMD EPYC : 8 canaux, ARM ThunderX2 : 8 canaux).
- Si la fréquence des cœurs stagne ou baisse, l'écart pourrait se réduire.

Top 500 - Évolution de la performance

Top 500


Evolution of the number of cores in the Top500
Machine #1
Average
Min
Max
Number of cores
10^0
10^1
10^2
10^3
10^4
10^5
10^6
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015
2017
Date

Top 500


Evolution of the performance per core in the Top500
Machine #1
Average
Min
Max
Performance (Gflop/s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015
2017
Date

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, FPGA ou PEZY-SC2 avec des processeurs standards par exemple).
- 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 et est sur certaines machines beaucoup plus basse que sur un simple PC portable (IBM Blue Gene, Intel Xeon Phi).
- 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.

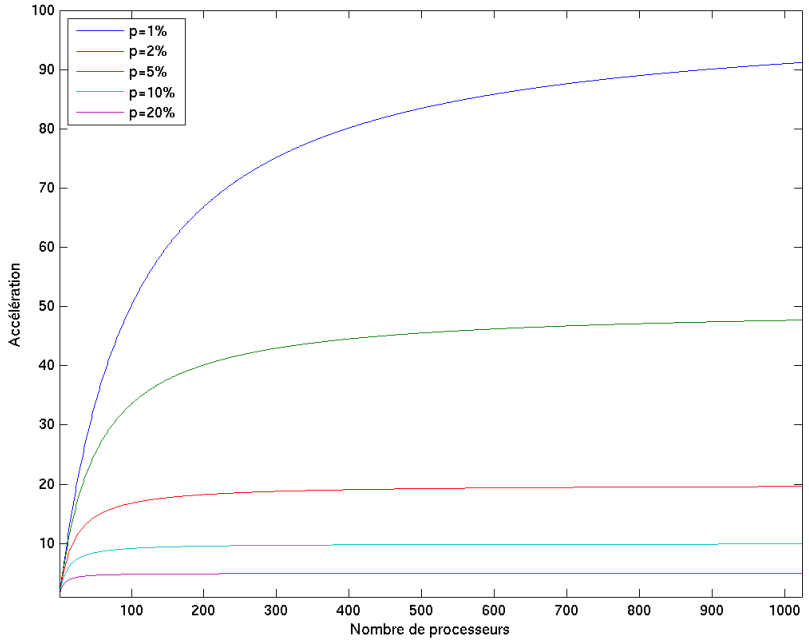
Interprétation

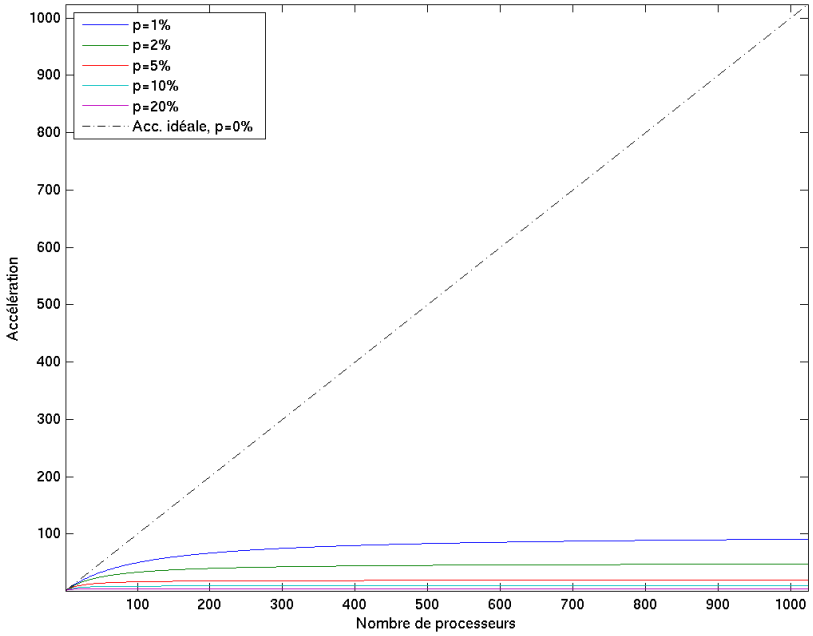
Quel que soit le nombre de cœurs, l'accélération est toujours inférieure à l'inverse du pourcentage que représente la partie purement séquentielle.

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

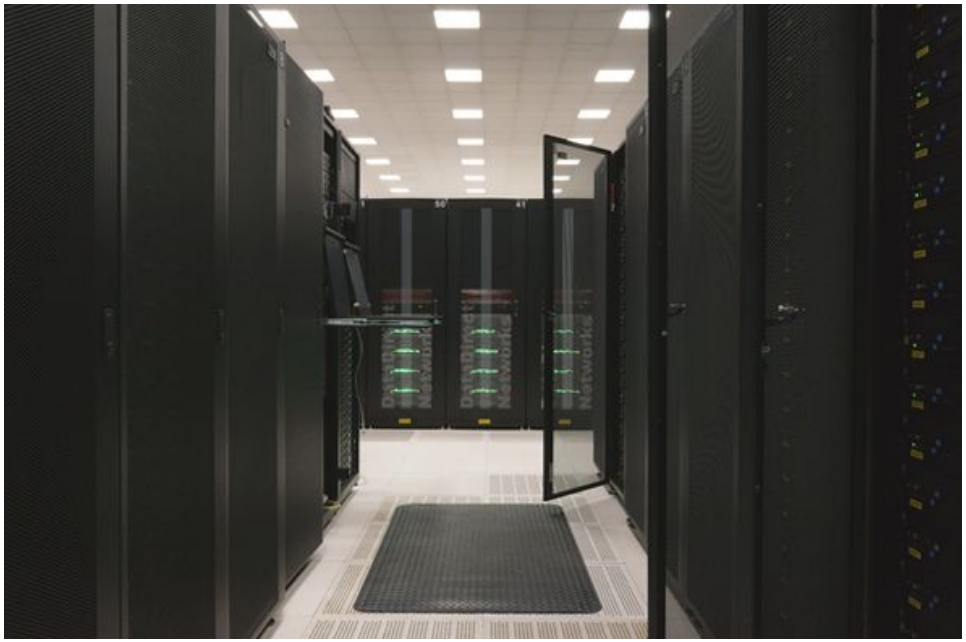
Evolution des méthodes de programmation

- 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 et semble devenir l'approche privilégiée pour les supercalculateurs.
- La programmation sur GPU se développe, mais reste complexe et nécessite un troisième niveau de parallélisme (MPI+OpenMP+GPU).
- D'autres formes de programmation hybride sont également testées (MPI + GPU...) avec généralement MPI comme ingrédient.
- De nouveaux langages de programmation parallèle apparaissent (UPC, Coarray-Fortran, langages PGAS, X10, Chapel...), mais ils sont en phase expérimentale (à des niveaux d'avancement très variables). Certains de ces langages sont très prometteurs. Il reste à voir si ils vont être utilisés dans de vraies applications.

Configuration IDRIS

Turing : IBM Blue Gene/Q

Ada : IBM x3750

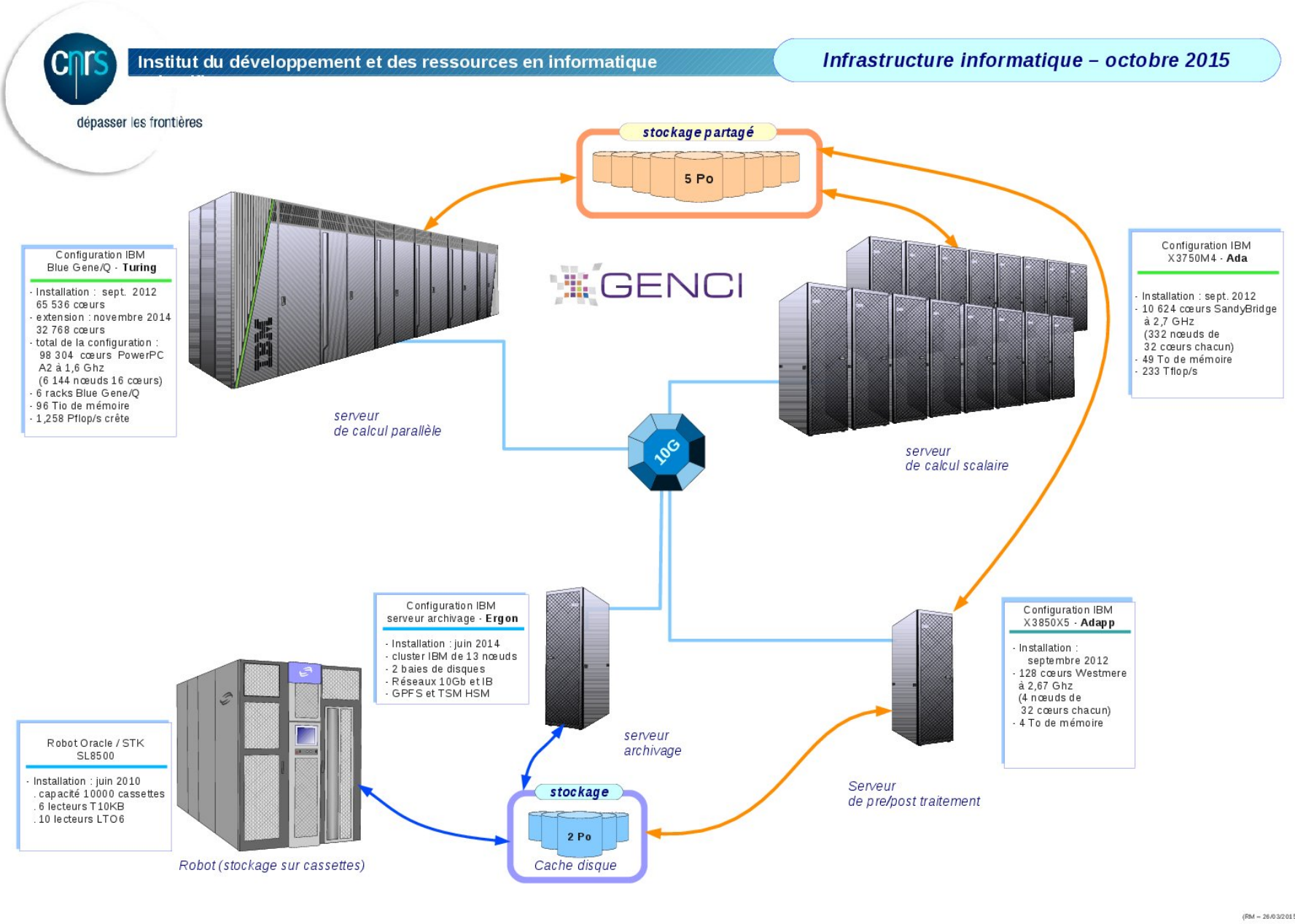


Configuration IDRIS

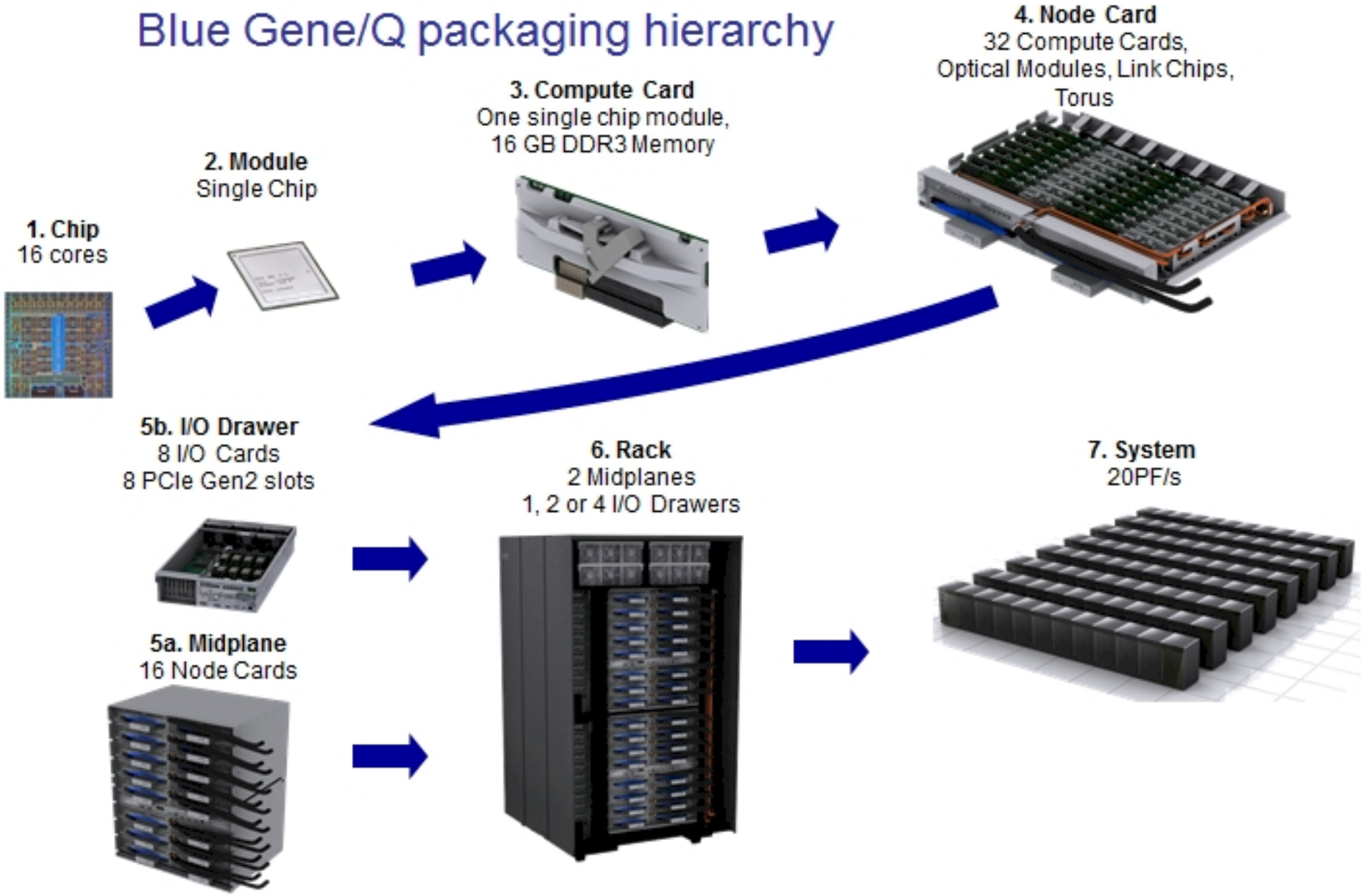
Chiffres importants

- **Turing : 6 *racks* Blue Gene/Q** :
  - 6.144 nœuds
  - 98.304 cœurs
  - 393.216 *threads*
  - 96 Tio
  - 1,258 Pflop/s
  - 636 kW (106 kW/*rack*)
- **Ada : 15 *racks* IBM x3750M4** :
  - 332 nœuds de calcul et 4 nœuds pour pré et post-traitement
  - 10.624 cœurs Intel SandyBridge à 2,7 GHz
  - 46 Tio
  - 230 Tflop/s
  - 366 kW
- 2,2 Po utiles et 50 Gio/s sur disques partagés entre BG/Q et Intel
- 1 MW pour la configuration complète (hors climatisation)

Configuration IDRIS



Architecture Blue Gene/Q



Programmation hybride

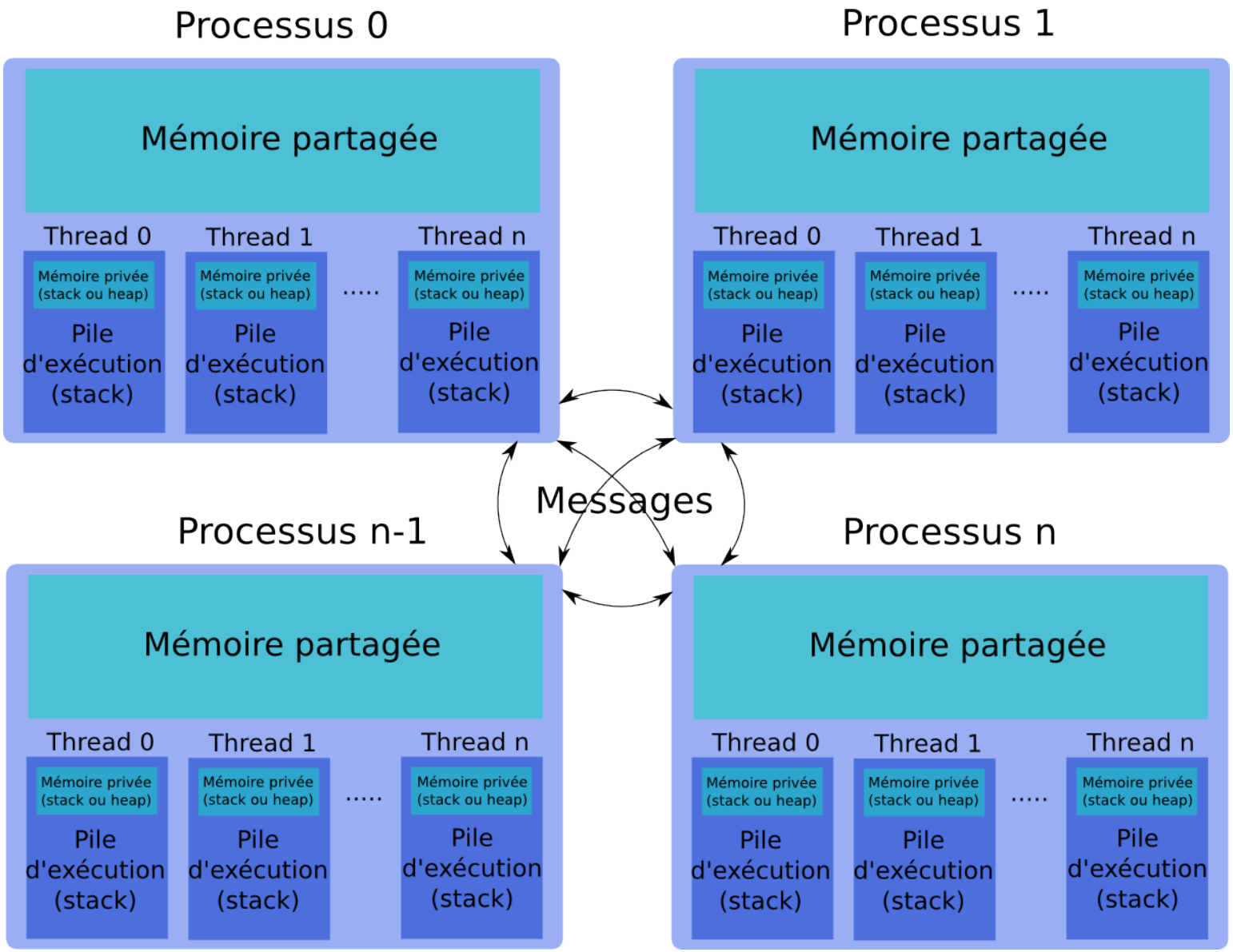
Définitions

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



Avantages de la programmation hybride (1)

- 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.
- Meilleure adéquation à l'architecture des calculateurs modernes (nœuds à mémoire partagée interconnectés, machines NUMA...), alors que MPI seul est une approche *flat*.
- Optimisation de la consommation de mémoire totale (grâce à l'approche mémoire partagée OpenMP, gain au niveau des données répliquées dans les processus MPI et de la mémoire utilisée par la bibliothèque MPI elle-même).
- Réduction de l'empreinte mémoire lorsque la taille de certaines structures de données dépend directement du nombre de processus MPI.
- 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).
- 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.

Raisons pour faire de la programmation hybride

Avantages de la programmation hybride (2)

- Moins de fichiers à gérer si on utilise une approche où le nombre de fichiers est proportionnel au nombre de processus MPI (approche très fortement déconseillée dans un cadre de parallélisme massif).
- Certaines architectures nécessitent de lancer plusieurs *threads* (ou processus) par cœur pour utiliser correctement les unités de calcul.
- 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

Applications pouvant en tirer parti

- Codes ayant une extensibilité MPI limitée (par des `MPI_Alltoall` par exemple).
- Codes nécessitant de l'équilibrage de charge dynamique.
- 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.
- Bibliothèque MPI locale peu performante pour les communications intra-nœud.
- De nombreuses applications massivement parallèle.
- 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.
- ...

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 et le *multithreading*

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 et le *multithreading*

Autres sous-programmes MPI

`MPI_Query_thread` retourne le niveau de support du processus appellant :

```
int MPI_Query_thread(int *provided)
MPI_QUERY_THREAD(PROVIDED, IERROR)
```

`MPI_Is_thread_main` retourne si le *thread* appellant est le *thread* principal ou pas (important si niveau `MPI_THREAD_FUNNELED` et pour l'appel à `MPI_Finalize`) :

```
int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)
```

MPI et le *multithreading*

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 (cela se vérifiait facilement sur Babel et Vargas).
- 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).

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.

