Programmation hybride MPI-OpenMP

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

CNRS — IDRIS / LA

Version 2.3 — 23 novembre 2015

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

OpenMP avancé

Programmation hybride

Sommaire III

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. C'est l'objet des deux chapitres suivants *MPI avancé* et *OpenMP avancé*.
- 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.

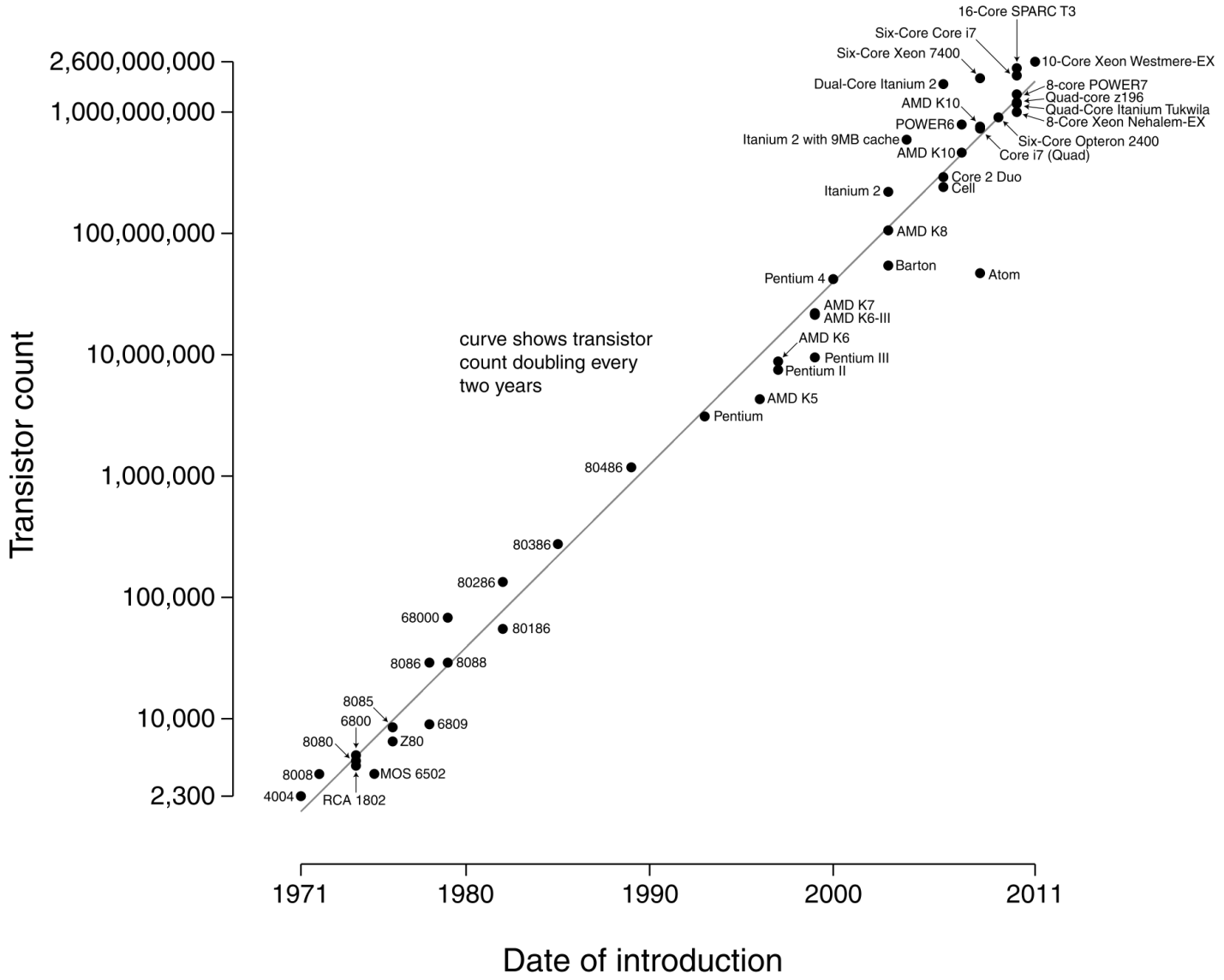
Loi de Moore et consommation électrique

- 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 Haswell-EX ont jusqu'à 18 cœurs et supportent 36 *threads* simultanément et les AMD Abu Dhabi 16 cœurs)

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

Loi de Moore



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.

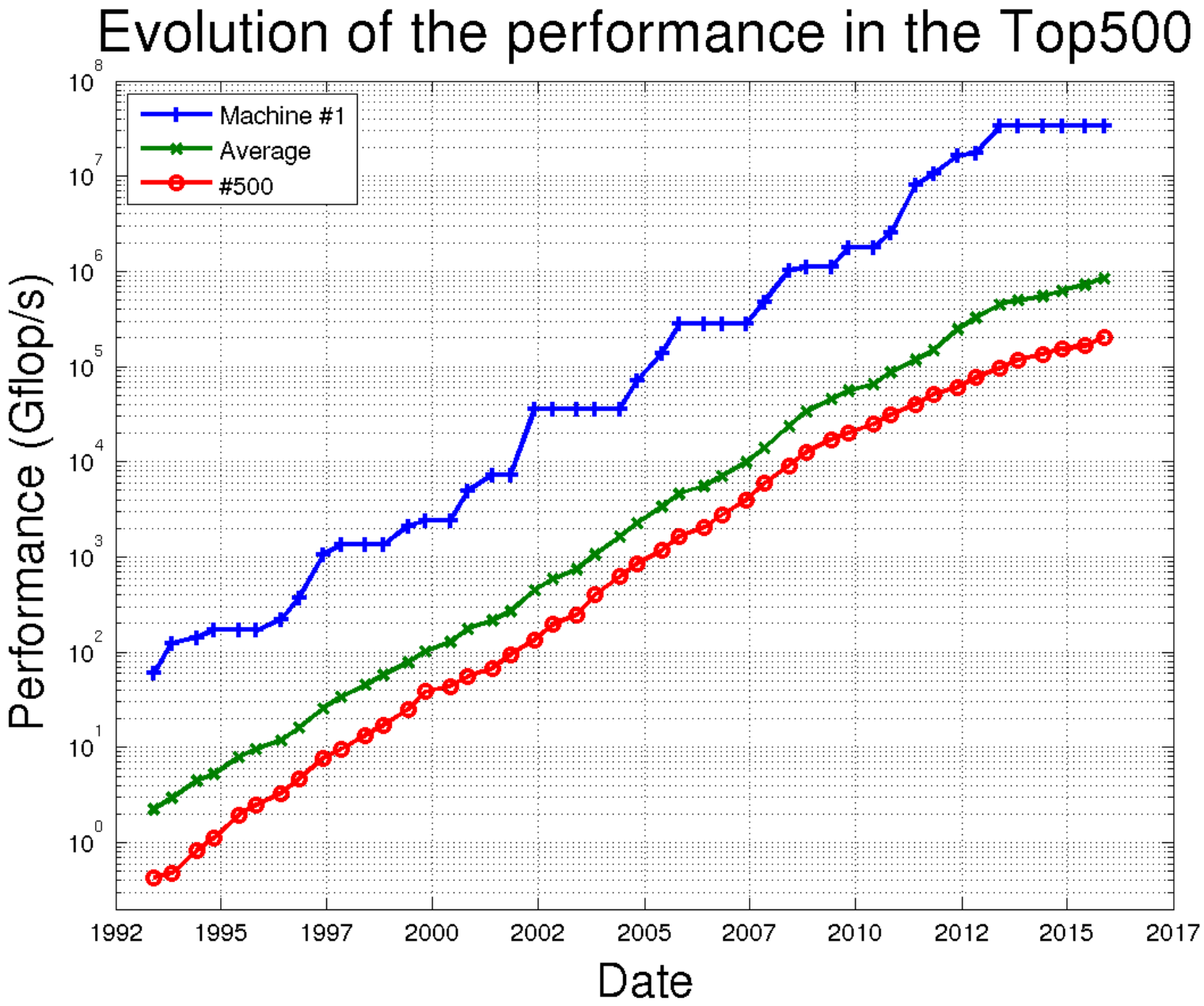
Conséquences

- 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.

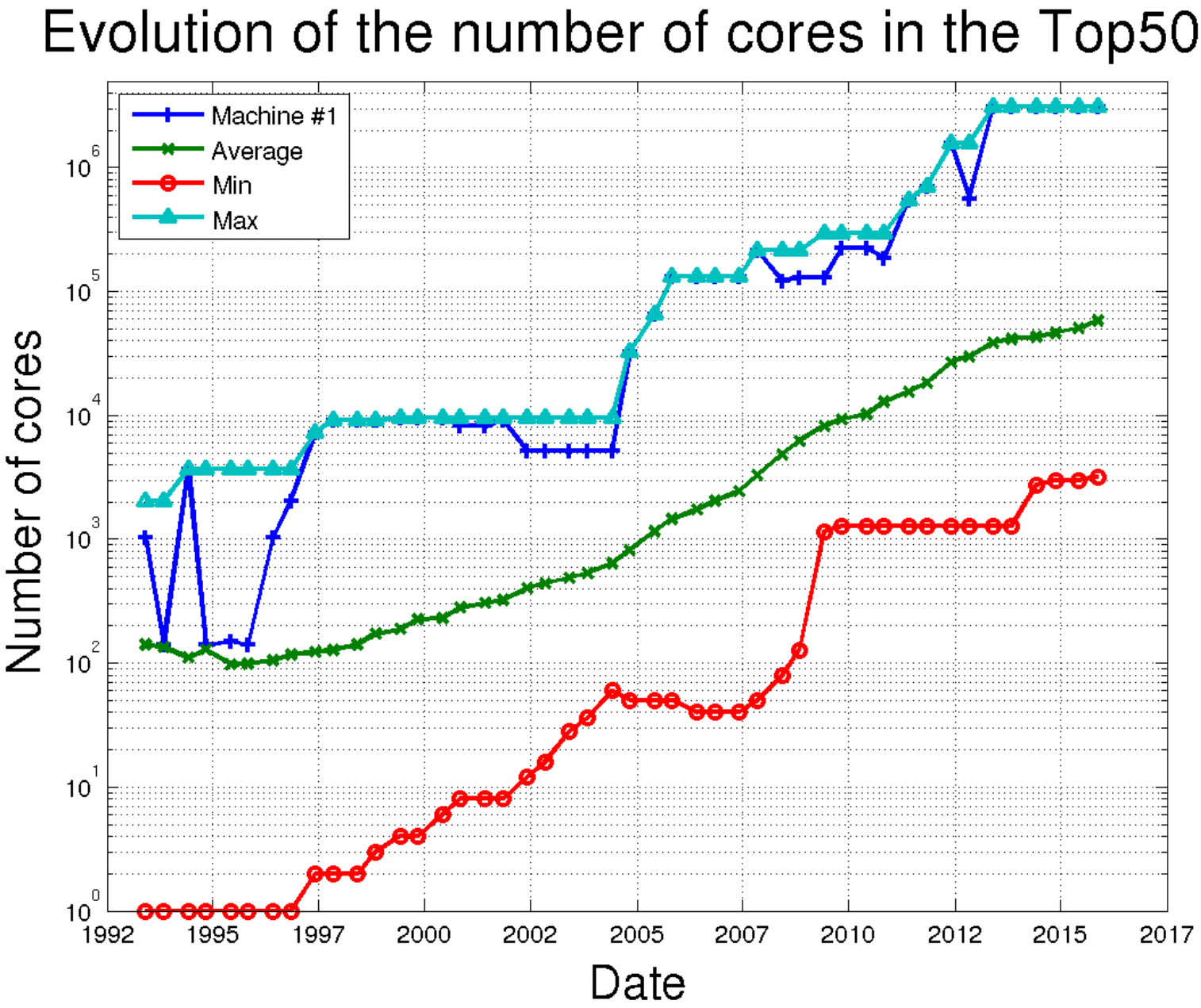
Solutions partielles

- L'ajout de mémoires caches est essentiel.
- Parallélisation des accès via plusieurs bancs mémoire comme sur les architectures vectorielles (Intel Haswell : 4 canaux et AMD : 4).
- Si la fréquence des cœurs stagne ou baisse, l'écart pourrait se réduire.

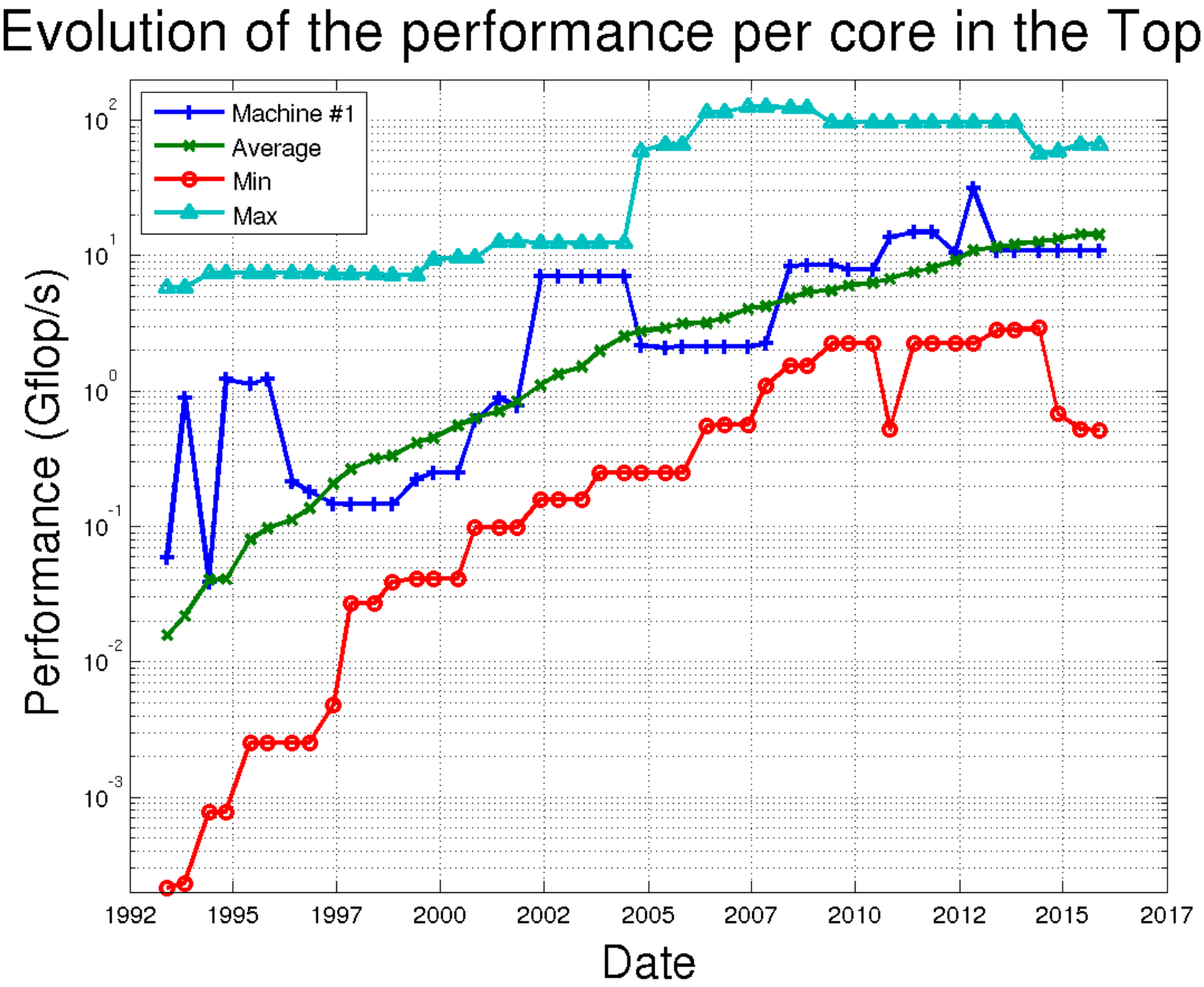
Top 500


Evolution of the performance in the Top500
Machine #1
Average
#500
Performance (Gflop/s)
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015
2017
Date

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 ou Xeons Phi avec des processeurs standards par exemple).
- L'architecture des machines se complexifie et le nombre de niveaux augmente (au niveau des processeurs/cœurs, accès à la 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).
- 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.

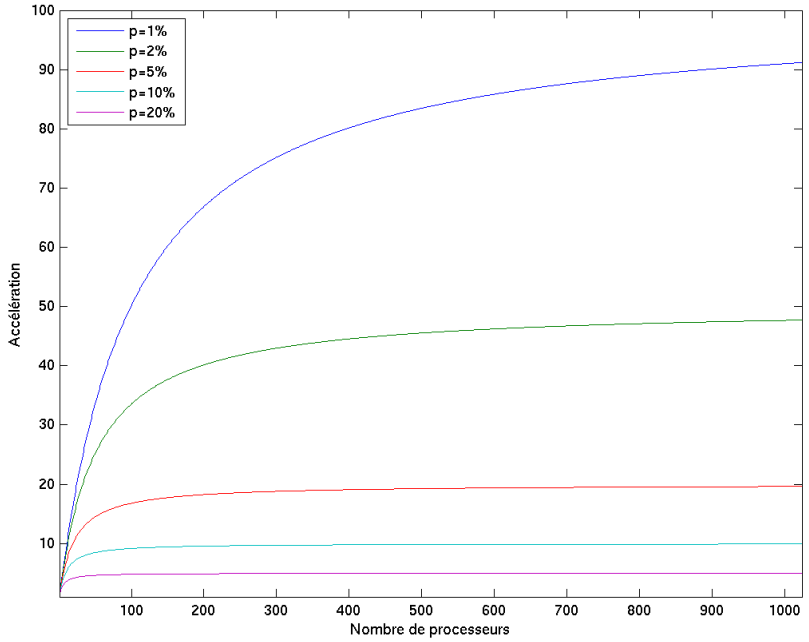
Interprétation

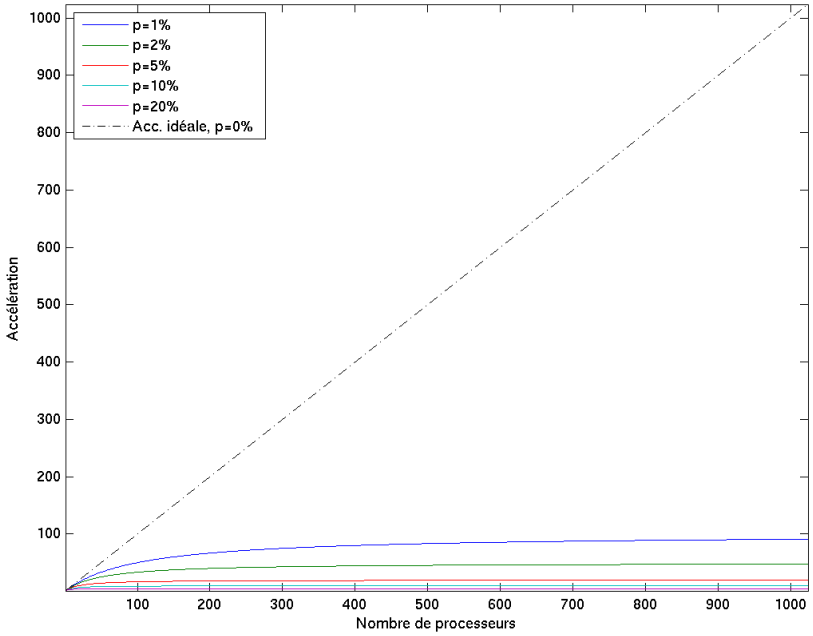
Quel que soit le nombre de cœurs, l'accélération est toujours inférieure à l'inverse du pourcentage que représente la partie purement séquentielle.

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

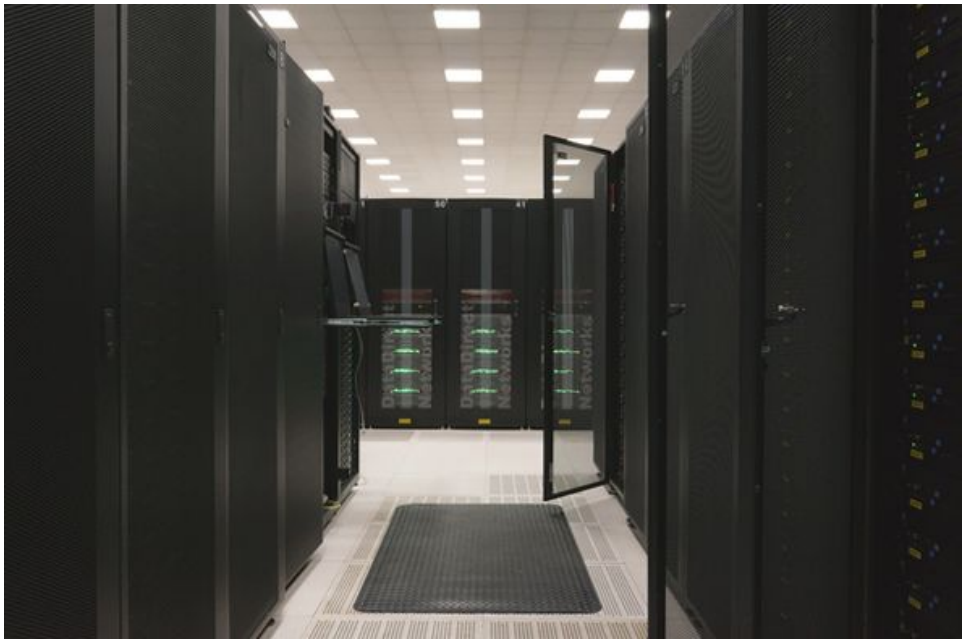
Evolution des méthodes de programmation

- 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 immature.
- 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

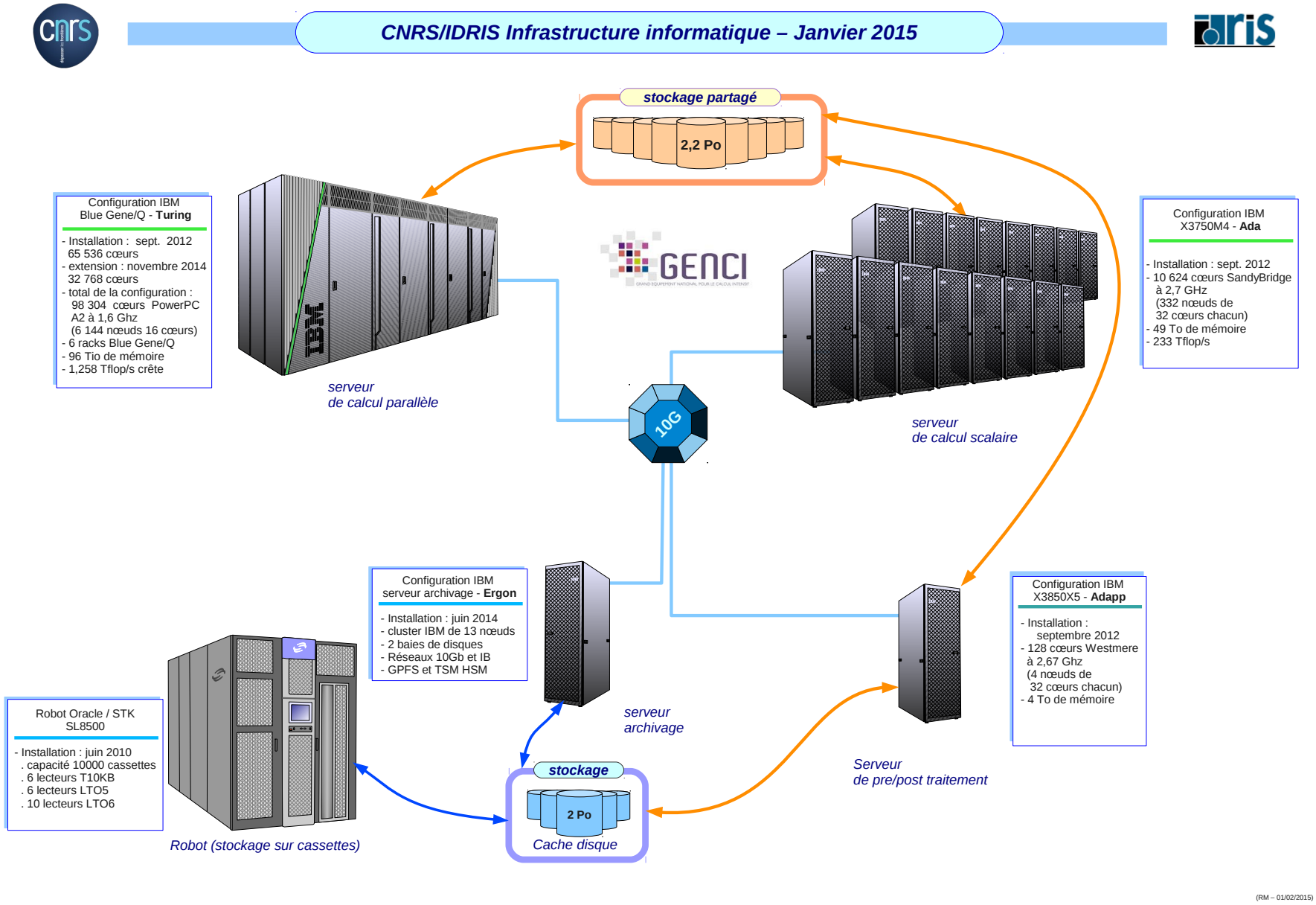


Configuration IDRIS

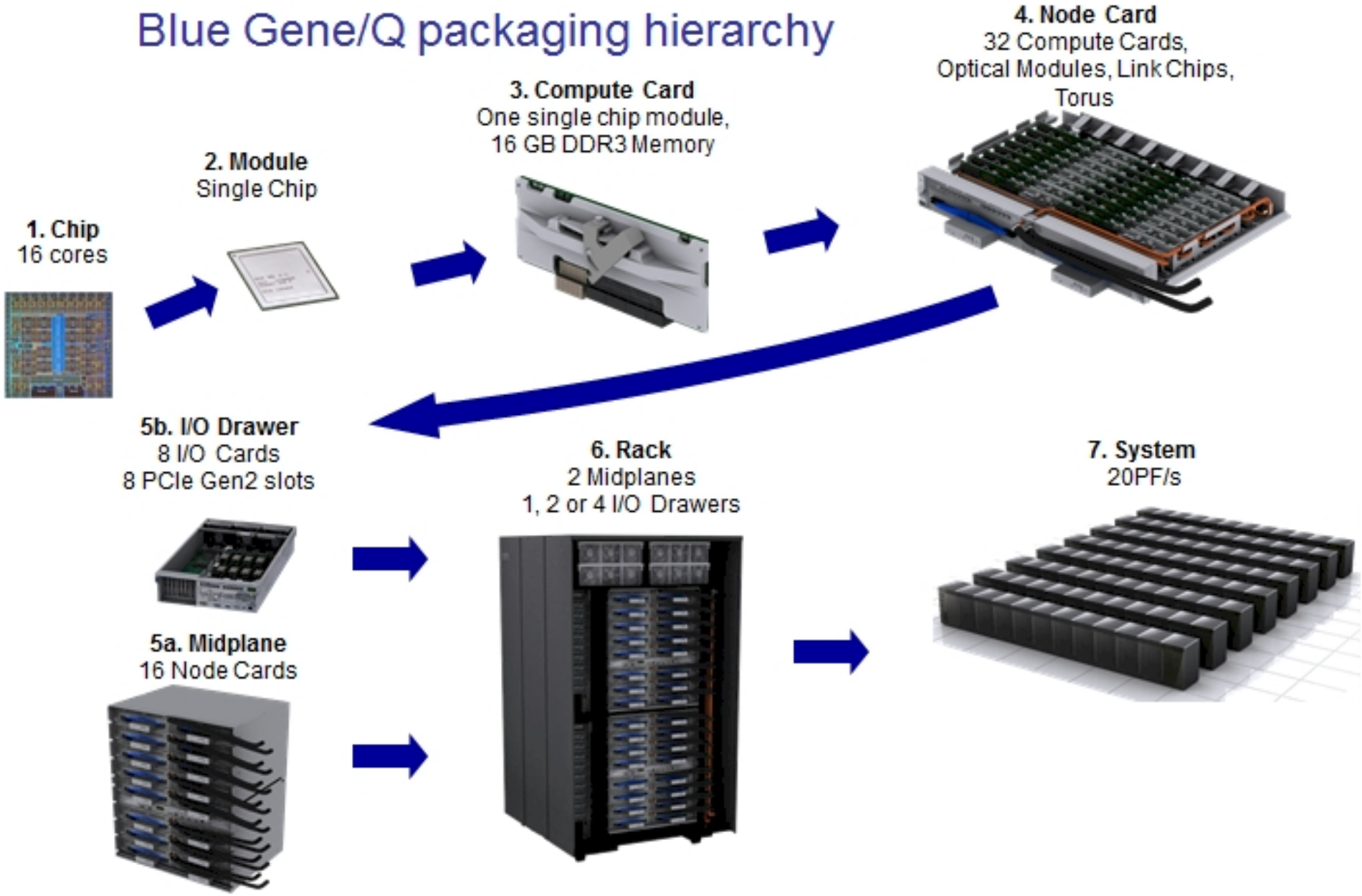
Chiffres importants

- **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

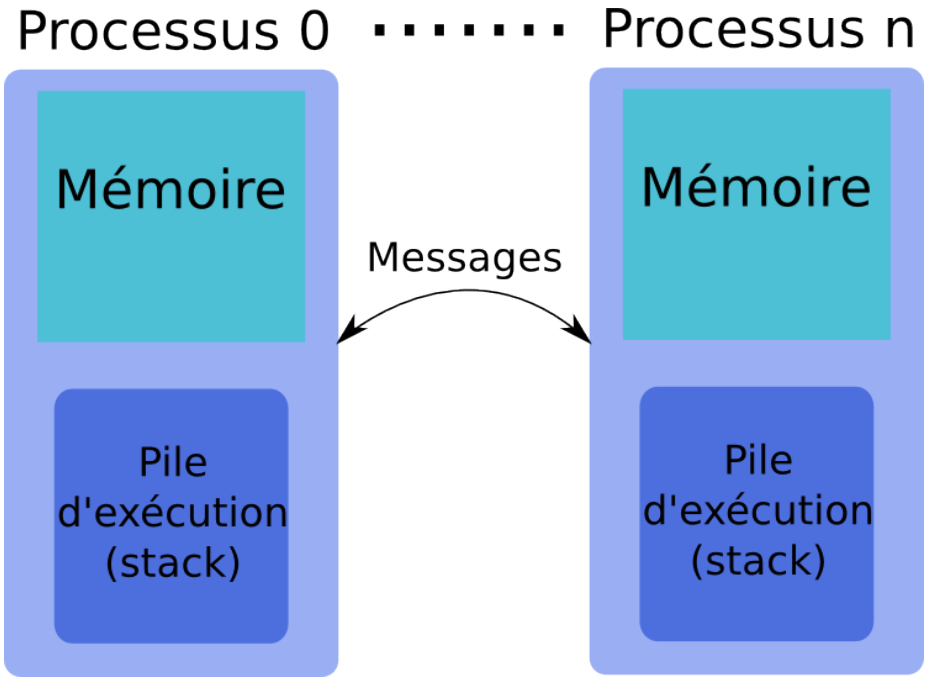


MPI avancé

Présentation de MPI

Présentation de MPI

- Paradigme de parallélisation pour architectures à mémoire distribuée basé sur l'utilisation d'une bibliothèque portable.
- MPI est basé sur une approche de communications entre processus par passage de messages.
- MPI fournit différents types de communications :
  - Point à point
  - Collectives
  - Copies mémoire à mémoire
- MPI fournit également les fonctionnalités suivantes (non exhaustif) :
  - Environnement d'exécution
  - Types de données dérivés
  - Communicateurs et topologies
  - Gestion dynamique de processus
  - Entrées-sorties parallèles
  - Interface de profilage



Présentation de MPI

Limitations de MPI

- L'accélération finale est limitée par la partie purement séquentielle du code (loi d'Amdhal).
- Les surcoûts liès à la bibliothèque MPI et la gestion de l'équilibrage de charge limitent l'extensibilité.
- Certains types de communications collectives prennent de plus en plus de temps lorsque le nombre de processus augmente (par exemple `MPI_Alltoall`).
- Pas de distinction entre processus tournant en mémoire partagée ou distribuée. Or, l'impact sur les performances des communications est majeur. La plupart des implémentations en tiennent compte, mais la norme ne permet pas de remontées d'informations vers l'application.
- Peu de moyens disponibles dans la norme pour mettre en adéquation le matériel et les processus MPI (placement des processus par exemple). Peut souvent se faire en dehors de la norme MPI.

Historique

Passé

- **Version 1.0** : en juin 1994, le forum MPI, avec la participation d'une quarantaine d'organisations, abouti à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages MPI.
- **Version 1.1** : juin 1995, avec seulement des changements mineurs.
- **Version 1.2** : en 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes.
- **Version 1.3** : septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI 2.1.
- **Version 2.0** : apparue en juillet 1997, cette version apportait des compléments essentiels volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc).
- **Version 2.1** : juin 2008, fusion des version 1.3 et 2.0 avec seulement des clarifications dans MPI 2.0 mais aucun changement.
- **Version 2.2** : septembre 2009, avec seulement de « petites » additions.