Implications des différents niveaux de support

Le niveau de support du *multithreading* fourni par la bibliothèque MPI impose certaines conditions et restrictions à l'utilisation d'OpenMP :

- `MPI_THREAD_SINGLE` : OpenMP ne peut pas être utilisé
- `MPI_THREAD_FUNNELED` : les appels MPI doivent être faits en dehors des régions parallèles OpenMP ou dans les regions OpenMP *master* ou dans des zones protégées par un appel à `MPI_Is_thread_main`
- `MPI_THREAD_SERIALIZED` : dans les régions parallèles OpenMP, les appels MPI doivent être réalisés dans des sections *critical* (si nécessaire pour garantir un seul appel MPI simultané)
- `MPI_THREAD_MULTIPLE` : aucune restriction

Etat des implémentations actuelles

Implémentation	*Niveau supporté*	*Remarques*
MPICH	`MPI_THREAD_MULTIPLE`	
OpenMPI	`MPI_THREAD_MULTIPLE`	Doit être compilé avec l'option *–enable-mpi-threads*
IBM Blue Gene/Q	`MPI_THREAD_MULTIPLE`	
IBM PEMPI	`MPI_THREAD_MULTIPLE`	
BullxMPI (1.2.8.4)	`MPI_THREAD_SERIALIZED`	
Intel - MPI	`MPI_THREAD_MULTIPLE`	Utiliser *-mt_mpi*
SGI - MPT	`MPI_THREAD_MULTIPLE`	Utiliser *-lmpi_mt*

Programmation hybride, l'aspect gain mémoire

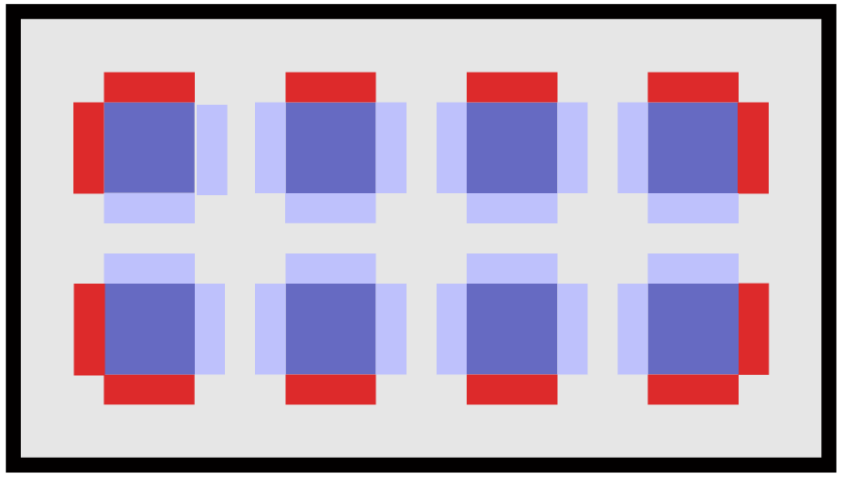
Pourquoi un gain mémoire ?

- 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.
- Le gain associé à la disparition des mailles fantômes intra-nœud est loin d'être négligeable. Il dépend fortement de l'ordre de la méthode utilisée, du type de domaine (2D ou 3D), du type de décomposition de domaine (suivant une seule dimension, suivant toutes les dimensions) et du nombre de cœurs du nœud SMP.
- L'empreinte mémoire des *buffers* systèmes associés à MPI est non négligeable et croît avec le nombre de processus. Par exemple, pour un réseau Infiniband avec 65.000 processus MPI, l'empreinte mémoire des *buffers* systèmes atteint 300 Mo par processus, soit pratiquement 20 To au total !

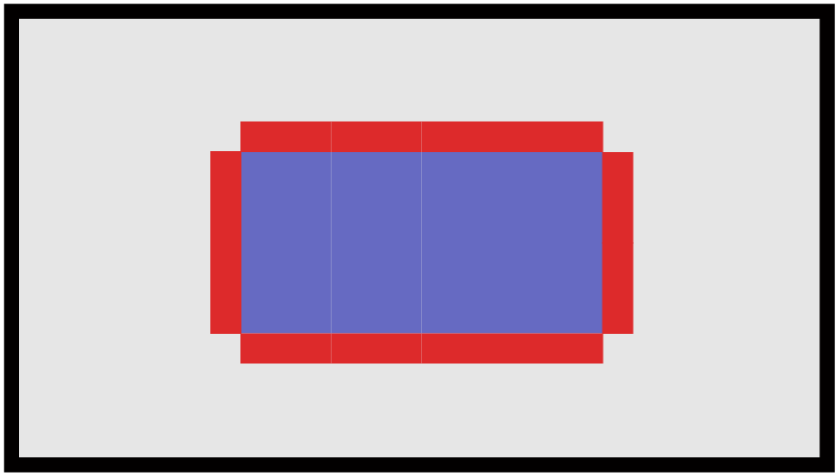
Programmation hybride, l'aspect gain mémoire

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

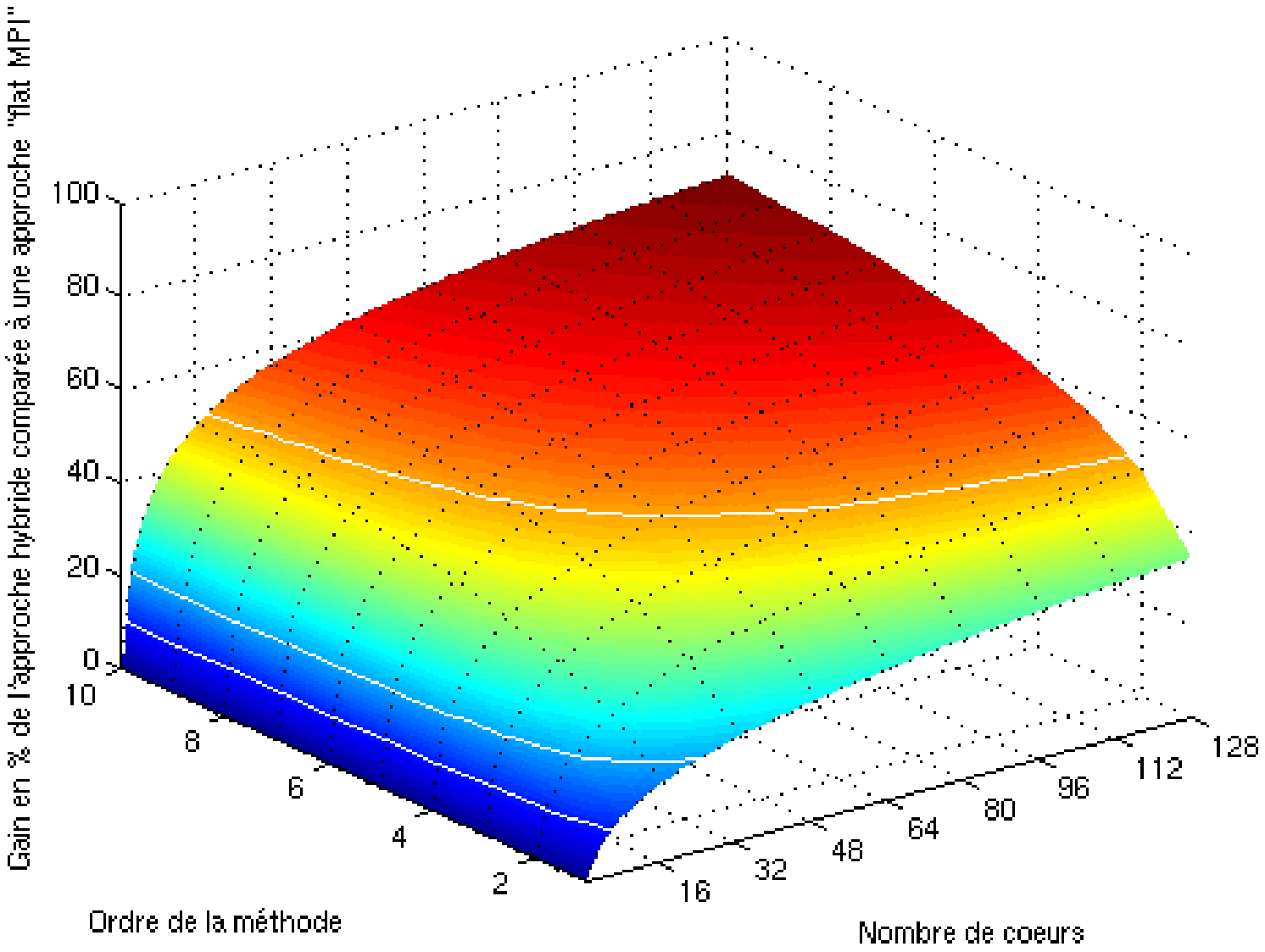
Programmation hybride, l'aspect gain mémoire

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)
- Ce qui va rendre quasi obligatoire le passage à la programmation hybride...

Utilisation optimale du réseau d'interconnexion

Comment optimiser l'utilisation du réseau d'interconnexion inter-nœud ?

- L'approche hybride vise à utiliser au mieux les ressources matérielles disponibles (mémoire partagée, hiérarchie mémoire, réseau de communication),
- Une des difficultés de la programmation hybride est de générer un nombre suffisant de flux de communications de façon à utiliser au mieux le réseau de commmunication inter-nœud.
- En effet, les débits des réseaux d'interconnexion inter-nœuds des architectures récentes sont élevés (débit crête de 10 Go/s sur Ada par exemple) et un seul flux de données ne peut le saturer, seule une fraction du réseau est réllement utilisée, le reste étant perdu...
- Développement IDRIS d'un petit benchmark SBPR (Saturation Bande Passante Réseau), simple test de PingPong en parallèle, visant à déterminer le nombre de flux concurrents nécéssaires pour saturer le réseau.

Utilisation optimale du réseau d'interconnexion

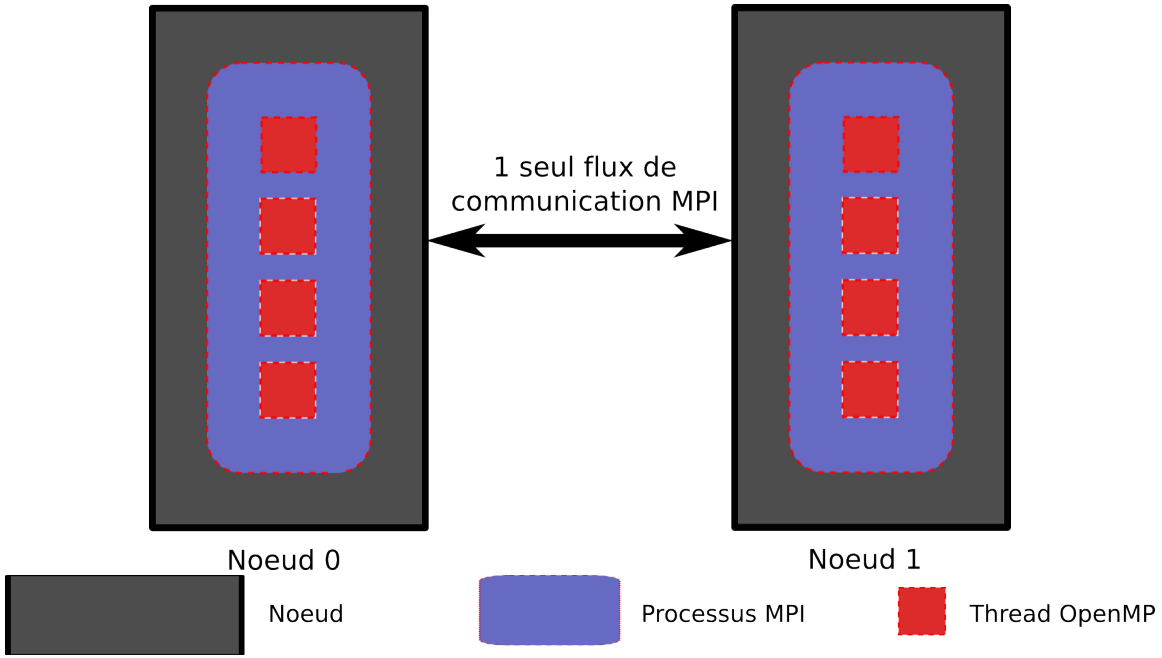
SBPR version `MPI_THREAD_FUNNELED`

Approche `MPI_THREAD_FUNNELED` :

- On augmente la bande passante réseau réellement utilisée en augmentant le nombre de processus MPI par nœud (i.e. on génère autant de flux de communication en parallèle que de processus MPI par nœud).
- La solution basique consistant à utiliser autant de *threads* OpenMP que de cœurs au sein d'un nœud et autant de processus MPI que de nœuds n'est généralement pas la meilleure les ressources n'étant pas utilisées de façon optimale, en particulier le réseau...
- On cherche à déterminer la valeur optimale du ratio entre le nombre de processus MPI par nœud et le nombre de threads OpenMP par processus MPI. Plus ce ratio est grand, meilleur est le débit du réseau inter-nœud, mais moins bonne est la granularité... Un compromis est à trouver.
- Le nombre de processus MPI (i.e. de flux de données à gérer simultanément) nécessaire pour saturer le réseau varie fortement d'une architecture à une autre.
- Cette valeur pourra être un bon indicateur du ratio optimal du nombre de processus MPI/nombre de *threads* OpenMP par nœud d'une application hybride.

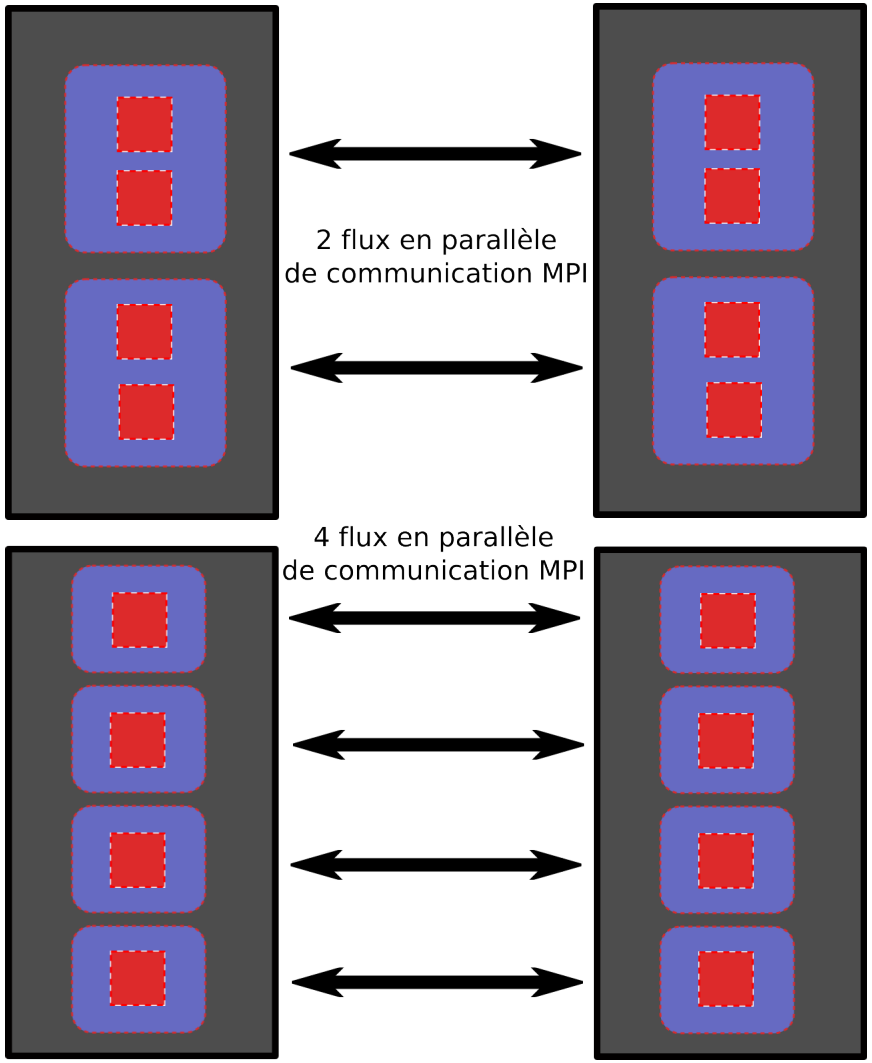
Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_FUNNELED` : exemple sur un nœud SMP à 4 cœurs (BG/P)



Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_FUNNELED` : exemple sur un nœud SMP à 4 cœurs (BG/P)



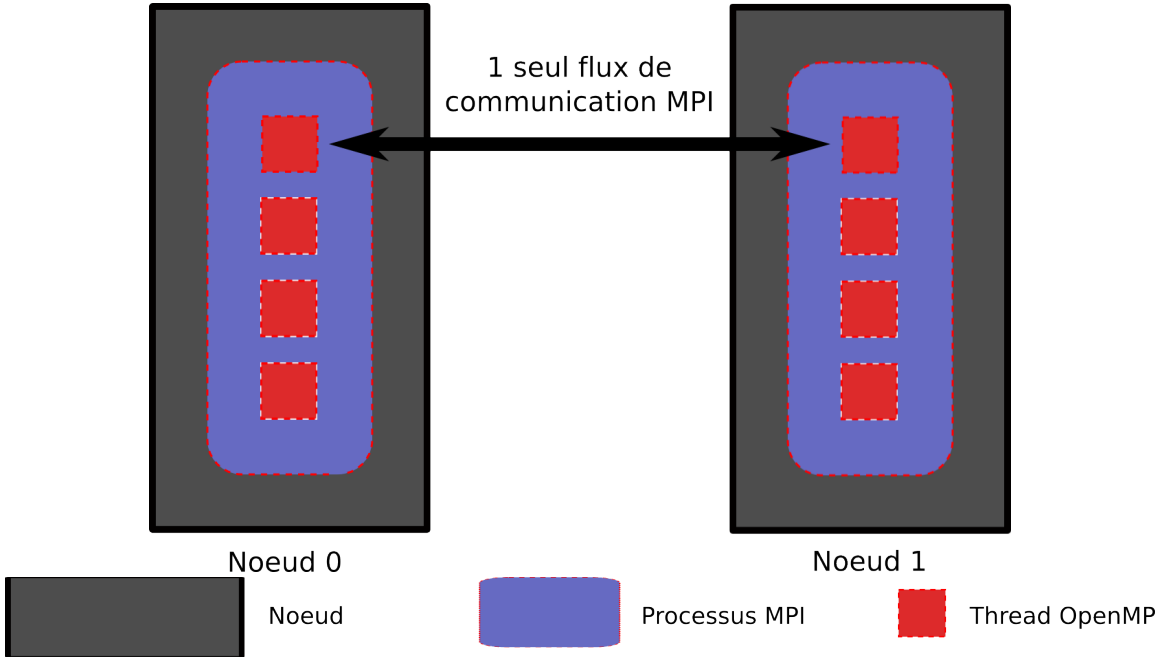
Utilisation optimale du réseau d'interconnexion

SBPR version `MPI_THREAD_MULTIPLE`

Approche `MPI_THREAD_MULTIPLE` :

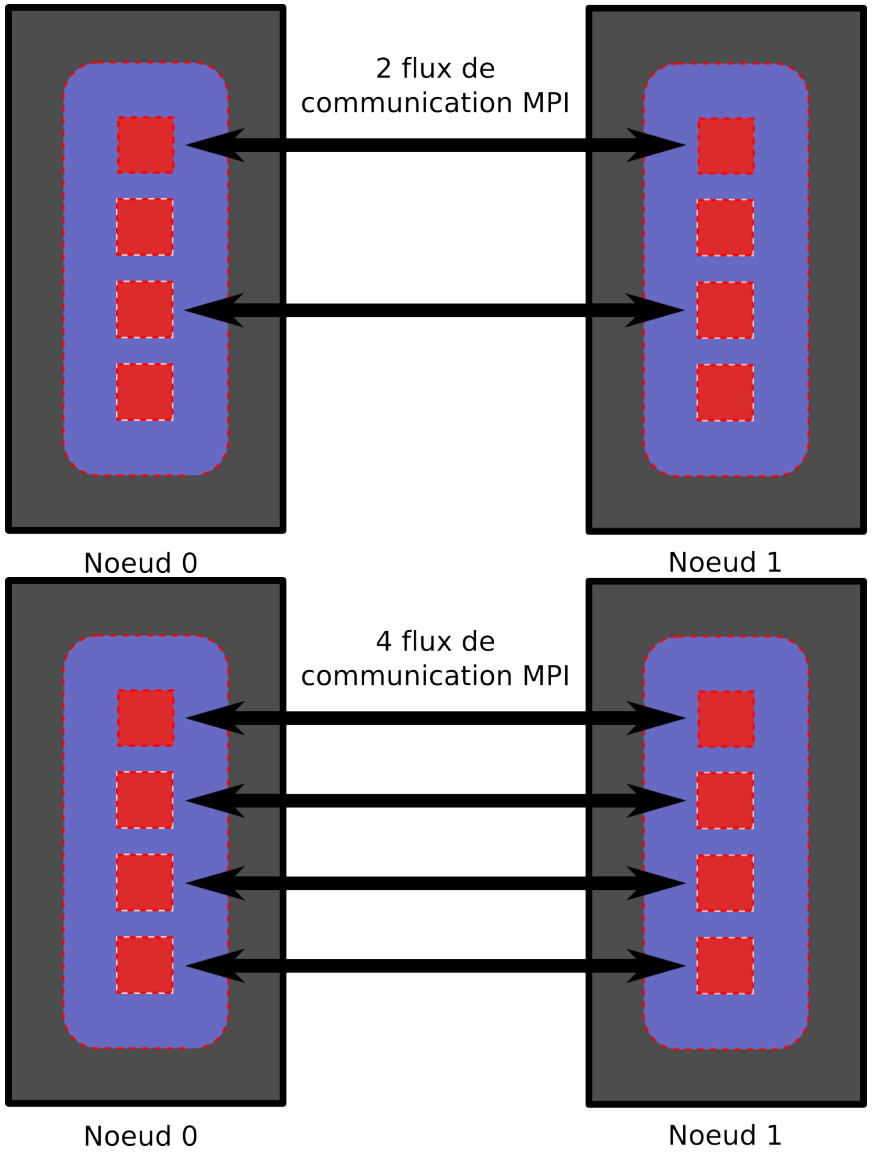
- On augmente la bande passante réseau réllement utilisée en augmentant le nombre de threads OpenMP qui participent aux communications.
- On a un unique processus MPI par nœud, on cherche le nombre minimum de threads de communication nécessaire pour saturer le réseau.

SBPR `MPI_THREAD_MULTIPLE` : exemple sur un nœud SMP à 4 cœurs (BG/P)



Utilisation optimale du réseau d'interconnexion

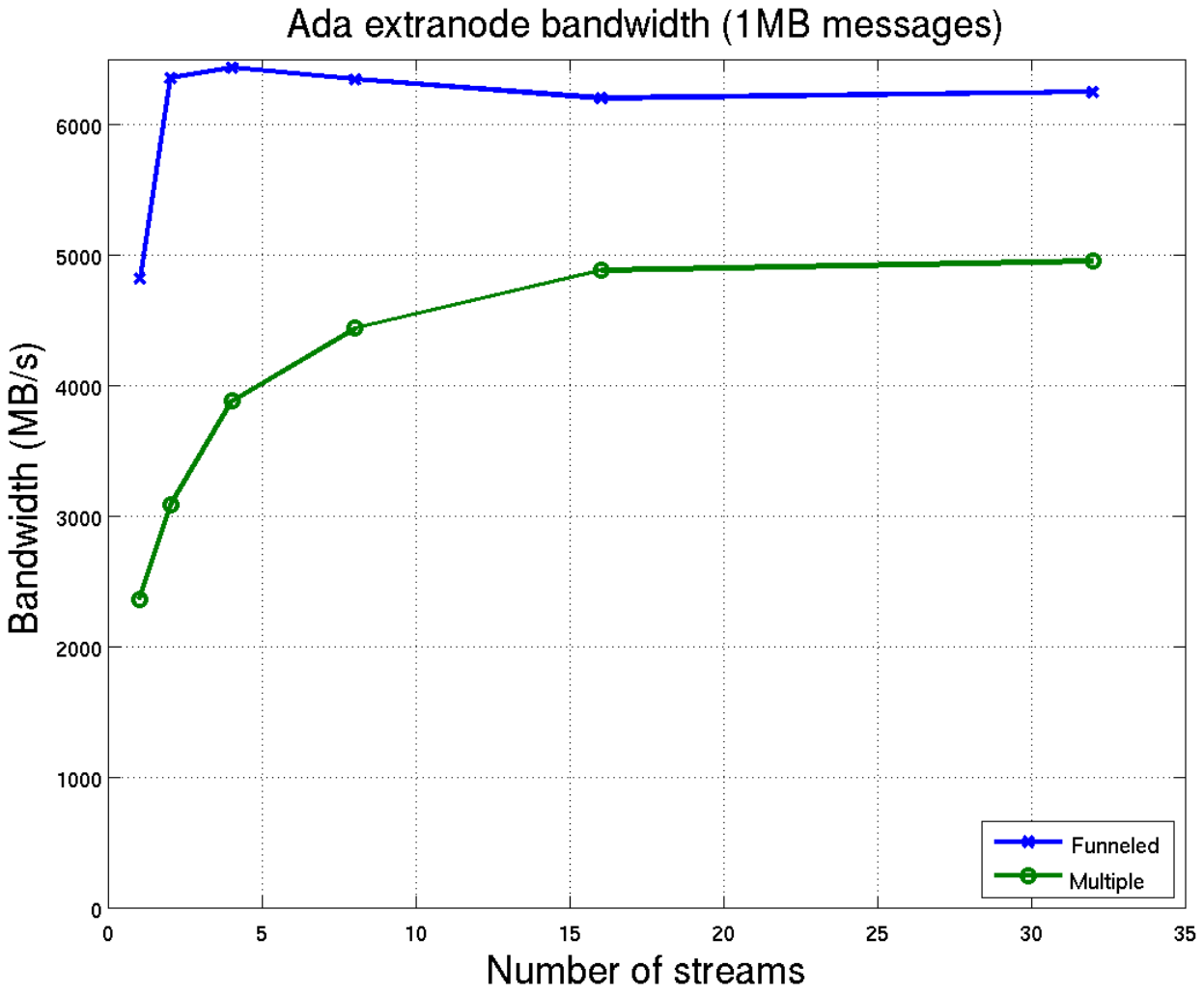
SBPR `MPI_THREAD_MULTIPLE` : exemple sur un nœud SMP à 4 cœurs (BG/P)



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Ada

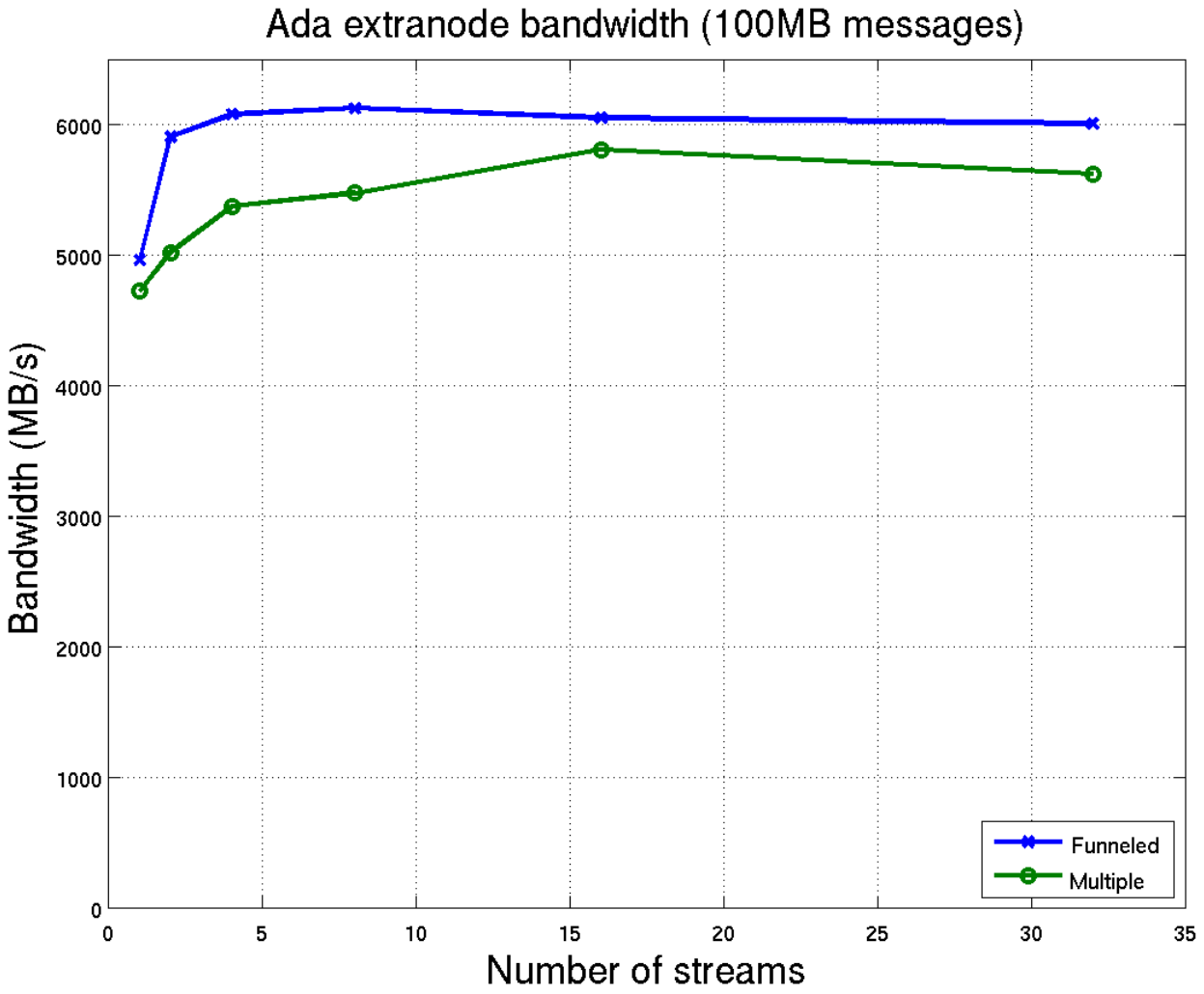
2 liens en // Infiniband FDR10, débit crête 10 Go/s.



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Ada

2 liens en // Infiniband FDR10, débit crête 10 Go/s.



Utilisation optimale du réseau d'interconnexion

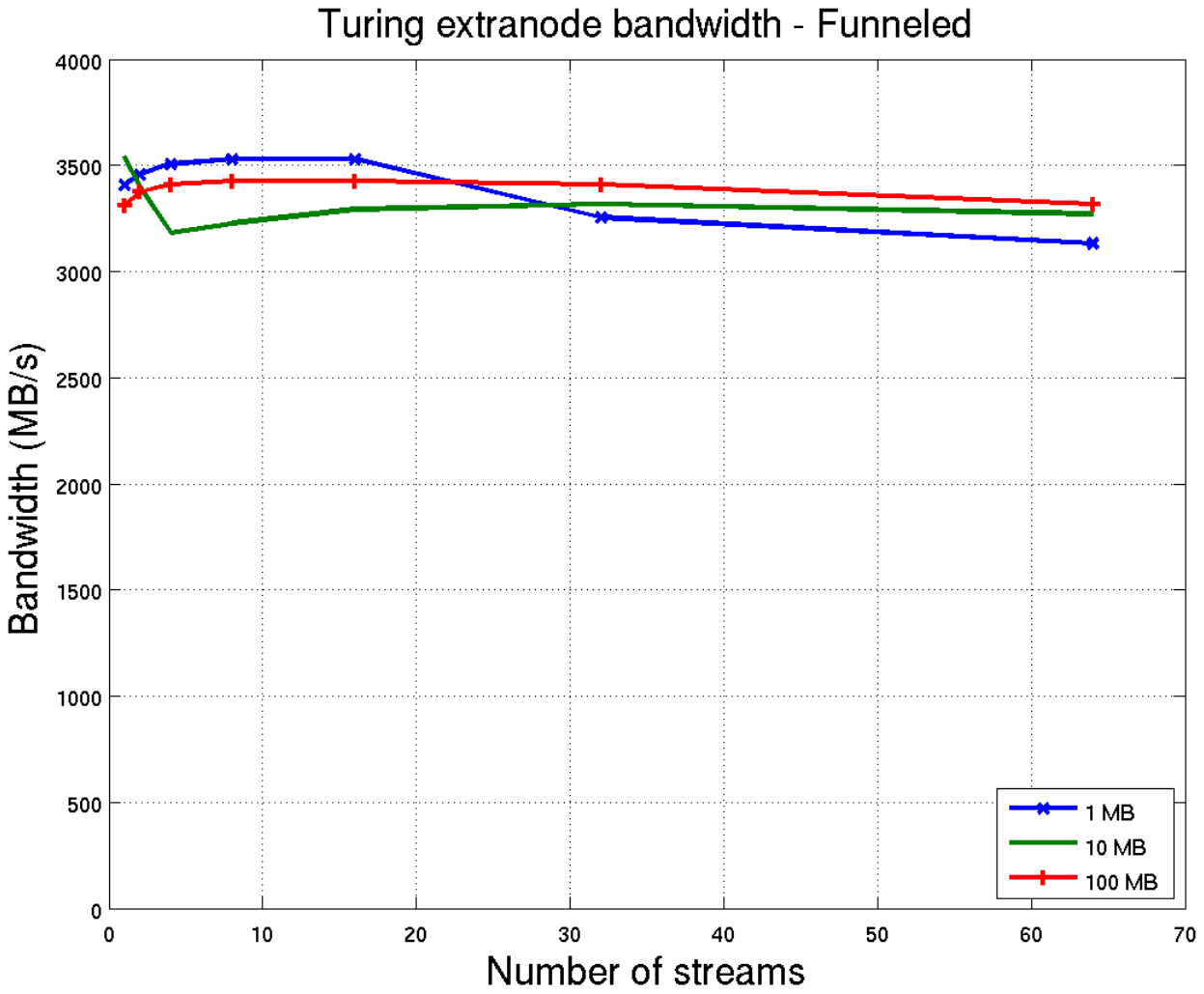
SBPR — Résultats sur Ada

- Avec 1 seul flux, on n'utilise qu'une partie de la bande passante réseau inter-nœud.
- En mode `MPI_THREAD_FUNNELED`, la saturation des liens réseaux inter-nœud d'Ada apparaît dès 2 flux en parallèle (i.e. 2 processus MPI par nœud).
- En mode `MPI_THREAD_MULTIPLE`, la saturation des liens réseaux inter-nœud d'Ada n'apparaît qu'à partir de 16 flux en parallèle (i.e. 16 threads participants aux communications par nœud).
- Les 2 approches `MPI_THREAD_FUNNELED` et `MPI_THREAD_MULTIPLE` sont parfaitement utilisables sur Ada avec néanmoins un avantage de performance pour la première.

Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Turing

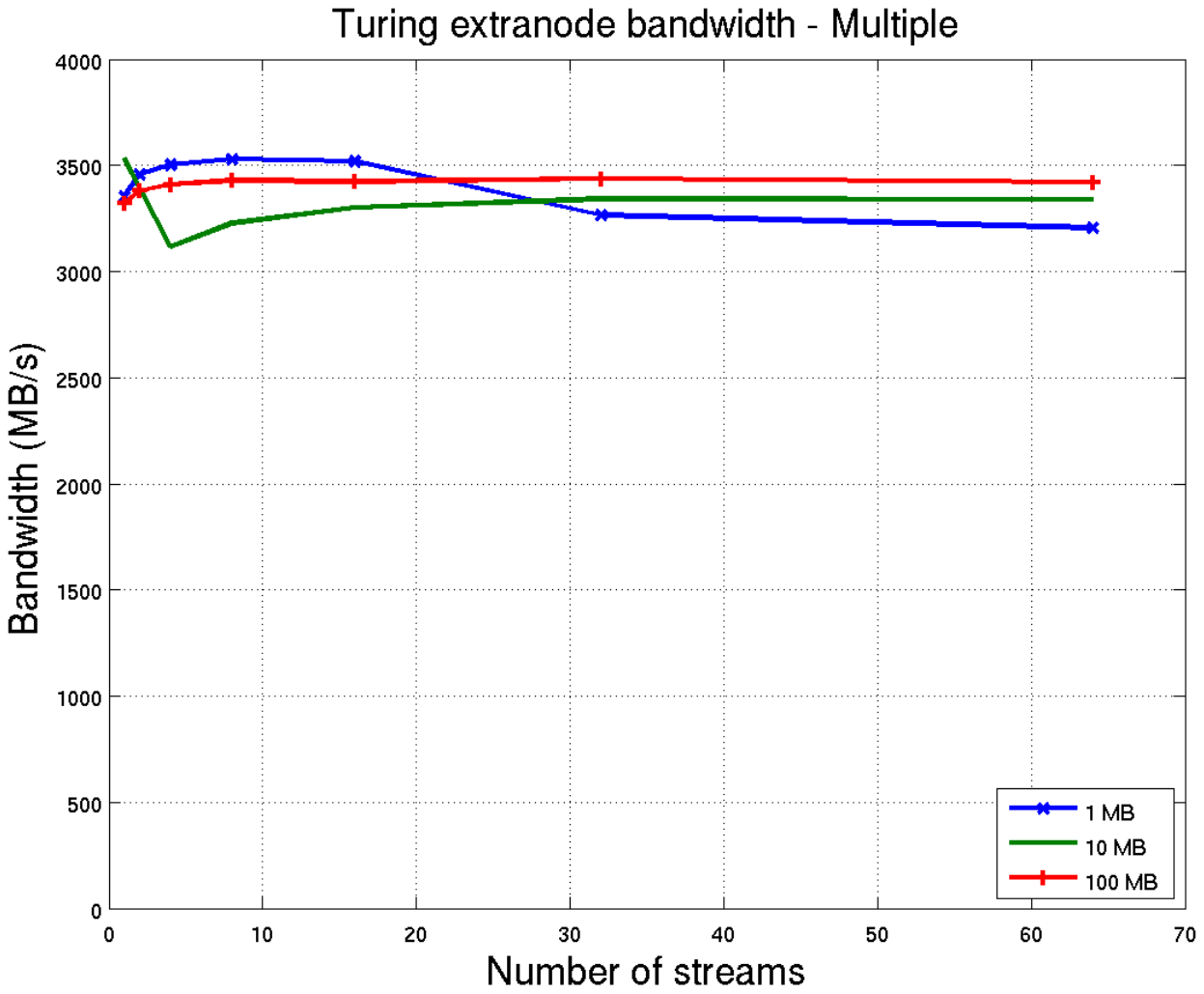
2 liens en // (direction E du tore 5D), débit crête 4 Go/s.



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Turing

2 liens en // (direction E du tore 5D), débit crête 4 Go/s.



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Turing

- L'utilisation d'un seul flux de données (i.e. 1 seul processus MPI ou thread par nœud) suffit à saturer totalement le réseau d'interconnexion entre deux nœuds voisins.
- Les performances des versions `MPI_THREAD_MULTIPLE` et `MPI_THREAD_FUNNELED` sont comparables.
- Le débit atteint est d'environ 3,5 Go/s, soit environ 85% de la bande passante crête réseau inter-nœud (selon la direction E du tore 5D).

Effets architecture non uniforme

Architecture non uniforme

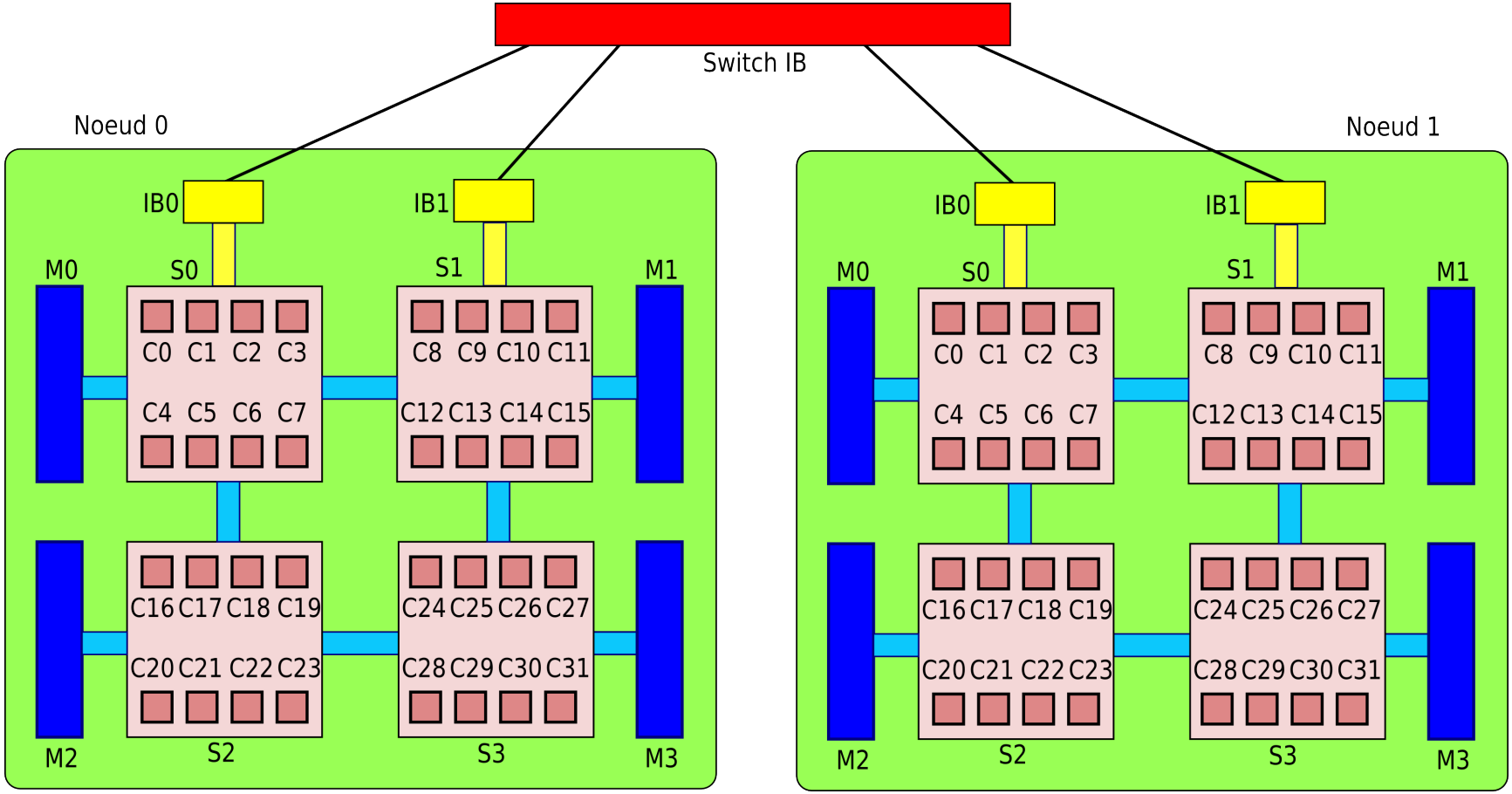
La plupart des machines de calcul modernes ont une architecture non uniforme :

- 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) => voir aussi placement des processus.