Historique

MPI 3.0

- Changements et ajouts importants par rapport à la version 2.2.
- Publié en septembre 2012.
- Principaux changements :
  - communications collectives non bloquantes ;
  - révision de l'implémentation des copies mémoire à mémoire ;
  - Fortran (2003-2008) *bindings* ;
  - suppression de l'interface C++ ;
  - interfaçage d'outils externes (pour le débogage et les mesures de performance) ;
  - etc.

MPI 3.1

- Ajouts mineurs, clarifications et corrections
- Publié en juin 2015

État des implémentations actuelles

État des implémentations actuelles

Implémentation	*Norme respectée*
MPICH	3.1 (depuis version 3.2, novembre 2015)
OpenMPI	2.1 (version 1.6.5, depuis 1.3.3) 3.0 (versions 1.8 et 1.10)
IBM Blue Gene/Q	2.2 (sans la gestion dynamique de processus ; basé sur MPICH2-1.5)
Intel MPI	3.0 (à partir version 5.0)
IBM PEMPI	2.2
BullxMPI	2.1 (version 1.2.8.4) 3.0 (version beta)
Cray	3.0

Remarque : la plupart des implémentations incluent une partie de la norme 3.0.

Types de communications MPI

Paradigmes de communications MPI

MPI fournit plusieurs approches pour réaliser des communications entre processus :

- Communications point à point bloquantes ou non bloquantes
- Communications point à point persistantes (voir annexes)
- Communications collectives bloquantes
- Communications collectives non bloquantes à partir de MPI 3.0
- Copies mémoire à mémoire (*one sided communications, RMA*)

Modes d'envoi point à point

Modes d'envoi point à point

Mode	*Bloquant*	*Non bloquant*
Envoi standard	`MPI_Send`	`MPI_Isend`
Envoi synchrone	`MPI_Ssend`	`MPI_Issend`
Envoi *bufferisé*	`MPI_Bsend`	`MPI_Ibsend`
Envoi en mode *ready*	`MPI_Rsend`	`MPI_Irsend`
Réception	`MPI_Recv`	`MPI_Irecv`

Modes d'envoi point à point

Rappels terminologiques

Il est important de bien comprendre la définition de certains termes au sens MPI.

- **Appel bloquant** : un appel est bloquant si l'espace mémoire servant à la communication peut être réutilisé immédiatement après la sortie de l'appel. Les données qui ont été ou seront envoyées sont celles qui étaient dans cet espace au moment de l'appel. Si il s'agit d'une réception, les données ont été reçues dans cet espace (si le code de retour est `MPI_SUCCESS`).
- **Appel non bloquant** : un appel non bloquant rend la main très rapidement, mais n'autorise pas la réutilisation immédiate de l'espace mémoire utilisé dans la communication. Il est nécessaire de s'assurer que la communication est bien terminée (avec `MPI_Wait` par exemple) avant de l'utiliser à nouveau.
- **Envoi synchrone** : un envoi synchrone implique une synchronisation entre les processus concernés. Il ne peut donc y avoir communication que si les deux processus sont prêts à communiquer. Un envoi ne pourra commencer que lorsque sa réception sera postée.
- **Envoi *bufferisé*** : un envoi *bufferisé* implique la recopie des données dans un espace mémoire intermédiaire. Il n'y a alors pas de couplage entre les deux processus de la communication. La sortie de ce type d'envoi ne signifie donc pas que la réception a eu lieu.

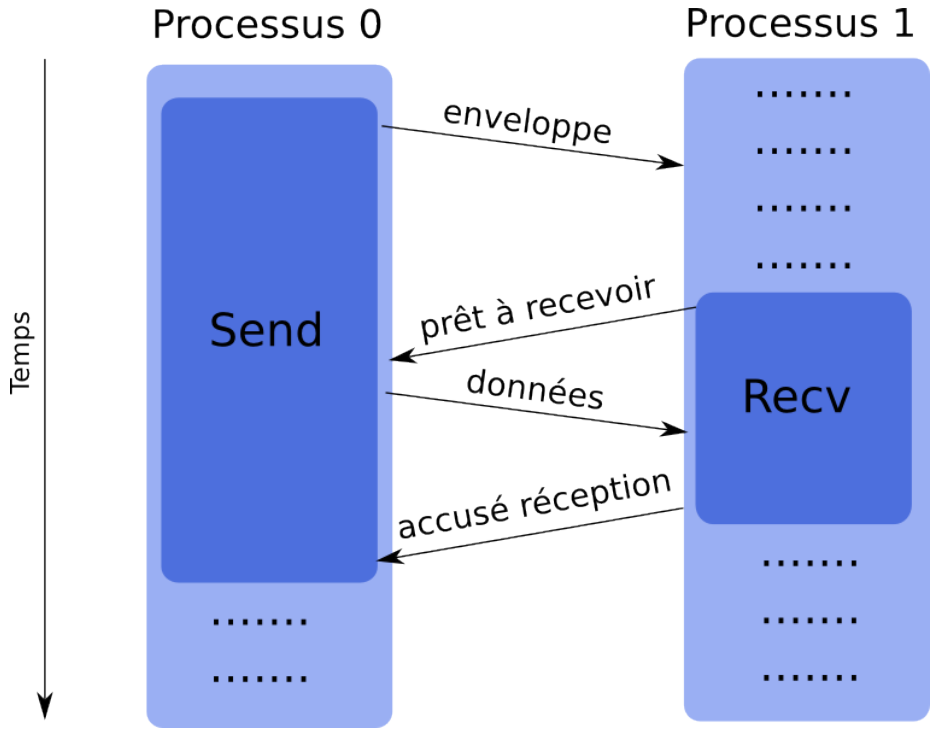
Envois synchrones

Envois synchrones

Un envoi synchrone se fait en appelant le sous-programme `MPI_Ssend` ou `MPI_Issend`.

Protocole de *rendez-vous*

Le protocole de *rendez-vous* est généralement celui employé pour les envois en mode synchrone (dépend de l'implémentation). L'accusé de réception est optionnel.



Envois synchrones

Avantages

- Consomment peu de ressources (pas de *buffer*)
- Rapides si le récepteur est prêt (pas de recopie dans un *buffer*)
- Garantie de la réception grâce à la synchronisation

Inconvénients

- Temps d'attente si le récepteur n'est pas là/pas prêt
- Risques de *deadlocks*

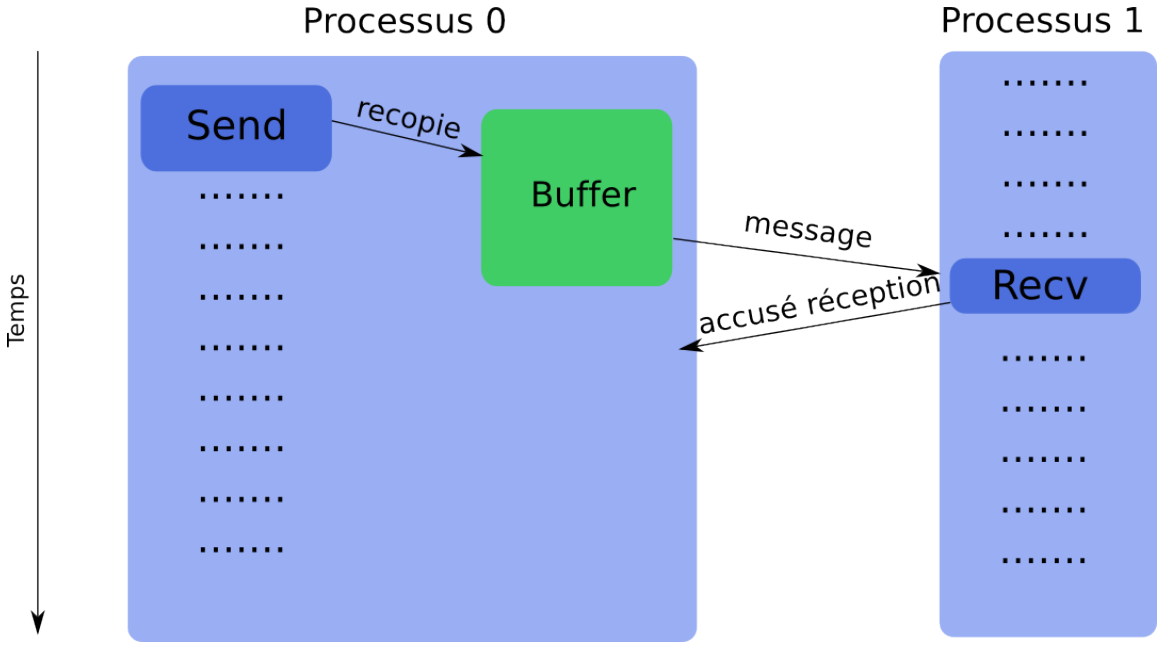
Envois *bufferisés*

Envois *bufferisés*

Un envoi *bufferisé* se fait en appellant le sous-programme `MPI_Bsend` ou `MPI_Ibsend`. Les *buffers* doivent être gérés manuellement (avec appels à `MPI_Buffer_attach` et `MPI_Buffer_detach`). Ils doivent être alloués en tenant compte des surcoûts mémoire des messages (en ajoutant la constante `MPI_BSEND_OVERHEAD` pour chaque instance de message).

Protocole avec *buffer* utilisateur du côté de l'émetteur

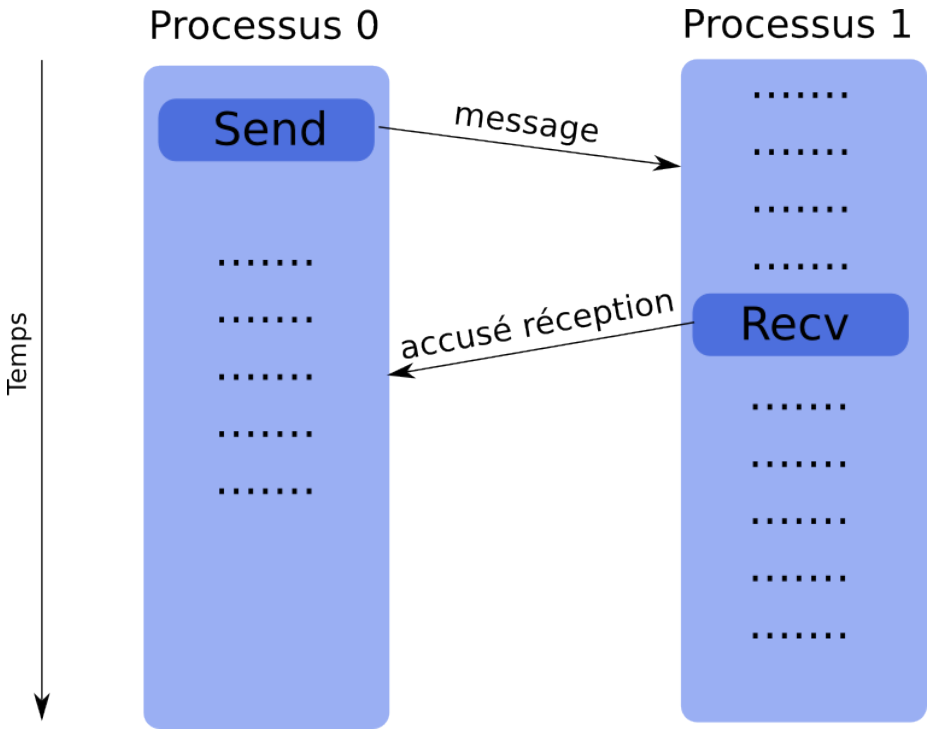
Cette approche est celle généralement employée pour les appels `MPI_Bsend` ou `MPI_Ibsend`. Dans cette approche, le *buffer* se trouve du côté de l'émetteur et est gérée explicitement par l'application. Un *buffer* géré par MPI peut exister du côté du récepteur. De nombreuses variantes sont possibles. L'accusé de réception est optionnel.



Envois *bufferisés*

Protocole *eager*

Le protocole *eager* est souvent employé pour les envois en mode standard pour les messages de petites tailles. Il peut aussi être utilisé pour les envois avec `MPI_Bsend` avec des petits messages (dépend de l'implémentation) et en court-circuitant le *buffer* utilisateur du côté de l'émetteur. Dans cette approche, le *buffer* se trouve du côté du récepteur. L'accusé de réception est optionnel.



Envois *bufferisés*

Avantages

- Pas besoin d'attendre le récepteur (recopie dans un *buffer*)
- Pas de risque de blocage (*deadlocks*)

Inconvénients

- Consomment plus de ressources (occupation mémoire par les *buffers* avec risques de saturation)
- Les *buffers* d'envoi utilisés dans les appels `MPI_Bsend` ou `MPI_Ibsend` doivent être gérés manuellement (souvent délicat de choisir une taille adaptée)
- Un peu plus lents que les envois synchrones si le récepteur est prêt
- Pas de garantie de la bonne réception (découplage envoi-réception)
- Risque de gaspillage d'espace mémoire si les *buffers* sont trop surdimensionnés
- L'application plante si les *buffers* sont trop petits
- Il y a aussi souvent des *buffers* cachés gérés par l'implémentation MPI du côté de l'expéditeur et/ou du récepteur (et consommant des ressources mémoire)

Envois standards

Envois standards

Un envoi standard se fait en appelant le sous-programme `MPI_Send` ou `MPI_Isend`.
Dans la plupart des implémentations, ce mode passe d'un mode *bufferisé* à un mode synchrone lorsque la taille des messages croît.

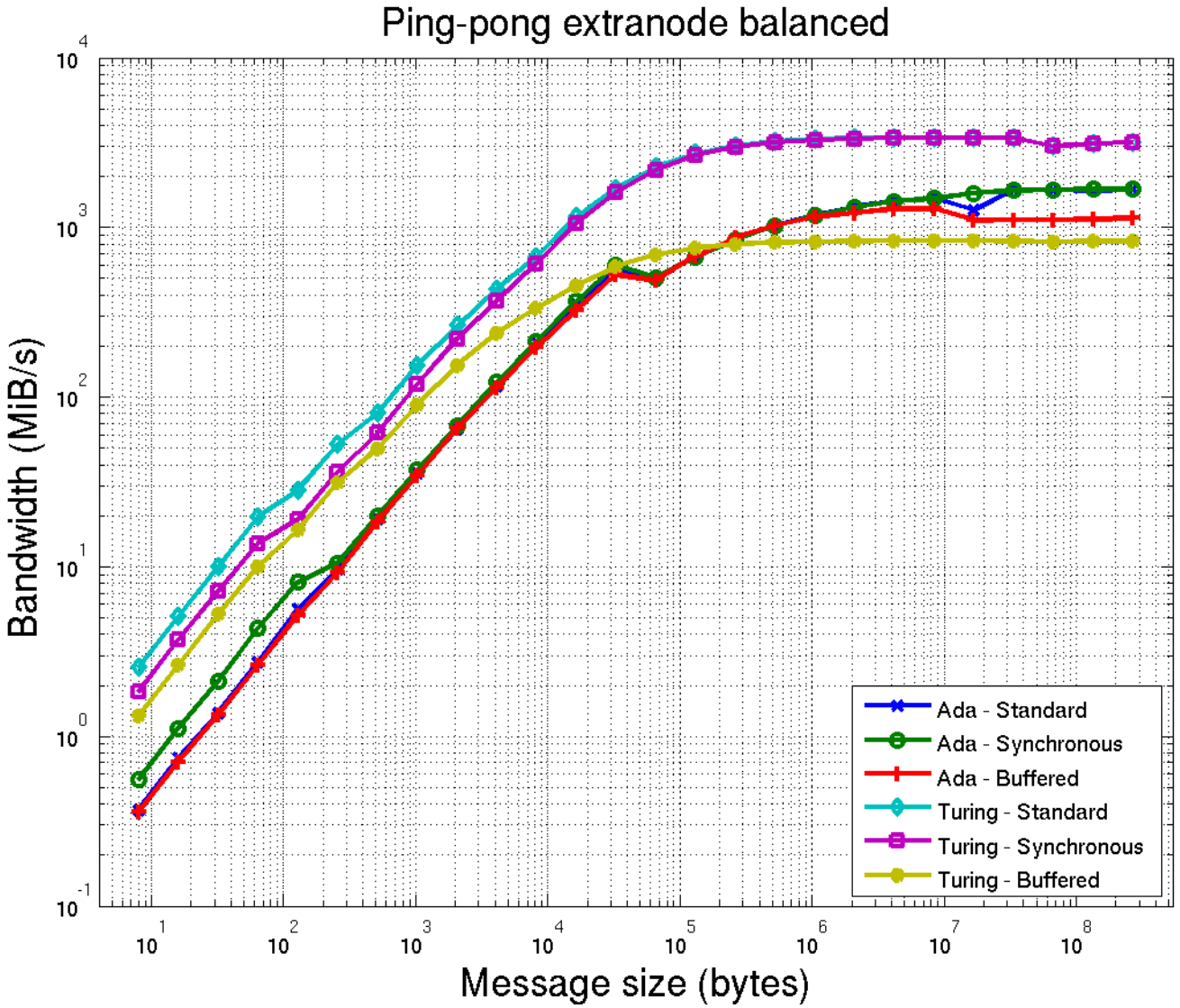
Avantages

- Souvent le plus performant (choix du mode le plus adapté par le constructeur)
- Le plus portable pour les performances

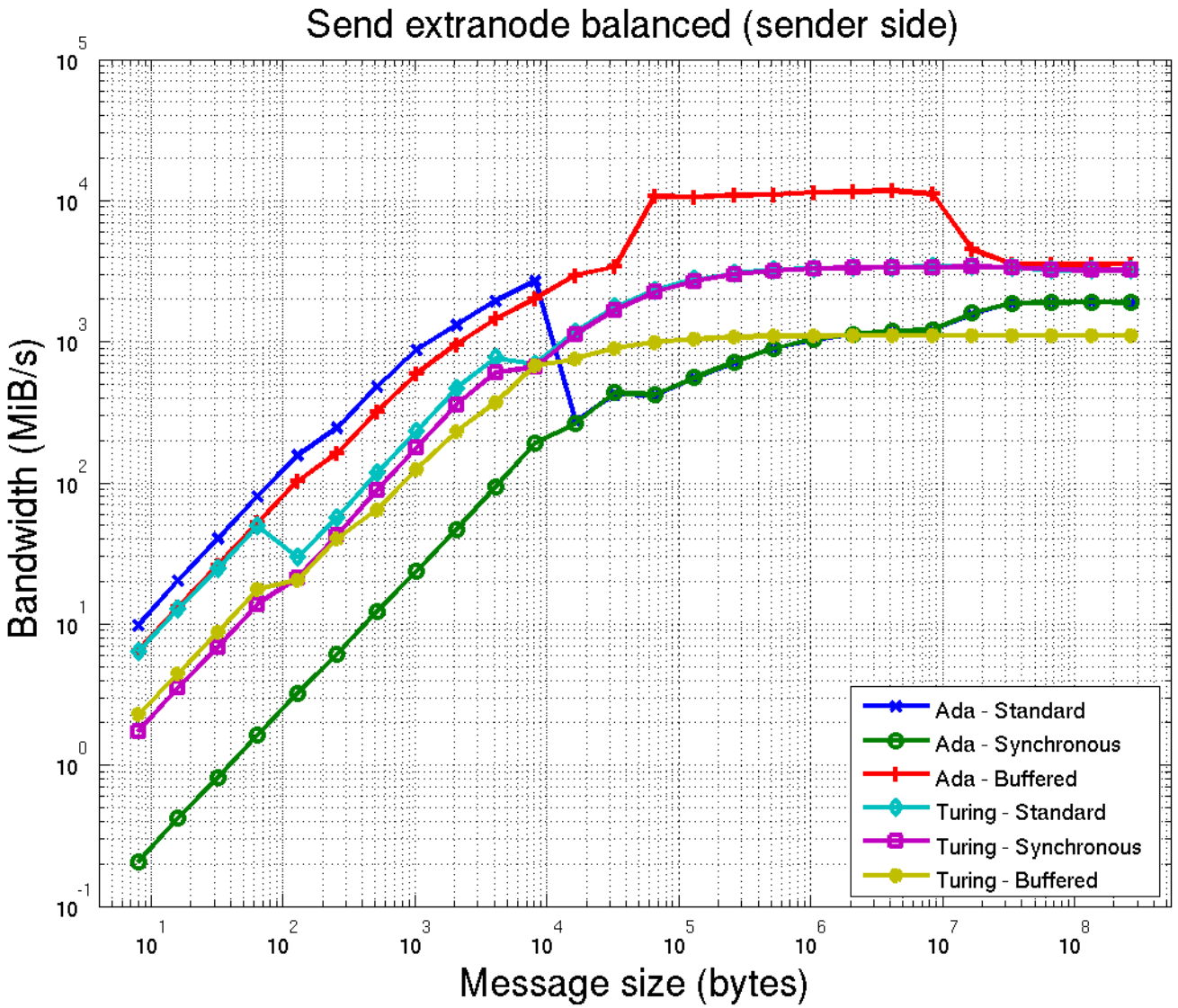
Inconvénients

- Peu de contrôle sur le mode réellement utilisé (souvent accessible via des variables d'environnement)
- Risque de *deadlock* selon le mode réel
- Comportement pouvant varier selon l'architecture et la taille du problème

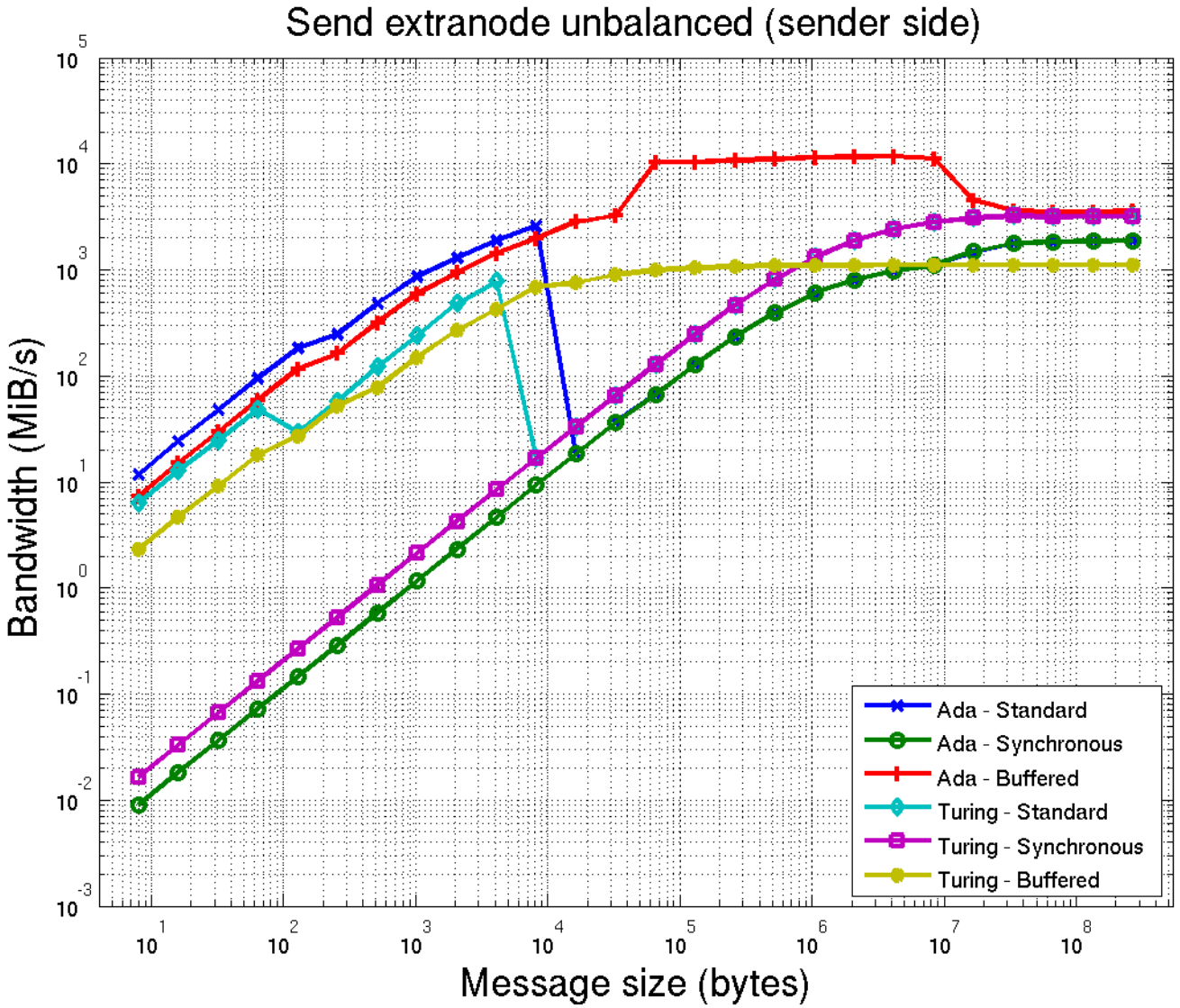
Modes d'envoi point à point



Modes d'envoi point à point



Modes d'envoi point à point



Communications collectives

Définitions et caractéristiques générales

Les communications collectives permettent de réaliser des communications impliquant plusieurs processus.

- Elles peuvent toujours être simulées par un ensemble de communications point à point (mais peuvent être fortement optimisées) avec éventuellement des opérations de réduction.
- Une communication collective implique l'ensemble des processus du communicateur utilisé.
- Jusqu'à MPI 2.2, il s'agissait d'appels bloquants (càd qu'un processus ne sort de l'appel que lorsque sa participation à la communication est terminée). La norme MPI 3.0 introduit les appels non bloquants pour les communications collectives.
- Les communications collectives n'impliquent pas de synchronisation globale (sauf `MPI_Barrier`) et n'en nécessitent pas.
- Elles n'interfèrent jamais avec les communications point à point.

Il faut **toujours** (ou presque) les préférer aux communications point à point.

Communications collectives

Catégories

Les communications collectives peuvent être séparées en 3 catégories :

- Synchronisations globales (`MPI_Barrier`) à n'utiliser que si nécessaire (rare)
- Tranferts/échanges de données
  - Diffusion de données (globales avec `MPI_Bcast`, sélectives avec `MPI_Scatter`)
  - Collecte de données (`MPI_Gather` et `MPI_Allgather`)
  - Echanges globaux (`MPI_Alltoall`)
- Opérations de réduction (`MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter` et `MPI_Scan`)

Communications collectives

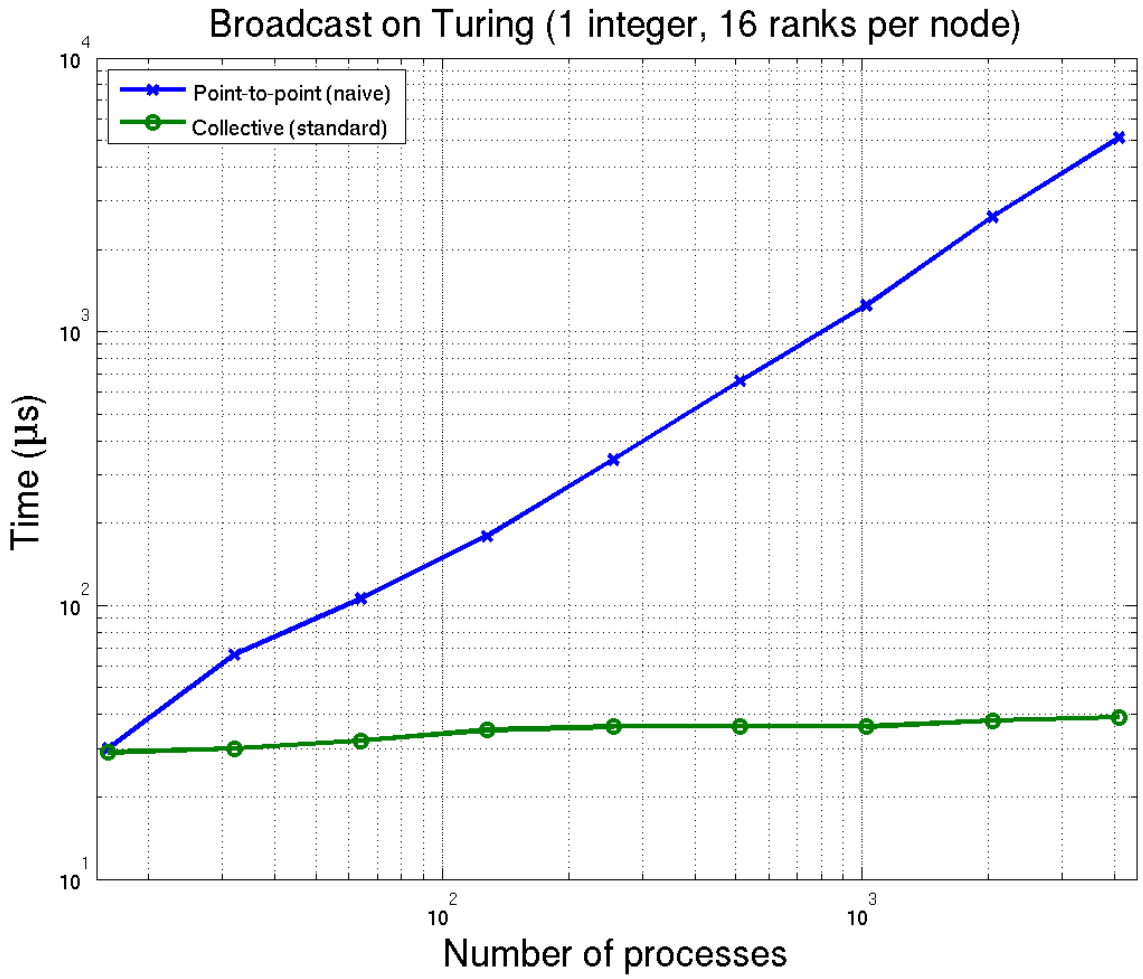
Avantages (par rapport au point à point)

- Sont fortement optimisées.
- Une série de communications point à point en une seule opération.

Inconvénients (par rapport au point à point)

- Peut cacher au programmeur un volume de transfert très important (par exemple, un `MPI_Alltoall` avec 1024 processus implique plus de 1 million de messages point à point).
- Pas d'appels non bloquants (plus vrai dans la norme MPI 3.0).
- Implique tous les processus du communicateur. Il faut donc créer des sous-communicateurs si tous les processus ne sont pas concernés par une communication collective.

Collectif vs point à point



Diffusion globale simulée avec des communications point à point (boucle de `MPI_Send` sur un processus et `MPI_Recv` sur les autres) versus un appel à `MPI_Bcast`

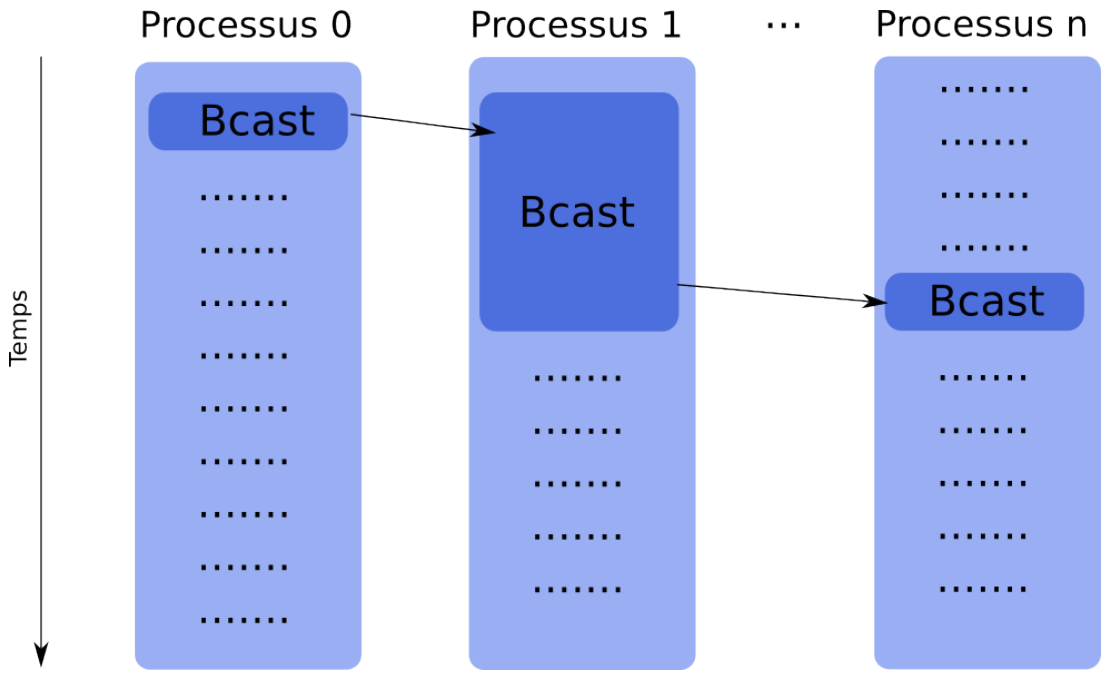
Communications collectives

Définition synchronisation globale

Une synchronisation globale (ou barrière) est une opération telle qu'à un instant donné l'ensemble des processus concernés seront dans un même appel. Un processus arrivant dans cet appel ne pourra pas en sortir tant que tous les autres n'y sont pas. Par contre, rien n'impose qu'ils en sortent tous simultanément.

Communications collectives synchronisantes ?

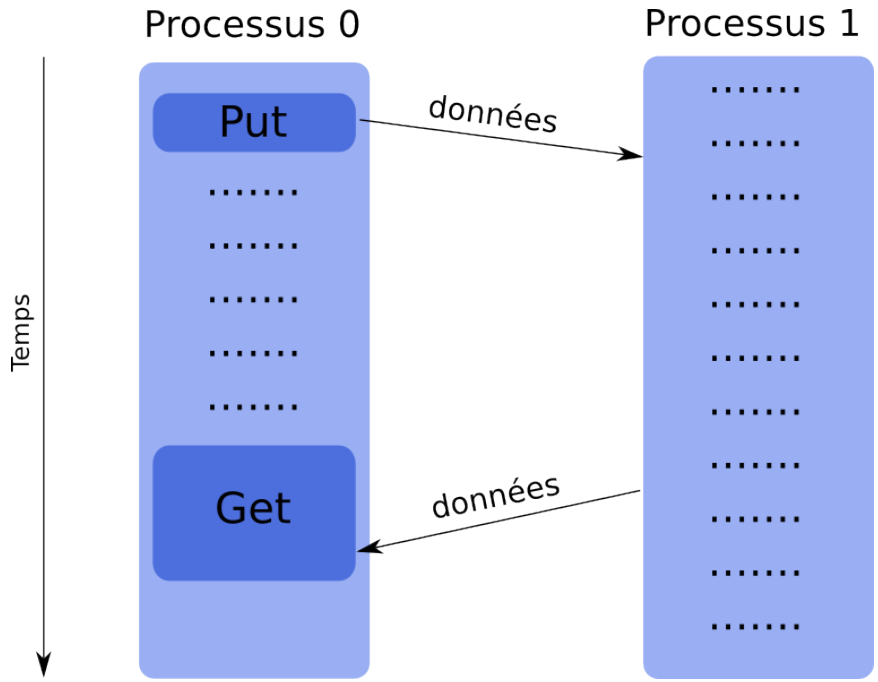
Les communications collectives n'impliquent pas de synchronisation globale (sauf `MPI_Barrier`) et n'en nécessitent pas. Les implémentations sont libres d'en mettre dans tous les appels collectifs et les développeurs doivent s'assurer que leurs applications fonctionnent dans ces cas.



Communications mémoire à mémoire (RMA)

Définition

Les communications mémoire à mémoire (ou RMA pour *Remote Memory Access* ou *one sided communications*) consistent à accéder en écriture ou en lecture à la mémoire d'un processus distant sans que ce dernier doive gérer cet accès explicitement. Le processus cible n'intervient donc pas lors du transfert.



Communications mémoire à mémoire (RMA)

Approche générale

- Création d'une fenêtre mémoire avec `MPI_Win_create` pour autoriser les transferts RMA dans cette zone.
- Accès distants en lecture ou écriture en appellant `MPI_Put`, `MPI_Get` ou `MPI_Accumulate`.
- Libération de la fenêtre mémoire avec `MPI_Win_free`.

Méthodes de synchronisation

Pour s'assurer d'un fonctionnement correct, il est obligatoire de réaliser certaines synchronisations. 3 méthodes sont disponibles :

- Communication à cible active avec synchronisation globale (`MPI_Win_fence`) ;
- Communication à cible active avec synchronisation par paire (`MPI_Win_Start` et `MPI_Win_Complete` pour le processus origine ; `MPI_Win_Post` et `MPI_Win_Wait` pour le processus cible) ;
- Communication à cible passive sans intervention de la cible (`MPI_Win_lock` et `MPI_Win_unlock`).

Communications mémoire à mémoire (RMA)

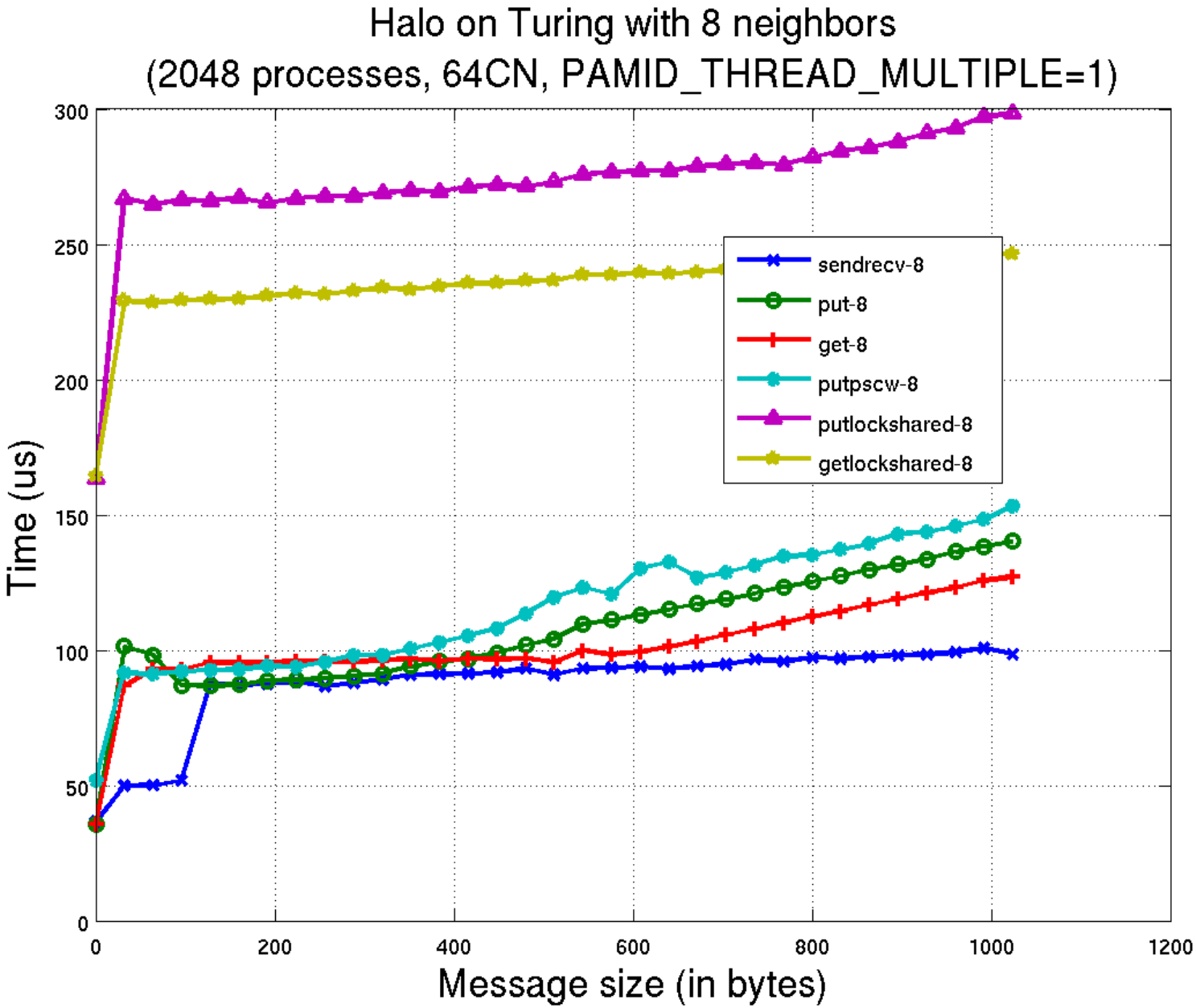
Avantages

- Permet de mettre en place plus efficacement certains algorithmes.
- Plus performant que les communications point à point sur certaines machines (utilisation de matériels spécialisés tels que moteur DMA, coprocesseur, mémoire spécialisée...).
- Possibilité pour l'implémentation de regrouper plusieurs opérations.

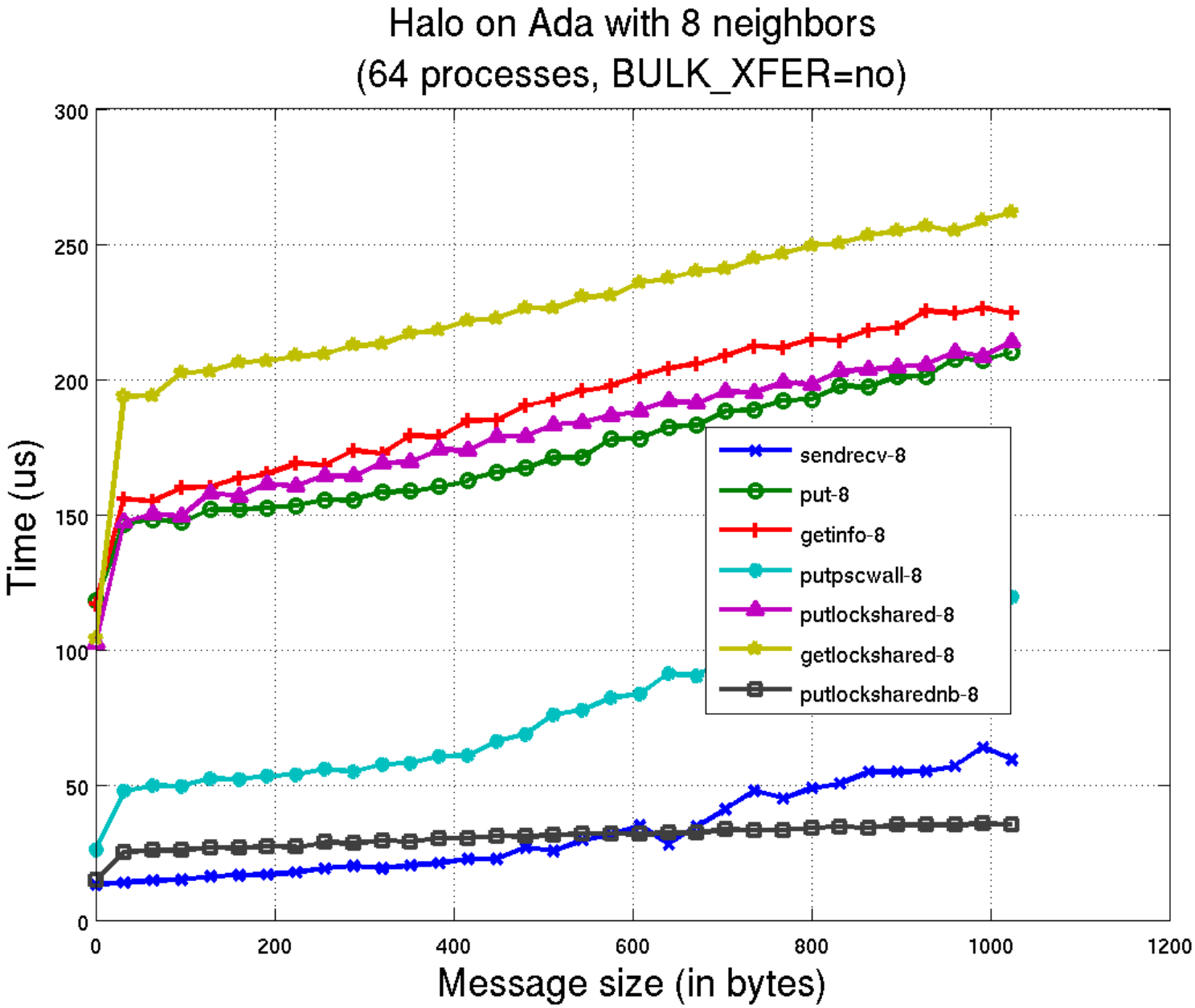
Inconvénients

- La gestion des synchronisations est délicate.
- Complexité et risques d'erreurs élevés.
- Pour les synchronisations cible passive, obligation d'allouer la mémoire avec `MPI_Alloc_mem` qui ne respecte pas la norme Fortran (utilisation de pointeurs Cray non supportés par certains compilateurs).
- Moins performant que les communications point à point sur certaines machines.

Communications mémoire à mémoire (RMA)



Communications mémoire à mémoire (RMA)



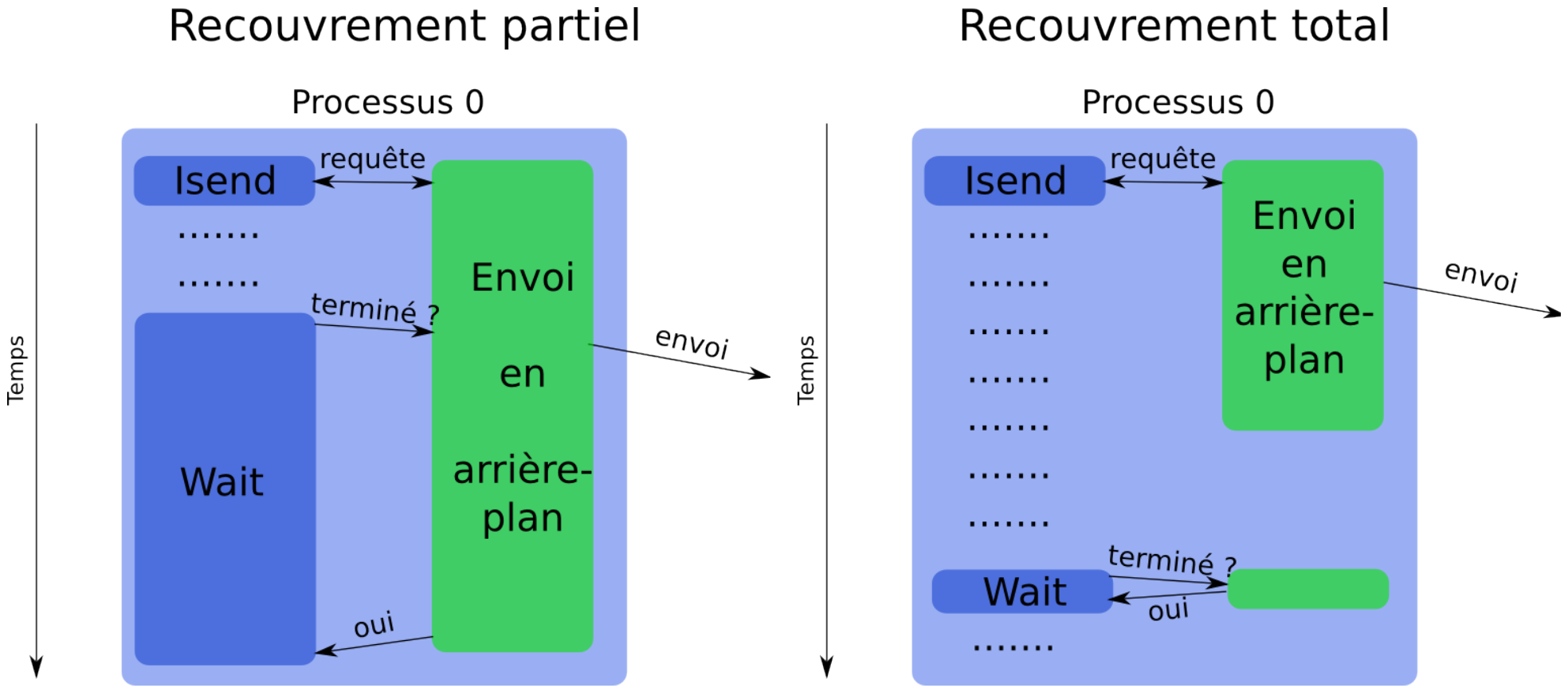
Recouvrement calculs-communications

Présentation

Le recouvrement des communications par des calculs est une méthode permettant de réaliser des opérations de communications en arrière-plan pendant que le programme continue de s'exécuter.

- Il est ainsi possible, si l'architecture matérielle et logicielle le permet, de masquer tout ou partie des coûts de communications.
- Le recouvrement calculs-communications peut être vu comme un niveau supplémentaire de parallélisme.
- Cette approche s'utilise dans MPI par l'utilisation de sous-programmes non-bloquants (i.e. `MPI_Isend`, `MPI_Irecv` et `MPI_Wait`).

Recouvrement calculs-communications



Recouvrement calculs-communications

Avantages

- Possibilité de masquer tout ou partie des coûts des communications (si l'architecture le permet)
- Pas de risques de *deadlock*

Inconvénients

- Surcoûts plus importants (plusieurs appels pour un seul envoi ou réception, gestion des requêtes)
- Complexité plus élevée et maintenance plus compliquée
- Peu performant sur certaines machines (par exemple avec transfert commençant seulement à l'appel de `MPI_Wait`)
- Risque de perte de performance sur les noyaux de calcul (par exemple gestion différenciée entre la zone proche de la frontière d'un domaine et la zone intérieure entraînant une moins bonne utilisation des caches mémoire)
- Limité aux communications point à point (a été étendu aux collectives dans MPI 3.0)

Recouvrement calculs-communications

Utilisation

L'envoi d'un message se fait en 2 étapes :

- Initier l'envoi ou la réception par un appel à un sous-programme commençant par `MPI_Isend` ou `MPI_Irecv` (ou une de leurs variantes)
- Attendre la fin de la contribution locale par un appel à `MPI_Wait` (ou à une de ses variantes).

Les communications se recouvrent par toutes les opérations qui se déroulent entre ces deux étapes. L'accès aux données en cours de réception est interdit avant la fin de l'appel à `MPI_Wait`.

Recouvrement calculs-communications

Exemple

```
do i=1,niter
  ! Initie les communications
  call MPI_Irecv(data_ext,  sz,MPI_REAL,dest,tag,comm, &
                 req(1),ierr)
  call MPI_Isend(data_bound,sz,MPI_REAL,dest,tag,comm, &
                 req(2),ierr)

  ! Calcule le domaine interieur (data_ext et data_bound
  ! non utilises) pendant que les communications ont lieu
  call calcul_domaine_interieur(data_int)

  ! Attend la fin des communications
  call MPI_Waitall(2,req,MPI_STATUSES_IGNORE,ierr)

  ! Calcule le domaine exterieur
  call calcul_domaine_exterieur(data_int,data_bound,data_ext)
end do
```

Recouvrement calculs-communications

Niveau de recouvrement sur différentes machines

Machine	*Niveau*
Blue Gene/Q, PAMID_THREAD_MULTIPLE=0	32%
Blue Gene/Q, PAMID_THREAD_MULTIPLE=1	100%
Ada	33% en extranœud 0% en intranœud
NEC SX-8	10%
CURIE (en 2012)	0%

Mesures faites en recouvrant un noyau de calcul et un noyau de communication de mêmes durées et en utilisant différents schémas de communications (intra/extra-nœuds, par paires, processus aléatoires...).
Selon le schéma de communication, les résultats peuvent être totalement différents.

Un recouvrement de 0% signifie que la durée totale d'exécution vaut 2x la durée d'un noyau de calcul (ou communication).
Un recouvrement de 100% signifie que la durée totale vaut 1x la durée d'un noyau de calcul (ou communication).

Types dérivés

Présentation

- Les types dérivés permettent de représenter des structures de données de n'importe quelle complexité.
- Un niveau d'abstraction important peut être ainsi atteint et permet de cacher la complexité sous-jacente.
- Peuvent être utilisés dans toutes les communications MPI, mais également dans les entrées-sorties.

Types dérivés existants

Types dérivés existants

Type	*Données multiples*	*Non contigu*	*Espacement quelconque*	*Hétérogène*
Types prédéfinis				
`MPI_Type_contiguous`	✓			
`MPI_Type_vector`	✓	✓		
`MPI_Type_indexed`	✓	✓	✓	
`MPI_Type_struct`	✓	✓	✓	✓

`MPI_Type_vector` et `MPI_Type_indexed` donnent les écarts en multiples entiers du type de base.

Les variantes `MPI_Type_create_hvector` et `MPI_Type_create_hindexed` permettent de fournir des écarts en octets.

Il existe également deux autres types dérivés : `MPI_Type_create_subarray` pour créer des sous-tableaux et `MPI_Type_create_darray` pour des tableaux distribués sur un ensemble de processus.

Types dérivés : avantages

Avantages

- Lisibilité
- Niveau d'abstraction élevé
- Types non-contigus supportés
- Types hétérogènes supportés
- Regroupement de messages
- Utilisables aussi dans les I/O

Types dérivés : inconvénients

Inconvénients

- Peuvent être complexes à mettre en place
- Types hétérogènes délicats à mettre en œuvre
- Niveau d'abstraction élevé $\Rightarrow$ perte de maîtrise
- Performances souvent moindres

Types dérivés : alternatives

Copies manuelles dans structures intermédiaires

Gestion des données à la main avec copie dans des structures intermédiaires

- Souvent le plus performant
- Implique des copies mémoire à mémoire supplémentaires
- Utilisation de plus de ressources mémoire
- Gestion manuelle de l'espace mémoire
- N'utilise pas les possibilités de certains sous-systèmes de communications (*scatter-gather hardware*) ou systèmes de fichiers parallèles spécialisés (PVFS par exemple)

Messages séparés

Envoi des données dans des messages séparés.

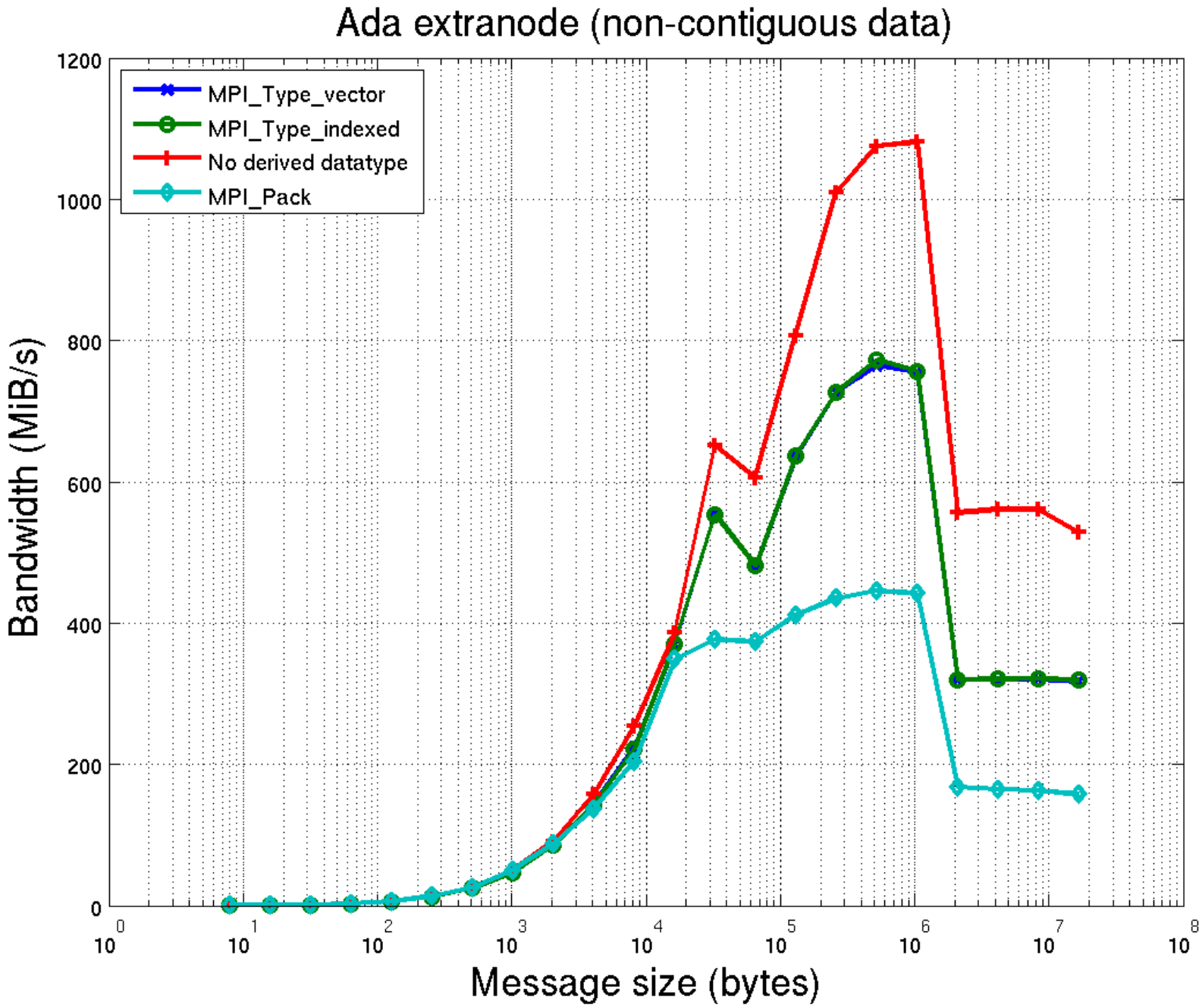
Attention : si nombreux messages, **très mauvaises performances**. A éviter absolument.

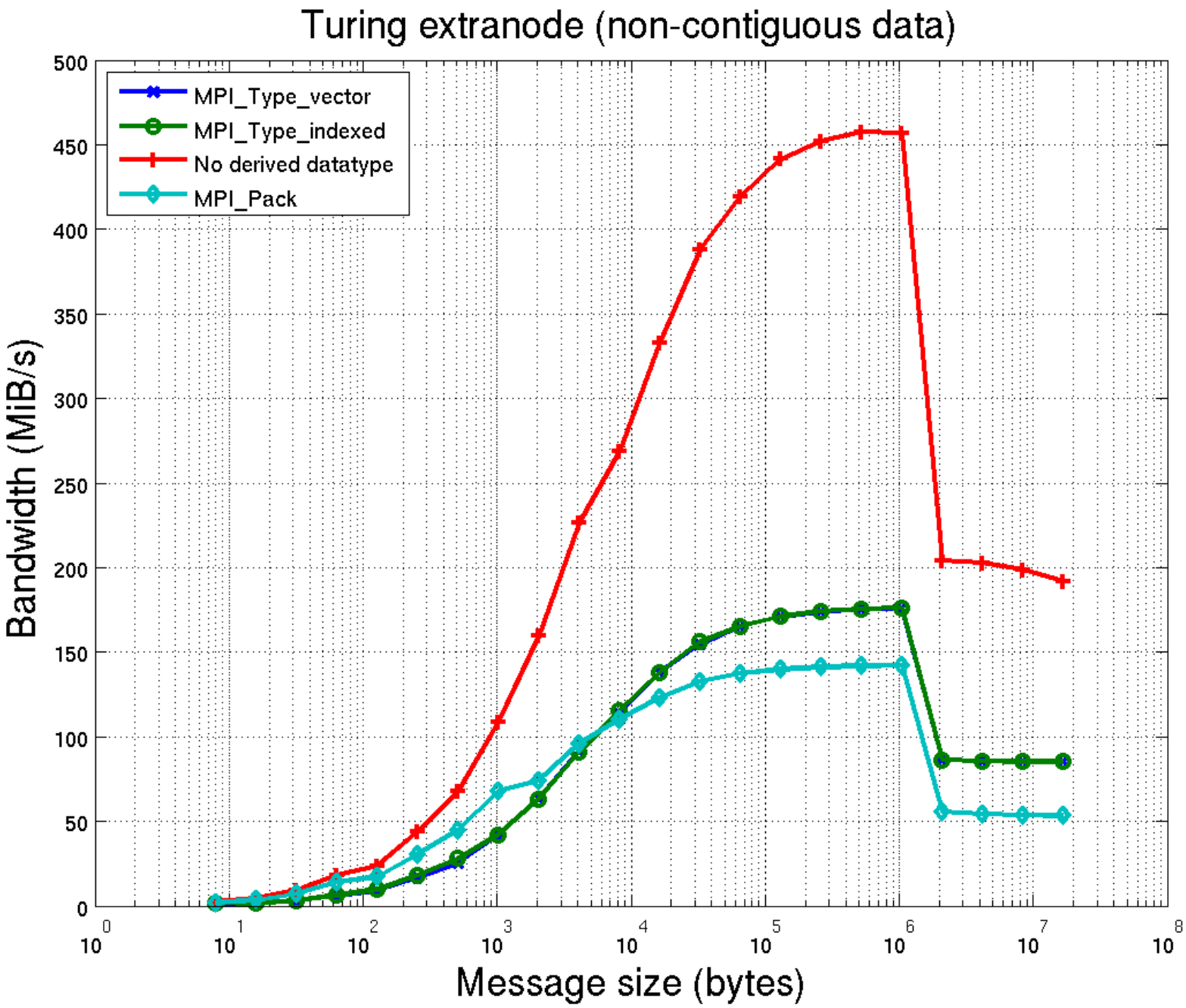
Types dérivés : alternatives

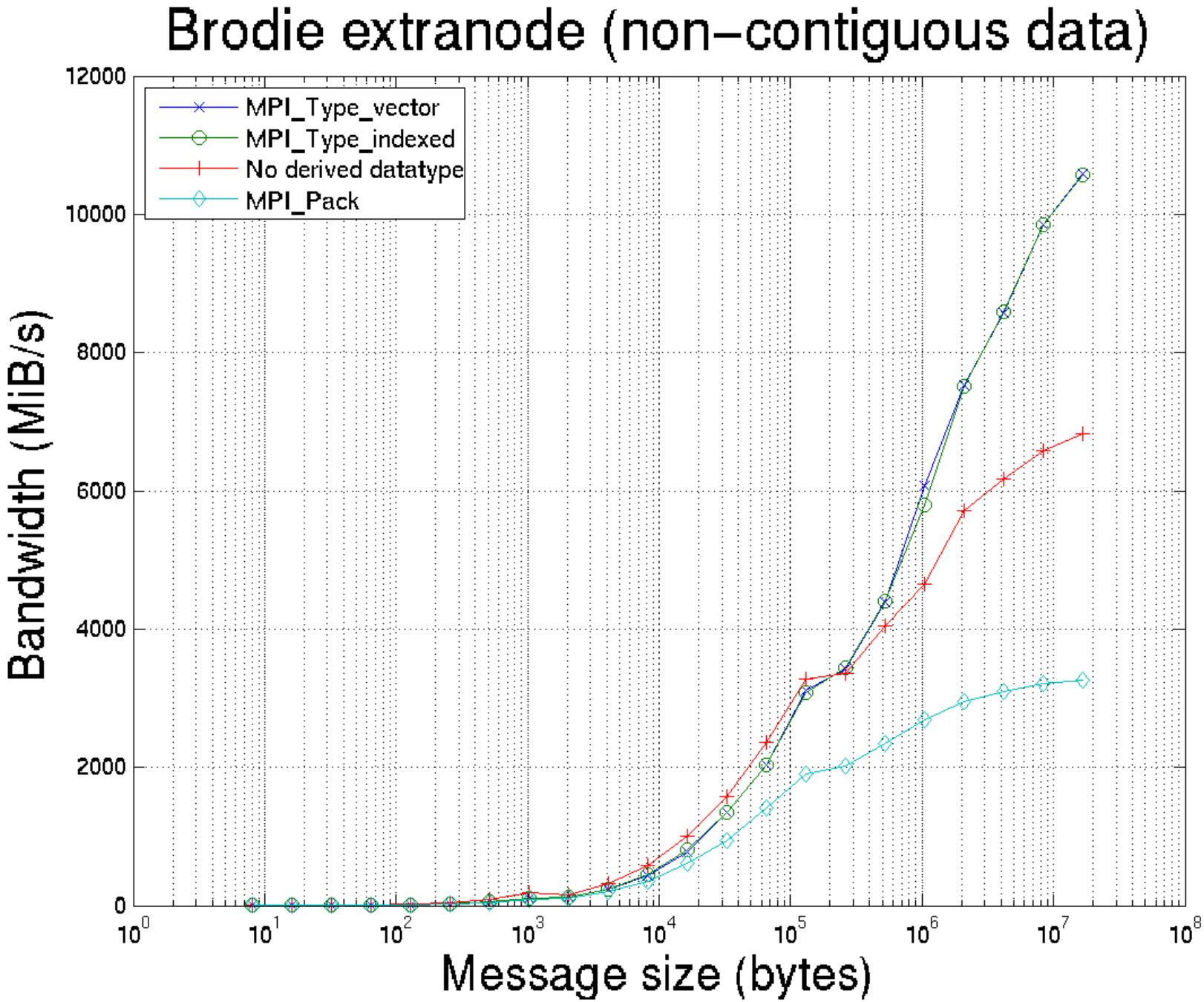
`MPI_Pack/MPI_Unpack`

Utilisation des sous-programmes `MPI_Pack/MPI_Unpack`

- Souvent peu performant
- Mêmes défauts que la gestion manuelle avec copie dans des structures intermédiaires
- Possibilité de recevoir un message en plusieurs fois ou de le préparer en plusieurs fois
- Peut être utilisé avec des types dérivés


Ada extranode (non-contiguous data)
MPI_Type_vector
MPI_Type_indexed
No derived datatype
MPI_Pack
Bandwidth (MiB/s)
Message size (bytes)
0
200
400
600
800
1000
1200
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8


Turing extranode (non-contiguous data)
MPI_Type_vector
MPI_Type_indexed
No derived datatype
MPI_Pack
Bandwidth (MiB/s)
Message size (bytes)
0
50
100
150
200
250
300
350
400
450
500
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8


Brodie extranode (non−contiguous data)
MPI_Type_vector
MPI_Type_indexed
No derived datatype
MPI_Pack
Bandwidth (MiB/s)
Message size (bytes)
0
2000
4000
6000
8000
10000
12000
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8

Equilibrage de charge

Définitions

- Une application parallèle MPI a un équilibre de charge parfait lorsque tous les processus prennent le même temps entre deux synchronisations (explicites ou non).
- Une quantité de travail différente entre les processus entraîne un déséquilibre de charge avec des temps d'attente pour les plus rapides, des désynchronisations et une extensibilité moindre.
- Le processus le plus lent est dimensionnant.