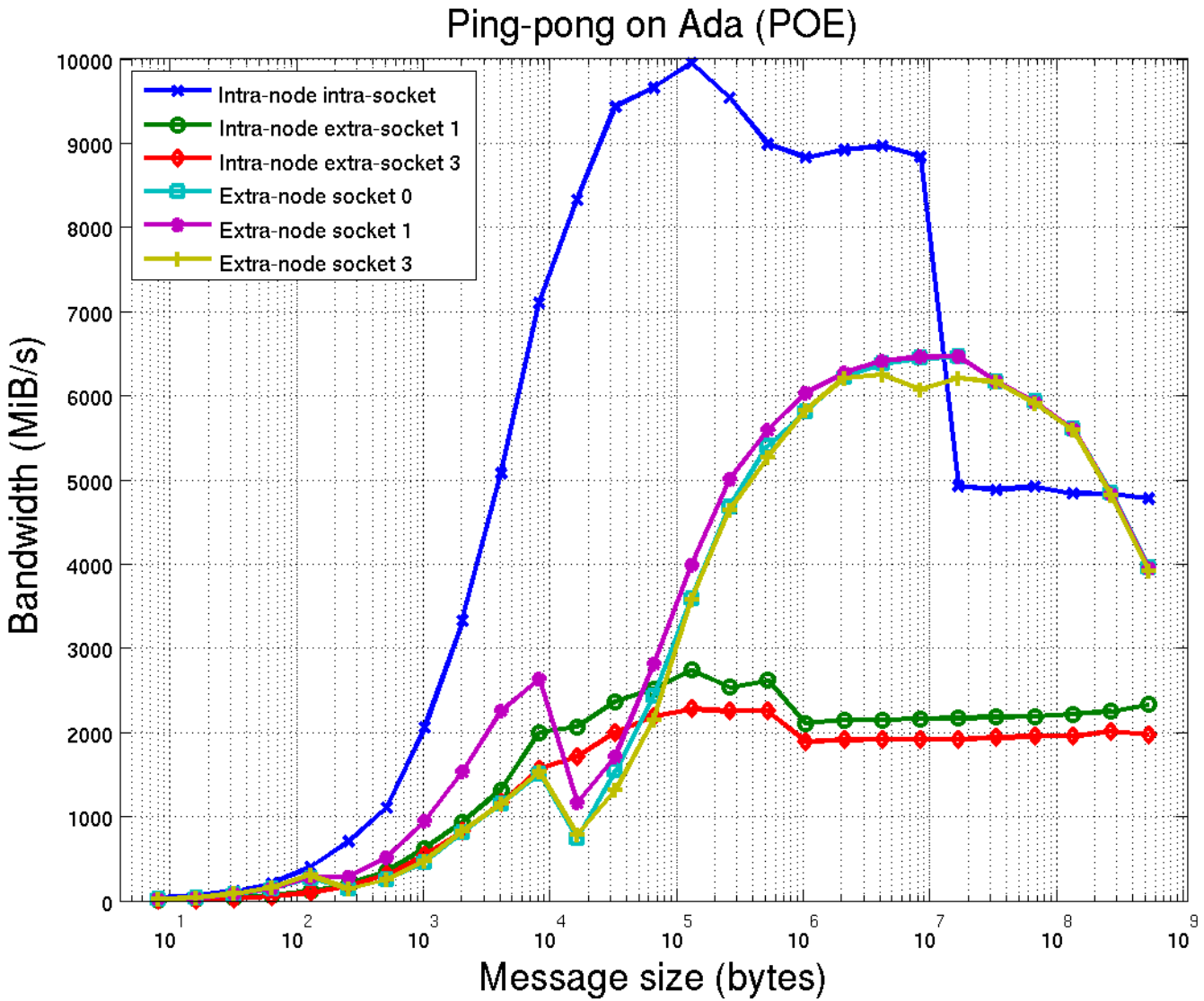
Effets

- Les performances des communications MPI diffèrent d'un cœur à l'autre même à l'intérieur d'un nœud.
- Le placement des processus est important à l'intérieur d'un nœud et entre les nœuds.
- La complexité des architectures actuelles rend difficile la compréhension des problèmes de performance et l'optimisation.

Architecture non uniforme sur Ada



Ping Pong sur Ada



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-zone fournit 3 applications différentes avec 8 tailles de problème différentes.
- *Benchmark* utilisé assez couramment.
- Les sources sont disponibles à l'adresse : `http://www.nas.nasa.gov/Resources/Software/software.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.
- L'approche hybride devrait améliorer la situation.

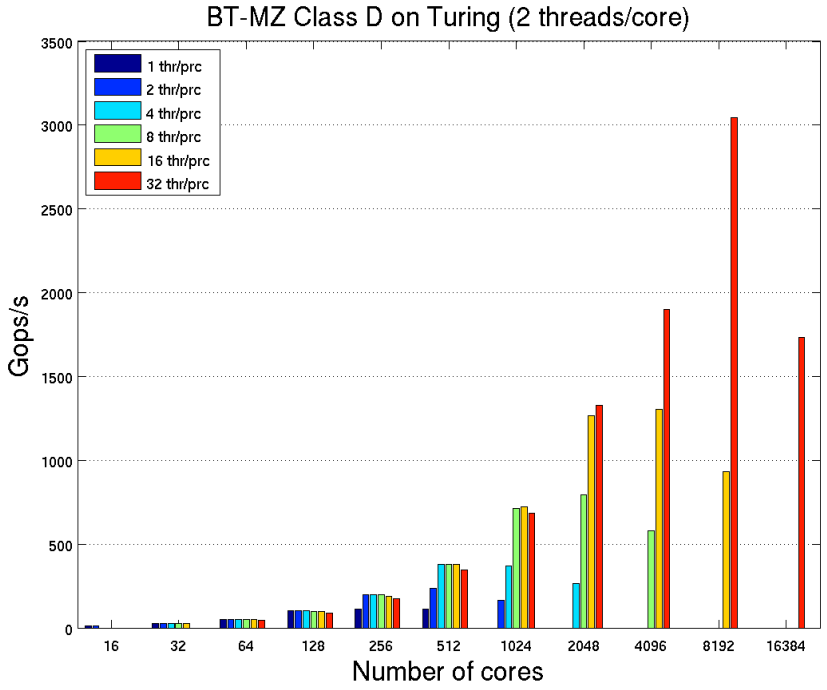
Application choisie : SP-MZ

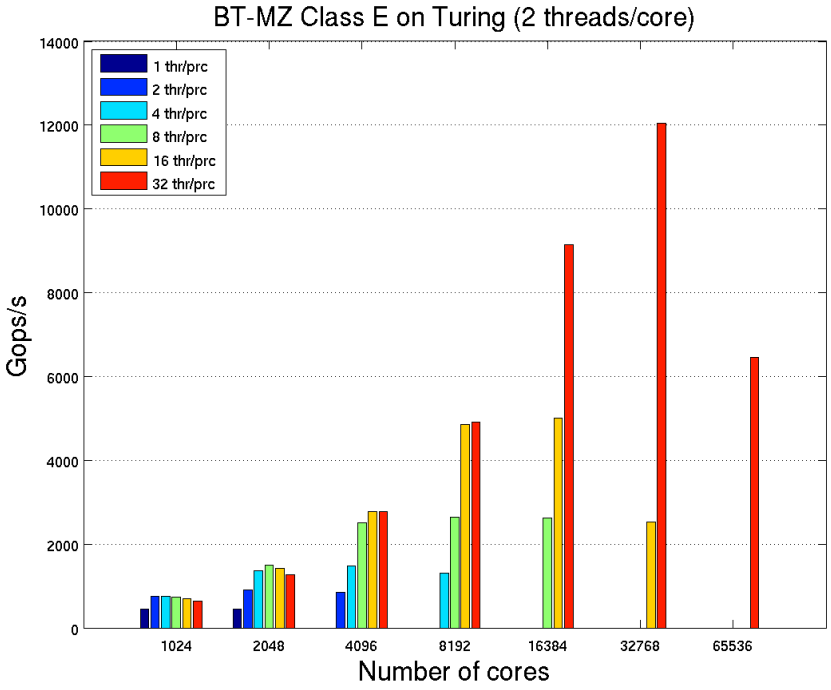
SP-MZ : méthode de résolution pentadiagonale scalaire.

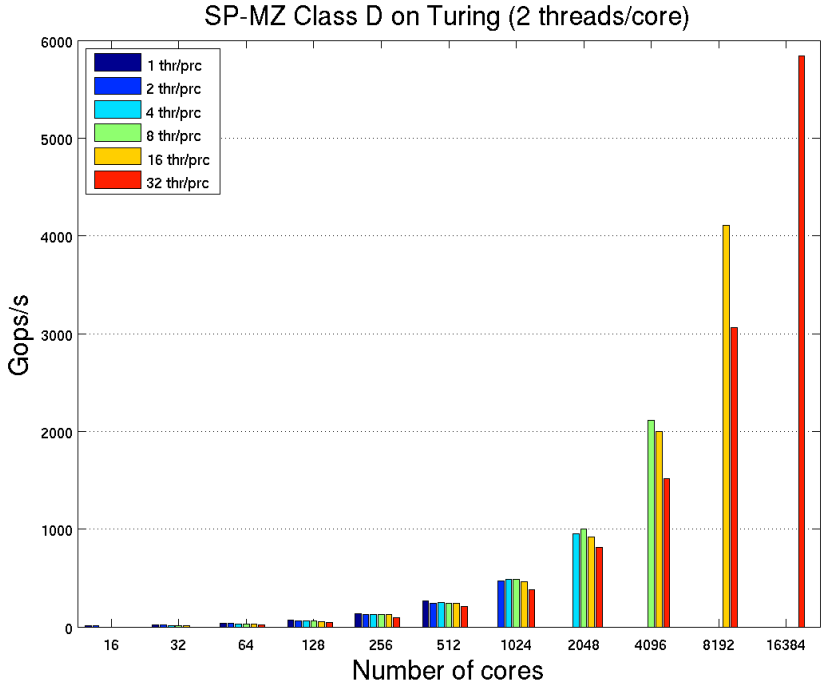
- Toutes les tailles de zones sont égales. Parfait équilibrage de charge.
- L'approche hybride ne devrait rien apporter sur ce point.

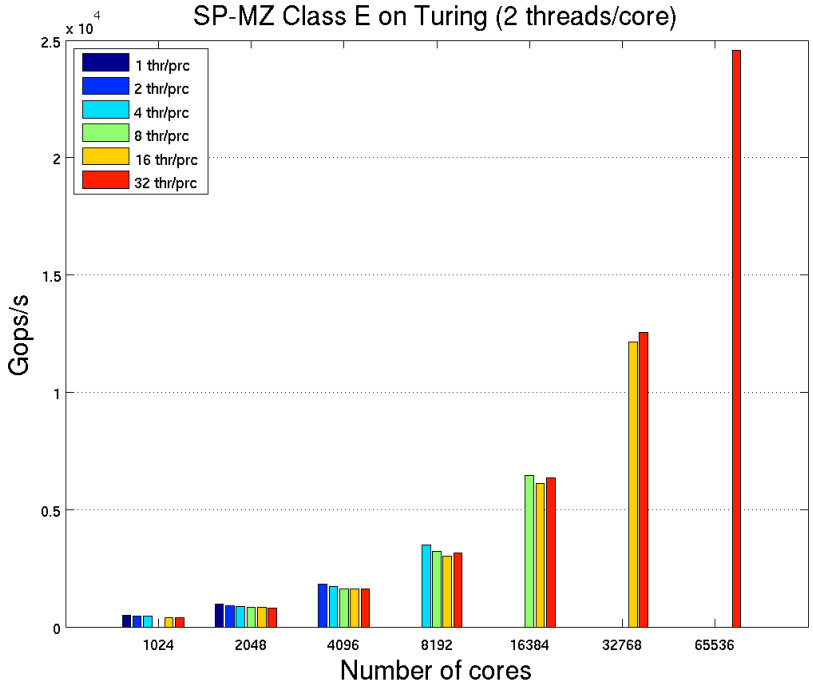
Tailles de problème sélectionnées

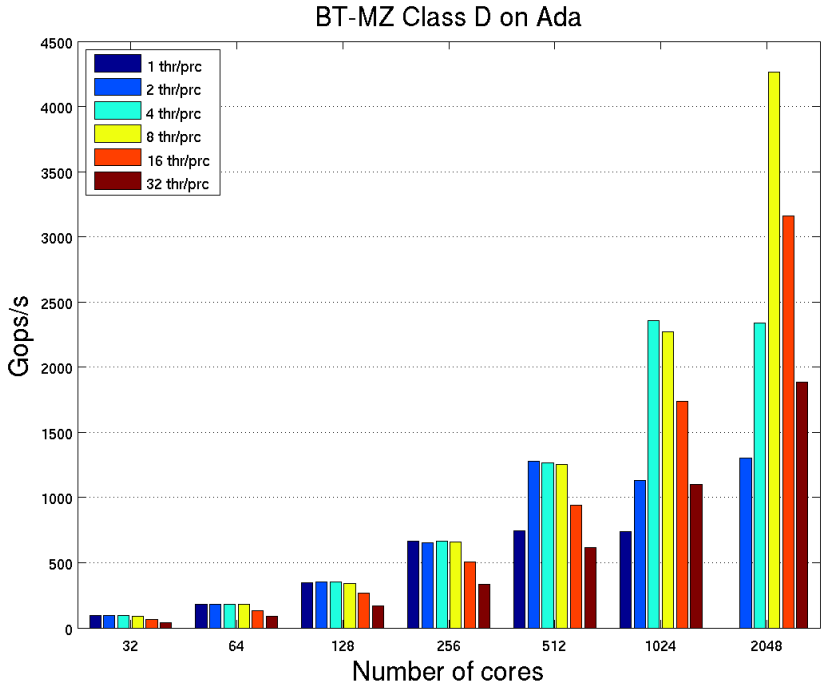
- Classe D : 1024 zones (et donc limité à 1024 processus MPI), 1632 x 1216 x 34 points de maillage (13 Gio)
- Classe E : 4096 zones (et donc limité à 4096 processus MPI), 4224 x 3456 x 92 points de maillage (250 Gio)

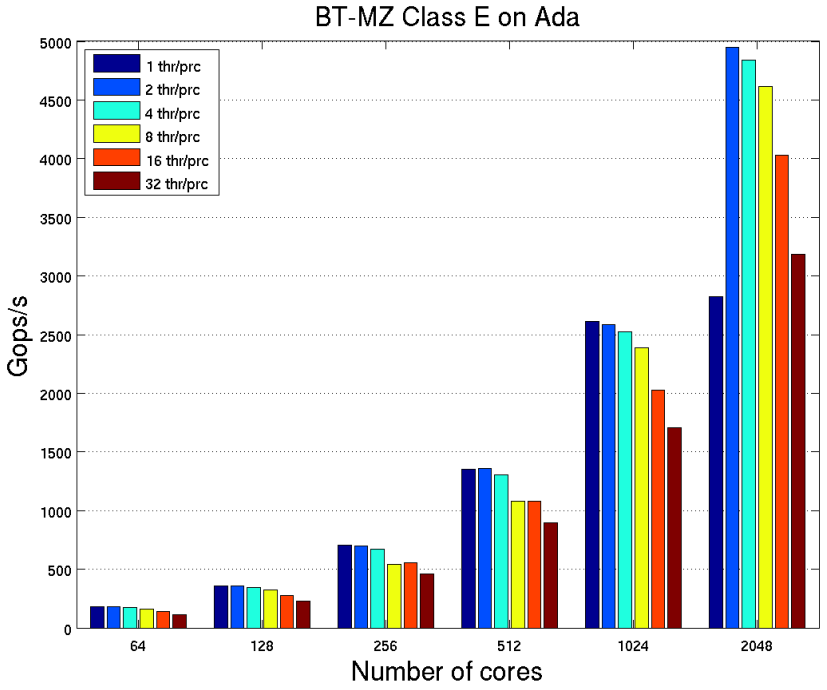

BT-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384

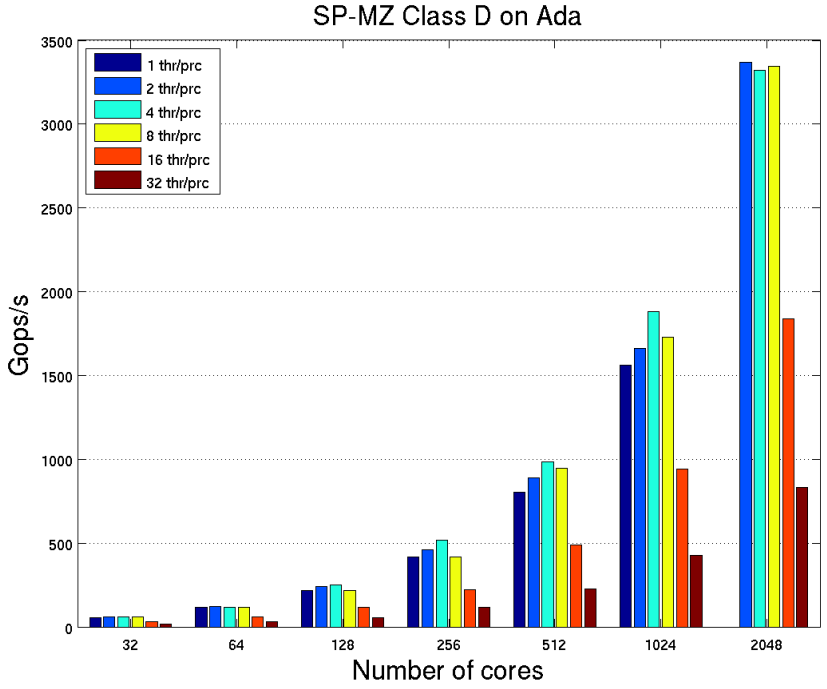

BT-MZ Class E on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
1024
2048
4096
8192
16384
32768
65536

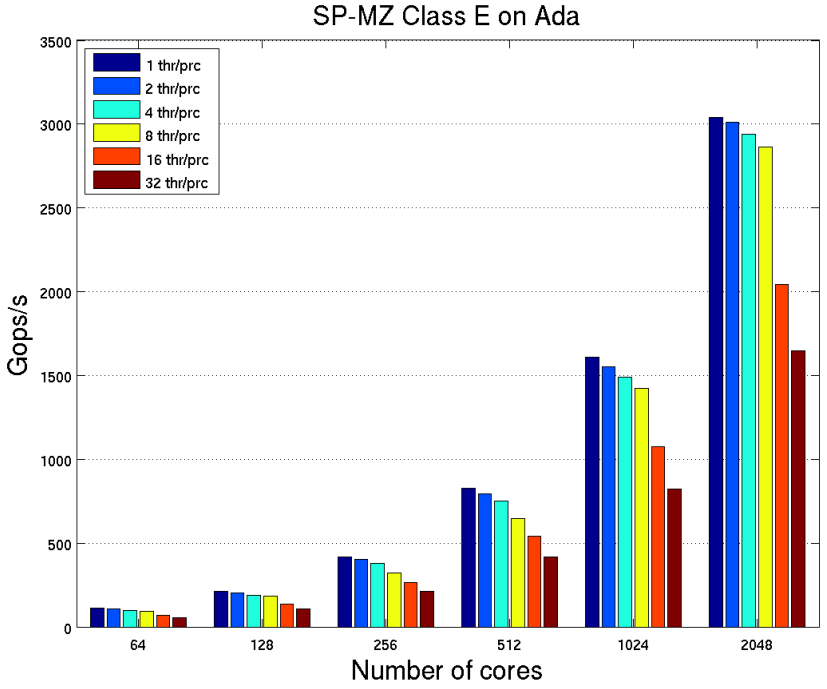

SP-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384


SP-MZ Class E on Turing (2 threads/core)
x 10^4
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
1024
2048
4096
8192
16384
32768
65536


BT-MZ Class D on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
4000
4500
32
64
128
256
512
1024
2048


BT-MZ Class E on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
64
128
256
512
1024
2048


SP-MZ Class D on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
32
64
128
256
512
1024
2048


SP-MZ Class E on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
64
128
256
512
1024
2048

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.

Code HYDRO

Présentation du code HYDRO (1)

- C'est le code utilisé pour les TPs du cours hybride.
- Code d'hydrodynamique, maillage cartésien 2D, méthode volumes finis, résolution d'un problème de Riemann aux interfaces par une méthode de Godunov.
- Depuis quelques années, dans le cadre de la veille technologique de l'IDRIS, ce code sert de *benchmark* pour les nouvelles architectures, de la simple carte graphique à la machine petaflopique.
- Il s'est enrichi au cours des années avec de nouvelles implémentations (nouveaux langages, nouveaux paradigmes de parallélisation).
- 1500 lignes de code dans sa version F90 monoprocesseur.