Une autre forme de déséquilibre est le déséquilibre de mémoire entre processus. Cela peut devenir problématique sur les machines ayant peu de mémoire par nœud.

Equilibrage de charge

Quelques causes de déséquilibre

- Mauvais partage des données dès la conception
- Adaptation dynamique de maillage (AMR)
- Processus itératifs avec nombre d'itérations différent selon les variables, mailles...
- Sources externes à l'application : *OS jitter*, ressources non dédiées...

Quelques pistes pour (ré)équilibrer

- Rééquilibrage dynamique en cours d'exécution avec échanges de mailles entre les processus (utilisation de la courbe de remplissage spatial de Hilbert...)
- Approche maître(s)-esclaves
- Utilisation de bibliothèques de partitionnement (PT-SCOTCH, ParMETIS...)
- Plusieurs sous-domaines par processus
- Approche hybride MPI-OpenMP

Placement des processus

Définitions

- Chaque processus MPI a un rang dans le communicateur `MPI_COMM_WORLD` (allant de 0 à Nprocessus-1).
- Chaque processus MPI est placé de façon statique sur un nœud de la machine (pas de migrations entre nœuds en cours d'exécution).
- Sur de nombreuses machines de calcul, chaque processus est également placé de façon statique sur un cœur donné (procédé appellé *binding* ou affinité) (pas de migrations entre cœurs en cours d'exécution).
- Le placement (ou *mapping*) correspond à la relation entre le rang du processus et sa position sur la machine.

Placement des processus

Importance

Plus le nombre de processus est élevé, plus il y a de communications et plus la distance moyenne (le nombre de liens/bus mémoires et réseaux à traverser) entre chaque processus augmente.

- La latence augmente avec la distance.
- La contention du réseau ou de la mémoire augmente si des messages traversent plusieurs liens.

L'impact d'un mauvais placement peut être très élévé. L'idéal est de ne communiquer qu'entre voisins proches.

Placement des processus

Adaptation logiciel

- Utiliser `MPI_Dims_create` et `MPI_Cart_create` et laisser le plus de liberté possible à MPI (ne pas imposer de dimensions et autoriser la renumérotation des processus).
- Connaître son application et ses modes/*patterns* de communications.
- Communiquer avec ses voisins les plus proches.
- Découper ses maillages pour coller au mieux à la machine.

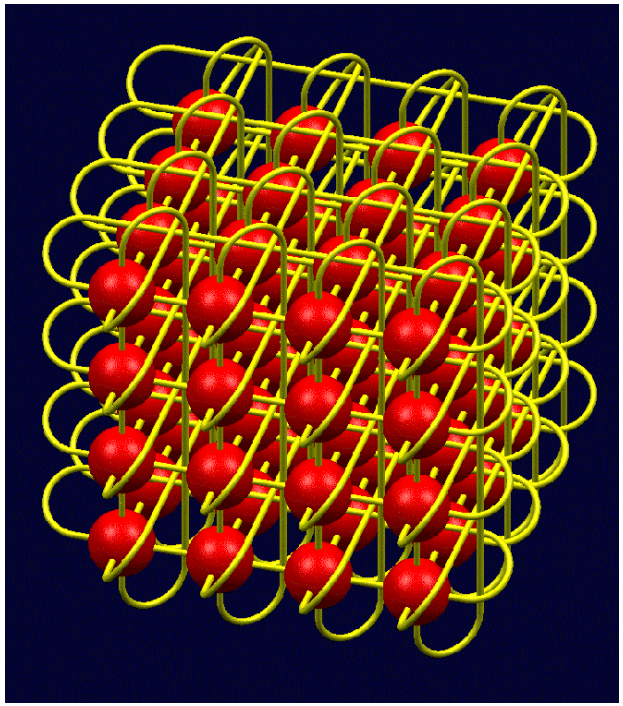
Outils de placement

- *numactl* permet de choisir l'affinité sur de nombreux systèmes Linux.
- La bibliothèque *hwloc* donne accès à de nombreuses informations sur la topologie des nœuds d'une machine et peut contrôler le *binding*.
- Sur Blue Gene/Q, le placement des processus se fait en utilisant l'option *--mapping* de *runjob*.

Exemple : topologie de la Blue Gene/P

Topologie de la Blue Gene/P

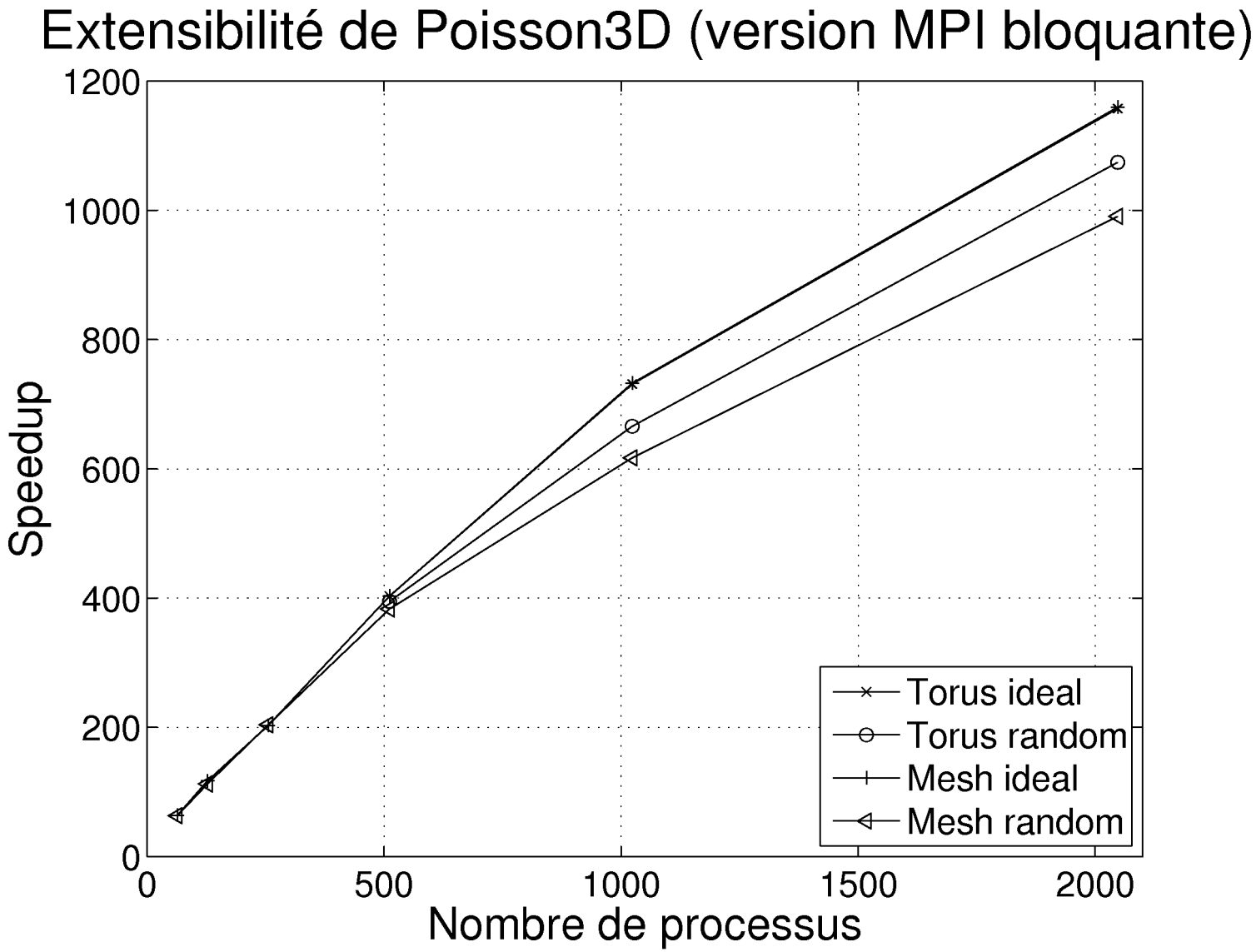
- Pour les communications point-à-point et certaines collectives, la topologie réseau est un tore 3D.
- Chaque nœud de calcul est connecté à ses 6 voisins par des liens réseaux bidirectionnels.
- L'idéal est de ne communiquer qu'avec ses voisins directs.



Topologies cartésiennes

Les topologies cartésiennes 3D et 4D peuvent être placées de façon optimale par MPI sur le tore 3D (4D si en mode DUAL ou VN).

Exemple d'extensibilité sur Blue Gene/P



OpenMP avancé

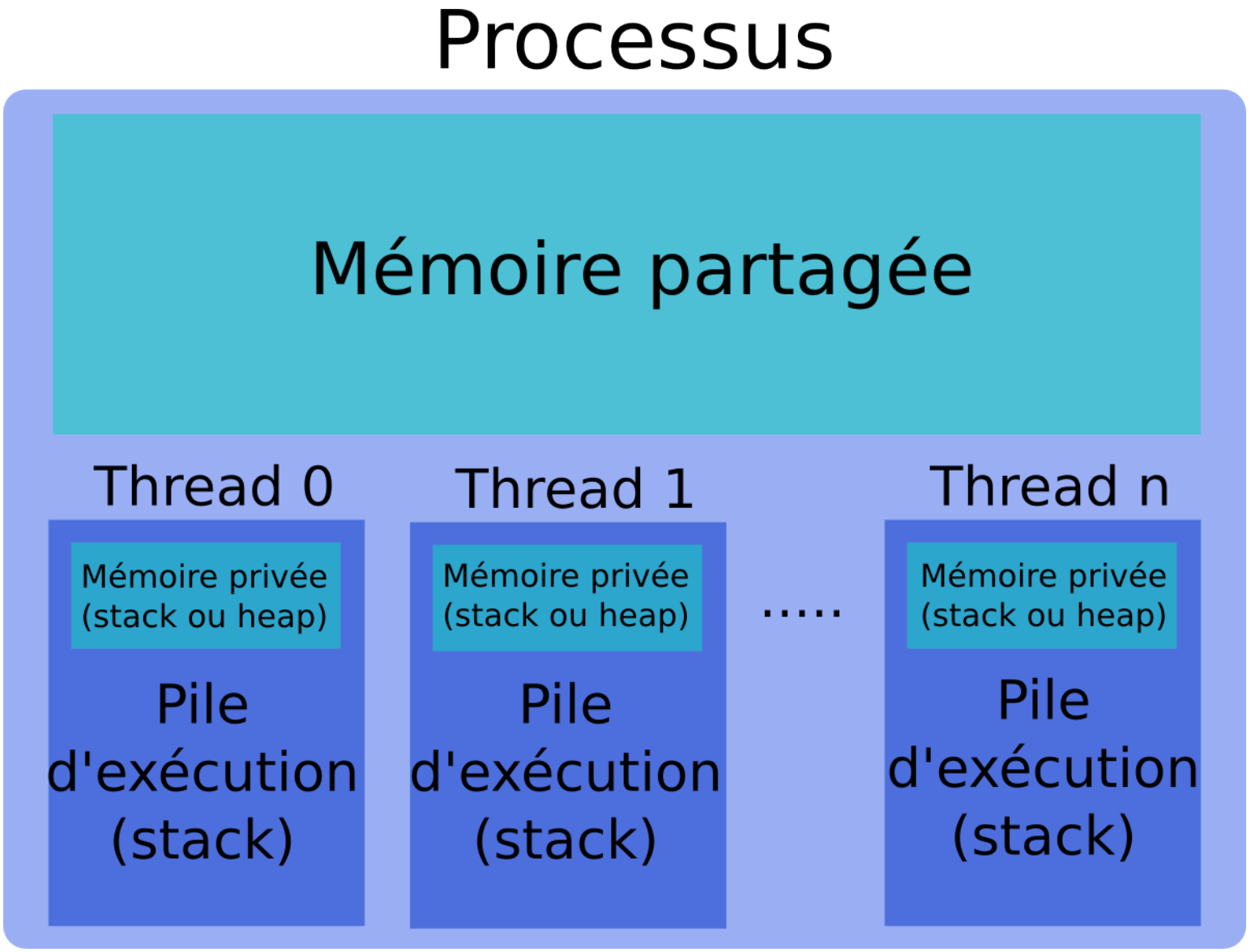
OpenMP en deux mots...

Présentation d'OpenMP

- Paradigme de parallélisation pour architecture à mémoire partagée basé sur des directives à insérer dans le code source (C, C++, Fortran).
- OpenMP est constitué d'un jeu de directives, d'une bibliothèque de fonctions et d'un ensemble de variables d'environnement.
- OpenMP fait partie intégrante de tout compilateur Fortran/C/C++ récent.
- Il permet de gérer :
  - la création des processus légers,
  - le partage du travail entre ces processus légers,
  - les synchronisations (explicites ou implicites) entre tous les processus légers,
  - le statut des variables (privées ou partagées).

OpenMP en deux mots...

Schéma de principe



Limitations d'OpenMP

Limitations d'OpenMP (1)

- L'extensibilité du code parallèle est physiquement limitée par la taille du nœud à mémoire partagée sur lequel il s'exécute.
- En pratique, la bande passante mémoire cumulée à l'intérieur d'un nœud SMP peut encore plus limiter le nombre de cœurs utilisables efficacement. Les contentions dues à la limitation de la bande passante mémoire peuvent souvent être levées en optimisant l'utilisation des caches.
- Attention aux surcoûts implicites d'OpenMP lors de la création des processus légers, de la synchronisation entre les processus légers (implicite ou explicite) ou du partage du travail comme le traitement des boucles parallèles par exemple.
- D'autres surcoûts directement liés à l'architecture de la machine cible peuvent aussi influer sur les performances, comme le *false sharing* sur des caches partagés ou l'aspect NUMA des accès mémoire.
- Le *binding* des processus légers sur les cœurs physiques de la machine sont à la charge du système (accessible au développeur en utilisant certaines bibliothèques). Son impact sur les performances peut être très important.

Limitations d'OpenMP

Limitations d'OpenMP (2)

- OpenMP ne gère pas l'aspect localisation des données, ce qui pose un vrai problème sur architecture fortement NUMA et interdit toute utilisation sur architecture à base de GPGPU.
- OpenMP n'est pas adapté à des problématiques dynamiques (i.e. dont la charge de travail évolue rapidement au cours de l'exécution).
- Comme pour n'importe quelle parallélisation de code, l'accélération finale sera limitée par la partie purement séquentielle du code (loi d'Amdahl).

L'approche classique *Fine-grain (FG)*

Définition

OpenMP grain fin ou *Fine-grain (FG)* : utilisation des directives OpenMP pour partager le travail entre les *threads*, en particulier au niveau des boucles parallèles avec la directive `DO`.

Avantages

- Simplicité de mise en oeuvre ; la parallélisation ne dénature pas le code ; une seule version du code à gérer pour la version séquentielle et parallèle.
- Une approche incrémentale de la parallélisation du code est possible.
- Si on utilise les directives OpenMP de partage du travail (`WORKSHARE`, `DO`, `SECTION`), alors des synchronisations implicites gérées par OpenMP simplifient grandement la programmation (i.e. boucle parallèle avec réduction).
- Sur les boucles parallèles, un équilibrage dynamique de charge peut être réalisé via les options de la clause `SCHEDULE` (`DYNAMIC`, `GUIDED`).

L'approche classique *Fine-grain (FG)*

Inconvénients

- Les surcoûts dus au partage du travail et à la création/gestion des threads peuvent se révéler importants, particulièrement lorsque la granularité du code parallèle est petite/faible.
- Certains algorithmes ou nids de boucles peuvent ne pas être directement parallélisables car ils requièrent des synchronisations plus fines que de simples barrières, exclusions mutuelles ou exécution *single*.
- Dans la pratique, on observe une extensibilité limitée des codes (en tout cas inférieure à celle du même code parallélisé avec MPI), même lorsqu'ils sont bien optimisés. Plus la granularité du code est fine, plus ce phénomème s'accentue.

L'approche *Coarse-grain (CG)*

Définition

OpenMP gros grain SPMD ou *Coarse-grain (CG)* : encapsulation du code dans une seule région parallèle et distribution du travail sur les *threads* à la main.

Avantages

- Pour des codes à petite granularité, les surcoûts de partage du travail sont bien plus faibles qu'avec l'approche *Fine-grain*.
- Très bonne extensibilité lorsque le code s'y prête (comparable et même le plus souvent meilleure que celle obtenue en parallélisant le code avec MPI).
- On verra dans la suite que c'est l'approche qui donne les meilleures performances sur un nœud SMP.

L'approche *Coarse-grain (CG)*

Inconvénients

- Approche très intrusive dans le code. Il n'est plus possible d'avoir une seule et unique version du code à gérer.
- L'approche incrémentale de parallélisation du code n'est plus possible.
- Les synchronisations (globales ou au niveau des *threads*) sont ENTIÈREMENT à la charge du programmeur.
- Le partage du travail et l'équilibrage de charge est aussi de la responsabilité du programmeur.
- Au final, la mise en œuvre se révèle au moins aussi complexe qu'une parallélisation avec MPI.

OpenMP *Coarse-grain* : le partage du travail

Exemple d'une boucle parallèle

Considérons la simple boucle parallèle ci-dessous répétée $nbIter = 10^6$ fois.

```
do iter=1,nbIter
   do i=1,n
      B(i) =  B(i) * a + iter
   enddo
enddo
```

La parallélisation *Fine-grain* de la boucle interne est très simple : la répartition du travail se fait en utilisant la directive OpenMP `DO`. Via la clause `SCHEDULE(RUNTIME)`, le mode de répartition du traitement des blocs d'itérations par les *threads* se fera à l'exécution. Un équilibrage dynamique de charge peut se faire avec les modes `DYNAMIC` ou `GUIDED`, mais attention aux surcoûts potentiels à l'exécution !

```
do iter=1,nbIter
   !$OMP DO SCHEDULE(RUNTIME)
   do i=1,n
      B(i) =  B(i) * a + iter
   enddo
enddo
```

OpenMP *Coarse-grain* : le partage du travail

Exemple d'une boucle parallèle

Parallélisation *Coarse-grain* version 1 — avec équilibrage de charge statique, chaque *thread* traitant des itérations non consécutives :

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
do iter=1,nbIter
   do i=1+myOMPRank,n,nbOMPThreads
      B(i) =  B(i) * a + iter
   enddo
enddo
```

Cela revient en fait à une boucle parallèle dont la répartition est de type `STATIC` avec une taille de bloc égale à 1 (i.e. à une seule itération).

OpenMP *Coarse-grain* : le partage du travail

Exemple d'une boucle parallèle (suite)

Parallélisation *Coarse-grain* version 2 — sans équilibrage de charge, chaque *thread* traitant un bloc d'itérations consécutives :

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
nbnLoc = n/nbOMPThreads
iDeb = 1+myOMPRank*nbnLoc
iFin = iDeb+nbnLoc-1
if (myOMPRank==nbOMPThreads-1) iFin = n
do iter=1,nbIter
   do i=iDeb,iFin
      B(i) =  B(i) * a + iter
   enddo
enddo
```

OpenMP *Coarse-grain* : le partage du travail

Exemple d'une boucle parallèle (suite)

Parallélisation *Coarse-grain* version 3 — avec équilibrage de charge statique, chaque *thread* traitant un bloc d'itérations consécutives :

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
iDeb = 1+(myOMPRank*n)/nbOMPThreads
iFin = ((myOMPRank+1)*n)/nbOMPThreads
do iter=1,nbIter
   do i=iDeb,iFin
      B(i) =  B(i) * a + iter
   enddo
enddo
```

Coarse-grain versus *Fine-grain*

Coarse-grain versus *Fine-grain* : surcoûts dus au partage du travail

	Mono	1 *core*	2 *cores*	4 *cores*	8 *cores*	16 *cores*	24 *cores*	32 *cores*
FG	2.75	2.65	5.44	7.26	14.75	65.97	187.2	458.4
CG V1	2.75	6.04	13.84	17.06	21.53	46.96	57.0	58.7
CG V2	2.75	6.19	3.10	1.54	0.79	0.40	0.29	0.26
CG V3	2.75	6.19	3.09	1.54	0.77	0.39	0.27	0.19

Conclusions

- La boucle parallèle version *Fine-grain* donne des résultats catastrophiques, les surcoûts étant si importants que les temps de restitution explosent avec le nombre de *threads*. Les résultats sont encore pire si on utilise le mode `DYNAMIC` de répartition des itérations sur les *threads* puisque, par défaut, si on ne spécifie pas de taille de bloc, il crée autant de blocs que d'itérations !
- Les deux seules versions qui sont extensibles sont les versions CG V2 et V3, l'impact de l'équilibrage de charge n'apparaissant qu'au delà de 24 *threads*. Les performances, rapportées à la granularité de la boucle parallélisée, sont excellentes (accélération de 14,5 sur 32 *threads*).
- L'approche *Coarse-grain* limite l'impact des surcoûts ce qui permet une extensibilité optimale du code parallélisé.

Approche *Coarse-grain*

Approche *Coarse-grain* : impact au niveau du code source

- Contrairement à l'approche *Fine-grain* qui est très peu intrusive dans le code source (uniquement des ajouts de directives OpenMP), l'approche *Coarse-grain* requiert beaucoup de réécriture de code, la nécéssité d'introduire de nouvelles variables, etc.
- Un exemple classique simple : comment paralléliser une boucle avec une réduction dans une version *Coarse-grain* ? On ne peut plus utiliser la clause REDUCTION de la directive DO. Il va falloir l'émuler à la main...
- Chaque *thread* va calculer une réduction locale dans une variable privée, puis va accumuler les réductions locales dans une même variable partagée, mais un seul *thread* à la fois !
- Pour la version *FG*, une seule directive OpenMP (2 lignes modifiées ou ajoutées) est utilisée. Pour la version *CG*, il faut introduire ou modifier pas moins de 14 lignes de code et utiliser 4 directives ou appels de fonction de la bibliothèque OpenMP ! ! !

Calcul de π

Exemple du calcul de π, version séquentielle

```
program pi
  !                    __
  !  But : calcul de  || par la methode des rectangles (point milieu).
  !
  !                     / 1
  !                    |       4              __
  !                    |   ----------  dx = ||
  !                    |    1 + x**2
  !                   / 0

  implicit none
  integer, parameter :: n=30000000
  real(kind=8)       :: f, x, a, h, Pi_calcule
  integer            :: i

  ! Fonction instruction a integrer
  f(a) = 4.0_8 / ( 1.0_8 + a*a )

  ! Longueur de l'intervalle d'integration.
  h = 1.0_8 / real(n,kind=8)

  ! Calcul de Pi
  Pi_calcule = 0.0_8
  do i = 1, n
    x = h * ( real(i,kind=8) - 0.5_8 )
    Pi_calcule = Pi_calcule + f(x)
  end do
  Pi_calcule = h * Pi_calcule

end program pi
```

Calcul de π

Exemple du calcul de π, version parallèle OpenMP *Fine-grain*

```
program pi
  !                      __
  !  But : calcul de || par la methode des rectangles (point milieu).
  !
  !                     / 1
  !                    |       4              __
  !                    |   ----------  dx = ||
  !                    |    1 + x**2
  !                   / 0

  implicit none
  integer, parameter :: n=30000000
  real(kind=8)       :: f, x, a, h, Pi_calcule
  integer            :: i

  ! Fonction instruction a integrer
  f(a) = 4.0_8 / ( 1.0_8 + a*a )

  ! Longueur de l'intervalle d'integration.
  h = 1.0_8 / real(n,kind=8)

  ! Calcul de Pi
  Pi_calcule = 0.0_8
  !$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:Pi_calcule)
  do i = 1, n
    x = h * ( real(i,kind=8) - 0.5_8 )
    Pi_calcule = Pi_calcule + f(x)
  end do
  !$OMP END PARALLEL DO
  Pi_calcule = h * Pi_calcule
end program pi
```

Calcul de π

Exemple du calcul de π, version parallèle OpenMP *Coarse-grain*

```
program pi
!$ use OMP_LIB
  implicit none
  integer, parameter :: n=30000000
  real(kind=8)       :: f, x, a, h, Pi_calcule, Pi_calcule_loc
  integer            :: i, iDeb, iFin, myOMPRank, nbOMPThreads
  ! Fonction instruction a integrer
  f(a) = 4.0_8 / ( 1.0_8 + a*a )

  ! Initialisation de myOMPRank et nbOMPThreads
  myOMPRank=0
  nbOMPThreads=1
  ! Longueur de l'intervalle d'integration.
  h = 1.0_8 / real(n,kind=8)
  ! Calcul de Pi
  Pi_calcule = 0.0_8
  Pi_calcule_loc = 0.0_8
  !$OMP PARALLEL PRIVATE(x,myOMPRankiDeb,iFin) FIRSTPRIVATE(Pi_calcule_loc)
  !$ myOMPRank   = OMP_GET_THREAD_NUM()
  !$ nbOMPThreads = OMP_GET_NUM_THREADS()
  iDeb = 1+(myOMPRank*n)/nbOMPThreads
  iFin = ((myOMPRank+1)*n)/nbOMPThreads
  do i = iDeb, iFin
    x = h * ( real(i,kind=8) - 0.5_8 )
    Pi_calcule_loc = Pi_calcule_loc + f(x)
  end do
  !$OMP ATOMIC
  Pi_calcule = Pi_calcule + Pi_calcule_loc
  !$OMP END PARALLEL
  Pi_calcule = h * Pi_calcule
end program pi
```

Synchronisations « fines »

Synchronisation « fine » entre deux ou plusieurs *threads*

- Les synchronisations fournies par OpenMP (`BARRIER`, `SINGLE`, `ATOMIC`, etc) ne sont pas adaptées à la parallélisation de type *Coarse-grain* qui requièrent des synchronisations plus bas niveau.
- Hélas, rien n'est disponible à cet effet dans OpenMP. Il va donc revenir au programmeur d'émuler ces fonctionnalités en utilisant les variables partagées et la directive `FLUSH` pour échanger de l'information entre deux ou plusieurs autres *threads*.
- Certains algorithmes ne sont pas parallélisables sans avoir recours à ce type de synchronisation.
- C'est lourd à coder, source d'erreurs et cela dénature le code, mais c'est indispensable...

CG — Synchronisations « fines »

La directive OpenMP `FLUSH`

- Rappel : la directive OpenMP `FLUSH` permet de remettre à jour toute la hiérarchie mémoire d'un *thread* associée aux variables partagées listées en argument. Si il n'y a pas d'argument à la commande `FLUSH`, toutes les variables partagées visibles par le *thread* sont mises à jour (extrêmement coûteux !). Plus précisément, sur le *thread* qui fait l'appel au `FLUSH` :
  - les variables partagées listées en argument et ayant été mises à jour depuis le dernier `FLUSH` voient leur valeur recopiée en mémoire partagée,
  - les variables partagées listées en argument et n'ayant pas été mises à jour depuis le dernier `FLUSH` voient leur ligne de cache invalidée. Ainsi, toute nouvelle référence à cette variable correspondra à une lecture en mémoire partagée de la valeur de la variable.
- Supposons qu'un algorithme composé de plusieurs parties $(T_1, T_2, \ldots, T_n)$ possède les dépendances suivantes :

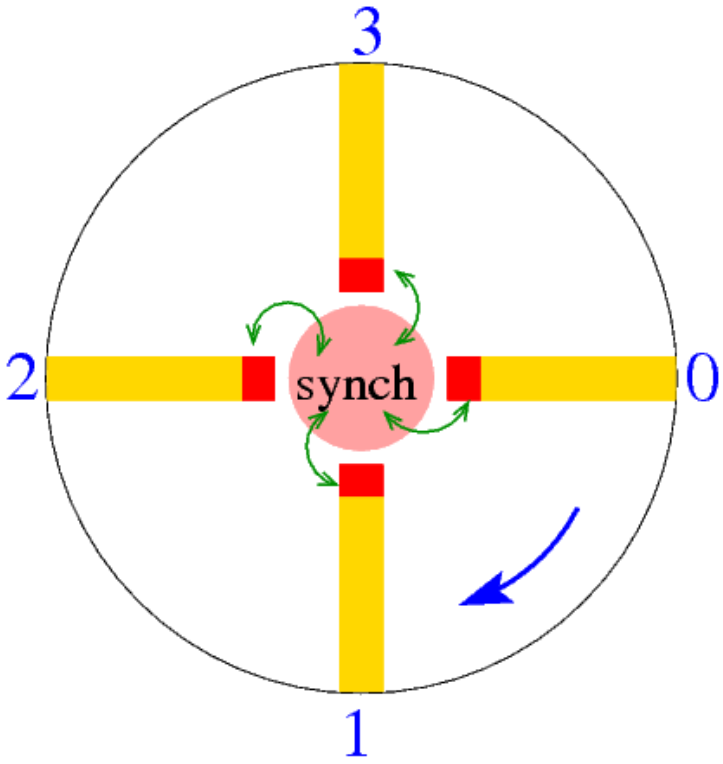
  $\forall i = 1, \ldots, n-1 \quad T_{i+1}$ ne peut commencer à s'exécuter que lorsque T_i est terminé

  La synchronisation fine entre deux *threads* nécessaire pour gérer ces dépendances peut s'implémenter comme dans le code suivant :

Gestion des dépendances

Exemple de dépendances gérées « à la main »

```
program anneau1
 !$ use OMP_LIB
 implicit none
 integer :: rang,nb_taches,synch=0
 !$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
   do
     !$OMP FLUSH(synch)
     if(synch==mod(rang-1+nb_taches,nb_taches)) &
       exit
   end do
   print *,"Rang:",rang,";synch:",synch
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
end program anneau1
```



Gestion des dépendances

Exemple piège facile

```
program anneau2-faux
 !$ use OMP_LIB
 integer :: rang,nb_taches,synch=0,compteur=0
 !$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
   if (rang == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_taches-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   compteur=compteur+1
   print *,"Rang:",rang,";synch:",synch,";compteur:",compteur
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau2-faux
```

Gestion des dépendances

Exemple piège difficile et vicieux

```
program anneau3-faux
 !$ use OMP_LIB
 integer :: rang,nb_taches,synch=0,compteur=0
 !$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM(); nb_taches=OMP_GET_NUM_THREADS()
   if (rang == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_taches-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   print *,"Rang:",rang,";synch:",synch,"
   !$OMP FLUSH(compteur)
   compteur=compteur+1
   !$OMP FLUSH(compteur)
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau3-faux
```

Synchronisations « fines »

Commentaires sur les codes précédents

- Dans anneau2-faux, on n'a pas *flushé* la variable partagée *compteur* avant et après l'avoir incrémentée. Le résultat final peut potentiellement être faux.
- Dans anneau3-faux, le compilateur peut inverser les lignes :

```
compteur=compteur+1
!$OMP FLUSH(compteur)
```

  et les lignes :

```
synch=rang
!$OMP FLUSH(synch)
```

  libérant le *thread* qui suit avant que la variable *compteur* n'ait été incrémentée... Là encore, le résultat final pourrait potentiellement être faux.
- Pour résoudre ce problème, il faut *flusher* les deux variables *compteur* et *synch* juste après l'incrémentation de la variable *compteur*, ainsi on impose un ordre au compilateur.
- Le code corrigé se trouve ci-dessous...

Gestion des dépendances

Le code corrigé...