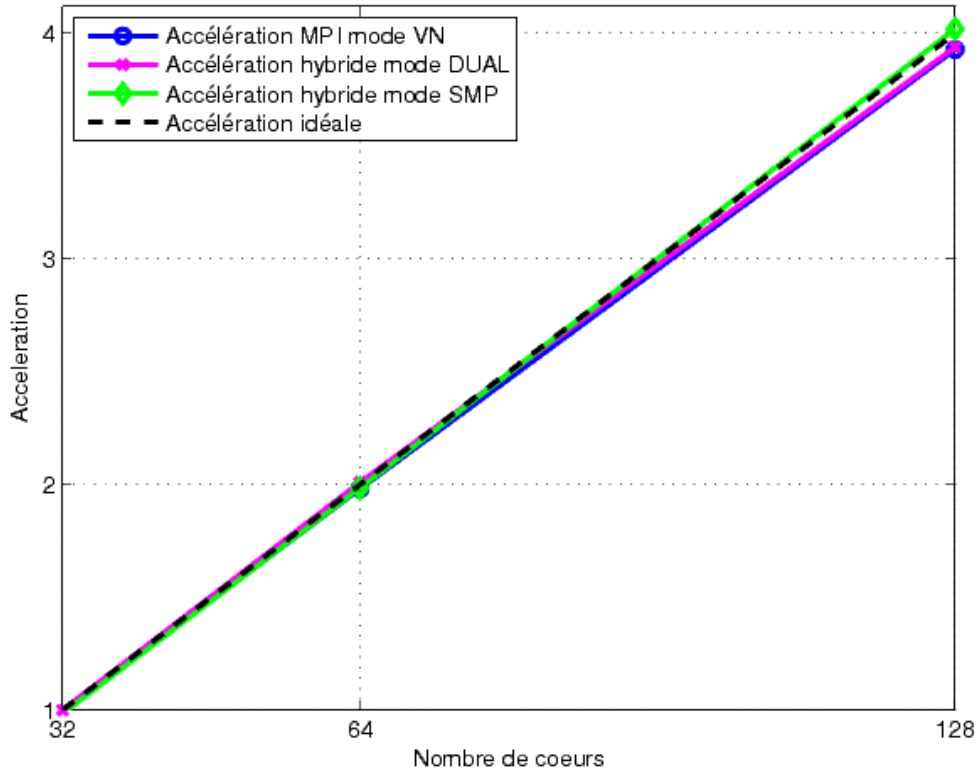
Présentation du code HYDRO (2)

- Aujourd'hui, il existe les versions d'hydro suivantes :
  - Version originale, monoprocesseur F90 (P.-Fr. Lavallée, R. Teyssier)
  - Version monoprocesseur C (G. Colin de Verdière)
  - Version parallèle MPI F90 (1D P.-Fr. Lavallée, 2D Ph. Wautelet)
  - Version parallèle MPI C (2D Ph. Wautelet)
  - Version parallèle OpenMP *Fine-grain* et *Coarse-grain* F90 (P.-Fr. Lavallée)
  - Version parallèle OpenMP *Fine-grain* C (P.-Fr. Lavallée)
  - Version parallèle hybride MPI2D-OpenMP *Coarse-grain* (P.-Fr. Lavallée, Ph. Wautelet)
  - Version C GPGPU CUDA, HMPP, OpenCL (G. Colin de Verdière)
- Plusieurs autres versions sont en cours de développement : UPC, CAF, PGI accelerator, CUDA Fortran, Pthreads

Résultats 128 cœurs Babel

Résultats pour le domaine nx = 100000, ny = 1000

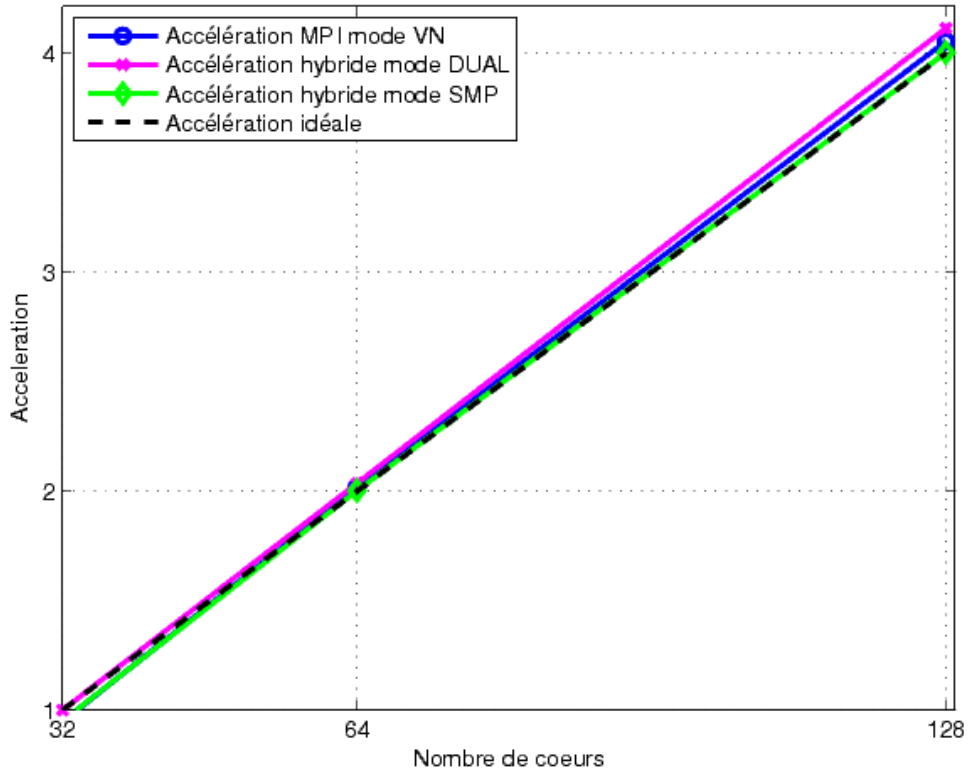
Time (s)	32 *cores*	64 *cores*	128 *cores*
Mode VN	49.12	24.74	12.47
Mode DUAL	49.00	24.39	12.44
Mode SMP	49.80	24.70	12.19



Résultats 128 cœurs Babel

Résultats pour le domaine nx = 10000, ny = 10000

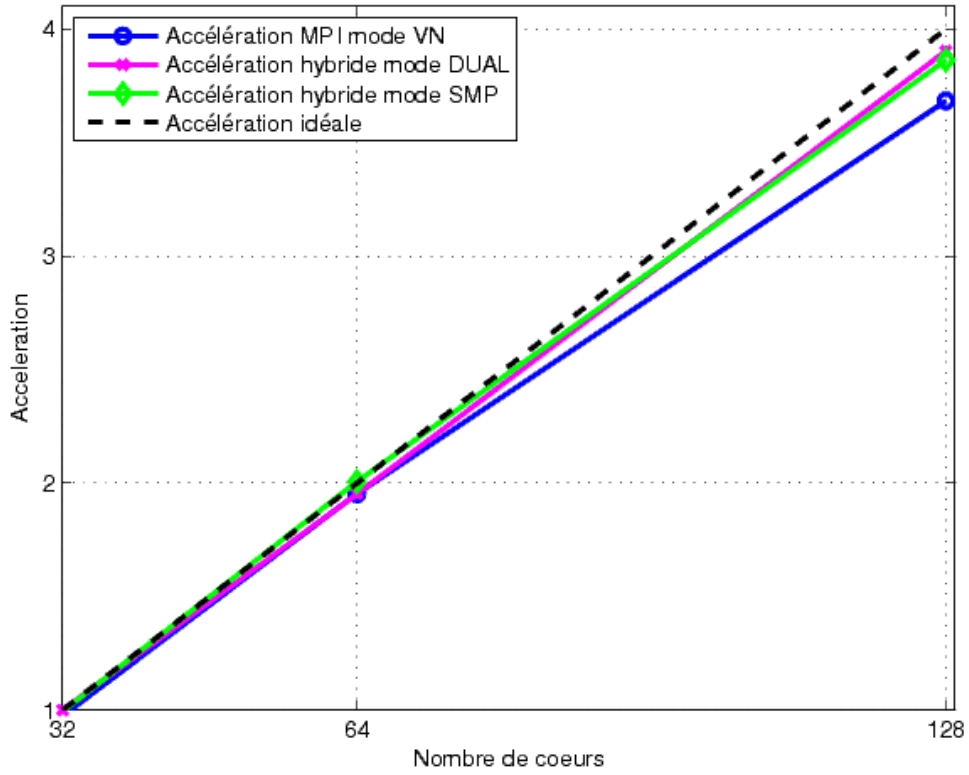
Time (s)	32 *cores*	64 *cores*	128 *cores*
Mode VN	53.14	24.94	12.40
Mode DUAL	50.28	24.70	12.22
Mode SMP	52.94	25.12	12.56



Résultats 128 cœurs Babel

Résultats pour le domaine nx = 1000, ny = 100000

Time (s)	32 *cores*	64 *cores*	128 *cores*
Mode VN	60.94	30.40	16.11
Mode DUAL	59.34	30.40	15.20
Mode SMP	59.71	29.58	15.36



Résultats 10 *racks* Babel - *Weak Scaling*

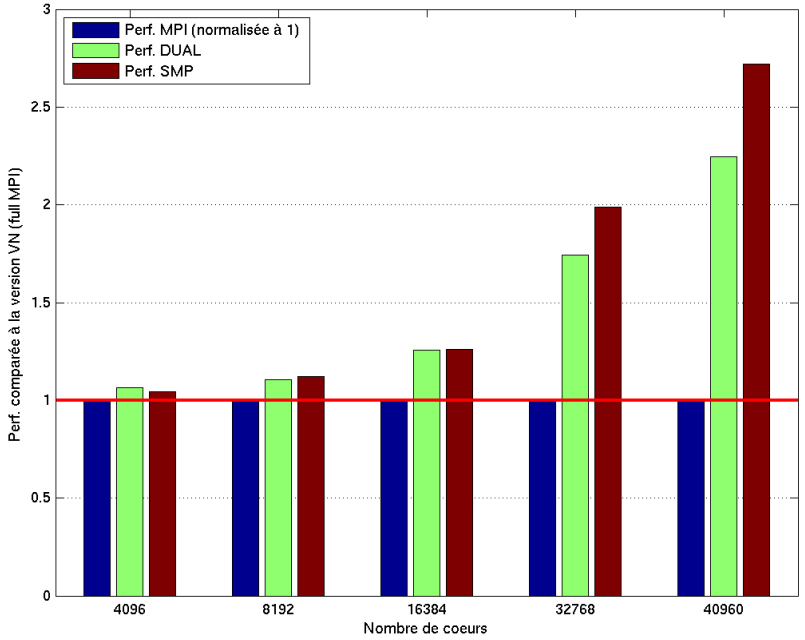
Caractéristiques des domaines utilisés pour le *Weak Scaling*

- Sur 4096 cœurs, nombre de points total du domaine : 16 10^8
  - 400000x4000 : domaine étiré suivant la première dimension
  - 40000x40000 : domaine carré
  - 4000x400000 : domaine étiré suivant la seconde dimension
- Sur 8192 cœurs, nombre de points total du domaine : 32 10^8
  - 800000x4000 : domaine étiré suivant la première dimension
  - 56568x56568 : domaine carré
  - 4000x800000 : domaine étiré suivant la seconde dimension
- Sur 16384 cœurs, nombre de points total du domaine : 64 10^8
  - 1600000x4000 : domaine étiré suivant la première dimension
  - 80000x80000 : domaine carré
  - 4000x1600000 : domaine étiré suivant la seconde dimension
- Sur 32768 cœurs, nombre de points total du domaine : 128 10^8
  - 3200000x4000 : domaine étiré suivant la première dimension
  - 113137x113137 : domaine carré
  - 4000x3200000 : domaine étiré suivant la seconde dimension
- Sur 40960 cœurs, nombre de points total du domaine : 16 10^9
  - 4000000x4000 : domaine étiré suivant la première dimension
  - 126491x126491 : domaine carré
  - 4000x4000000 : domaine étiré suivant la seconde dimension

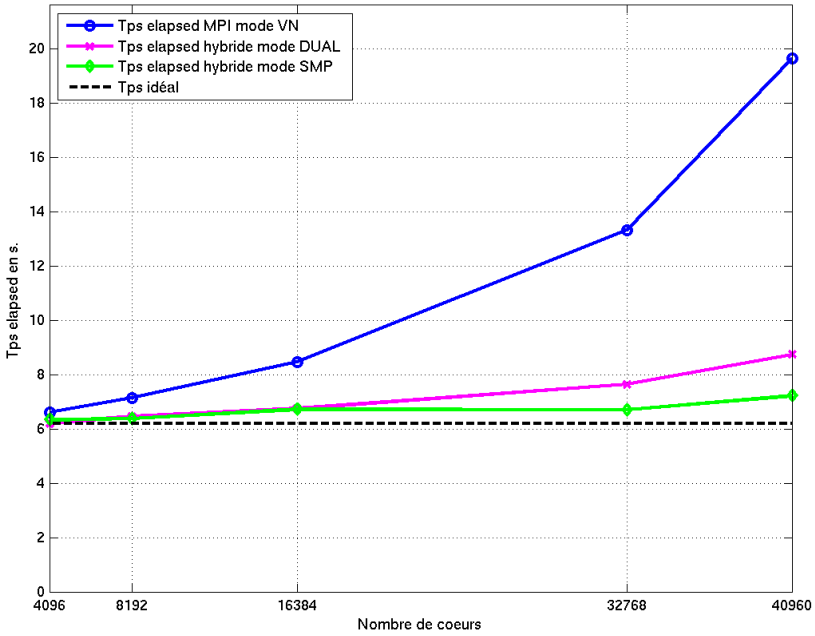
Résultats 10 *racks* Babel - *Weak Scaling*

Résultats pour le domaine étiré suivant la première dimension

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	6.62	7.15	8.47	13.89	19.64
Mode DUAL	6.21	6.46	6.75	7.85	8.75
Mode SMP	6.33	6.38	6.72	7.00	7.22



Performances comparées a la version MPI

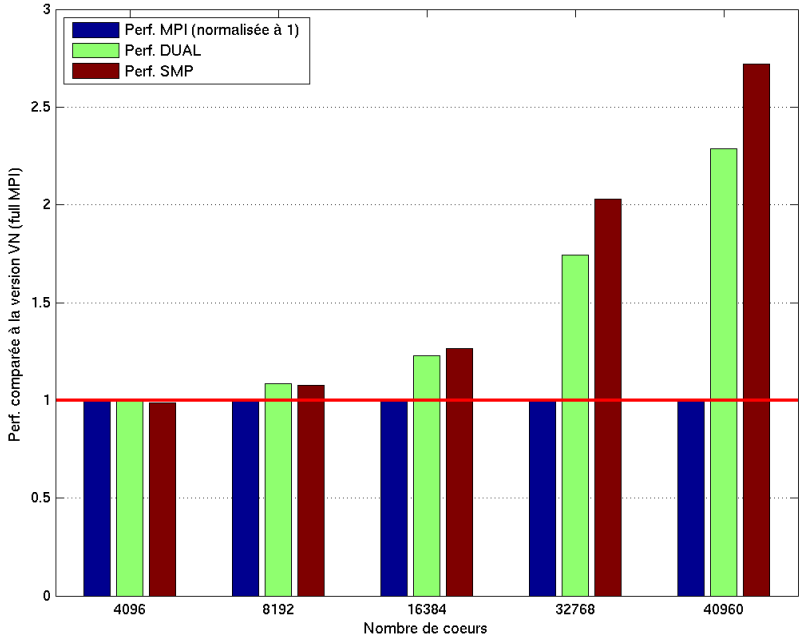


Temps *elapsed* d'exécution

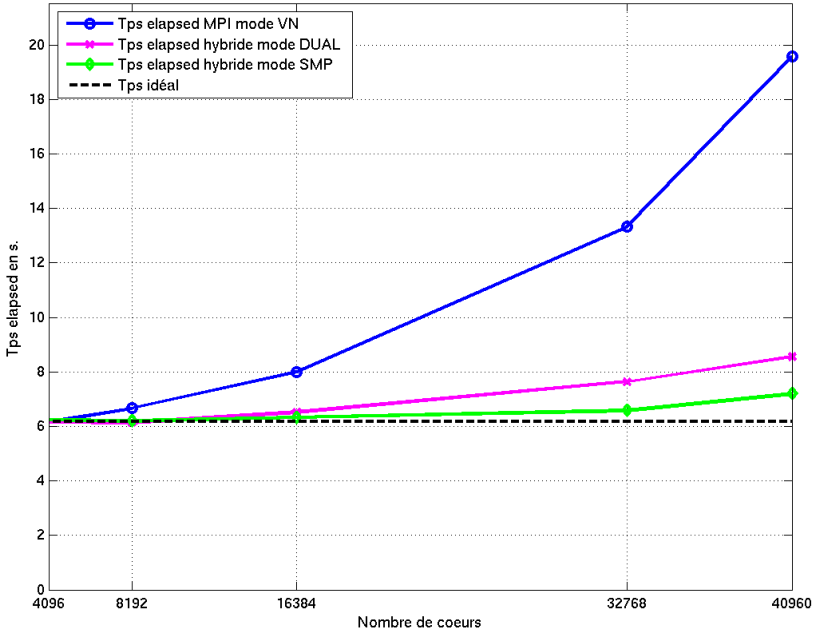
Résultats 10 *racks* Babel - *Weak Scaling*

Résultats pour le domaine carré

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	6.17	6.67	8.00	13.32	19.57
Mode DUAL	6.17	6.14	6.52	7.64	8.56
Mode SMP	6.24	6.19	6.33	6.57	7.19



Performances comparées a la version MPI

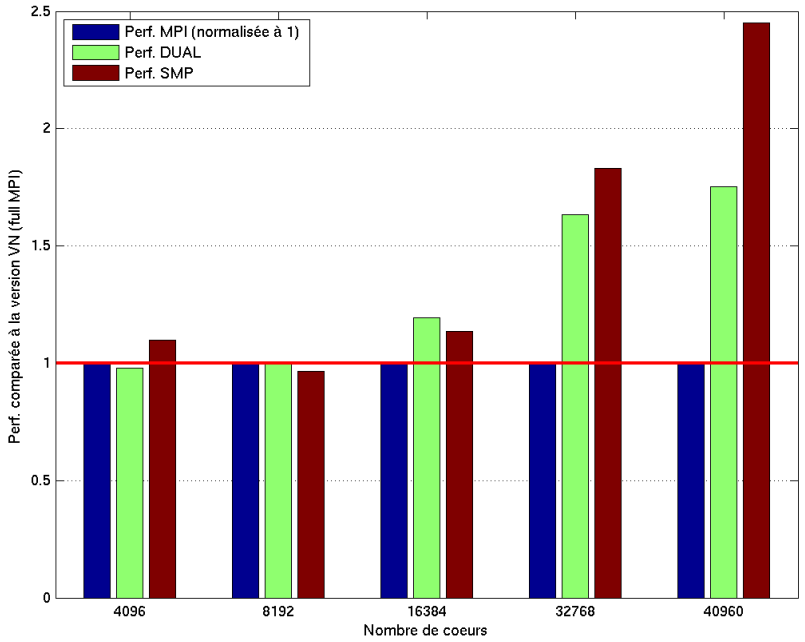


Temps *elapsed* d'exécution

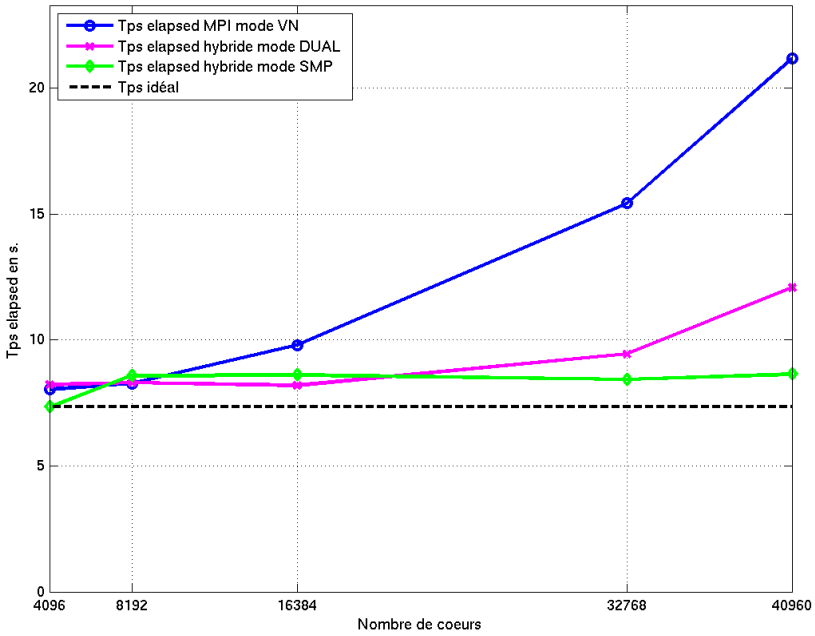
Résultats 10 *racks* Babel - *Weak Scaling*

Résultats pour le domaine étiré suivant la deuxième dimension

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	8.04	8.28	9.79	15.42	21.17
Mode DUAL	8.22	8.30	8.20	9.44	12.08
Mode SMP	7.33	8.58	8.61	8.43	8.64



Performances comparées a la version MPI



Temps *elapsed* d'exécution

Résultats sur 10 *racks* Babel - *Weak Scaling*

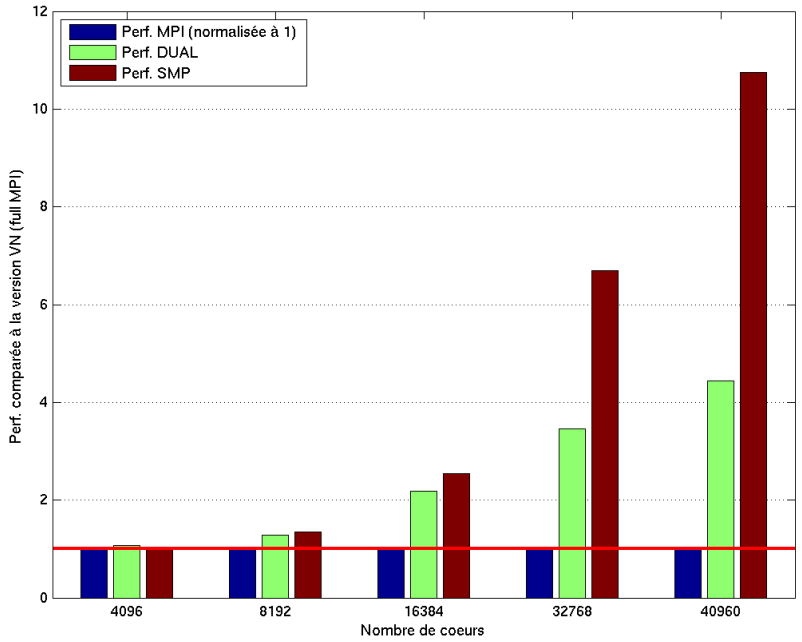
Interprétation des résultats

- Les résultats du *Weak Scaling* obtenus en utilisant jusqu'à 40960 cœurs de calcul sont très intéressants. De nouveaux phénomèmes apparaissent avec ce nombre élevé de cœurs.
- L'extensibilité de la version *flat* MPI montre ses limites. Elle peine à *scaler* jusqu'à 16384 cœurs, puis les temps de restitution explosent au-delà.
- Comme on s'y attendait, la version hybride DUAL, mais encore plus la version SMP se comportent très bien jusqu'à 32768 cœurs, avec des temps de restitution quasi-constants. Sur 40960 cœurs, la version SMP affiche un très léger surcoût, surcoût qui devient significatif pour la version DUAL.
- En *Weak Scaling*, la limite d'extensibilité de la version *flat* MPI est de 16384 cœurs, celle de la version DUAL de 32768 cœurs et celle de la version SMP n'est pas encore atteinte sur 40960 cœurs !
- Sur 40960 cœurs, la version hybride SMP est entre 2.5 et 3 fois plus rapide que la version pure MPI.
- Il est clair qu'avec ce type de méthode de parallélisation (i.e. décomposition de domaine), le passage à l'échelle (ici au-dessus de 16K cœurs) nécessite clairement le recours à la parallélisation hybride. Point de salut uniquement avec MPI !