```
program anneau4
 !$ use OMP_LIB
 integer :: rang,nb_taches,synch=0,compteur=0
 !$OMP PARALLEL PRIVATE(rang,nb_taches)
   rang=OMP_GET_THREAD_NUM()
   nb_taches=OMP_GET_NUM_THREADS()
   if (rang == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_taches-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rang-1) exit
     end do
   end if
   print *,"Rang:",rang,";synch:",synch,"
   !$OMP FLUSH(compteur)
   compteur=compteur+1
   !$OMP FLUSH(compteur,synch)
   synch=rang
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Compteur = ",compteur
end program anneau4
```

Nid de boucles avec double dépendance

Description et analyse du problème

- Considérons le code suivant :

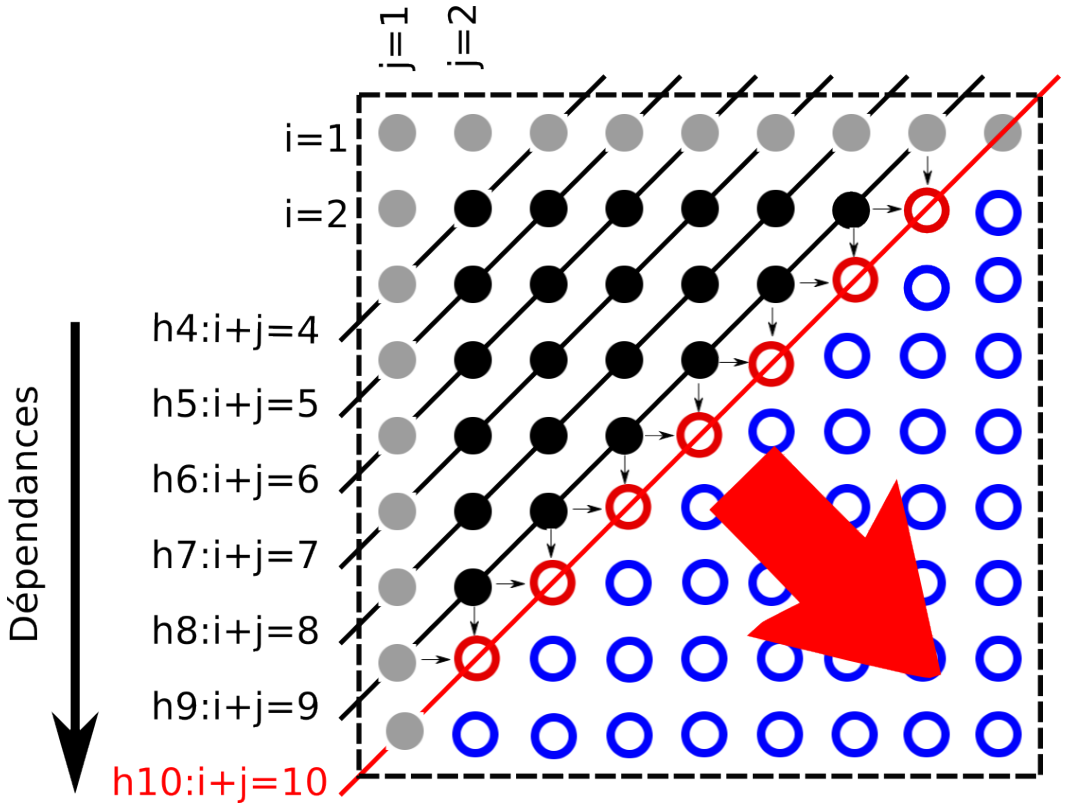
```
! Boucles avec double dependance
do j = 2, ny
   do i = 2, nx
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   end do
end do
```

- C'est un problème classique en parallélisme qui se retrouve par exemple dans les *NAS Parallel Benchmarks* (application LU).
- Du fait de la dépendance arrière en i et en j, ni la boucle en i, ni la boucle en j ne sont parallèles (i.e. chaque itération en i ou j dépend de l'itération précédente).
- Paralléliser avec la directive OpenMP `PARALLEL DO` la boucle en i ou la boucle en j donnerait des résultats faux.
- Pourtant, il est quand même possible d'exhiber du parallélisme de ce nid de boucles en effectuant les calculs dans un ordre qui ne casse pas les dépendances.
- Il existe au moins deux méthodes pour paralléliser ce nid de boucles : l'algorithme de *l'hyperplan* et celui du *software pipelining*.

Algorithme de l'hyperplan

Comment exhiber du parallélisme ?

- Le principe est simple : nous allons travailler sur des hyperplans d'équation : $i + j = cste$ qui correspondent à des diagonales de la matrice.
- Sur un hyperplan donné, les mises à jour des éléments de cet hyperplan sont indépendantes les unes des autres, donc ces opérations peuvent être réalisées en parallèle.
- Par contre, il existe une relation de dépendance entre les hyperplans ; on ne peut pas mettre à jour d'éléments de l'hyperplan H_n tant que la mise à jour de ceux de l'hyperplan H_{n-1} n'est pas terminée.



Algorithme de l'hyperplan (2)

Réécriture du code

- L'algorithme de l'hyperplan nécessite donc une réécriture du code, avec une boucle externe sur les hyperplans (non parallèle à cause des dépendances entre hyperplans) et une boucle parallèle interne sur les éléments appartenant à l'hyperplan qui peuvent être mis à jour dans un ordre quelconque.
- Le code peut se réécrire sous la forme suivante :

```
do h = 1,nbre_hyperplan       ! boucle non //, dépendance entre les hyperplans
   call calcul(INDI,INDJ,h)   ! calcul tab. d'indices i et j des éléments des hyperpla
   do e = 1,nbre_element_hyperplan  ! boucle sur le nombre d'éléments de l'hyperplan
      i = INDI(e)
      j = INDJ(e)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3  ! MAJ de l'élément V(i,j)
   enddo
enddo
```

Quelques remarques complémentaires

- Une fois le code réécrit, la parallélisation est très simple, la boucle interne étant parallèle. Nous n'avons pas eu besoin d'avoir recours à des synchronisations fines pour implémenter l'algorithme de l'hyperplan.
- Les performances obtenues ne sont hélas pas optimales, la cause principale est la médiocre utilisation des caches due à l'accès en diagonale (donc non contigu en mémoire) des éléments de la matrice V.

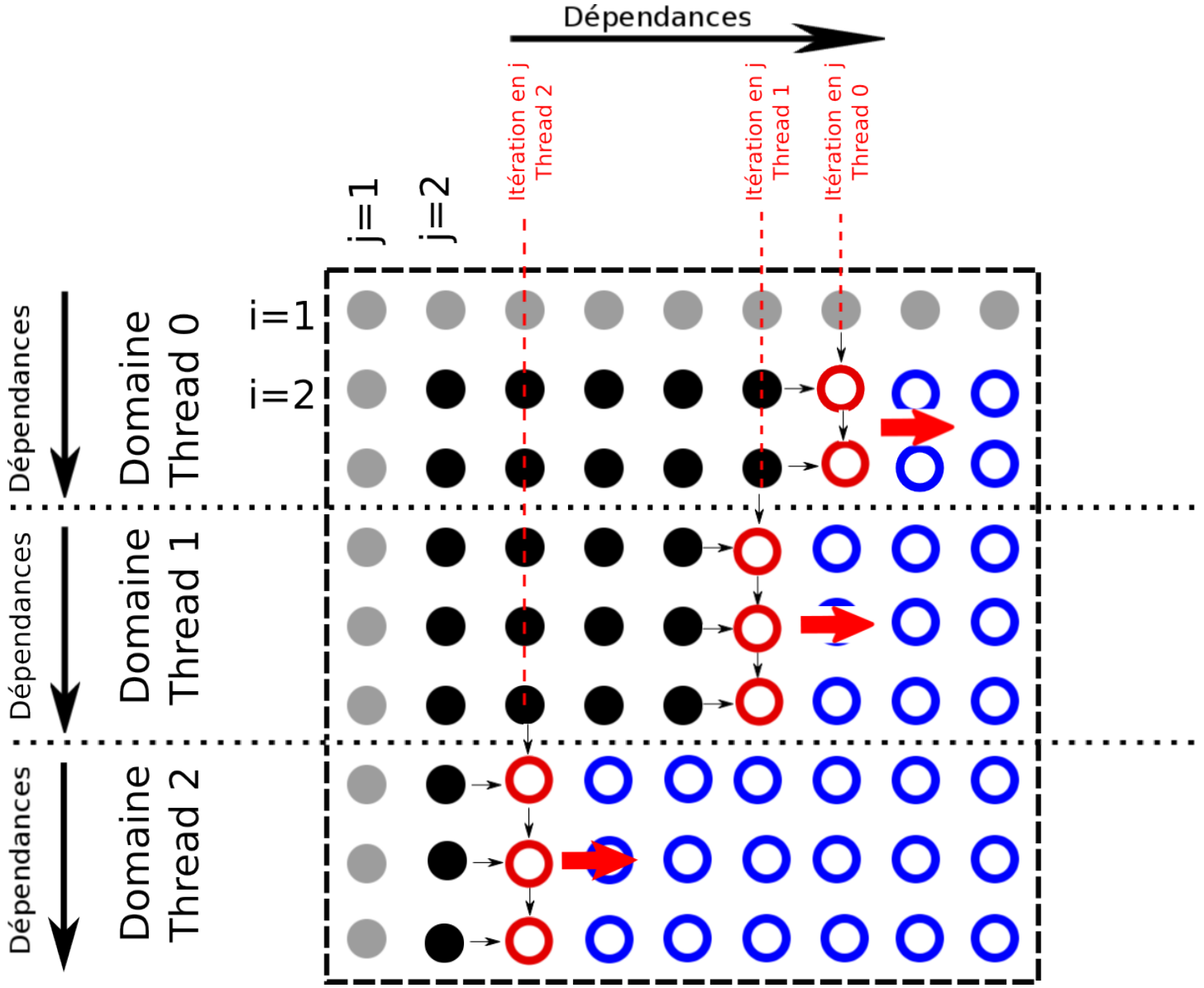
Algorithme *software pipelining*

Comment exhiber du parallélisme ?

- Le principe est simple : nous allons paralléliser par blocs la boucle la plus interne et jouer sur les itérations de la boucle externe pour veiller à ne pas casser de dépendance en synchronisant finement les *threads* entre eux.
- On découpe la matrice en tranches horizontales et on attribue chaque tranche à un *thread*.
- Les dépendances de l'algorithme nous imposent alors que le *thread* 0 doit toujours traiter une itération de la boucle externe j qui doit être supérieure à celle du *thread* 1, qui elle-même doit être supérieure à celle du *thread* 2 et ainsi de suite... Si on ne respecte pas cette condition, on casse des dépendances !
- Concrètement, lorsqu'un *thread* a terminé de traiter la j^{eme} colonne de son domaine, il doit vérifier avant de continuer que le *thread* qui le précède a lui-même terminé de traiter la colonne suivante (i.e. la $j + 1^{eme}$). Si ce n'est pas le cas, il faut le faire attendre jusqu'à ce que cette condition soit remplie.
- Pour implémenter cet algorithme, il faut constamment synchroniser les *threads* deux à deux et ne libérer un *thread* que lorsque la condition énoncée précédemment est réalisée.

Algorithme software pipelining (2)

Les dépendances...



Algorithme software pipelining (3)

Implémentation

- Finalement, l'implémentation de cette méthode peut se faire de la façon suivante :

```
myOMPRank = ...
nbOMPThrds = ...
call calcul_borne(iDeb,iFin)
do j= 2,n
   ! On bloque le thread (sauf le 0) tant que le
   ! précedent n'a pas fini le traitement
   ! de l'itération j+1
   call sync(myOMPRank,j)
   ! Boucle // distribuée sur les threads
   do i = iDeb,iFin
      ! MAJ de l'élément V(i,j)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   enddo
enddo
```

Performances MPI/OpenMP-FG/OpenMP-CG

Caractéristiques du code et de la machine cible

- Les tests de performance ont été faits avec le code d'hydrodynamique HYDRO (celui utilisé dans les TP)
- La machine cible est un nœud mémoire partagée de Vargas (IBM SP6, 32 cœurs par nœud)
- On a comparé trois versions parallélisées d'HYDRO :
  1. Une version parallélisée avec MPI, décomposition de domaine 2D, utilisation de types dérivés, pas de recouvrement calculs-communications,
  2. Une version parallélisée avec OpenMP de type *Fine-grain*, *scheduling* de type `STATIC`,
  3. Une version parallélisée avec OpenMP de type *Coarse-grain*, décomposition de domaine 2D, synchronisation *thread* à *thread* pour gérer les dépendances.

Performances MPI/OpenMP-FG/OpenMP-CG

Caractéristiques des jeux de données utilisés

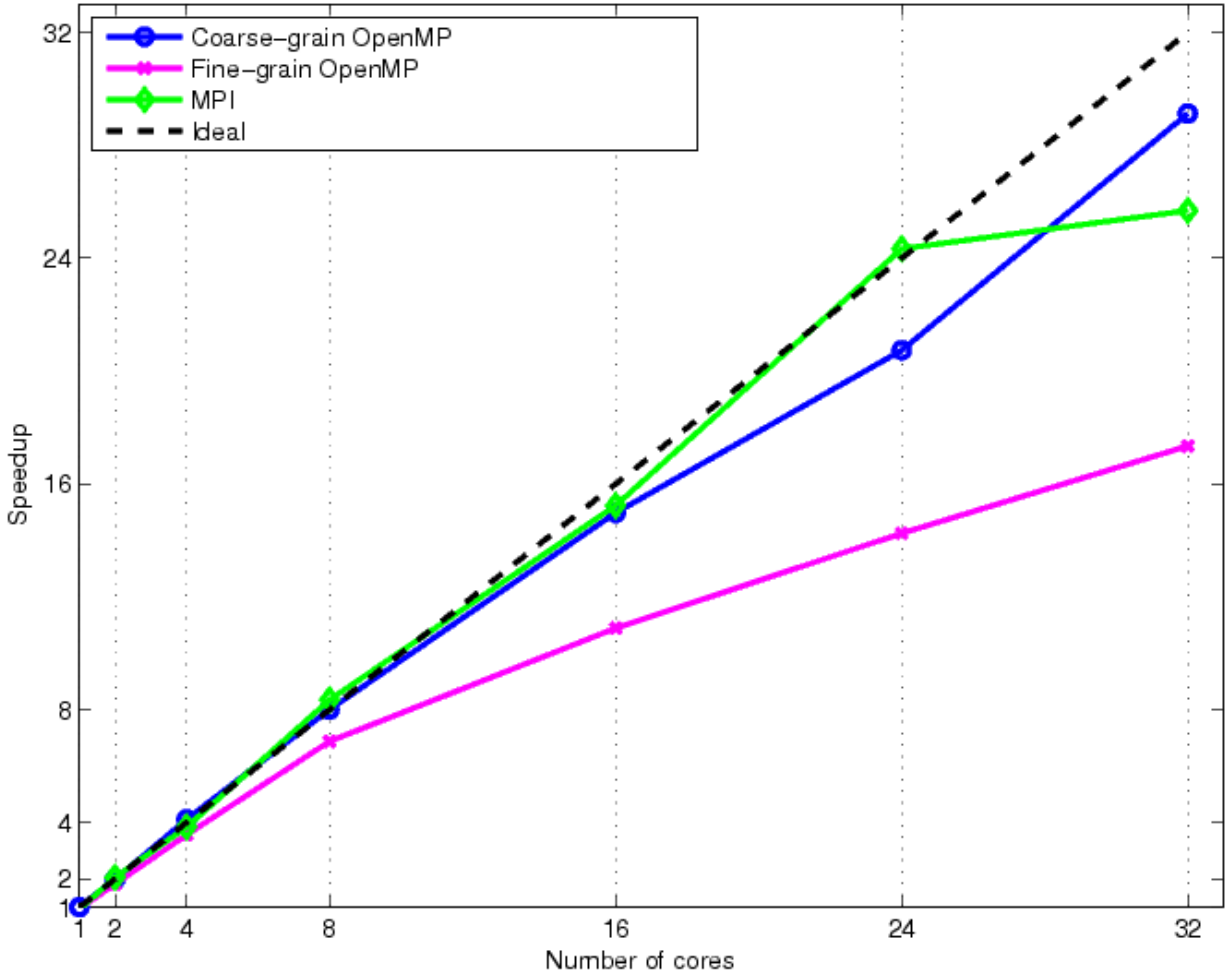
On a utilisé trois jeux de tests, de même taille totale (i.e. même nombre total de points du domaine), mais de répartitions différentes dans les deux directions x et y :

1. Domaine allongé suivant la direction y : $nx = 1000$ et $ny = 100000$,
2. Domaine carré : $nx = ny = 10000$,
3. Domaine allongé suivant la direction x : $nx = 100000$ et $ny = 1000$.

Performances MPI/OpenMP-FG/OpenMP-CG

Résultats pour le domaine $nx = 100000, ny = 1000$

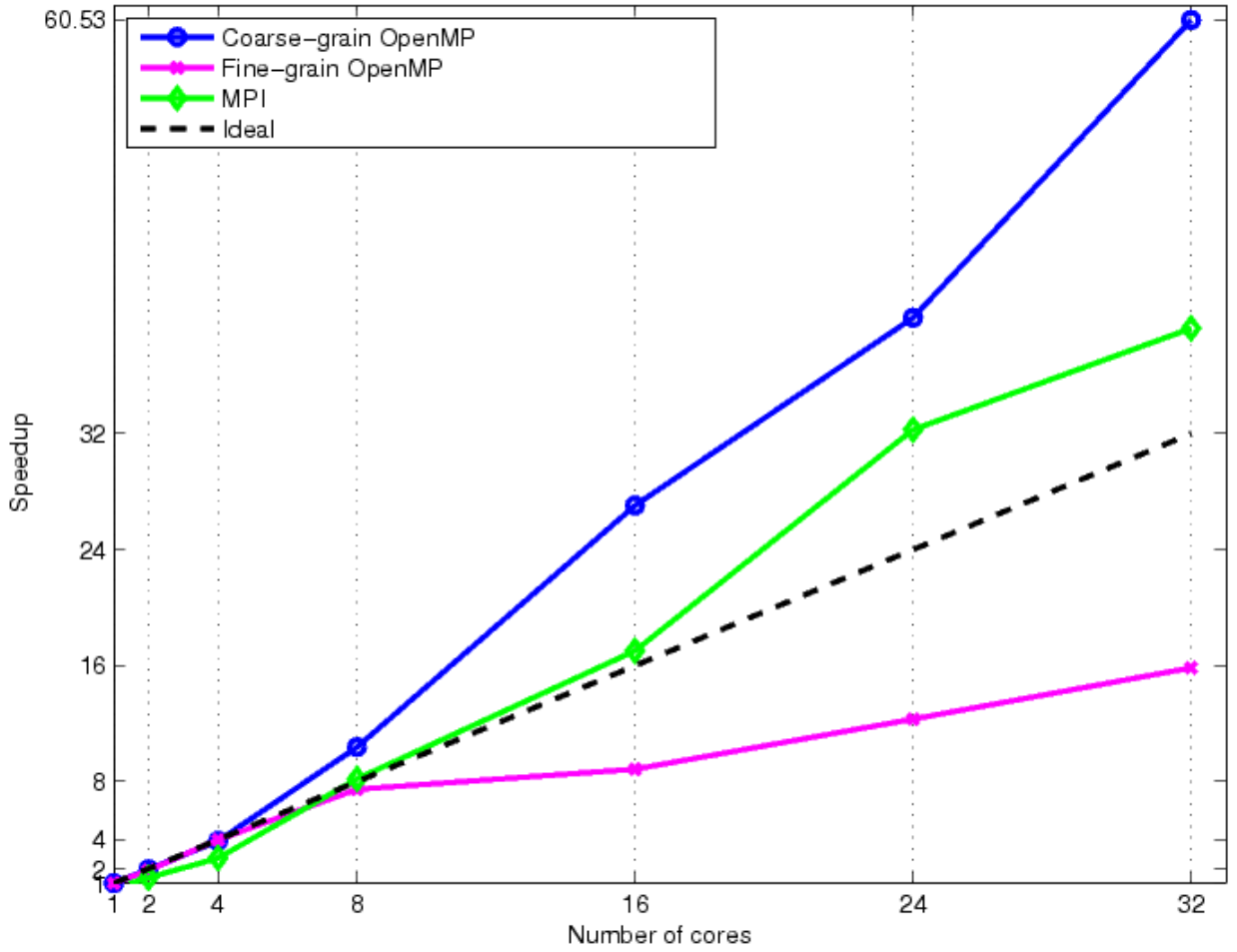
Tps (s)	Mono	1 *core*	2 *cores*	4 *cores*	8 *cores*	16 *cores*	24 *cores*	32 *co.*
mpi2D	361.4	383.8	170.9	91.4	42.0	23.0	14.4	13.6
ompfg	361.4	371.6	193.2	98.4	51.0	32.2	24.6	20.2
ompcg	361.4	350.4	177.3	85.3	43.8	23.4	16.9	12.0



Performances MPI/OpenMP-FG/OpenMP-CG

Résultats pour le domaine $nx = 1000, ny = 100000$

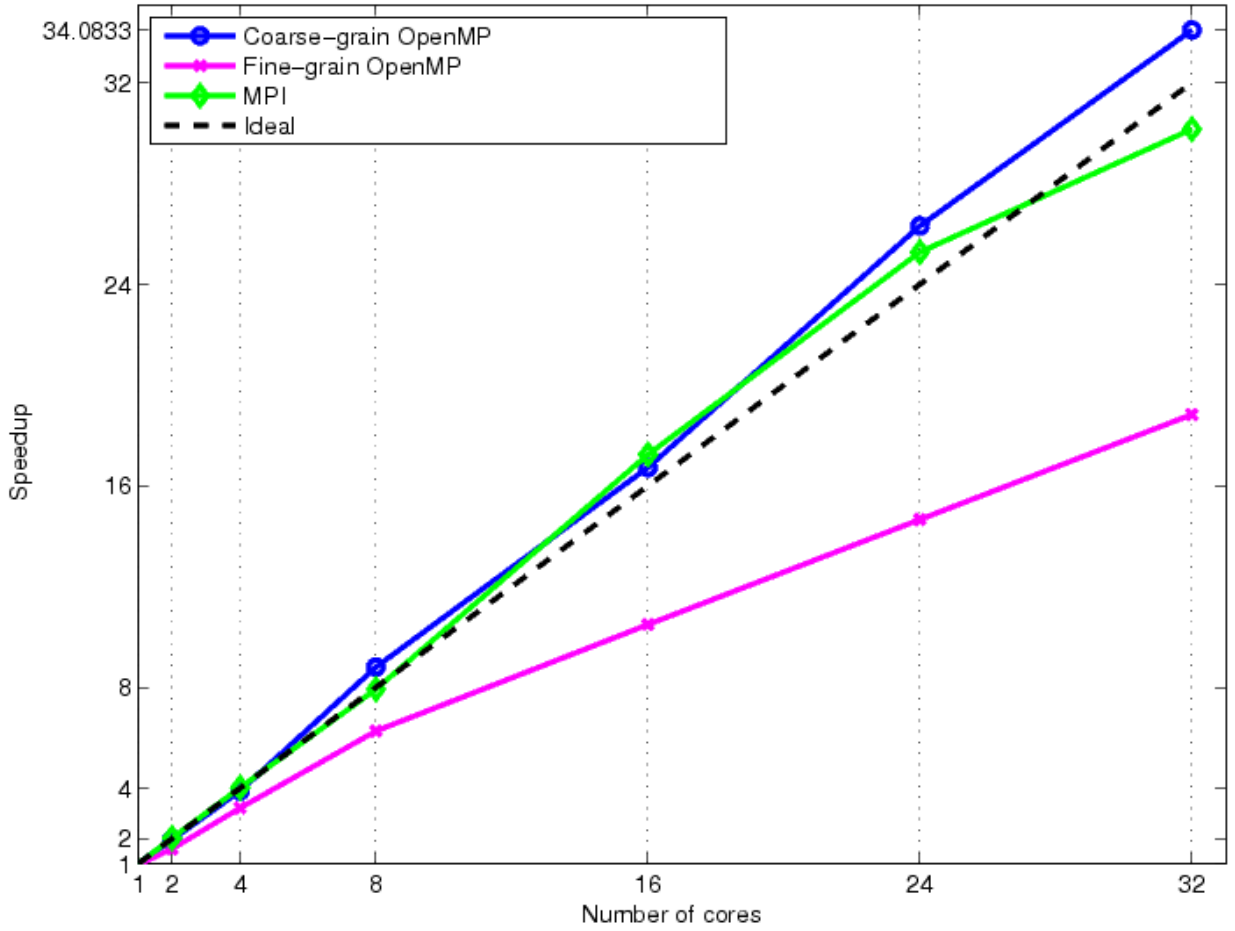
Tps (s)	Mono	1 *core*	2 *cores*	4 *cores*	8 *cores*	16 *cores*	24 *cores*	32 *co.*
mpi2D	1347.2	1310.6	638.3	318.7	106.1	51.0	26.9	22.1
ompfg	1347.2	879.0	461.0	217.9	116.2	98.0	70.5	54.8
ompcg	1347.2	868.0	444.1	222.7	83.7	32.1	21.7	14.3



Performances MPI/OpenMP-FG/OpenMP-CG

Résultats pour le domaine $nx = 10000, ny = 10000$

Tps (s)	Mono	1 *core*	2 *cores*	4 *cores*	8 *cores*	16 *cores*	24 *cores*	32 *co.*
mpi2D	449.9	456.0	223.7	112.2	56.8	26.1	17.8	14.9
ompfg	449.9	471.9	283.4	140.5	71.8	42.9	30.7	23.9
ompcg	449.9	455.7	230.8	115.9	51.1	26.9	17.1	13.2



Analyse des résultats (1)

Version OpenMP *Fine-grain*

- Bien qu'on travaille sur des domaines de taille identique, les performances obtenues sur un seul cœur varient du simple au triple. La raison est à chercher du côté de la bonne utilisation ou non des caches.
- Quel que soit le jeu de test et jusqu'à 4 cœurs, les trois versions donnent des performances plus ou moins comparables. Au-delà de 4 cœurs, la version OpenMP FG souffre d'un problème d'extensibilité dégradée comparée aux versions MPI et OpenMP CG.
- En conséquence, hormis pour un nombre limité de cœurs, la version OpenMP FG est toujours largement dominée par les versions MPI ou OpenMP CG, même si elle est extensible de façon régulière mais limitée jusqu'à 32 cœurs.

Analyse des résultats (2)

Versions MPI et OpenMP *Coarse-grain*

- Jusqu'à 24 cœurs, les versions MPI et OpenMP CG ont des extensibilités comparables et parfaites, parfois même super-linéaires (i.e. meilleur réutilisation des caches au fur et à mesure que la taille des sous-domaines locaux diminue).
- Au-delà de 24 cœurs, la version MPI semble marquer le pas, alors que la version OpenMP CG continue à être parfaitement extensible.
- Sur 32 cœurs, c'est toujours la version OpenMP CG qui donne les meilleurs résultats.
- Attention cependant, la version MPI peut encore être optimisée en implémentant par exemple le recouvrement calculs-communications, ce qui lui permettrait peut-être d'être extensible au-delà de 24 cœurs...

Conclusions

- La généralisation des machines à mémoire partagée ainsi que l'augmentation du nombre de cœurs disponibles au sein d'un nœud demande à ce que l'on reconsidère la façon dont on programme les applications.