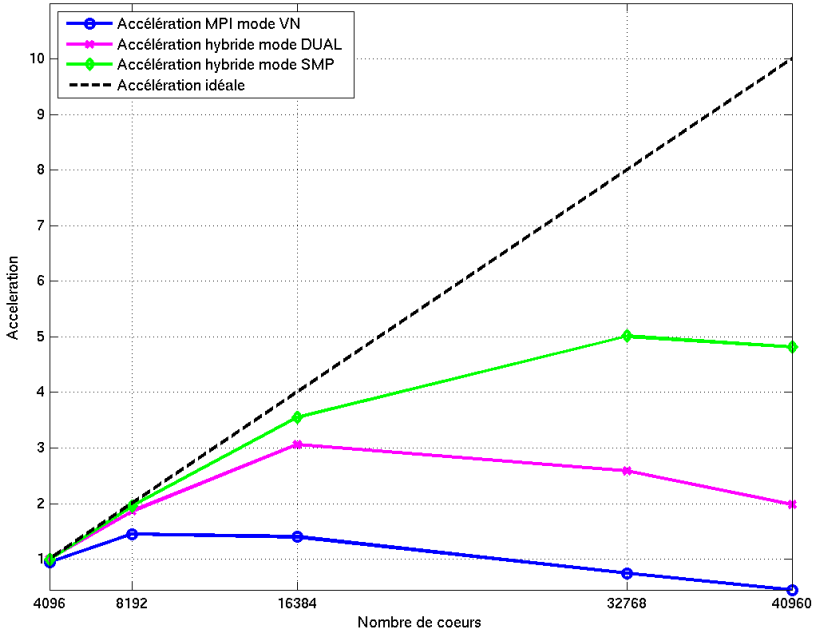
Résultats 10 *racks* Babel - *Strong Scaling*

Résultats pour le domaine nx = 400000, ny = 4000

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	6.62	4.29	4.44	8.30	13.87
Mode DUAL	6.21	3.34	2.03	2.40	3.13
Mode SMP	6.33	3.18	1.75	1.24	1.29



Performances comparées a la version MPI

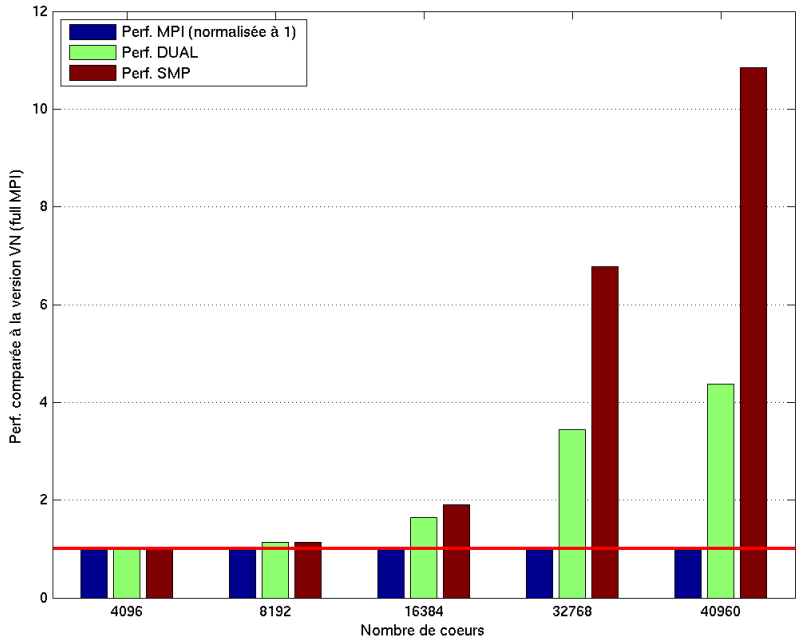


Extensibilité jusqu'à 40960 cœurs

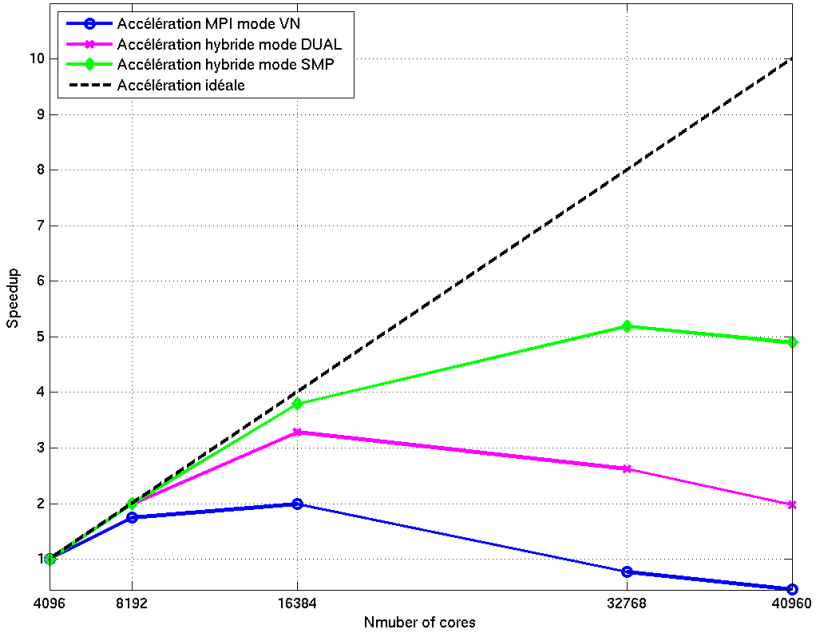
Résultats 10 *racks* Babel - *Strong Scaling*

Résultats pour le domaine nx = 40000, ny = 40000

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	6.17	3.54	3.10	8.07	13.67
Mode DUAL	6.17	3.10	1.88	2.35	3.12
Mode SMP	6.24	3.10	1.63	1.20	1.26



Performances comparées a la version MPI

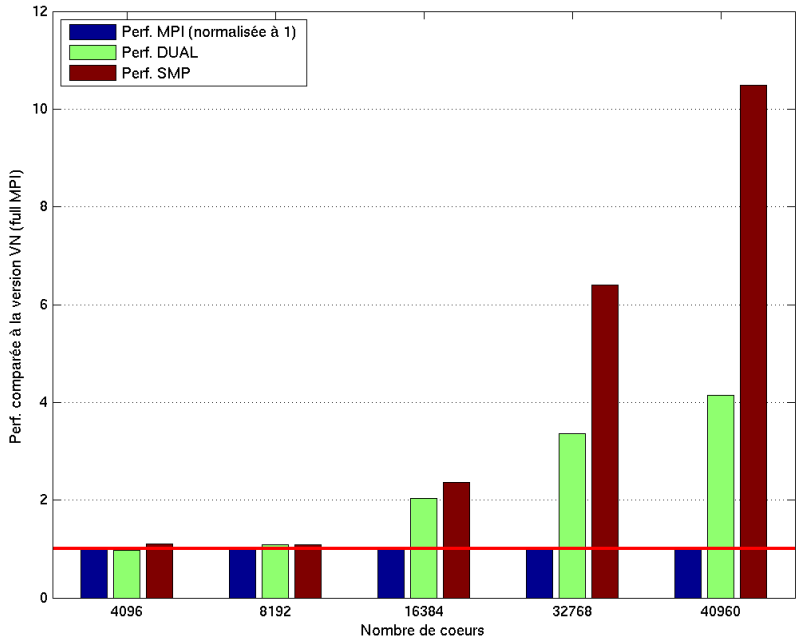


Extensibilité jusqu'à 40960 cœurs

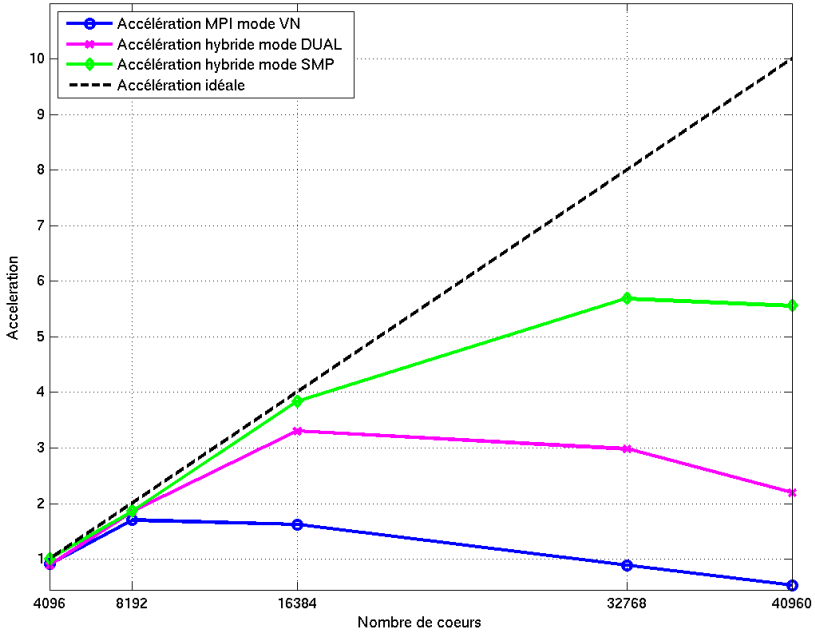
Résultats 10 *racks* Babel - *Strong Scaling*

Résultats pour le domaine nx = 4000, ny = 400000

Time (s)	4096 *cores*	8192 *cores*	16384 *cores*	32768 *cores*	40960 *c.*
Mode VN	8.04	4.31	4.52	8.26	13.85
Mode DUAL	8.22	3.96	2.22	2.46	3.34
Mode SMP	7.33	3.94	1.91	1.29	1.32



Performances comparées a la version MPI



Extensibilité jusqu'à 40960 cœurs

Résultats sur 10 *racks* Babel - *Strong Scaling*

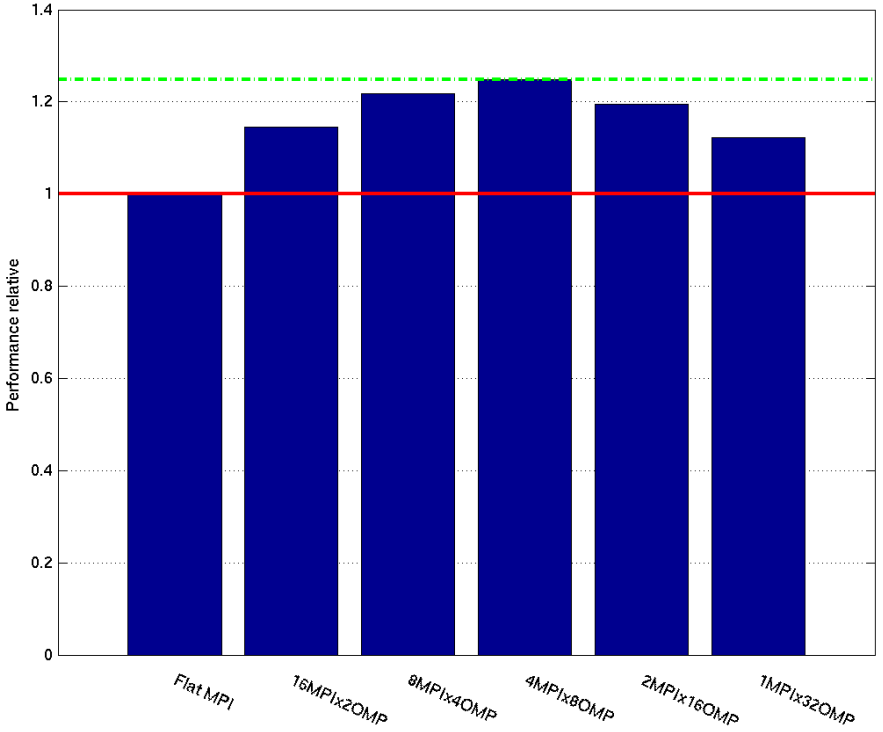
Interprétation des résultats

- Les résultats du *Strong Scaling* obtenus en utilisant jusqu'à 40960 cœurs de calcul sont très intéressants. Là encore, de nouveaux phénomèmes apparaissent avec ce nombre élevé de cœurs.
- L'extensibilité de la version *flat* MPI montre très rapidement ses limites. Elle peine à *scaler* jusqu'à 8192 cœurs, puis s'effondre au-delà.
- Comme on s'y attendait, la version hybride DUAL, mais encore plus la version SMP se comportent très bien jusqu'à 16384 cœurs, avec une accélération parfaitement linéaire. La version SMP continue de *scaler* (de façon non linéaire) jusqu'à 32768 cœurs, au-delà on n'améliore plus les performances...
- En *Strong Scaling*, la limite d'extensibilité de la version *flat* MPI est de 8192 cœurs, alors que celle de la version hybride SMP est de 32768 cœurs. On retrouve ici le facteur 4 qui correspond au nombre de cœurs d'un nœud de la BG/P !
- La meilleure version hybride (32768 cœurs) est entre 2.6 et 3.5 fois plus rapide que la meilleure version pure MPI (8192 cœurs).
- Il est clair qu'avec ce type de méthode de parallélisation (i.e. décomposition de domaine), le passage à l'échelle (ici au-dessus de 10K cœurs) nécessite clairement le recours à la parallélisation hybride. Point de salut uniquement avec MPI !

Résultats sur deux nœuds Vargas

Résultats pour le domaine nx = 100000, ny = 1000

MPI x OMP par nœud	Temps en s.	
	Mono	64 *cores*
32 x 1	361.4	7.00
16 x 2	361.4	6.11
8 x 4	361.4	5.75
4 x 8	361.4	5.61
2 x 16	361.4	5.86
1x 32	361.4	6.24

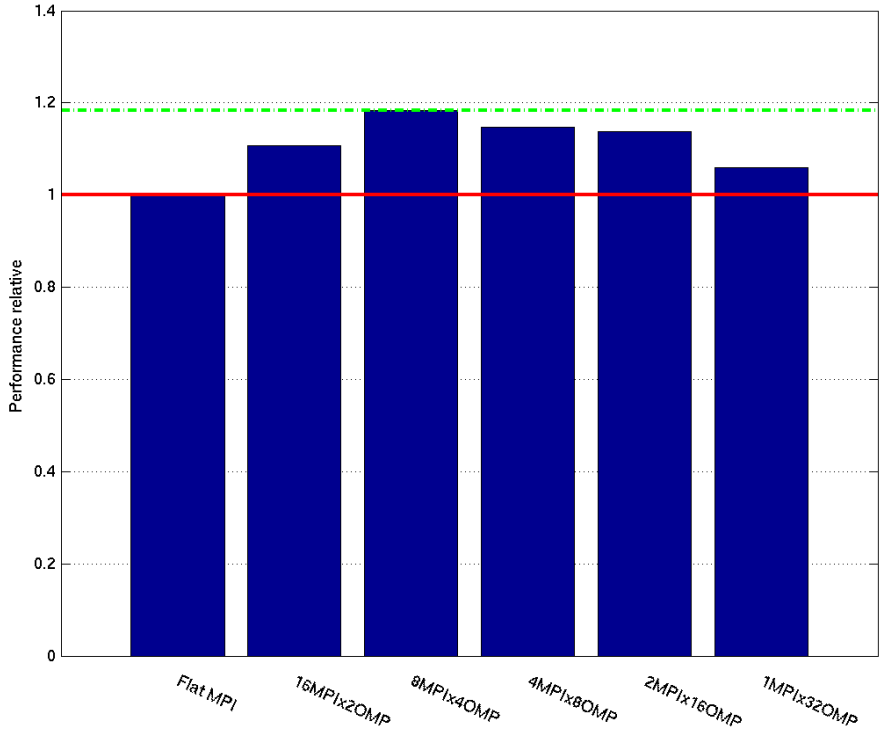


- La version hybride est toujours plus performante que la version pure MPI.
- Le gain maximum est supérieur à 20% pour les répartitions 8MPIx4OMP, 4MPIx8OMP et 2MPIx16OMP.

Résultats sur deux nœuds Vargas

Résultats pour le domaine nx = 10000, ny = 10000

MPI x OMP par nœud	Temps en s.	
	Mono	64 *cores*
32 x 1	449.9	6.68
16 x 2	449.9	6.03
8 x 4	449.9	5.64
4 x 8	449.9	5.82
2 x 16	449.9	5.87
1 x 32	449.9	6.31

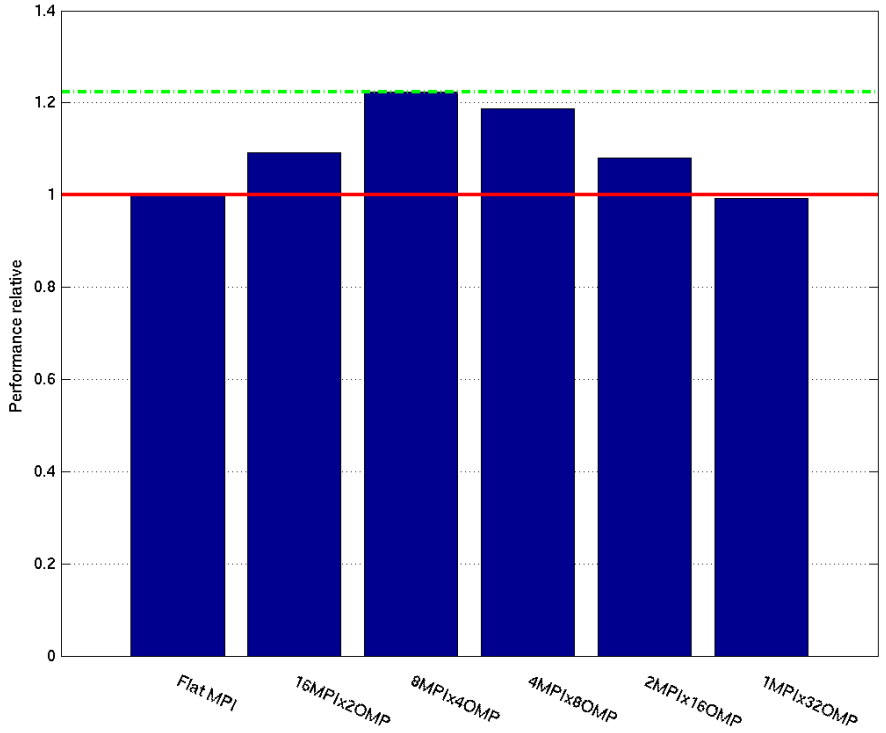


- La version hybride est toujours plus performante que la version pure MPI.
- Le gain maximum est de l'ordre 20% pour la répartition 8MPIx4OMP.

Résultats sur deux nœuds Vargas

Résultats pour le domaine nx = 1000, ny = 100000

MPI x OMP par nœud	Temps en s.	
	Mono	64 *cores*
32 x 1	1347.2	8.47
16 x 2	1347.2	7.75
8 x 4	1347.2	6.92
4 x 8	1347.2	7.13
2 x 16	1347.2	7.84
1 x 32	1347.2	8.53



- La version hybride est toujours plus performante que la version pure MPI.
- Le gain maximum est de l'ordre 20% pour la répartition 8MPIx4OMP.

Résultats sur deux nœuds Vargas

Interprétation des résultats

- Quel que soit le type de domaine, les versions *flat* MPI et un processus MPI par nœud donnent systématiquement les moins bons résultats.
- Les meilleurs résultats sont donc obtenus avec la version hybride et une répartition de 8 processus MPI par nœud et 4 *threads* OpenMP par processus MPI pour les deux derniers cas tests et de 4 processus MPI par nœud et 16 *threads* OpenMP par processus MPI pour le premier cas test.
- Nous retrouvons ici un ratio (i.e. nombre de processus MPI/nombre de *threads* OpenMP) proche de celui obtenu lors des tests de saturation du réseau d'interconnexion (début de saturation avec 8 processus MPI par nœud).
- Même avec une taille modeste en terme de nombre de cœurs utilisés, il est intéressant de noter que l'approche hybride l'emporte à chaque fois, parfois même avec des gains significatifs de performance.
- C'est très encourageant et cela incite à augmenter le nombre de cœurs utilisés.

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 en performance (à nombre fixe de cœurs d'exécution), grâce à une meilleure adaptation du code à l'architecture cible
  - Gain en terme d'extensibilité, permet 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 (multi-peta, exascale, ...)