- Au sein d'un nœud, si on recherche la performance maximale, c'est OpenMP version *Coarse-grain* qu'il faut utiliser. Cela nécessite un gros investissement et c'est au moins aussi compliqué à implémenter qu'une version MPI. Le débogage est particulièrement complexe. Cette approche est à réserver aux spécialistes qui maîtrisent parfaitement le parallélisme et ses pièges.
- La simplicité d'utilisation et la rapidité d'implémentation d'une version OpenMP *Fine-grain* sont ses principaux avantages. À condition de bien coder (minimisation des barrières de synchronisation et des régions parallèles), les performances selon le type d'algorithme (en particulier selon la granularité) peuvent aller de moyen à relativement bon. Le débogage reste là encore particulièrement complexe. Cette approche s'adresse à tous.
- MPI obtient de bonnes performances sur un nœud mémoire partagée, mais OpenMP version CG le surclasse en terme d'extensibilité. Il peut cependant être encore optimisé en implémentant le recouvrement calculs-communications. Il reste de toute façon indispensable dès qu'il faut aller au-delà de l'utilisation d'un simple nœud...

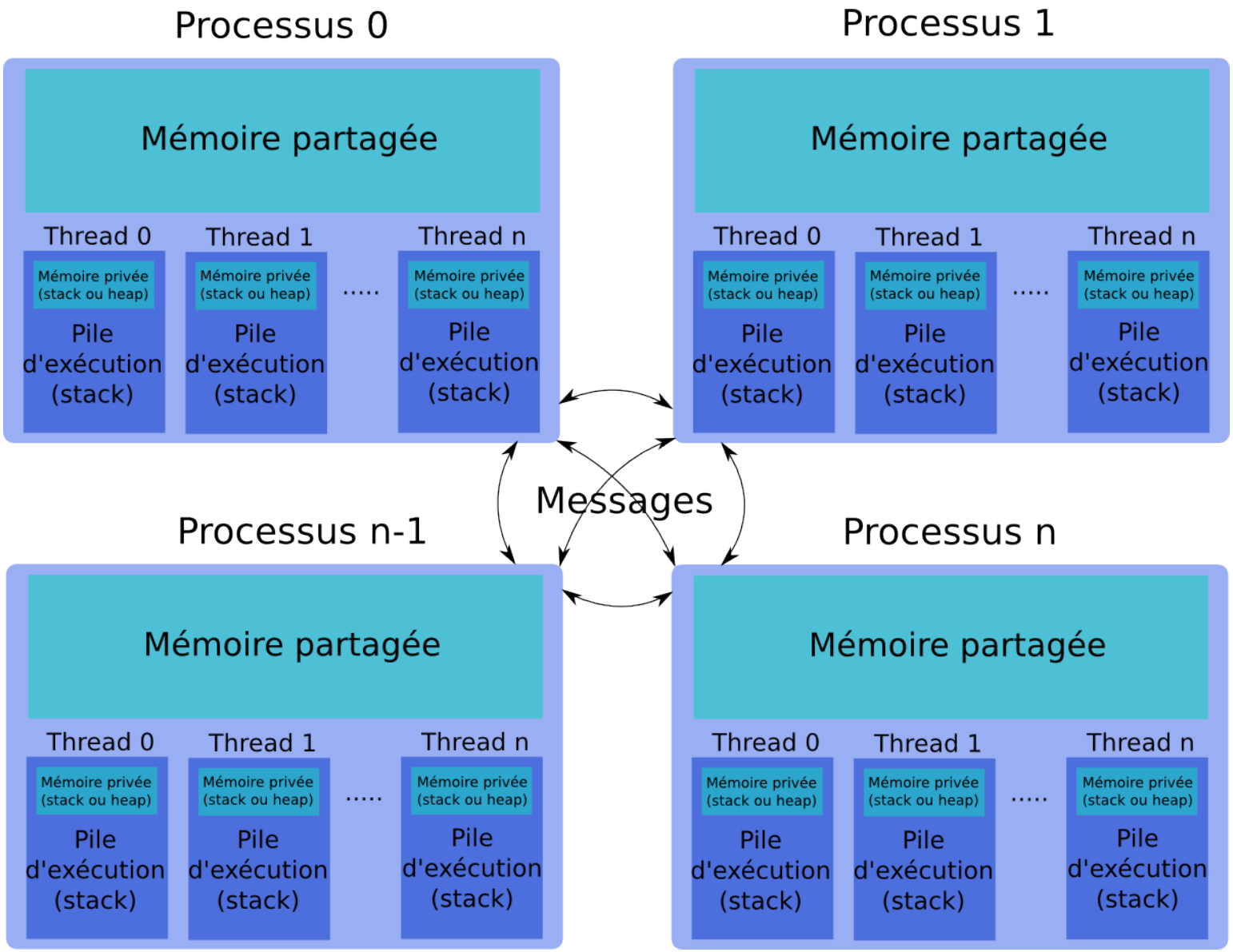
Programmation hybride

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



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

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

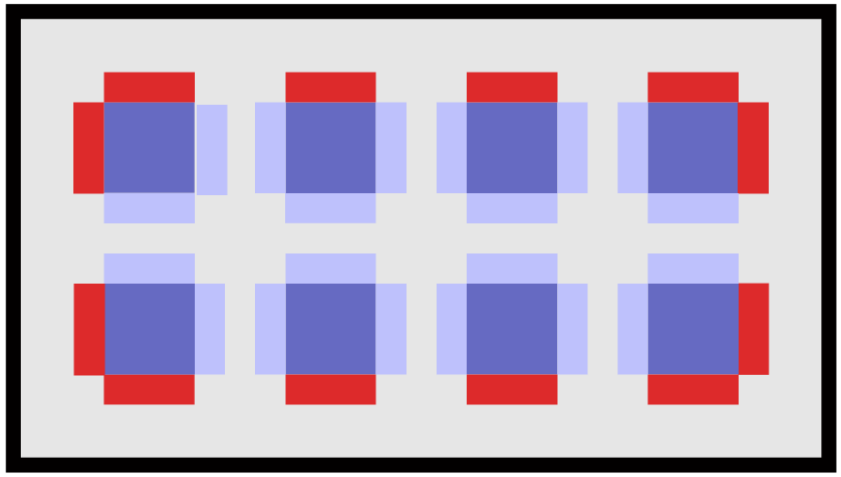
Pourquoi un gain mémoire ?

- 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 !

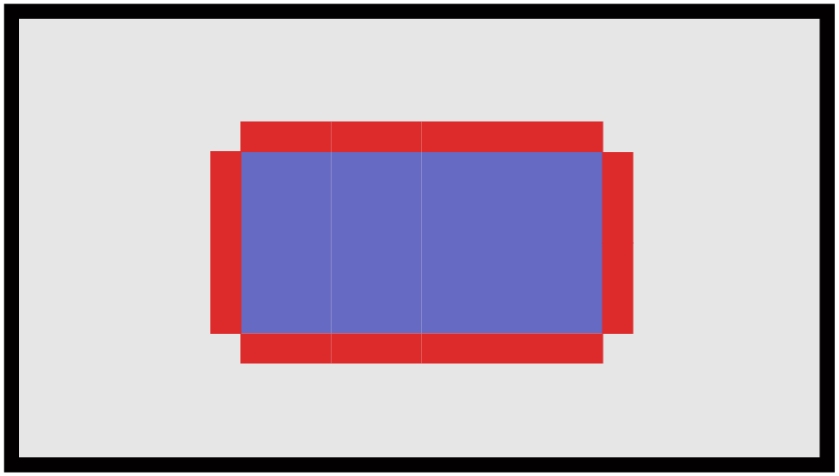
Programmation hybride, l'aspect gain mémoire

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

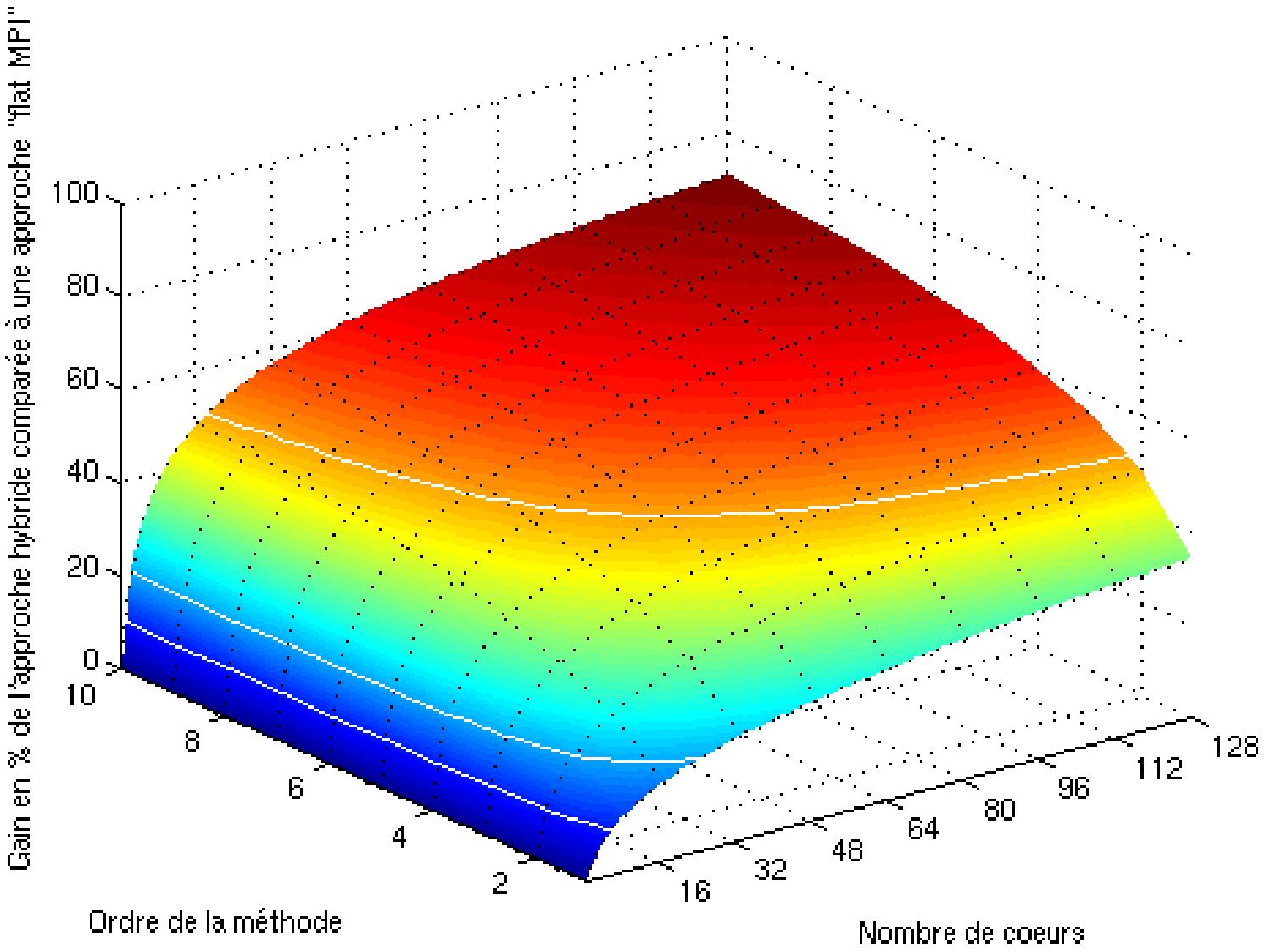
Programmation hybride, l'aspect gain mémoire

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 8 Go/s bidirectionnel sur Vargas 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

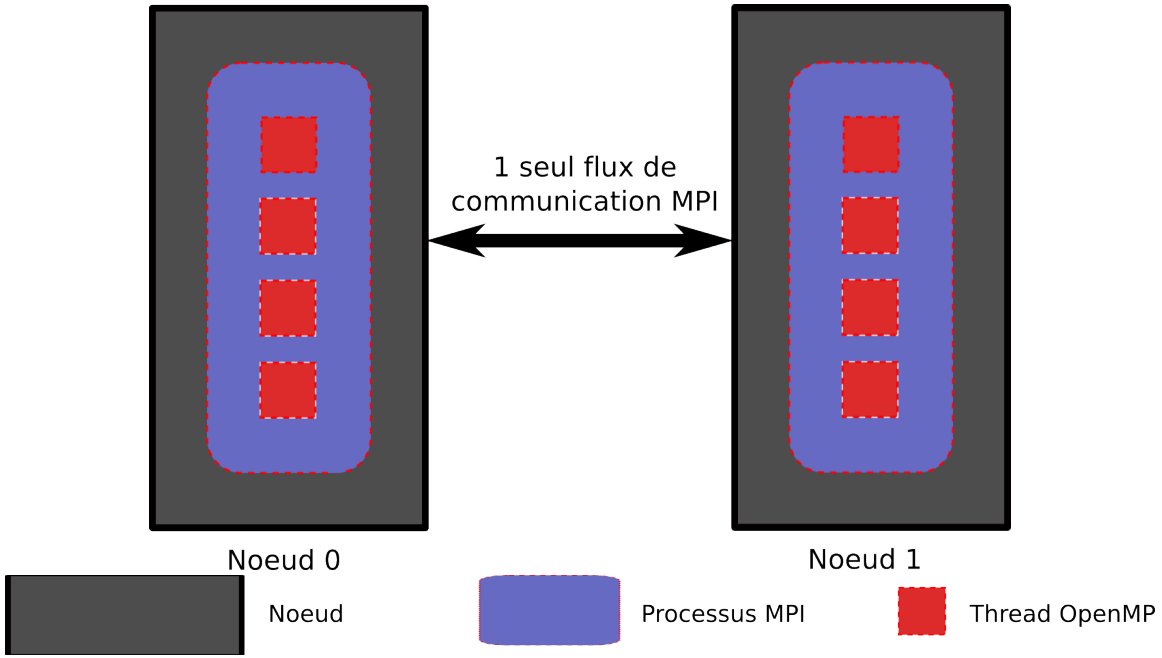
SBPR version `MPI_THREAD_FUNNELED`

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.

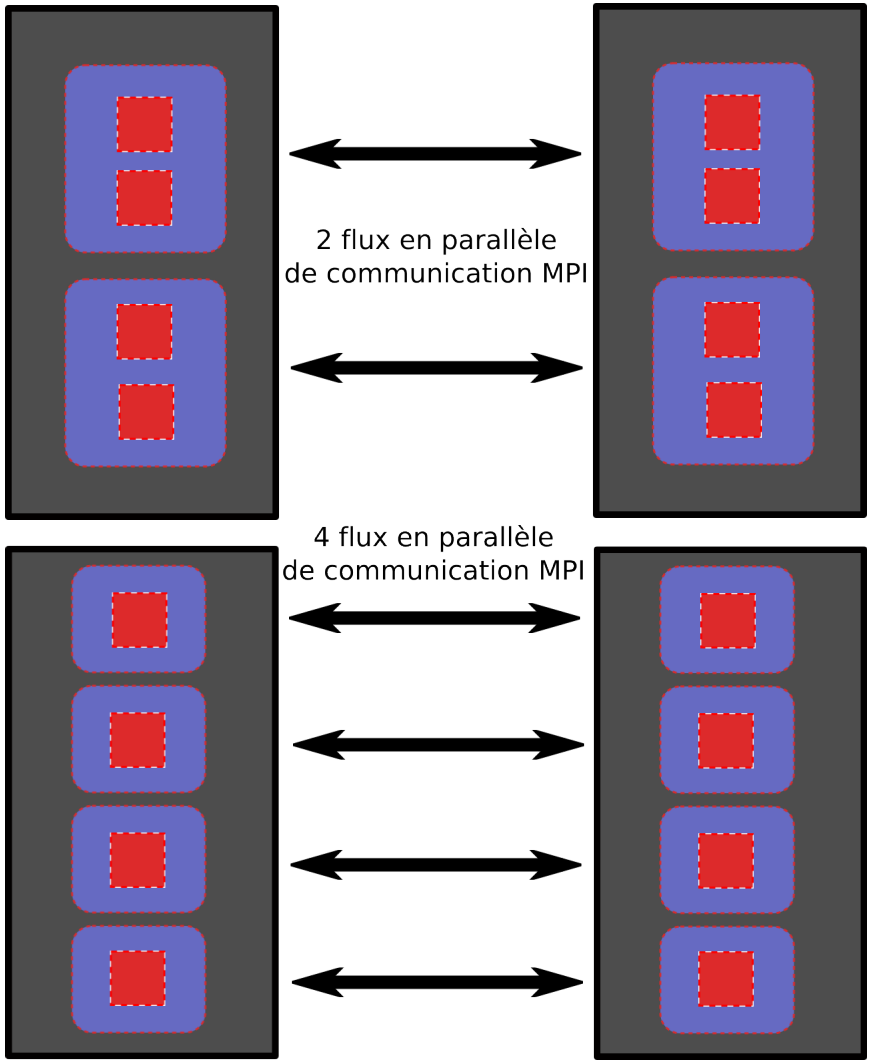
Utilisation optimale du réseau d'interconnexion

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)



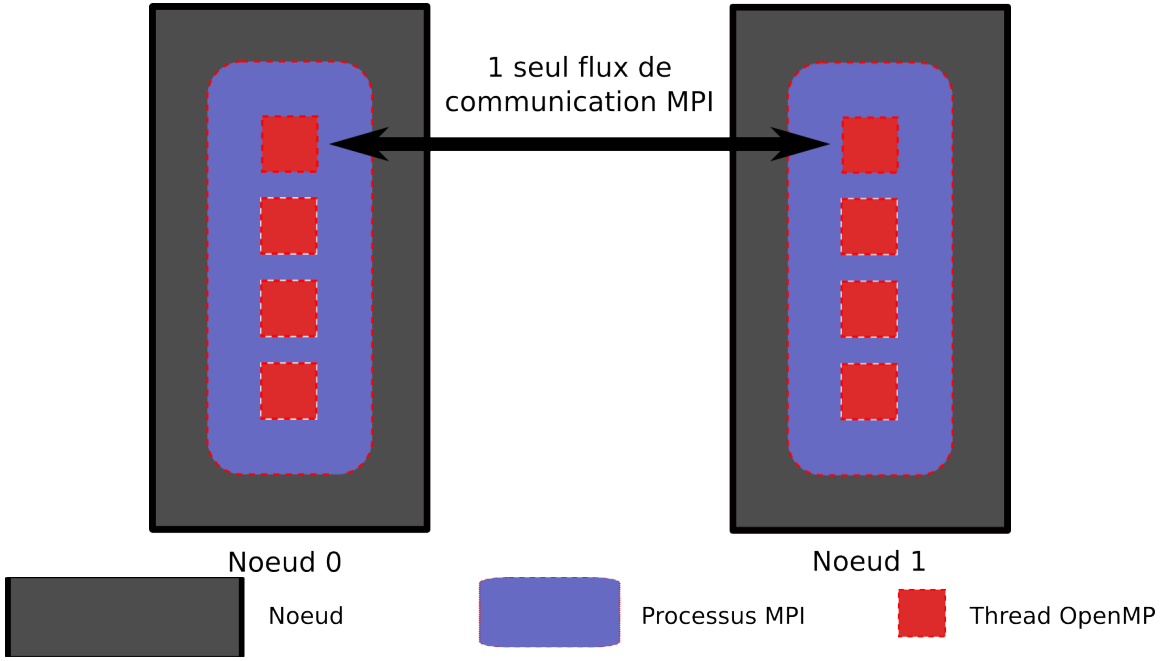
Utilisation optimale du réseau d'interconnexion

SBPR version `MPI_THREAD_MULTIPLE`

Approche `MPI_THREAD_MULTIPLE` :

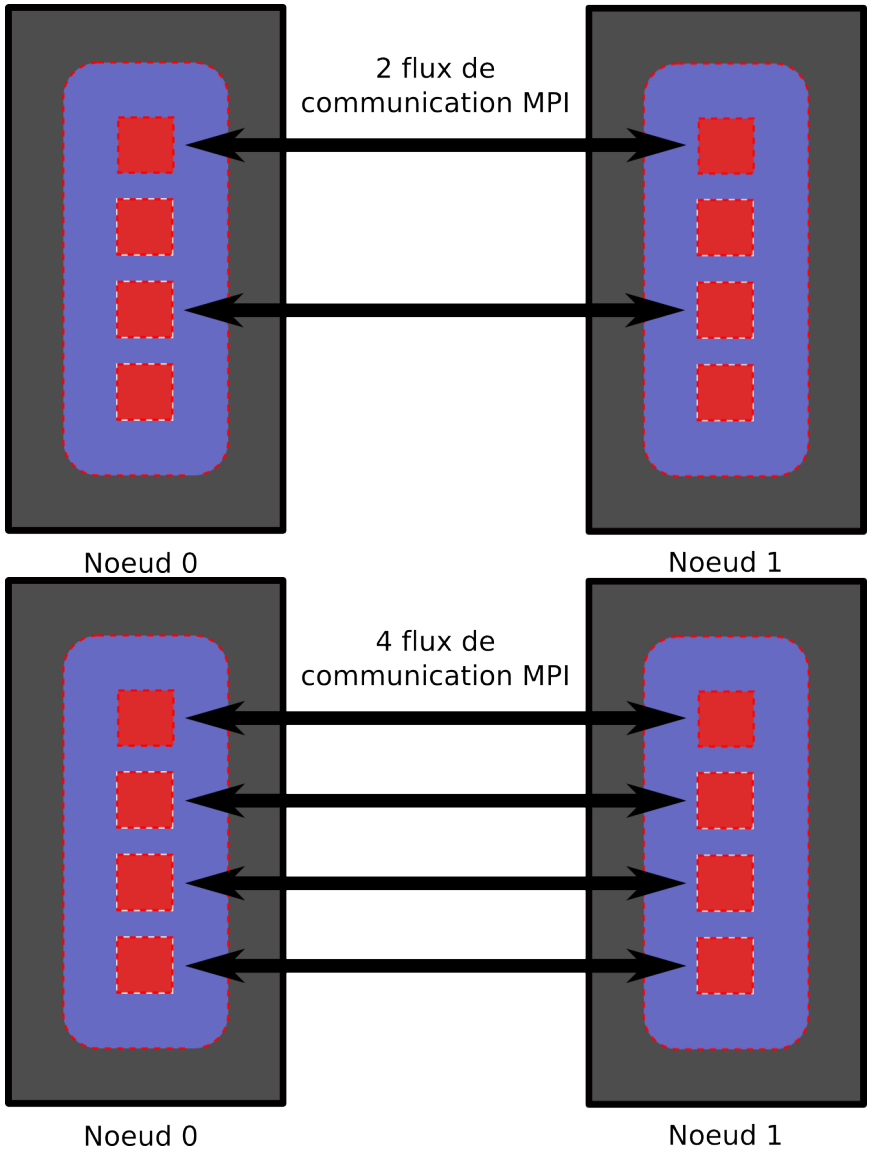
- On augmente la bande passante réseau réllement utilisée en augmentant le nombre de threads OpenMP qui participent aux communications.
- On a un unique processus MPI par nœud, on cherche le nombre minimum de threads de communication nécessaire pour saturer le réseau.

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



Utilisation optimale du réseau d'interconnexion

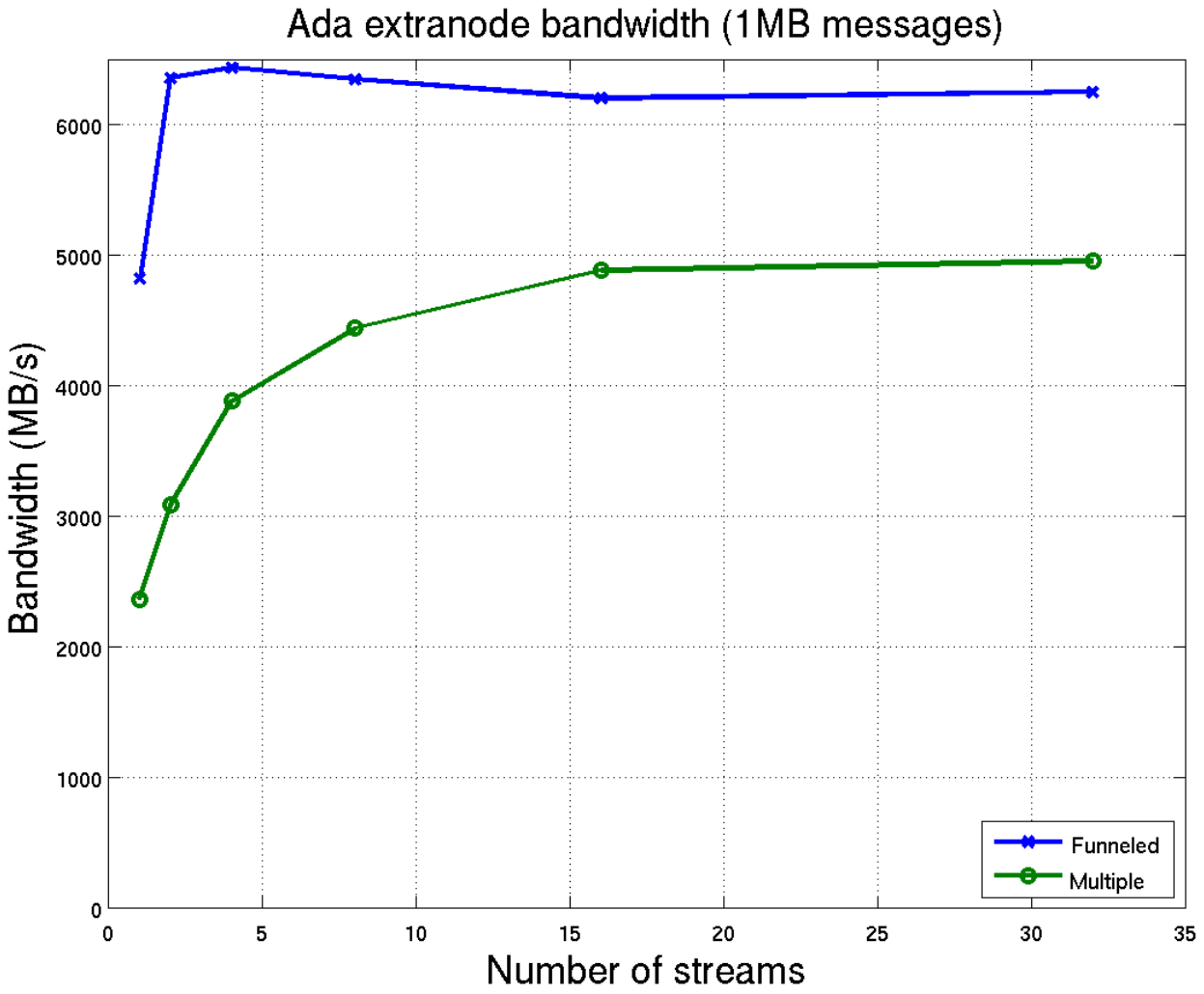
SBPR `MPI_THREAD_MULTIPLE` : exemple sur un nœud SMP à 4 cœurs (BG/P)



Utilisation optimale du réseau d'interconnexion

SBPR — Résultats sur Ada

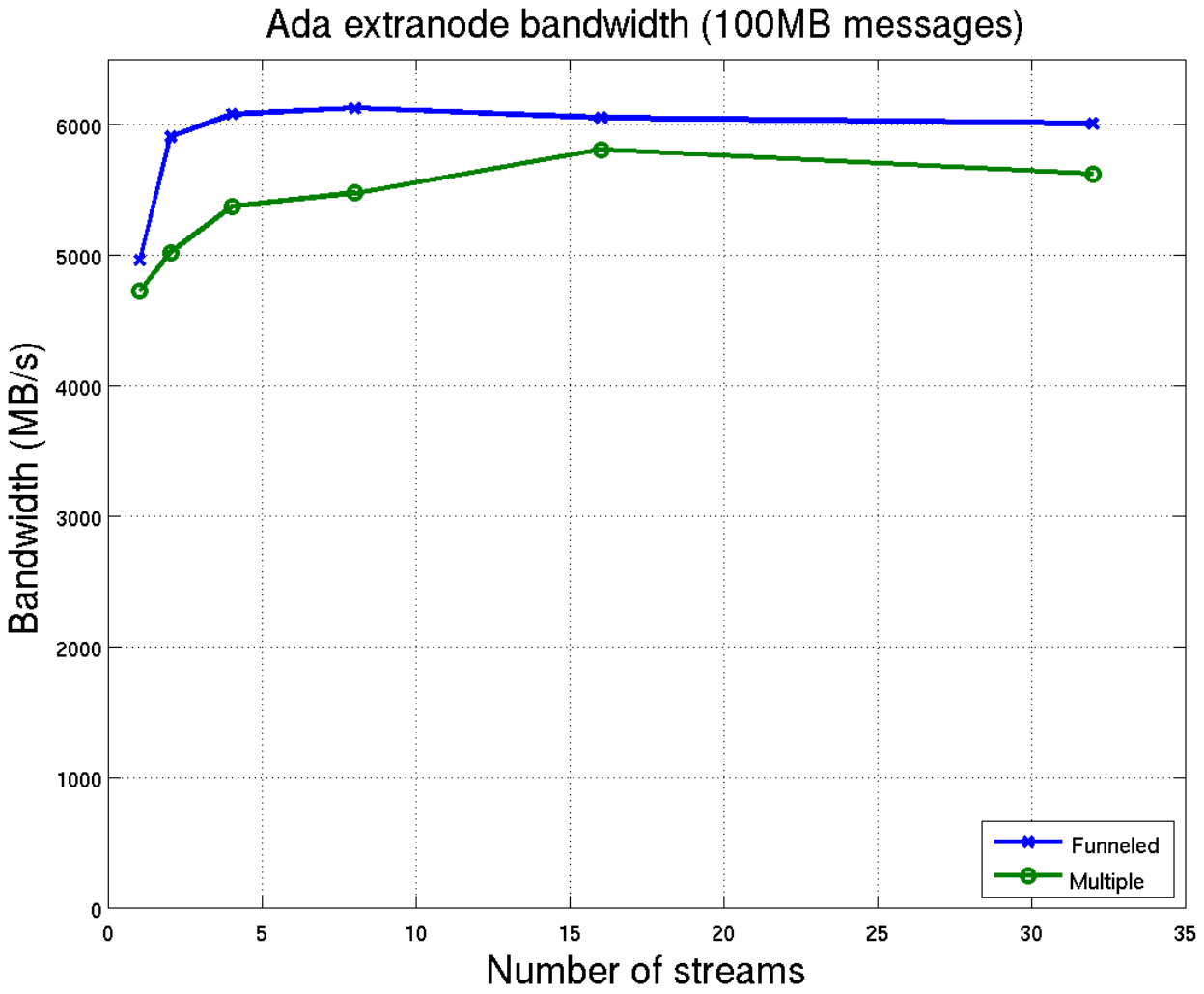
2 liens en // Infiniband FDR10, débit crête 10 Go/s.



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

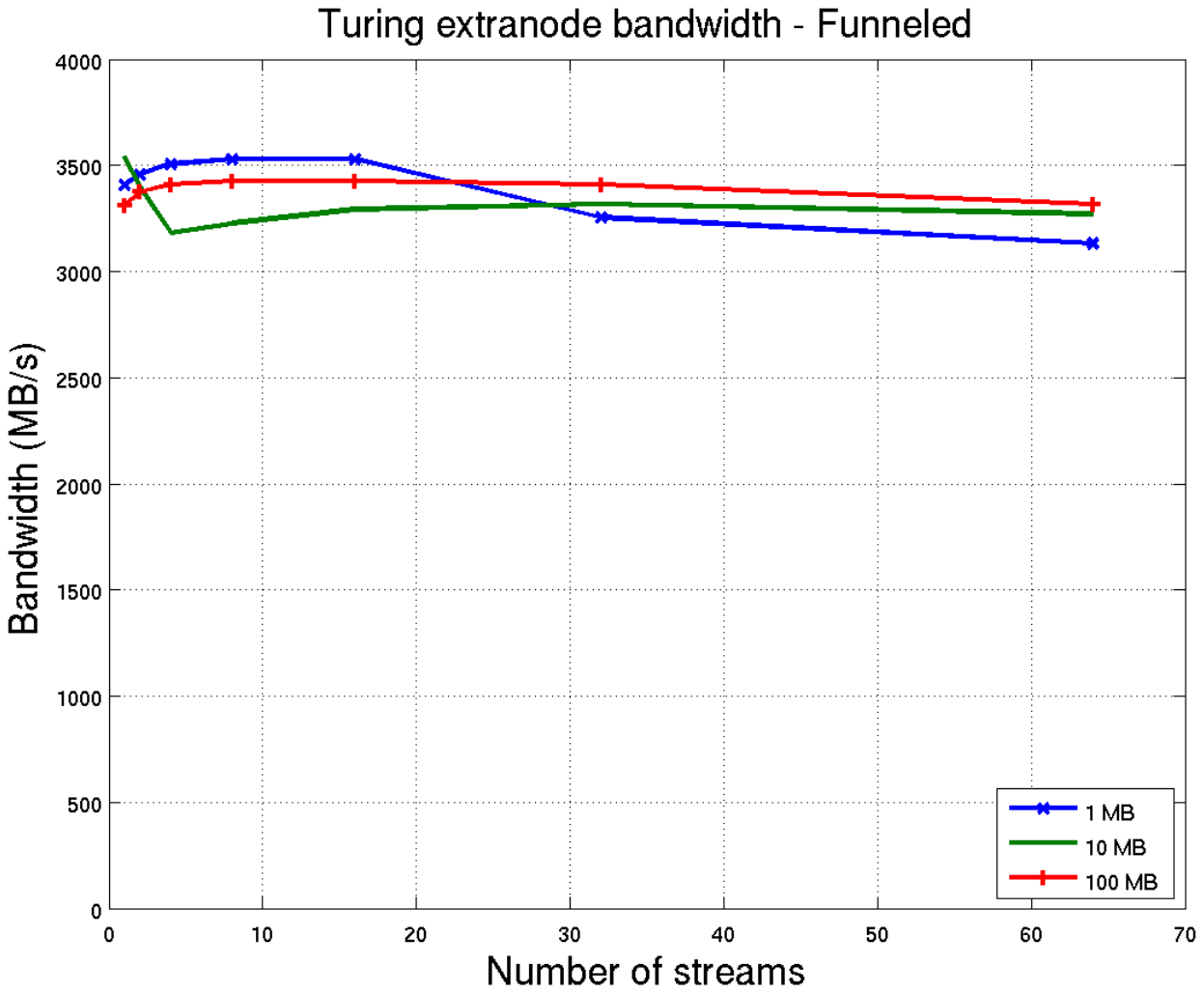
SBPR — Résultats sur Ada

- 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

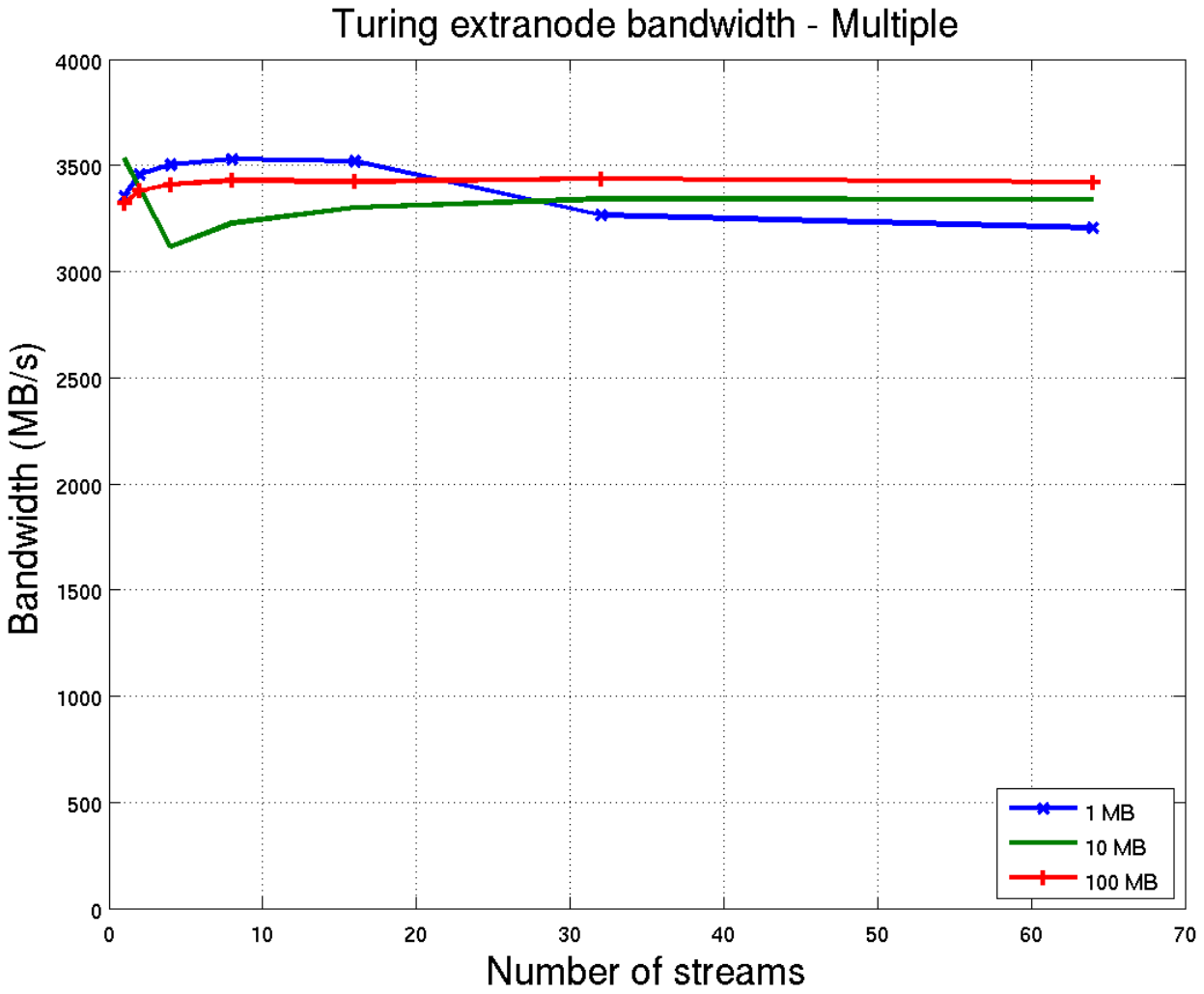
2 liens en // (direction E du tore 5D), débit crête 4 Go/s.



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

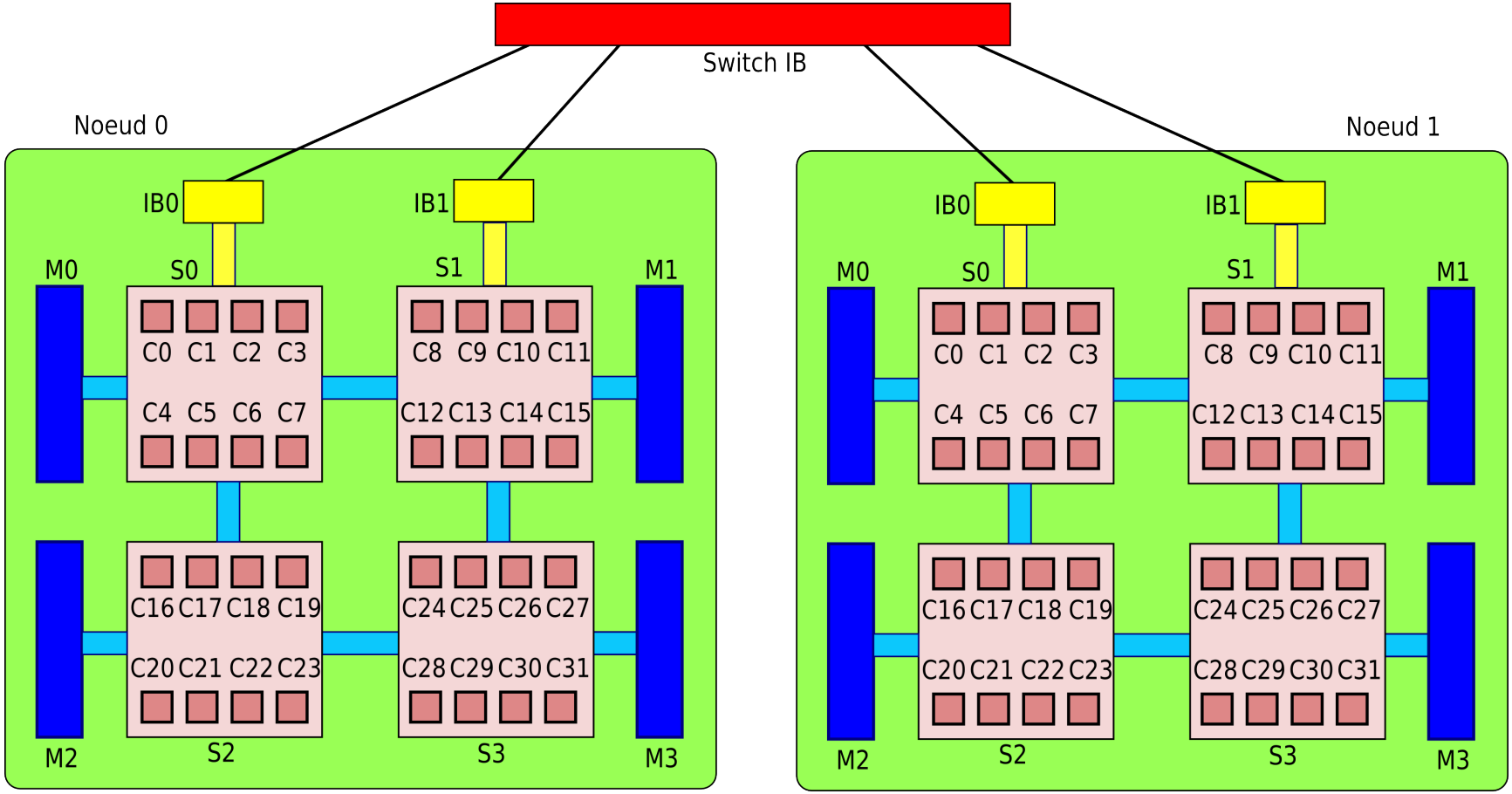
La plupart des machines de calcul modernes ont une architecture non uniforme :

- 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.

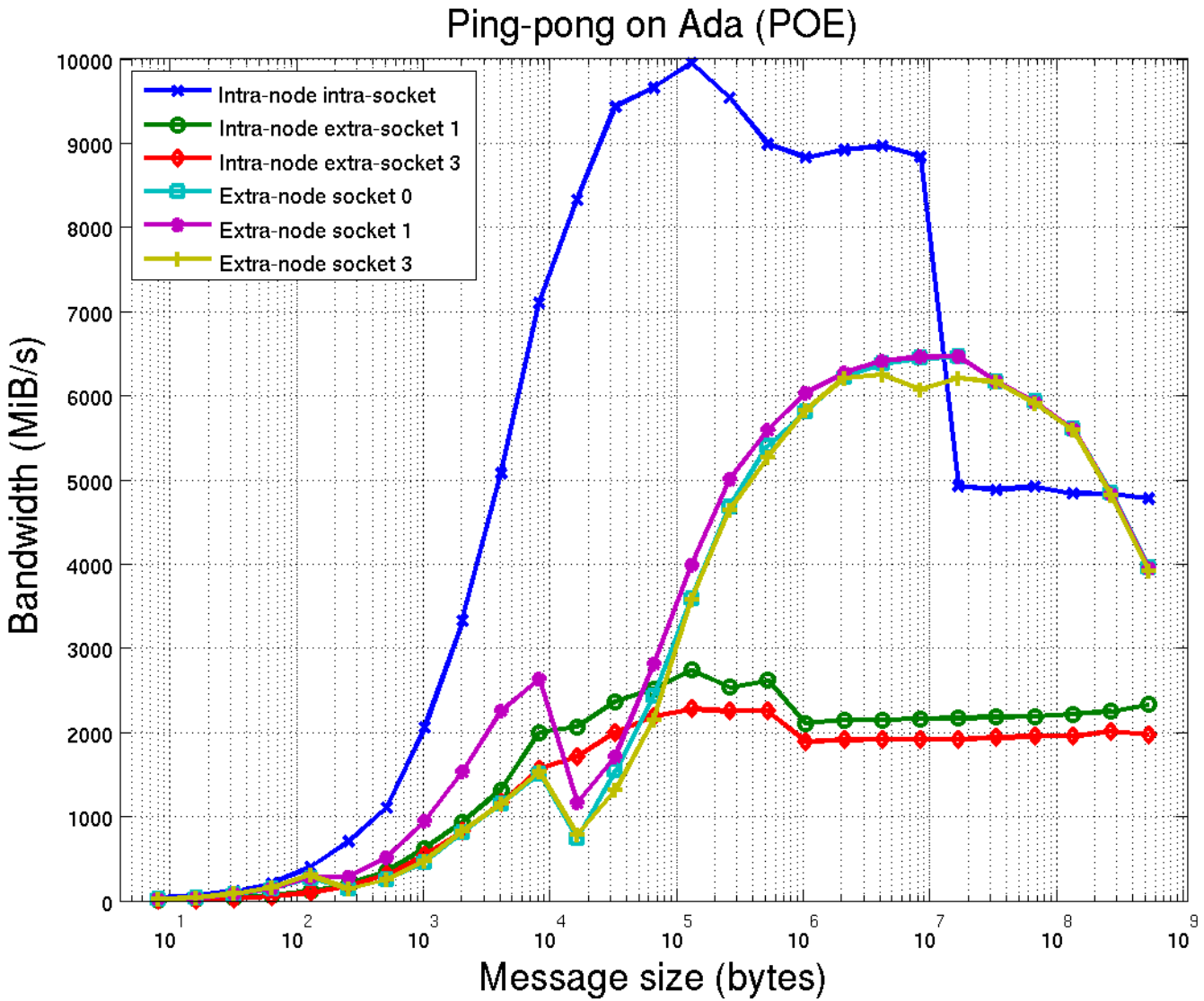
Effets

- 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.

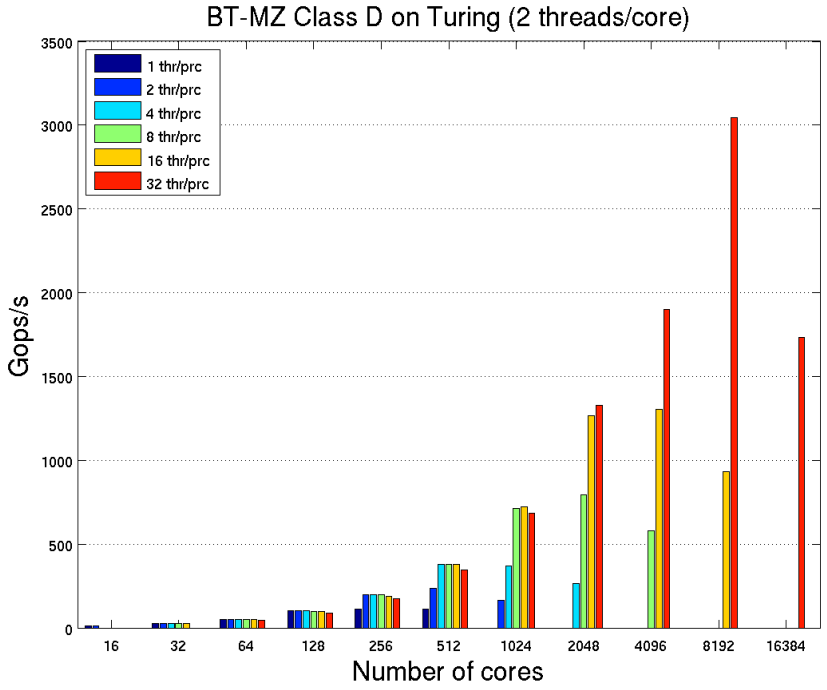
Application choisie : SP-MZ

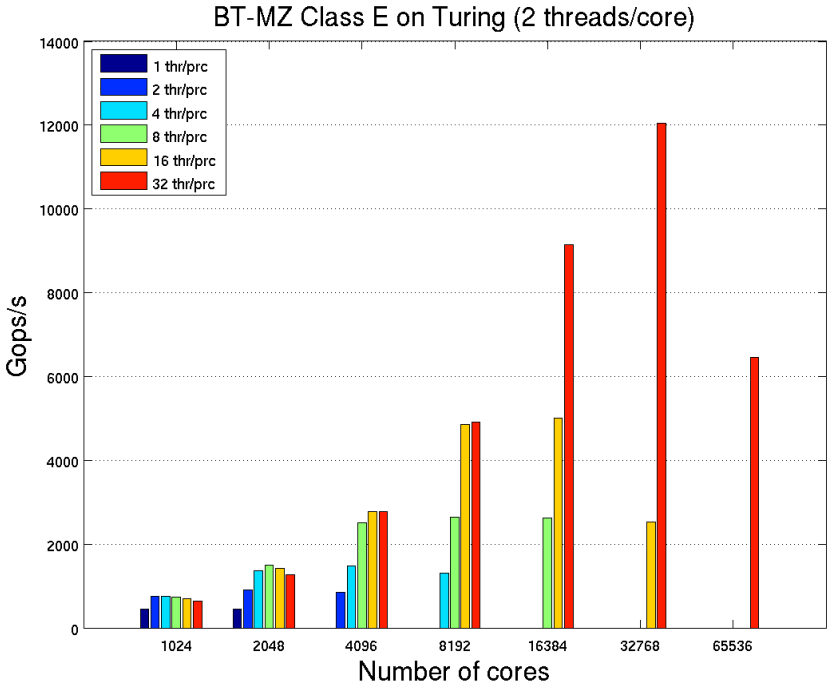
SP-MZ : méthode de résolution pentadiagonale scalaire.

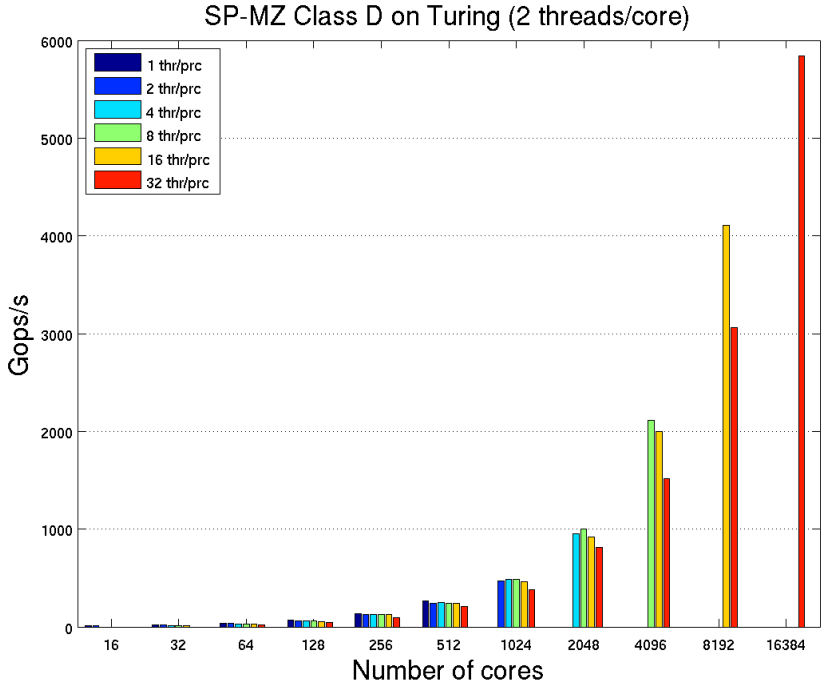
- Toutes les tailles de zones sont égales. Parfait équilibrage de charge.
- L'approche hybride ne devrait rien apporter sur ce point.

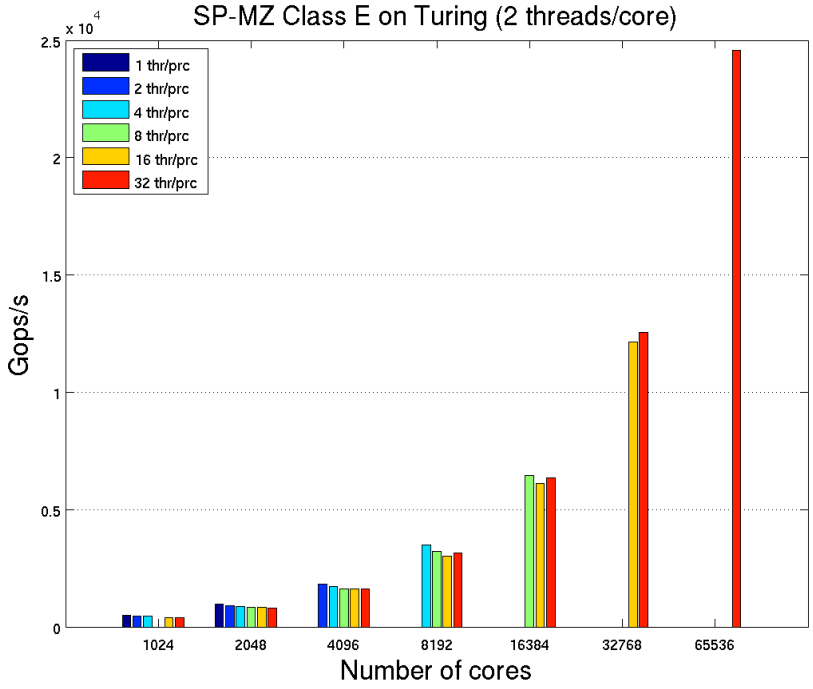
Tailles de problème sélectionnées

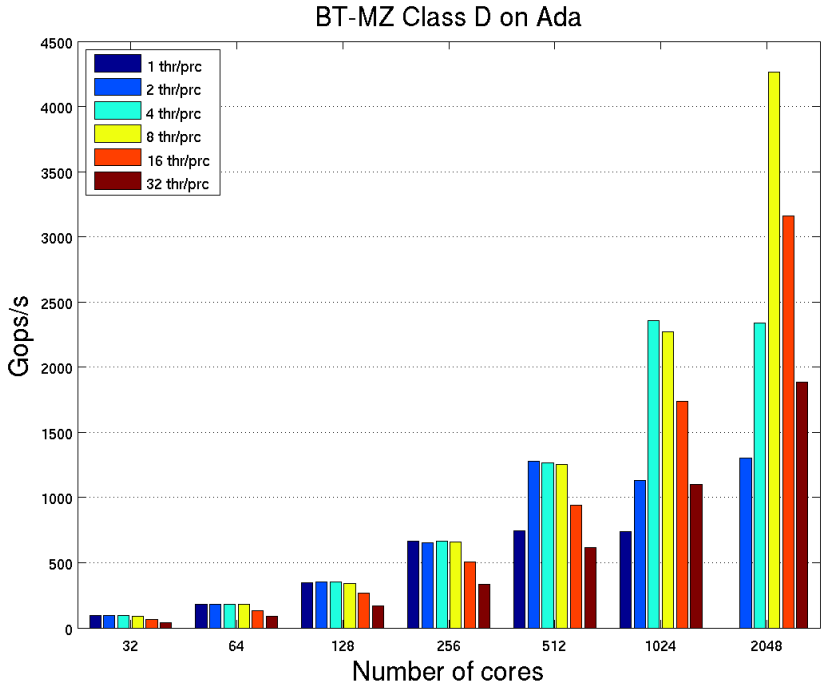
- Classe D : 1024 zones (et donc limité à 1024 processus MPI), 1632 x 1216 x 34 points de maillage (13 Gio)