Outils

SCALASCA

Description

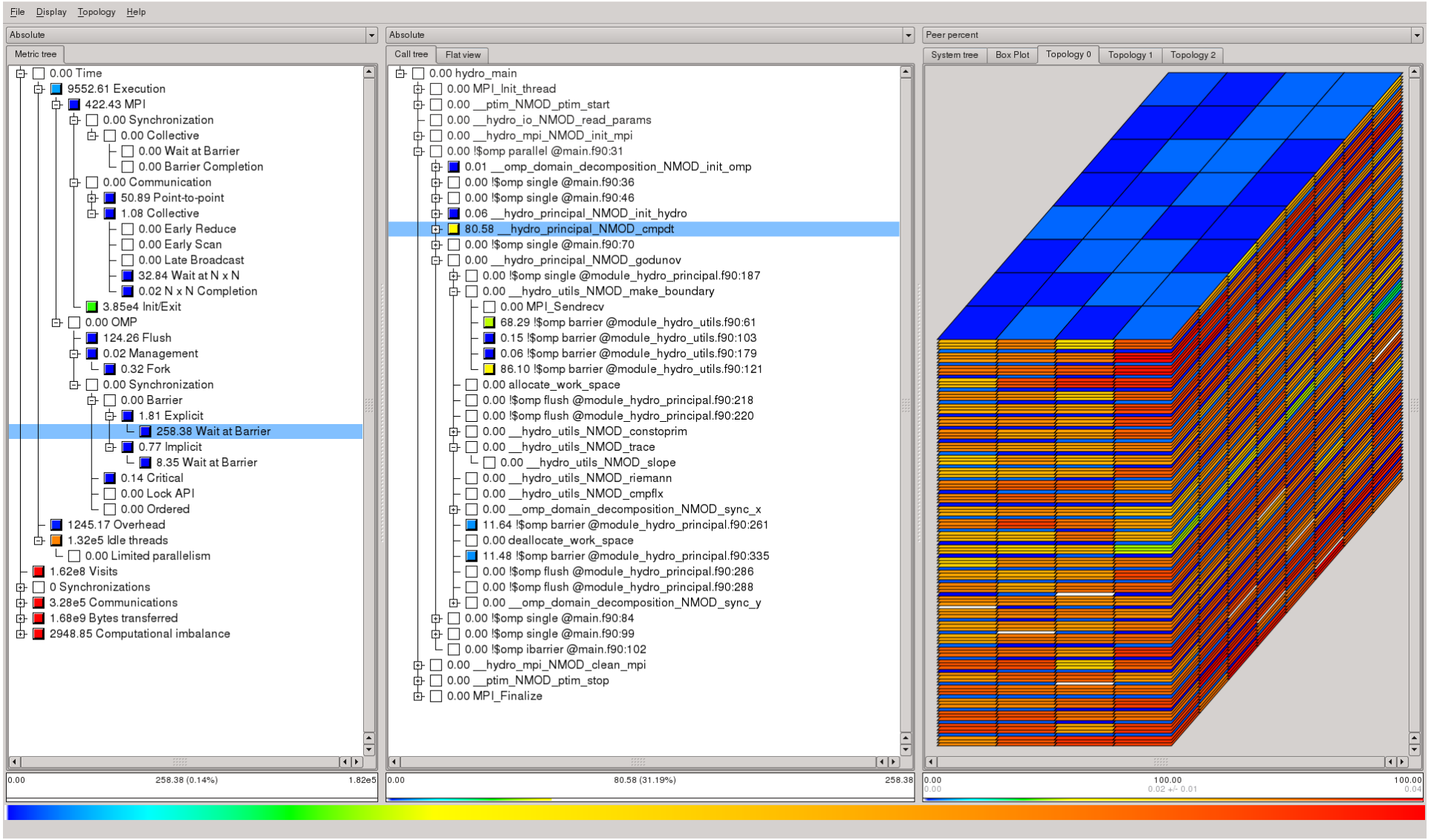
SCALASCA est un outil graphique d'analyse de performances pour applications parallèles. Principales caractéristiques :

- Support de MPI et des applications multithreadées/OpenMP
- Modes profilage ou prise de trace (limité à `MPI_THREAD_FUNNELED` pour les traces)
- Identification/analyse automatique des problèmes courants de performance (en mode trace)
- Nombre de processus illimité
- Support des compteurs hardware (via PAPI)

Utilisation

- Compiler votre application avec *skin f90* (ou autre compilateur)
- Exécuter avec *scan mpirun*. Option *-t* pour le mode trace.
- Visualiser les résultats avec *square*

SCALASCA


File Display Topology Help
Absolute
Metric tree
0.00 Time
9552.61 Execution
422.43 MPI
0.00 Synchronization
0.00 Collective
0.00 Wait at Barrier
0.00 Barrier Completion
0.00 Communication
50.89 Point-to-point
1.08 Collective
0.00 Early Reduce
0.00 Early Scan
0.00 Late Broadcast
32.84 Wait at N x N
0.02 N x N Completion
3.85e4 Init/Exit
0.00 OMP
124.26 Flush
0.02 Management
0.32 Fork
0.00 Synchronization
0.00 Barrier
1.81 Explicit
258.38 Wait at Barrier
0.77 Implicit
8.35 Wait at Barrier
0.14 Critical
0.00 Lock API
0.00 Ordered
1245.17 Overhead
1.32e5 Idle threads
0.00 Limited parallelism
1.62e8 Visits
0 Synchronizations
3.28e5 Communications
1.68e9 Bytes transferred
2948.85 Computational imbalance
0.00
258.38 (0.14%)
1.82e5
Absolute
Call tree
Flat view
0.00 hydro_main
0.00 MPI_Init_thread
0.00 __ptim_NMOD_ptim_start
0.00 __hydro_io_NMOD_read_params
0.00 __hydro_mpi_NMOD_init_mpi
0.00 !$omp parallel @main.f90:31
0.01 __omp_domain_decomposition_NMOD_init_omp
0.00 !$omp single @main.f90:36
0.00 !$omp single @main.f90:46
0.06 __hydro_principal_NMOD_init_hydro
80.58 __hydro_principal_NMOD_cmpdt
0.00 !$omp single @main.f90:70
0.00 __hydro_principal_NMOD_godunov
0.00 !$omp single @module_hydro_principal.f90:187
0.00 __hydro_utils_NMOD_make_boundary
0.00 MPI_Sendrecv
68.29 !$omp barrier @module_hydro_utils.f90:61
0.15 !$omp barrier @module_hydro_utils.f90:103
0.06 !$omp barrier @module_hydro_utils.f90:179
86.10 !$omp barrier @module_hydro_utils.f90:121
0.00 allocate_work_space
0.00 !$omp flush @module_hydro_principal.f90:218
0.00 !$omp flush @module_hydro_principal.f90:220
0.00 __hydro_utils_NMOD_constoprim
0.00 __hydro_utils_NMOD_trace
0.00 __hydro_utils_NMOD_slope
0.00 __hydro_utils_NMOD_riemann
0.00 __hydro_utils_NMOD_cmpflx
0.00 __omp_domain_decomposition_NMOD_sync_x
11.64 !$omp barrier @module_hydro_principal.f90:261
0.00 deallocate_work_space
11.48 !$omp barrier @module_hydro_principal.f90:335
0.00 !$omp flush @module_hydro_principal.f90:286
0.00 !$omp flush @module_hydro_principal.f90:288
0.00 __omp_domain_decomposition_NMOD_sync_y
0.00 !$omp single @main.f90:84
0.00 !$omp single @main.f90:99
0.00 !$omp ibarrier @main.f90:102
0.00 __hydro_mpi_NMOD_clean_mpi
0.00 __ptim_NMOD_ptim_stop
0.00 MPI_Finalize
0.00
80.58 (31.19%)
258.38
Peer percent
System tree
Box Plot
Topology 0
Topology 1
Topology 2
0.00
100.00
0.02 +/- 0.01
100.00
0.04

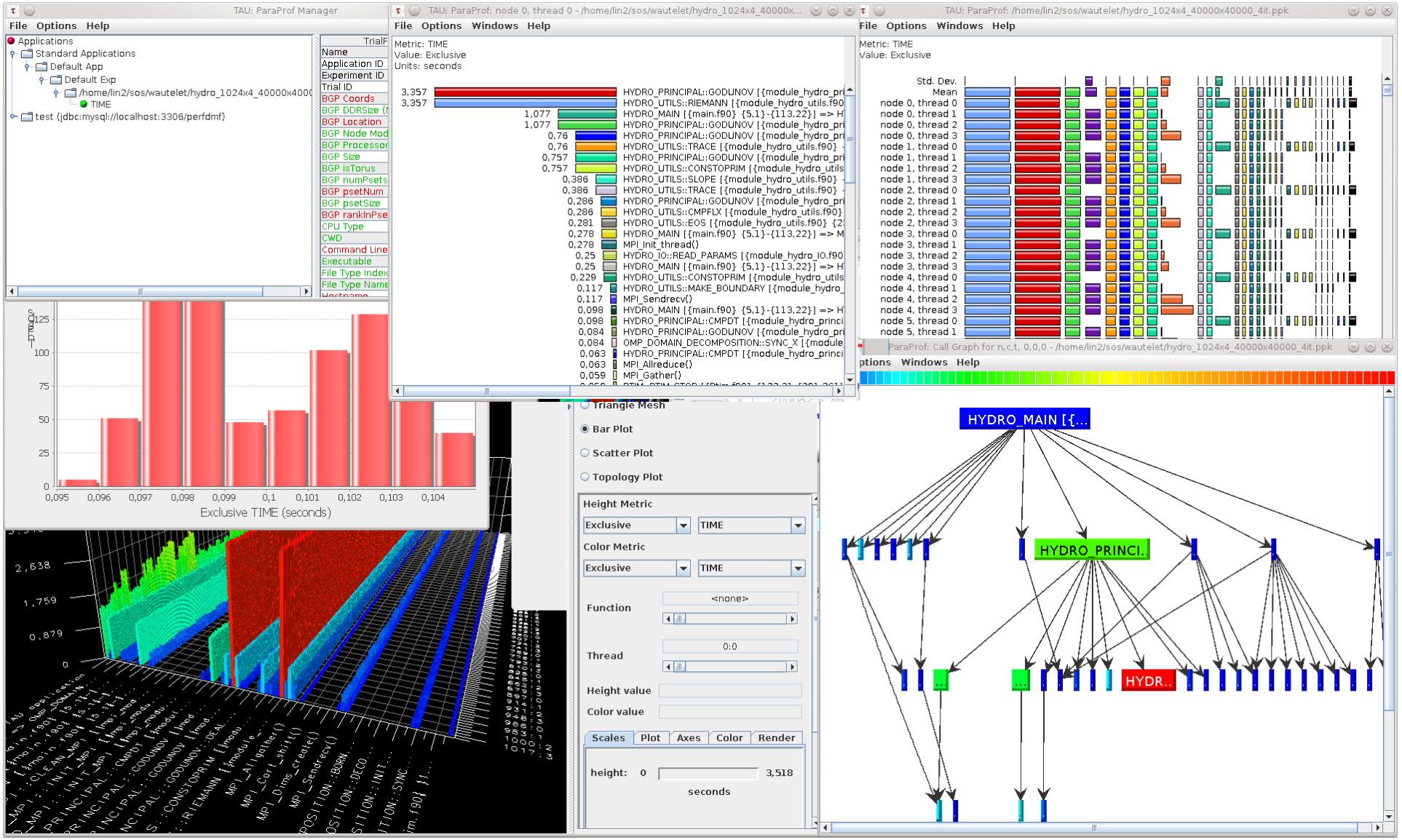
TAU

Description

TAU est un outil graphique d'analyse de performances pour applications parallèles.
Principales caractéristiques :

- Support de MPI et des applications multithreadées/OpenMP
- Mode profilage ou prise de trace
- Nombre de processus illimité
- Support des compteurs *hardware* (via PAPI)
- Instrumentation automatique des boucles
- Suivi des allocations mémoire
- Suivi des I/O
- Arbre d'appels
- Visualisation 3D (utile pour comparer les processus/*threads* entre-eux)

TAU


TAU: ParaProf Manager
File Options Help
Applications
Standard Applications
Default App
Default Exp
/home/lin2/sos/wautelet/hydro_1024x4_40000x4000
TIME
test (jdbc:mysql://localhost:3306/perfdmf)
TAU: ParaProf: node 0, thread 0 - /home/lin2/sos/wautelet/hydro_1024x4_40000x...
File Options Windows Help
Metric: TIME
Value: Exclusive
Units: seconds
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
HYDRO_UTILS::RIEMANN [{module_hydro_utils.f90
HYDRO_MAIN [{main.f90} {5,1}-{113,22}] => H
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
HYDRO_UTILS::TRACE [{module_hydro_utils.f90}
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
HYDRO_UTILS::CONSTOPRIM [{module_hydro_utils
HYDRO_UTILS::SLOPE [{module_hydro_utils.f90}
HYDRO_UTILS::TRACE [{module_hydro_utils.f90}
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
HYDRO_UTILS::CMPFLX [{module_hydro_utils.f90}
HYDRO_UTILS::EOS [{module_hydro_utils.f90}
HYDRO_MAIN [{main.f90} {5,1}-{113,22}] => M
MPI_Init_thread()
HYDRO_IO::READ_PARAMS [{module_hydro_IO.f90
HYDRO_MAIN [{main.f90} {5,1}-{113,22}] => H
HYDRO_UTILS::CONSTOPRIM [{module_hydro_utils
HYDRO_UTILS::MAKE_BOUNDARY [{module_hydro_
MPI_Sendrecv()
HYDRO_MAIN [{main.f90} {5,1}-{113,22}] => H
HYDRO_PRINCIPAL::CMPDT [{module_hydro_princi
HYDRO_PRINCIPAL::GODUNOV [{module_hydro_pri
OMP_DOMAIN_DECOMPOSITION::SYNC_X [{module
HYDRO_PRINCIPAL::CMPDT [{module_hydro_princi
MPI_Allreduce()
MPI_Gather()
TAU: ParaProf: /home/lin2/sos/wautelet/hydro_1024x4_40000x40000_4it.ppk
File Options Windows Help
Metric: TIME
Value: Exclusive
Std. Dev.
Mean
node 0, thread 0
node 0, thread 1
node 0, thread 2
node 0, thread 3
node 1, thread 0
node 1, thread 1
node 1, thread 2
node 1, thread 3
node 2, thread 0
node 2, thread 1
node 2, thread 2
node 2, thread 3
node 3, thread 0
node 3, thread 1
node 3, thread 2
node 3, thread 3
node 4, thread 0
node 4, thread 1
node 4, thread 2
node 4, thread 3
node 5, thread 0
node 5, thread 1
ParaProf: Call Graph for n,c,t, 0,0,0 - /home/lin2/sos/wautelet/hydro_1024x4_40000x40000_4it.ppk
ptions Windows Help
HYDRO_MAIN [{...
HYDRO_PRINCI..
HYDR..
Exclusive TIME (seconds)
Triangle Mesh
Bar Plot
Scatter Plot
Topology Plot
Height Metric
Exclusive
TIME
Color Metric
Exclusive
TIME
Function
<none>
Thread
0:0
Height value
Color value
Scales
Plot
Axes
Color
Render
height: 0
3,518
seconds

TotalView

Description

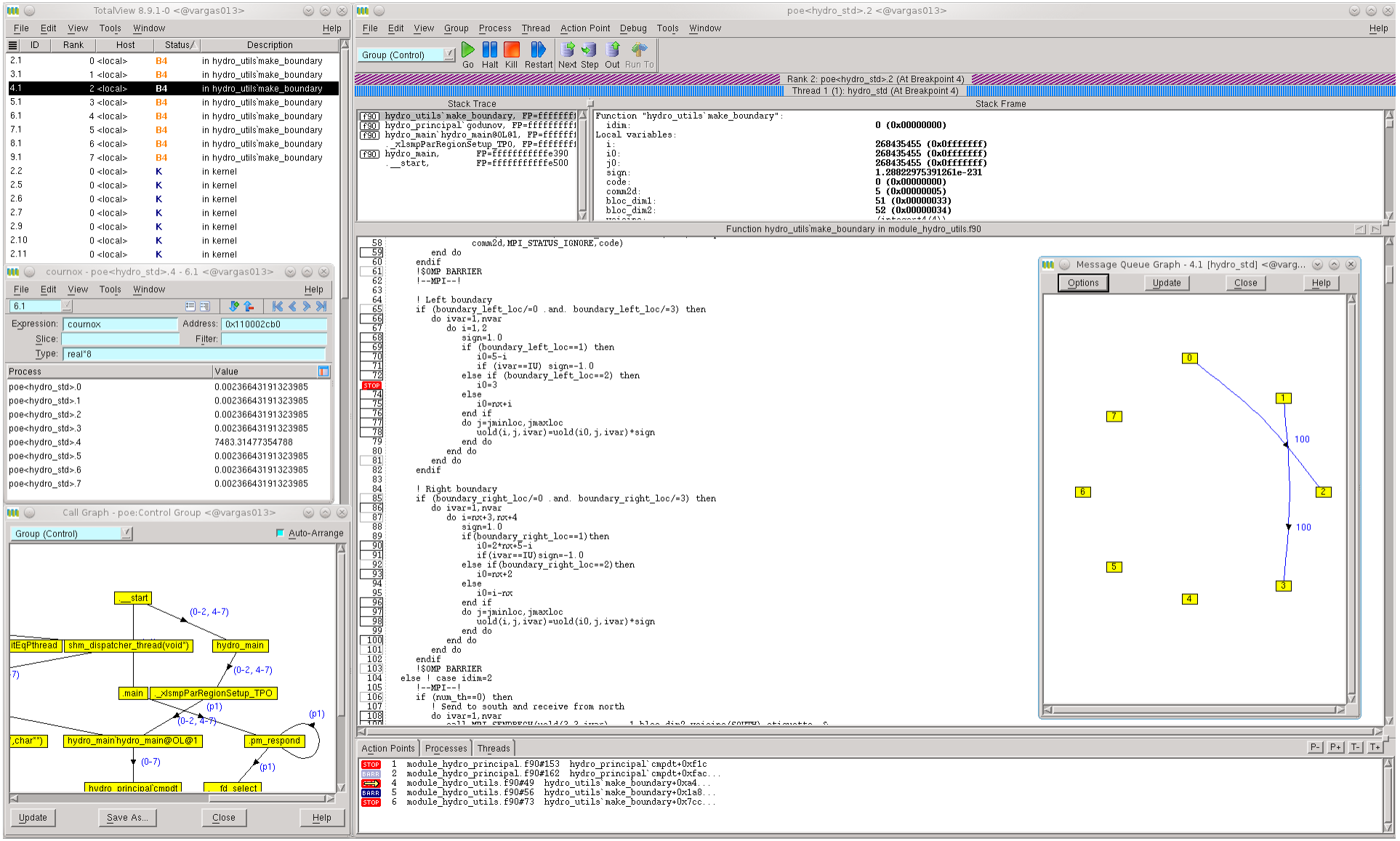
TotalView est un outil de débogage graphique d'applications parallèles. Principales caractéristiques :

- Support de MPI et des applications multithreadées/OpenMP
- Support des langages C/C++ et Fortran95
- Débogueur mémoire intégré
- Nombre de processus maximum selon licence

Utilisation

- Compiler votre application avec *-g* et un niveau d'optimisation pas trop élevé

TotalView


TotalView 8.9.1-0 <@vargas013>
File Edit View Tools Window Help
ID Rank Host Status Description
poe<hydro_std>.2 <@vargas013>
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To
Rank 2: poe<hydro_std>.2 (At Breakpoint 4)
Thread 1 (1): hydro_std (At Breakpoint 4)
Stack Trace
Stack Frame
Function "hydro_utils`make_boundary":
idim: 0 (0x00000000)
Local variables:
i: 268435455 (0x0fffffff)
i0: 268435455 (0x0fffffff)
j0: 268435455 (0x0fffffff)
sign: 1.2882297539126le-231
code: 0 (0x00000000)
comm2d: 5 (0x00000005)
bloc_dim1: 51 (0x00000033)
bloc_dim2: 52 (0x00000034)
Function hydro_utils`make_boundary in module_hydro_utils.f90
cournox - poe<hydro_std>.4 - 6.1 <@vargas013>
Expression: cournox Address: 0x110002cb0
Slice: Filter:
Type: real*8
Process Value
Call Graph - poe:Control Group <@vargas013>
Group (Control) Auto-Arrange
Update Save As... Close Help
Message Queue Graph - 4.1 [hydro_std] <@varg...
Options Update Close Help
Action Points Processes Threads

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.f90* 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 — 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.

Annexes

Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_FUNNELED` — Résultats sur Vargas

4 liens en // Infiniband DDR, débit crête 8 Go/s.

MPI x OMP par nœud	Débit cumulé en Mo/s Message de 1 Mo	Débit cumulé en Mo/s Message de 10 Mo	Débit cumulé en Mo/s Message de 100 Mo
1 x 32	1016	1035	959
2 x 16	2043	2084	1803
4 x 8	3895	3956	3553
8 x 4	6429	6557	5991
16 x 2	7287	7345	7287
32 x 1	7412	7089	4815

Interprétations

- Avec 1 seul flux, on n'utilise qu'un huitième de la bande passante réseau inter-nœud.
- La saturation des liens réseaux inter-nœud de Vargas commence à apparaître clairement à partir de 8 flux en parallèle (i.e. 8 processus MPI par nœud).
- La saturation est totale avec 16 flux en parallèle (i.e. 16 processus MPI par nœud).
- Avec 16 flux en parallèle, on obtient un débit de 7,35 Go/s, soit plus de 90% de la bande passante réseau crête inter-nœud disponible !

Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_FUNNELED` — Résultats sur Babel

Débit crête 425 Mo/s

MPI x OMP par nœud	Débit cumulé en Mo/s Message de 1 Mo	Débit cumulé en Mo/s Message de 10 Mo	Débit cumulé en Mo/s Message de 100 Mo
SMP (1 x 4)	373.5	374.8	375.0
DUAL (2 x 2)	374.1	374.9	375.0
VN (4 x 1)	374.7	375.0	375.0

Interprétations

- L'utilisation d'un seul flux de données (i.e. 1 seul processus MPI par nœud) suffit à saturer totalement le réseau d'interconnexion entre deux nœuds voisins.
- Le débit atteint est de 375 Mo/s, soit 88% de la bande passante crête réseau inter-nœud.

Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_MULTIPLE` — Résultats sur Vargas

4 liens en // Infiniband DDR, débit crête 8 Go/s.

MPI x OMP par nœud	Débit cumulé en Mo/s Message de 1 Mo	Débit cumulé en Mo/s Message de 10 Mo	Débit cumulé en Mo/s Message de 100 Mo
1 x 32 (1 flux)	548.1	968.1	967.4
1 x 32 (2 flux //)	818.6	1125	1016
1 x 32 (4 flux //)	938.6	1114	1031
1 x 32 (8 flux //)	964.4	1149	1103
1 x 32 (16 flux //)	745.1	1040	1004
1 x 32 (32 flux //)	362.2	825.1	919.9

Interprétations

- Les performances des versions `MPI_THREAD_MULTIPLE` et `MPI_THREAD_FUNNELED` sont très différentes, le débit n'augmente plus avec le nombre de flux en parallèle et reste constant.
- Avec 1 seul flux comme avec plusieurs, on n'utilise toujours qu'un huitième de la bande passante réseau inter-nœud. Elle n'est de ce fait jamais saturée !
- Cette approche `MPI_THREAD_MULTIPLE` (i.e. plusieurs *threads* communiquant simultanément au sein d'un même processus MPI) n'est donc absolument pas adaptée à la machine Vargas, mieux vaut lui préférer l'approche `MPI_THREAD_FUNNELED`.

Utilisation optimale du réseau d'interconnexion

SBPR `MPI_THREAD_MULTIPLE` — Résultats sur Babel

Débit crête 425 Mo/s

MPI x OMP par nœud	Débit cumulé en Mo/s Message de 1 Mo	Débit cumulé en Mo/s Message de 10 Mo	Débit cumulé en Mo/s Message de 100 Mo
SMP (1 flux)	372.9	374.7	375.0
SMP (2 flux //)	373.7	374.8	375.0
SMP (4 flux //)	374.3	374.9	375.0

Interprétations

- Les performances des versions `MPI_THREAD_MULTIPLE` et `MPI_THREAD_FUNNELED` sont comparables.
- L'utilisation d'un seul flux de données (i.e. 1 seul *thread* de communication par nœud) suffit à saturer totalement le réseau d'interconnexion entre deux nœuds voisins.
- Le débit atteint est de 375 Mo/s, soit 88% de la bande passante crête réseau inter-nœud.

Etude de cas : Poisson3D

Présentation de Poisson3D

Poisson3D est une application qui résout l'équation de Poisson sur le domaine cubique [0,1]x[0,1]x[0,1] par une méthode aux différences finies avec un solveur Jacobi.

$$\begin{cases} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} & = f(x,y,z) \quad \text{dans } [0,1]\text{x}[0,1]\text{x}[0,1] \\ u(x,y,z) & = 0. \quad \text{sur les frontières} \\ f(x,y,z) & = 2yz(y-1)(z-1) + 2xz(x-1)(z-1) + 2xy(x-1)(y-1) \\ u_{\text{exacte}}(x,y) & = xyz(x-1)(y-1)(z-1) \end{cases}$$

Solveur

La discrétisation se fait sur un maillage régulier dans les trois directions spatiales (pas $h = h_x = h_y = h_z$).

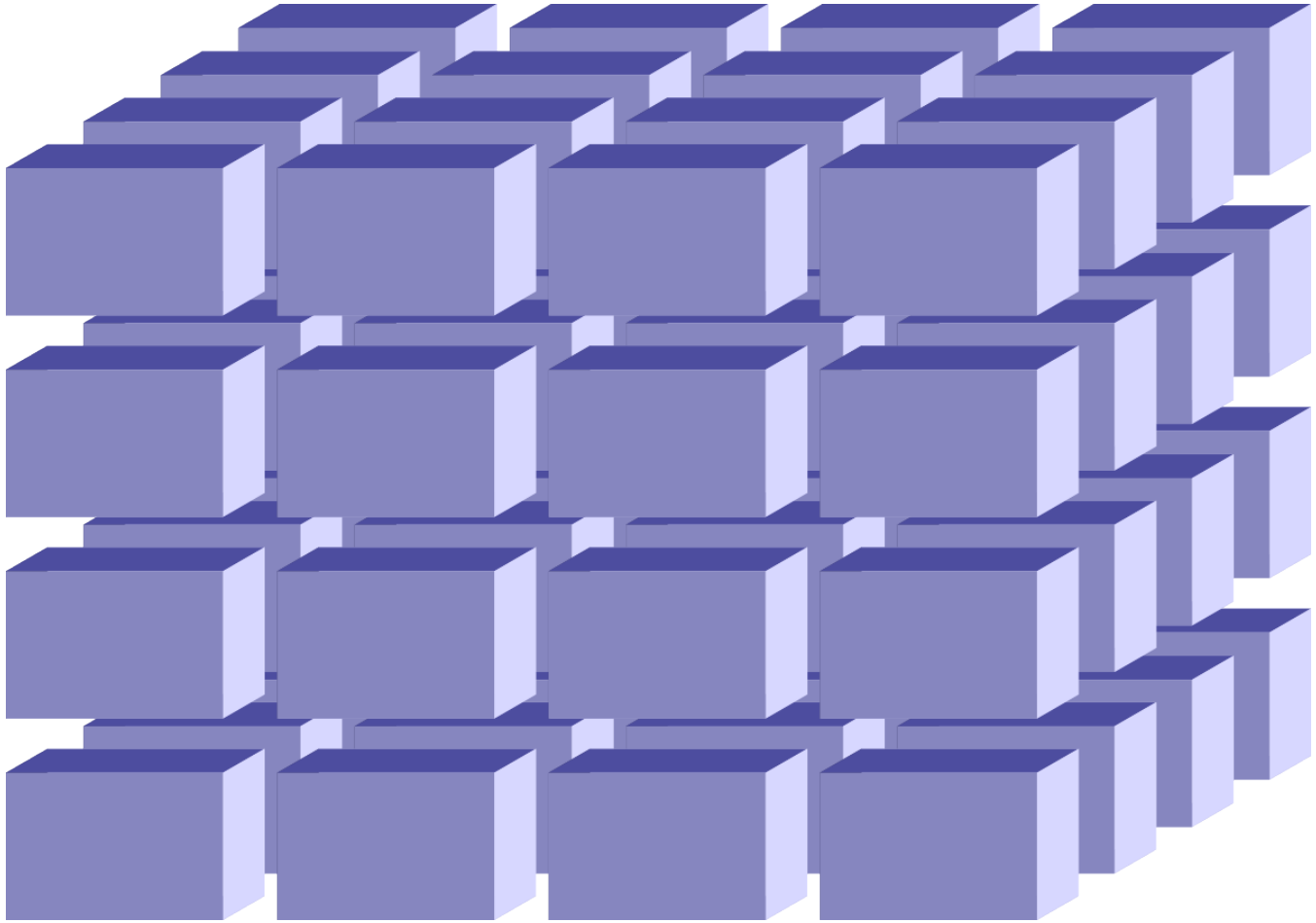
La solution est calculée par un solveur Jacobi où la solution de l'itération $n+1$ est calculée à partir de la solution précédente de l'itération n.

$$u_{ijk}^{n+1} = \frac{1}{6}(u_{i+1jk}^{n} + u_{i-1jk}^{n} + u_{ij+1k}^{n} + u_{ij-1k}^{n} + u_{ijk+1}^{n} + u_{ijk-1}^{n} - h^2 f_{ijk})$$

Etude de cas : Poisson3D

Décomposition de domaine 3D

Le domaine physique est découpé dans les 3 directions spatiales.



Etude de cas : Poisson3D sur Babel

Versions

4 versions différentes ont été développées :

1. Version MPI pur sans recouvrement calculs-communications
2. Version hybride MPI + OpenMP sans recouvrement calculs-communications
3. Version MPI pur avec recouvrement calculs-communications
4. Version hybride MPI + OpenMP avec recouvrement calculs-communications

Les versions OpenMP sont toutes *Fine-grain*.

Babel

Tous les tests ont été réalisés sur Babel qui était un système IBM Blue Gene/P de 10.240 nœuds chacun avec 4 cœurs et 2 Gio de mémoire.

Phénomènes intéressants

- Effets de cache
- Types dérivés
- Placement des processus

Etude de cas : Poisson3D sur Babel

Topologie cartésienne et utilisation des caches

Version	*Topologie*	*Time* (s)	*L1 read* (Tio)	*DDR read* (Tio)	*Torus send* (Gio)
MPI avec recouv	16x4x4	52.741	11.501	14.607	112.873
MPI avec recouv	4x16x4	39.039	11.413	7.823	112.873
MPI avec recouv	4x4x16	36.752	11.126	7.639	37.734

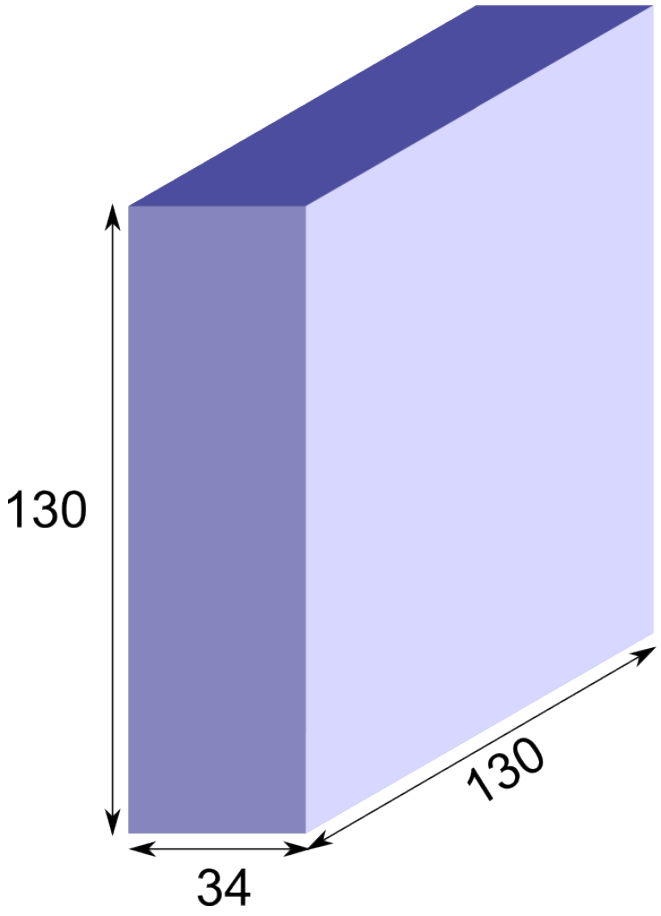
Exécutés sur 256 cœurs Blue Gene/P avec une taille de 512^3.

- L'effet du découpage de la topologie cartésienne est très important.
- Le phénomène semble dû à des effets de cache. En 512^3, les tableaux *u* et *u_nouveau* occupent 8 Mio/cœur.
- Selon la topologie, les accès à la mémoire centrale sont très différents (entre 7,6 Tio et 18,8 Tio en lecture). Le temps de restitution semble fortement corrélé à ces accès.

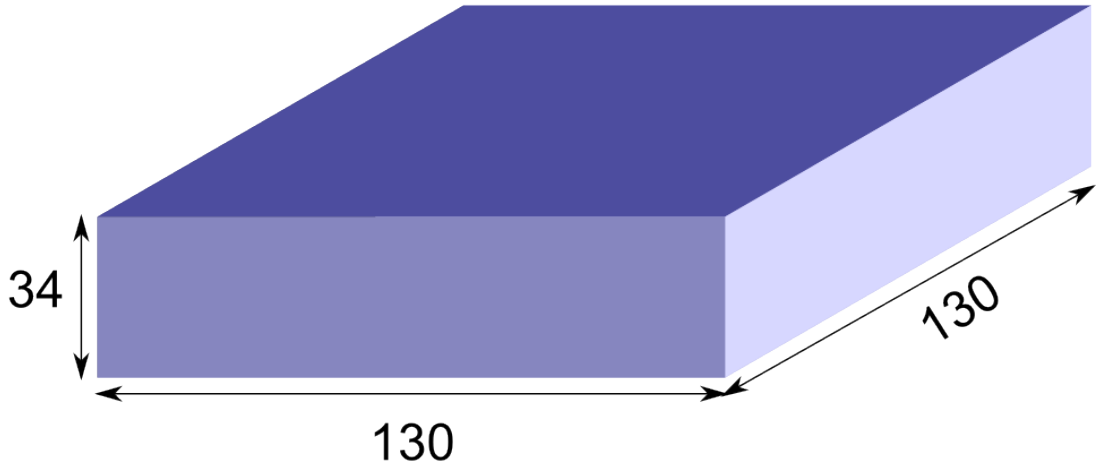
Etude de cas : Poisson3D

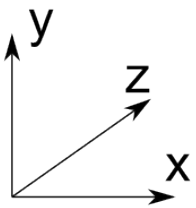
Forme des sous-domaines (512^3)

16x4x4



4x16x4





Etude de cas : Poisson3D sur Babel

Effets de cache

- L'effet de la forme de la topologie cartésienne s'explique par la disposition dans les caches.
- Les tableaux *u* et *u_nouveau* sont découpés dans la topologie 16*x*4*x*4 en $(34, 130, 130)$ et dans la topologie 4*x*16*x*4 en $(130, 34, 130)$.
- Dans le calcul du domaine extérieur, le calcul des faces à $i = cte$ entraîne l'utilisation d'un seul élement *u_nouveau* par ligne de cache L3 (qui contient 16 doubles).
- Les faces $i = cte$ étant 4x plus petites en 4x16x4 qu'en 16x4x4, une grande partie de l'écart vient de là.

Pour améliorer l'utilisation des caches, on peut calculer dans le domaine extérieur plus de plans $i = cte$ qu'avant.

Topologie	*Nplans*	*Time (s)*
4x16x4	1	39.143
4x16x4	16	35.614

Topologie	*Nplans*	*Time (s)*
16x4x4	1	52.777
16x4x4	16	41.559

Etude de cas : Poisson3D sur Babel

Effets de cache sur les types dérivés : analyse

La version hybride est presque toujours plus lente que la version MPI pure.

- A nombre de cœurs égal, les communications prennent 2 fois plus de temps dans la version hybride (256^3 sur 16 cœurs).
- La perte de temps vient de l'envoi des messages utilisant les types dérivés les plus discontigus en mémoire (plans YZ).
- La construction de ces types dérivés n'utilise qu'un seul élément par ligne de cache.
- En hybride, la communication et donc le remplissage du type dérivé est faite par un seul *thread* par processus.
- $\Rightarrow$ un seul flux en lecture (ou écriture) en mémoire par nœud de calcul. L'unité de *prefetching* n'est capable de stocker que 2 lignes de cache L3 par flux.
- En MPI pur, 4 processus par nœud lisent ou écrivent simultanément (sur des faces 4 fois plus petites).
- $\Rightarrow$ 4 flux simultanés et donc remplissage plus rapide

Etude de cas : Poisson3D sur Babel

Effets de cache sur les types dérivés : solution

- Remplacement des types dérivés par des tableaux faces 2D remplis manuellement.
- La copie vers et depuis ces faces est parallélisable en OpenMP.
- Le remplissage se fait maintenant en parallèle comme dans la version MPI pure.

Résultats de quelques tests (512^3) :

	MPI *std*	MPI *no deriv*	MPI+*OMP std*	MPI+*OMP no deriv*
64 cœurs	84.837s	84.390s	102.196s	88.527s
256 cœurs	27.657s	26.729s	25.977s	22.277s
512 cœurs	16.342s	14.913s	16.238s	13.193s

Des améliorations apparaissent aussi sur la version MPI pure.

Etude de cas : Poisson3D sur Babel

Communications MPI

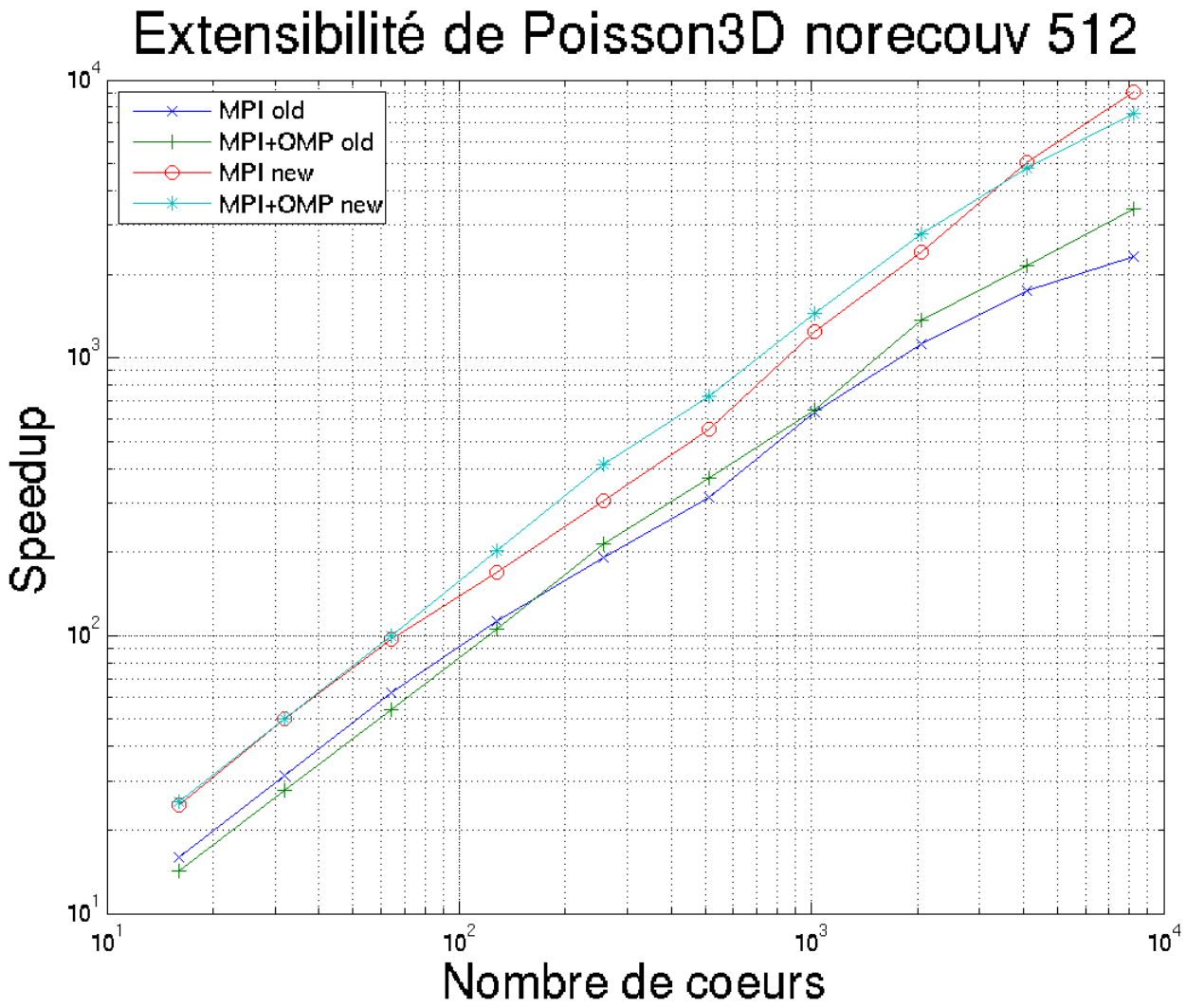
Les tests précédents donnaient des quantités de données envoyées sur le tore 3D variables en fonction de la topologie. Les causes en sont :

- Les messages envoyés entre processus à l'intérieur d'un nœud de calcul ne sont pas inclus. Une topologie dans laquelle les processus sont bien placés voit donc la quantité de données envoyées sur le réseau diminuer.
- Les mesures incluent aussi le trafic de transit à travers chaque nœud. Un message envoyé à un processus situé sur un nœud non-adjacent à celui de l'émetteur sera donc mesuré plusieurs fois (il génère un réel trafic et produit de la contention sur les liens réseaux).

Version	*Topologie*	*Time (s)*	*L1 read (Tio)*	*DDR read (Tio)*	*Torus send (Gio)*
MPI sans recouv	16x4x4	42.826	11.959	9.265	112.873
MPI avec recouv	8x8x4	45.748	11.437	10.716	113.142
MPI avec recouv	16x4x4	52.741	11.501	14.607	112.873
MPI avec recouv	32x4x2	71.131	12.747	18.809	362.979
MPI avec recouv	4x16x4	39.039	11.413	7.823	112.873
MPI avec recouv	4x4x16	36.752	11.126	7.639	37.734

Etude de cas : Poisson3D sur Babel

Comparaison versions optimisées contre versions originales (sans recouvrement)



Etude de cas : Poisson3D sur Babel

Observations

- La topologie cartésienne a un effet important sur les performances à cause d'effets sur la réutilisation des caches.
- La topologie cartésienne a un effet sur le volume de communication et donc sur les performances.
- L'utilisation des types dérivés a une influence sur les accès mémoire.
- Les versions hybrides sont (légèrement) plus performantes que les versions MPI tant que l'entièreté des tableaux de travail ne tiennent pas dans les caches L3.
- L'obtention de bonnes performances en hybride est possible, mais n'est pas toujours triviale.
- Des gains importants peuvent être réalisés (aussi en MPI pur).
- Une bonne compréhension de l'application et de l'architecture matérielle est nécessaire.
- L'intérêt de l'hybride n'apparaît pas clairement ici (au-delà d'une réduction de l'utilisation de la mémoire), probablement car Poisson3D en MPI pur a déjà une extensibilité très bonne et que l'approche OpenMP choisie est *Fine-grain*.