- Classe E : 4096 zones (et donc limité à 4096 processus MPI), 4224 x 3456 x 92 points de maillage (250 Gio)

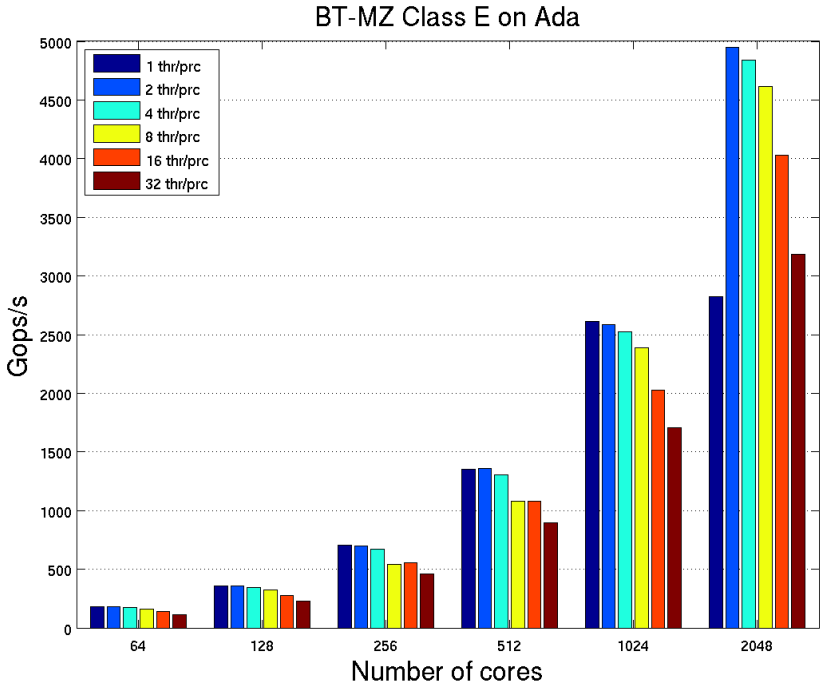

BT-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384

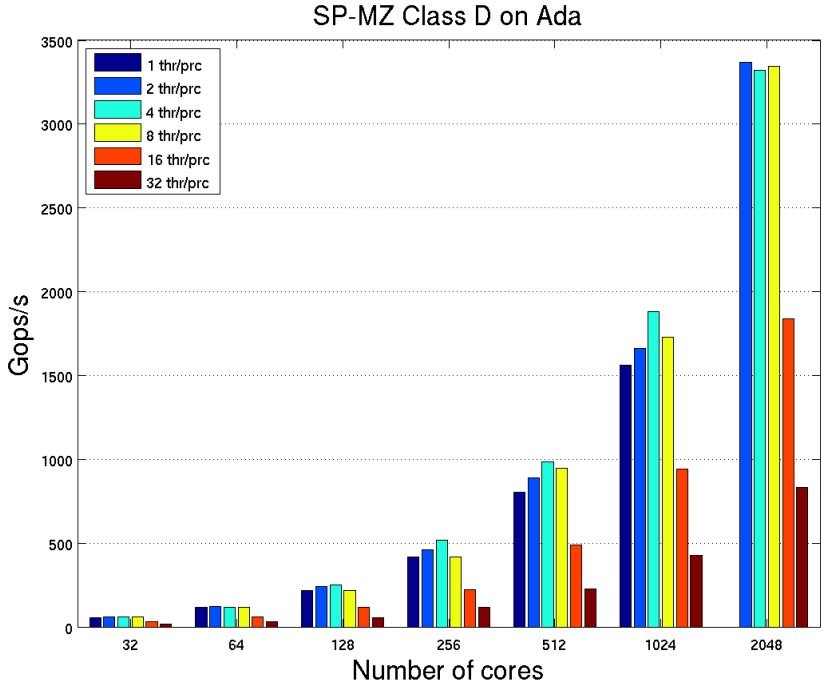

BT-MZ Class E on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
1024
2048
4096
8192
16384
32768
65536

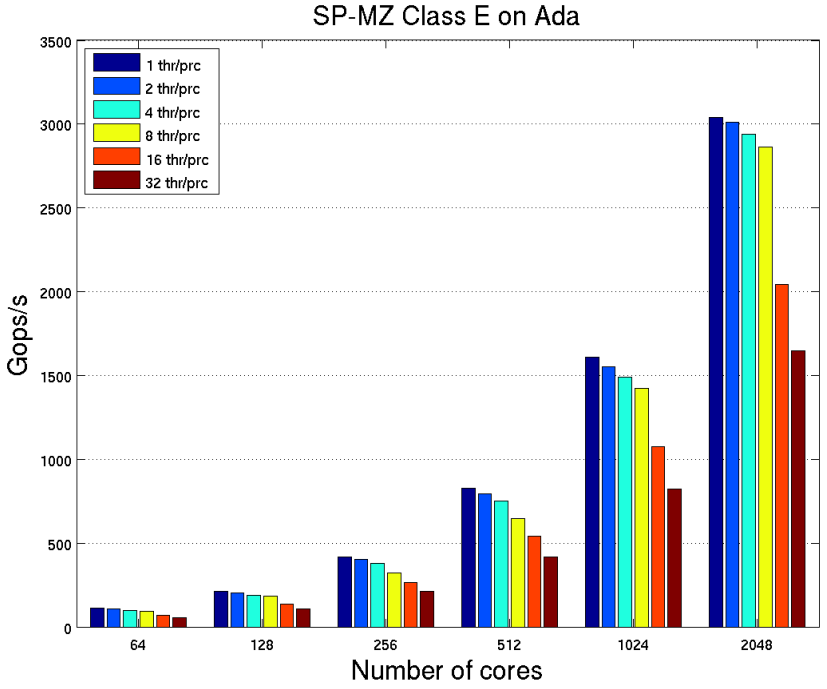

SP-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384


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.

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

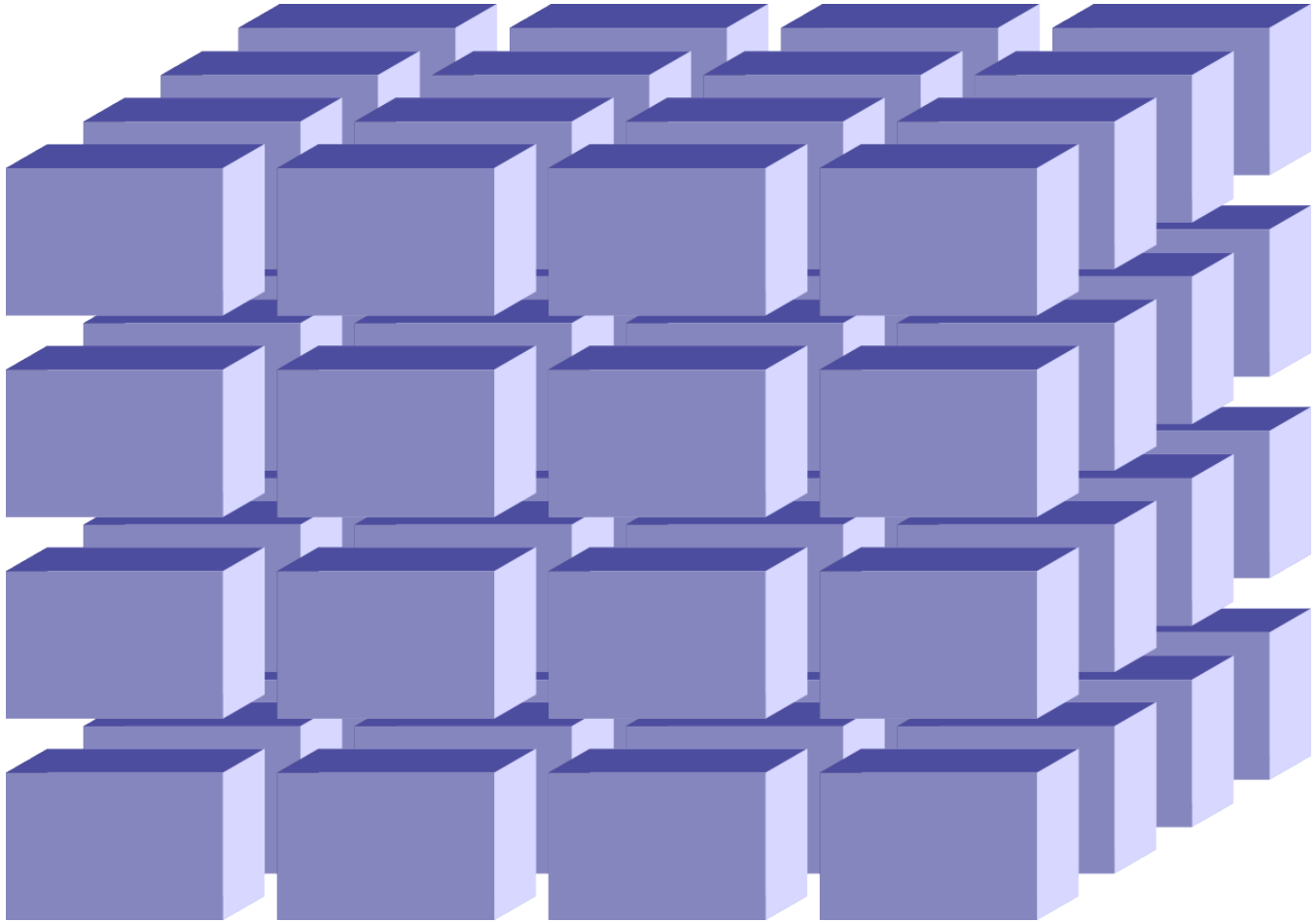
La discrétisation se fait sur un maillage régulier dans les trois directions spatiales (pas $h = h_x = h_y = h_z$).
La solution est calculée par un solveur Jacobi où la solution de l'itération $n+1$ est calculée à partir de la solution précédente de l'itération n.

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

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

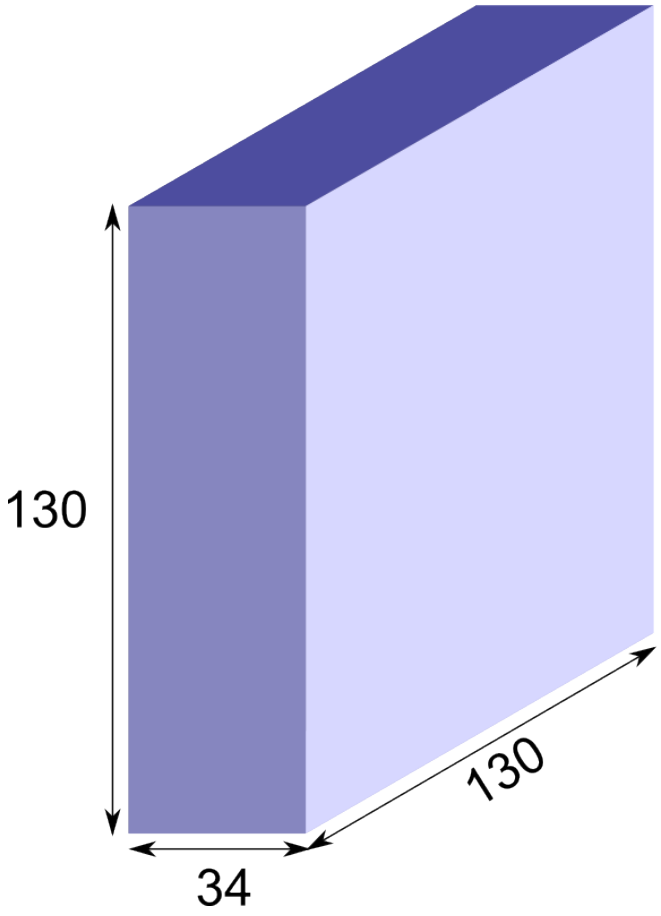
Exécutés sur 256 cœurs Blue Gene/P avec une taille de 512^3.

- 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.

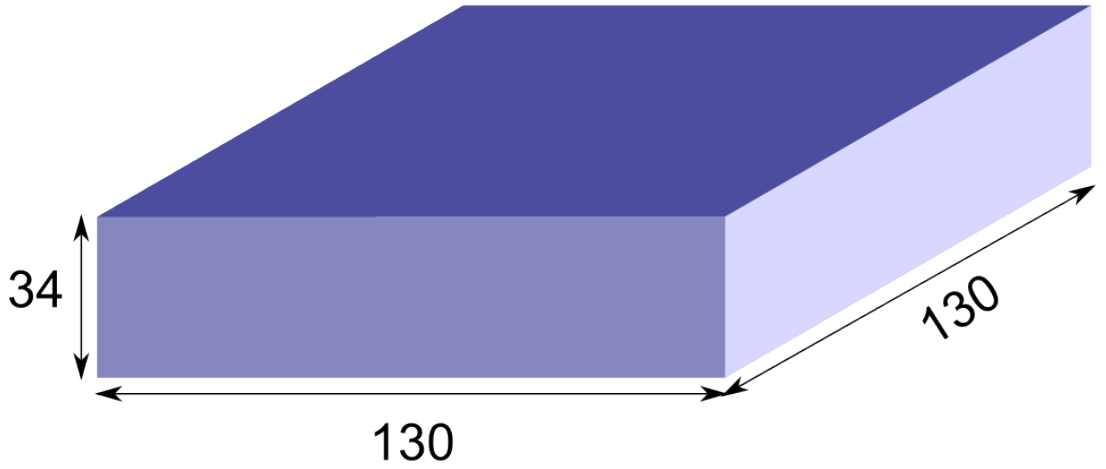
Etude de cas : Poisson3D

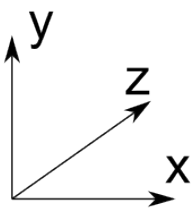
Forme des sous-domaines (512^3)

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

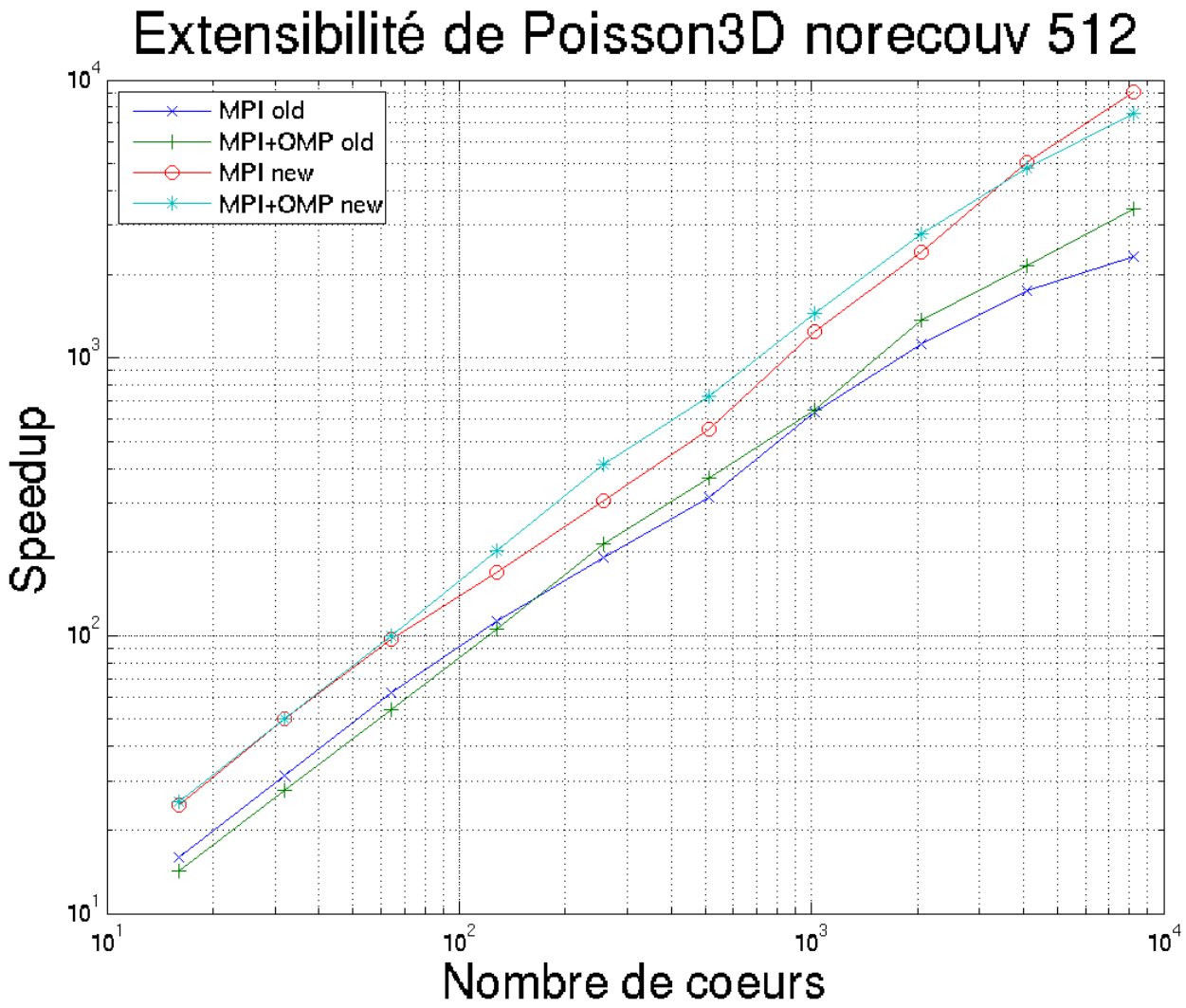
Les tests précédents donnaient des quantités de données envoyées sur le tore 3D variables en fonction de la topologie. Les causes en sont :

- 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*.

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.

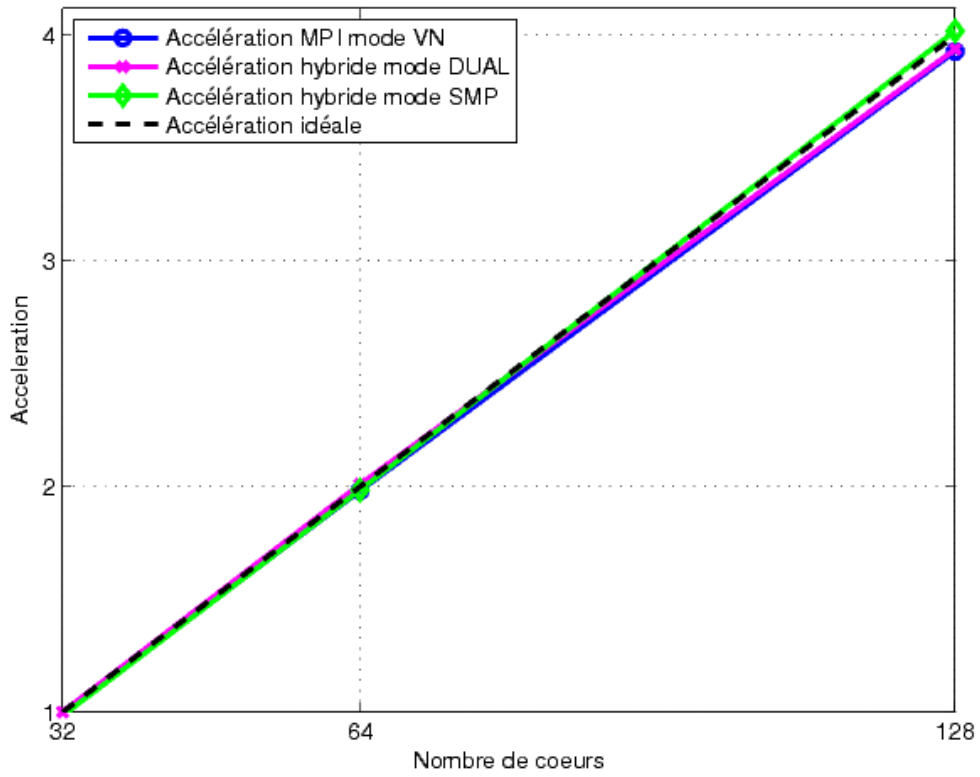
Présentation du code HYDRO (2)

- 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

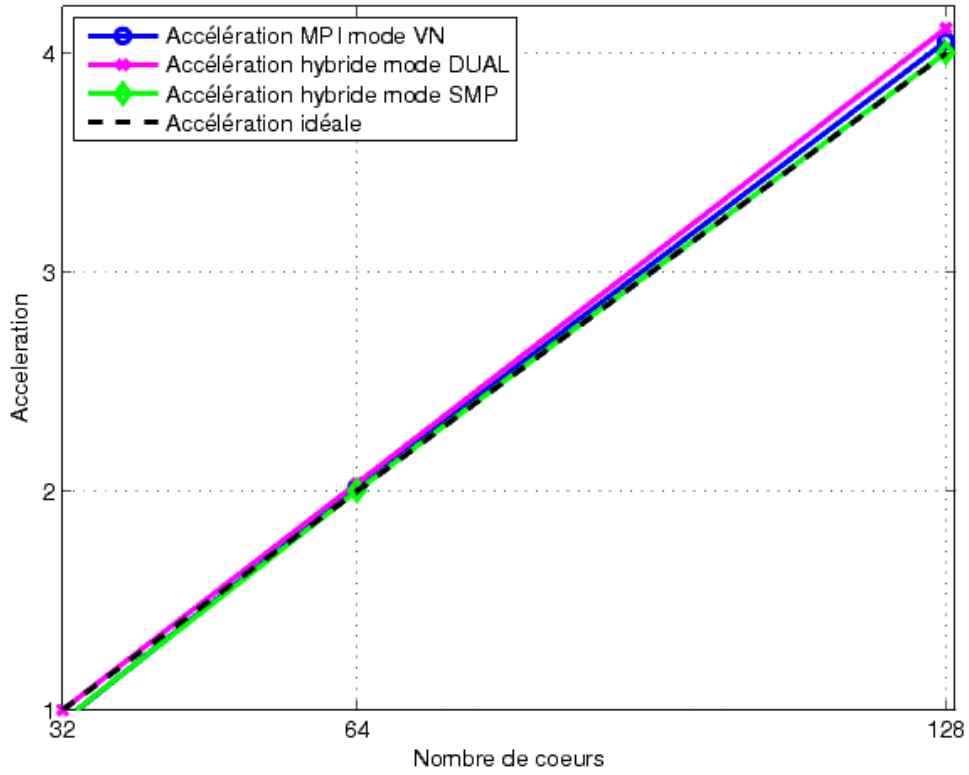
Time (s)	32 *cores*	64 *cores*	128 *cores*
Mode VN	49.12	24.74	12.47
Mode DUAL	49.00	24.39	12.44
Mode SMP	49.80	24.70	12.19



Résultats 128 cœurs Babel

Résultats pour le domaine nx = 10000, ny = 10000

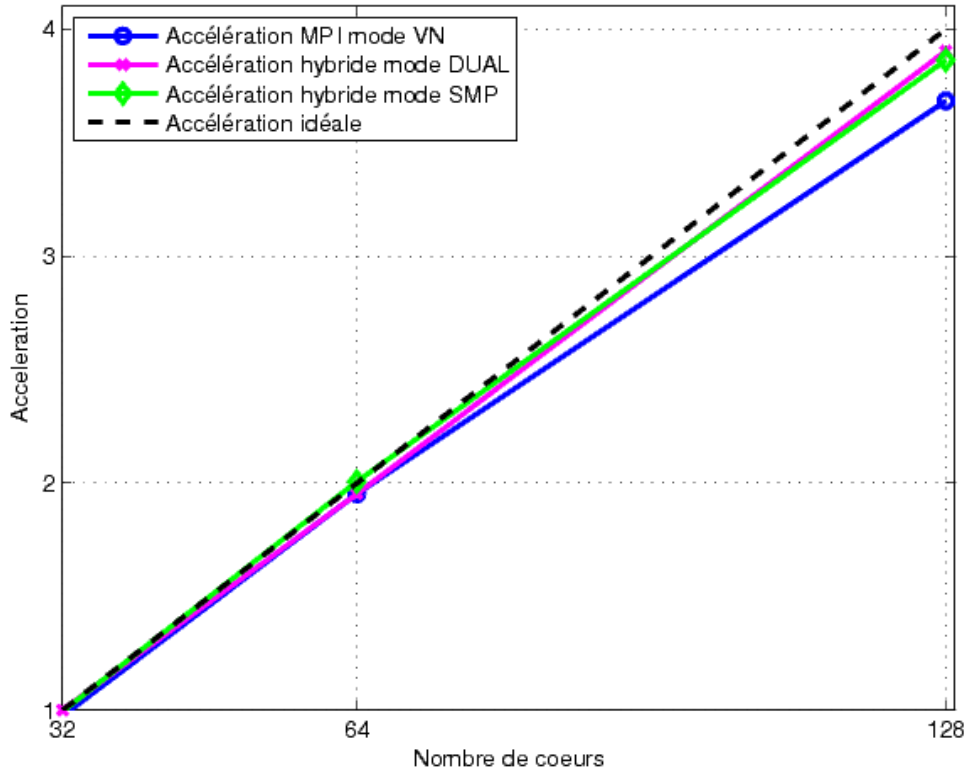
Time (s)	32 *cores*	64 *cores*	128 *cores*
Mode VN	53.14	24.94	12.40
Mode DUAL	50.28	24.70	12.22
Mode SMP	52.94	25.12	12.56



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*

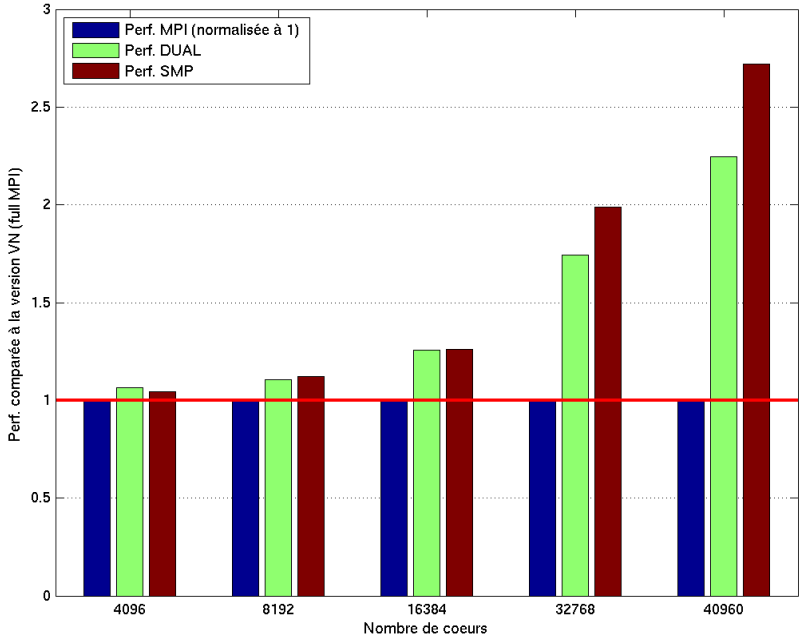
Caractéristiques des domaines utilisés pour le *Weak Scaling*

- 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

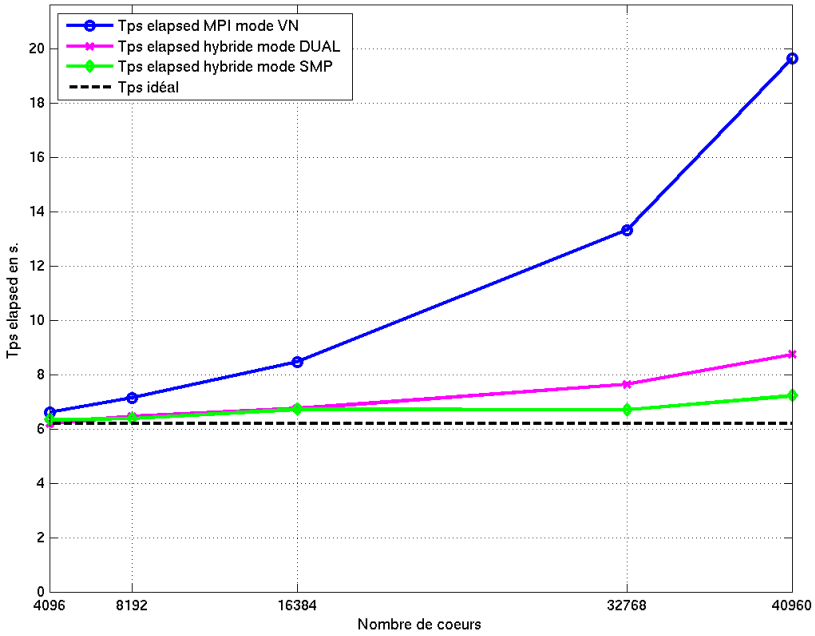
Résultats 10 *racks* Babel - *Weak Scaling*

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

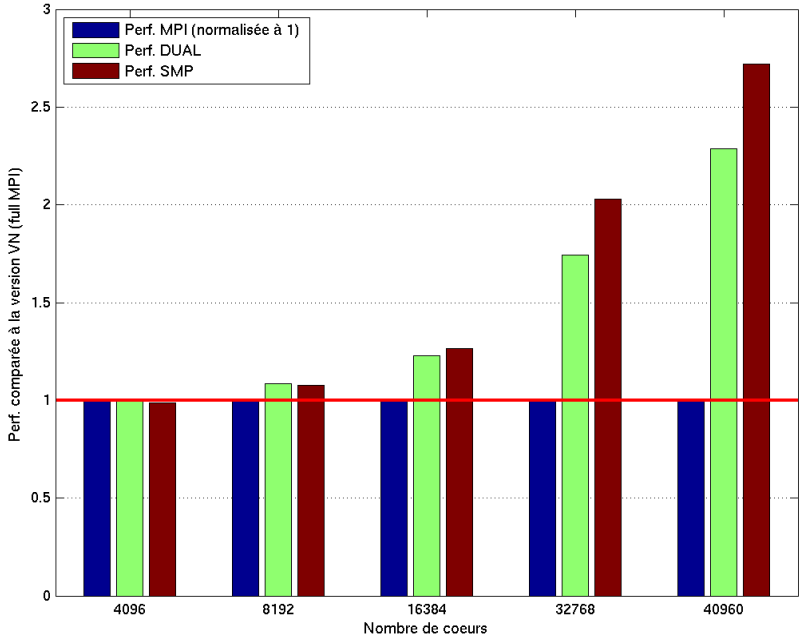


Temps *elapsed* d'exécution

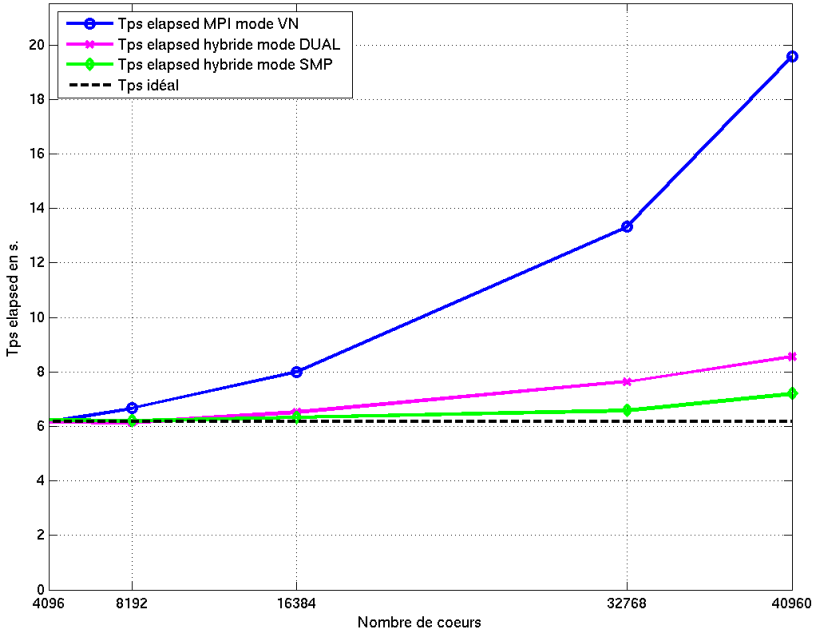
Résultats 10 *racks* Babel - *Weak Scaling*

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

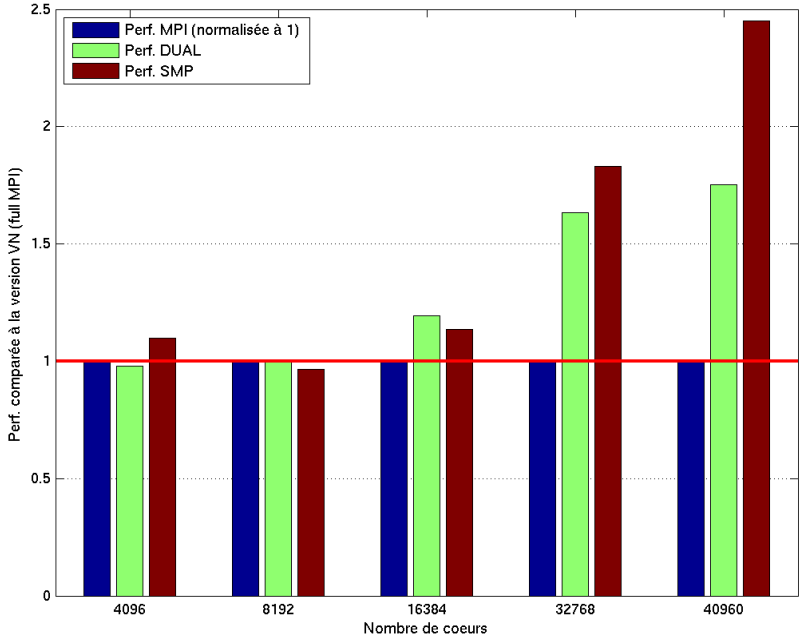


Temps *elapsed* d'exécution

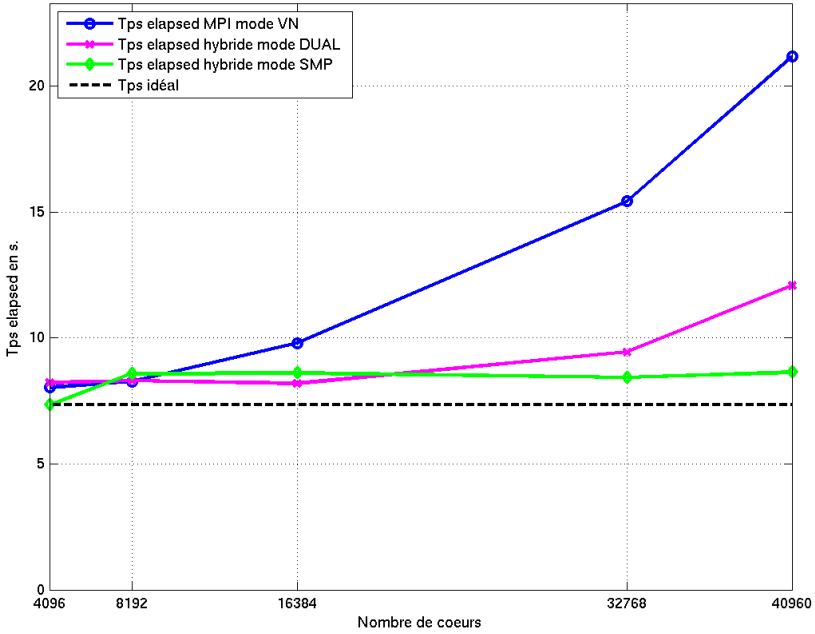
Résultats 10 *racks* Babel - *Weak Scaling*

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*

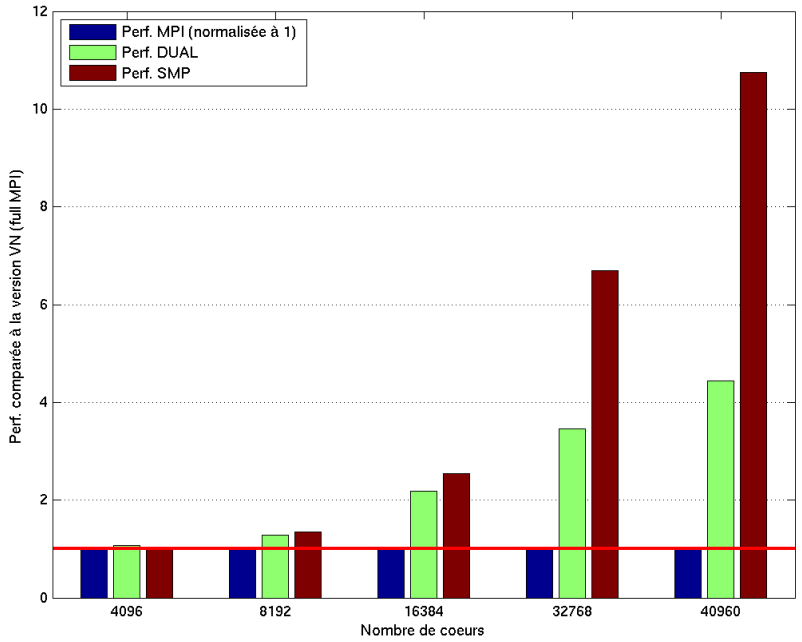
Interprétation des résultats

- 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 !

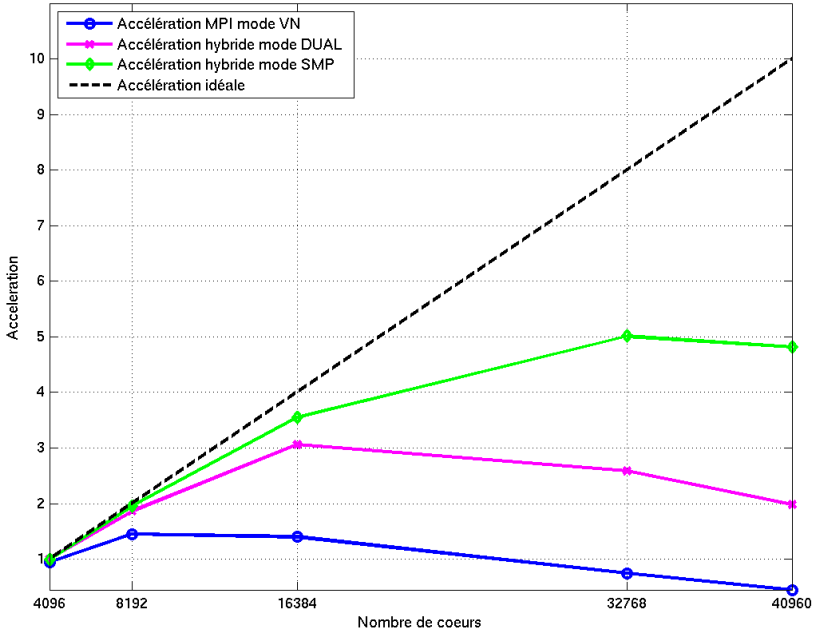
Résultats 10 *racks* Babel - *Strong Scaling*

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

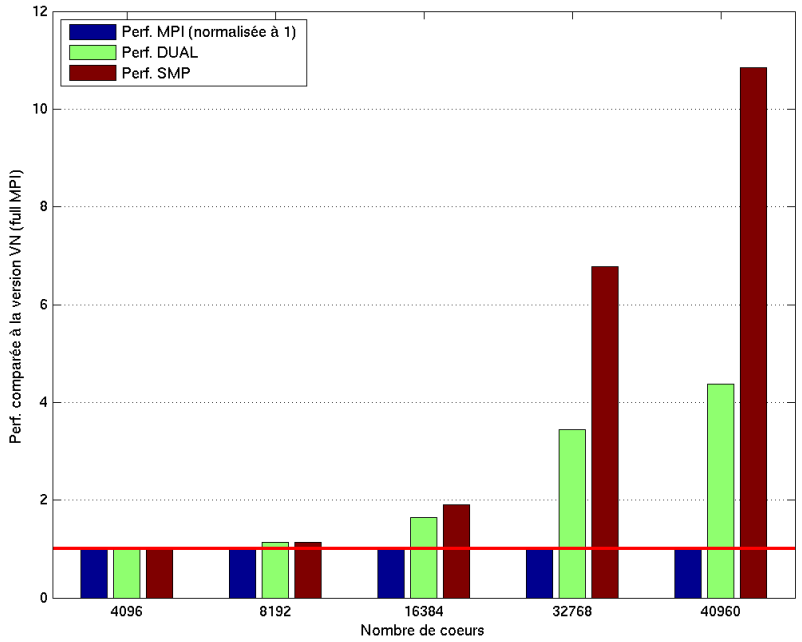


Extensibilité jusqu'à 40960 cœurs

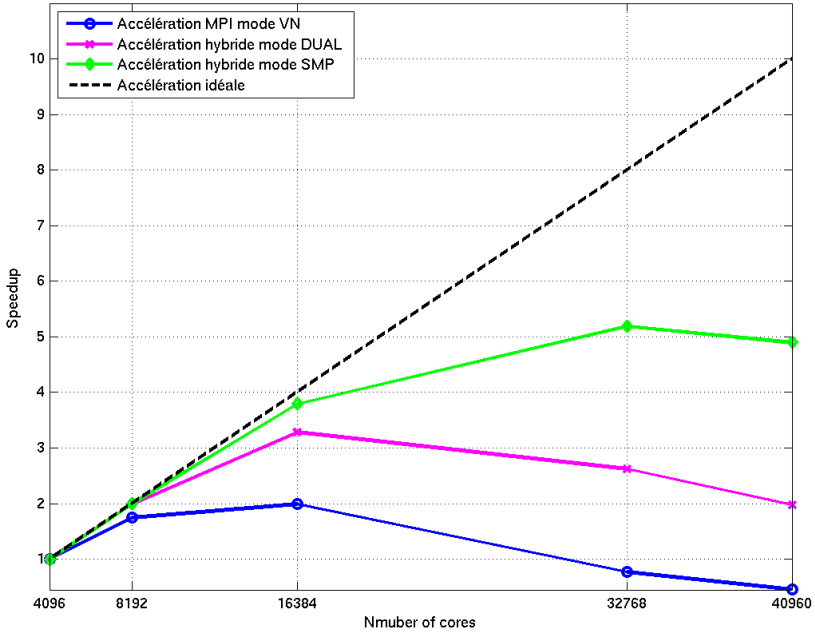
Résultats 10 *racks* Babel - *Strong Scaling*

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

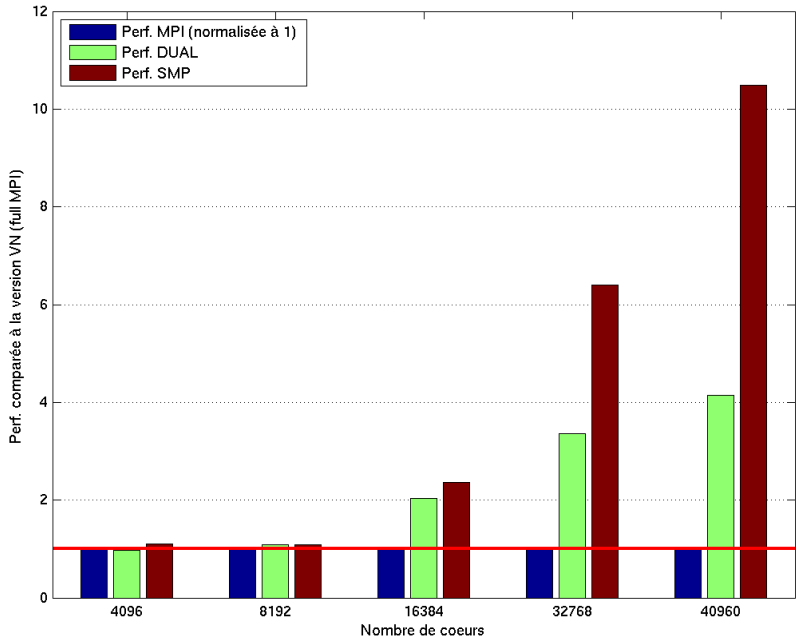


Extensibilité jusqu'à 40960 cœurs

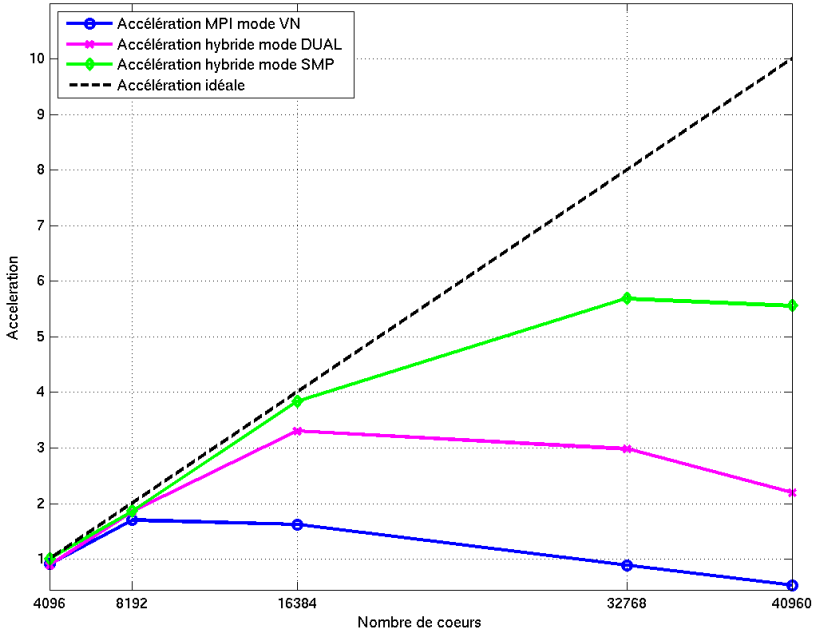
Résultats 10 *racks* Babel - *Strong Scaling*

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*

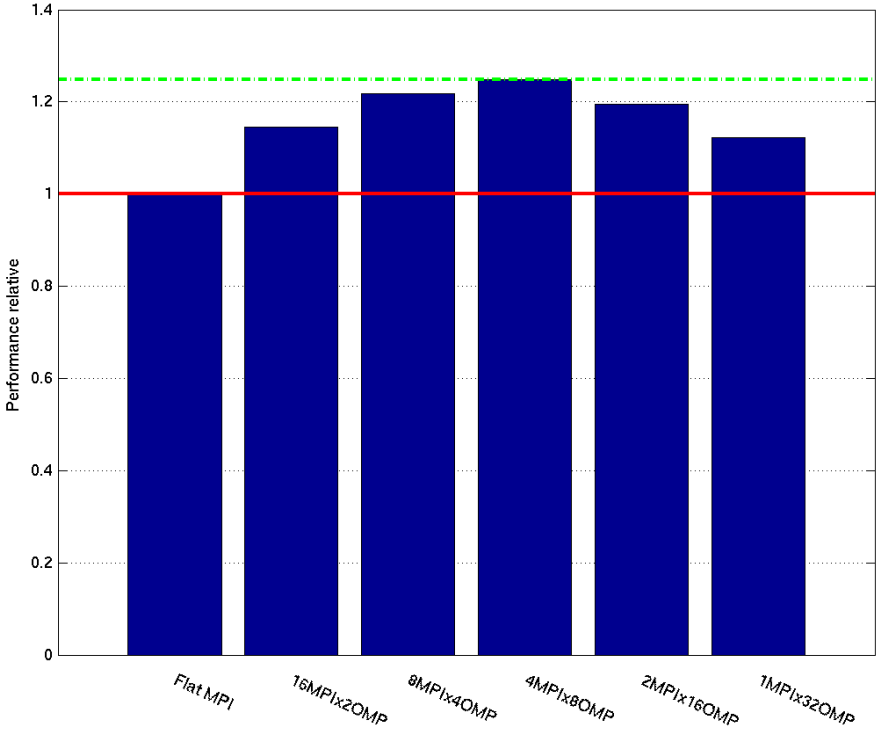
Interprétation des résultats

- 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

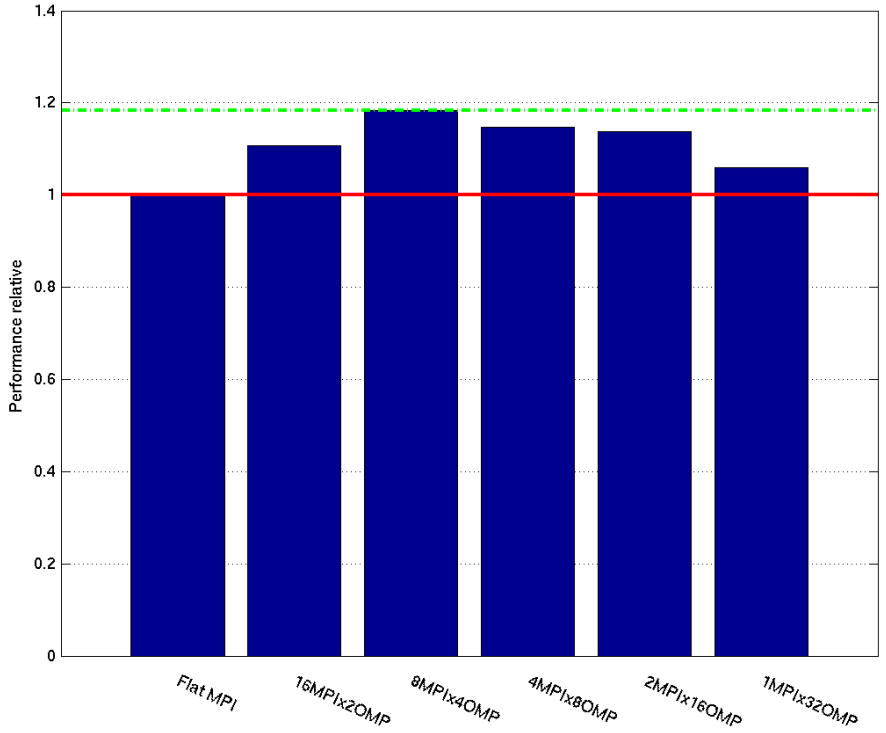


- La version hybride est toujours plus performante que la version pure MPI.
- Le gain maximum est supérieur à 20% pour les répartitions 8MPIx4OMP, 4MPIx8OMP et 2MPIx16OMP.

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

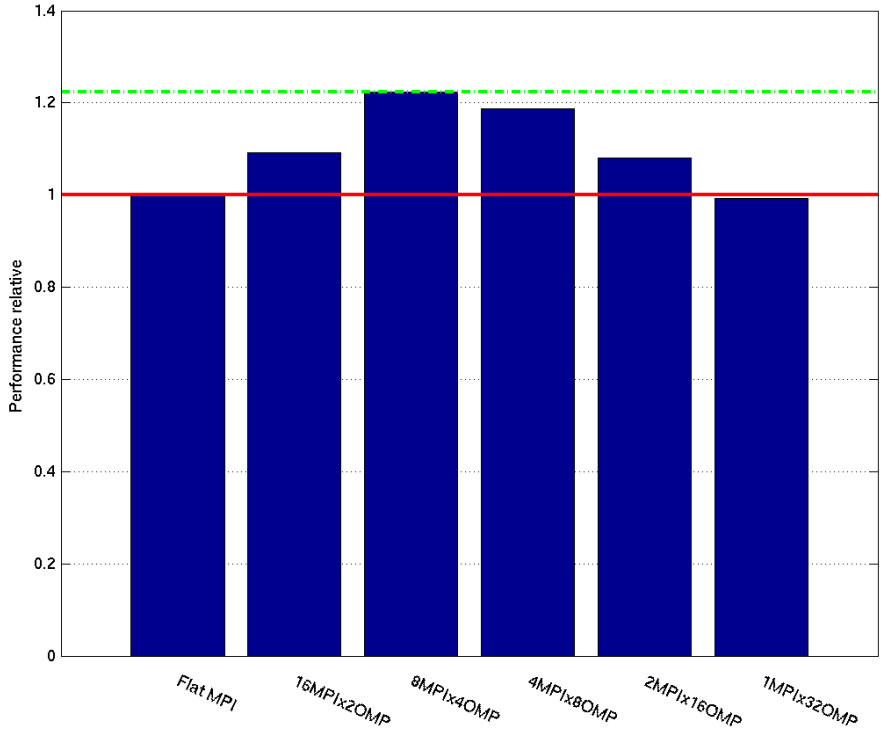


- La version hybride est toujours plus performante que la version pure MPI.
- Le gain maximum est de l'ordre 20% pour la répartition 8MPIx4OMP.

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

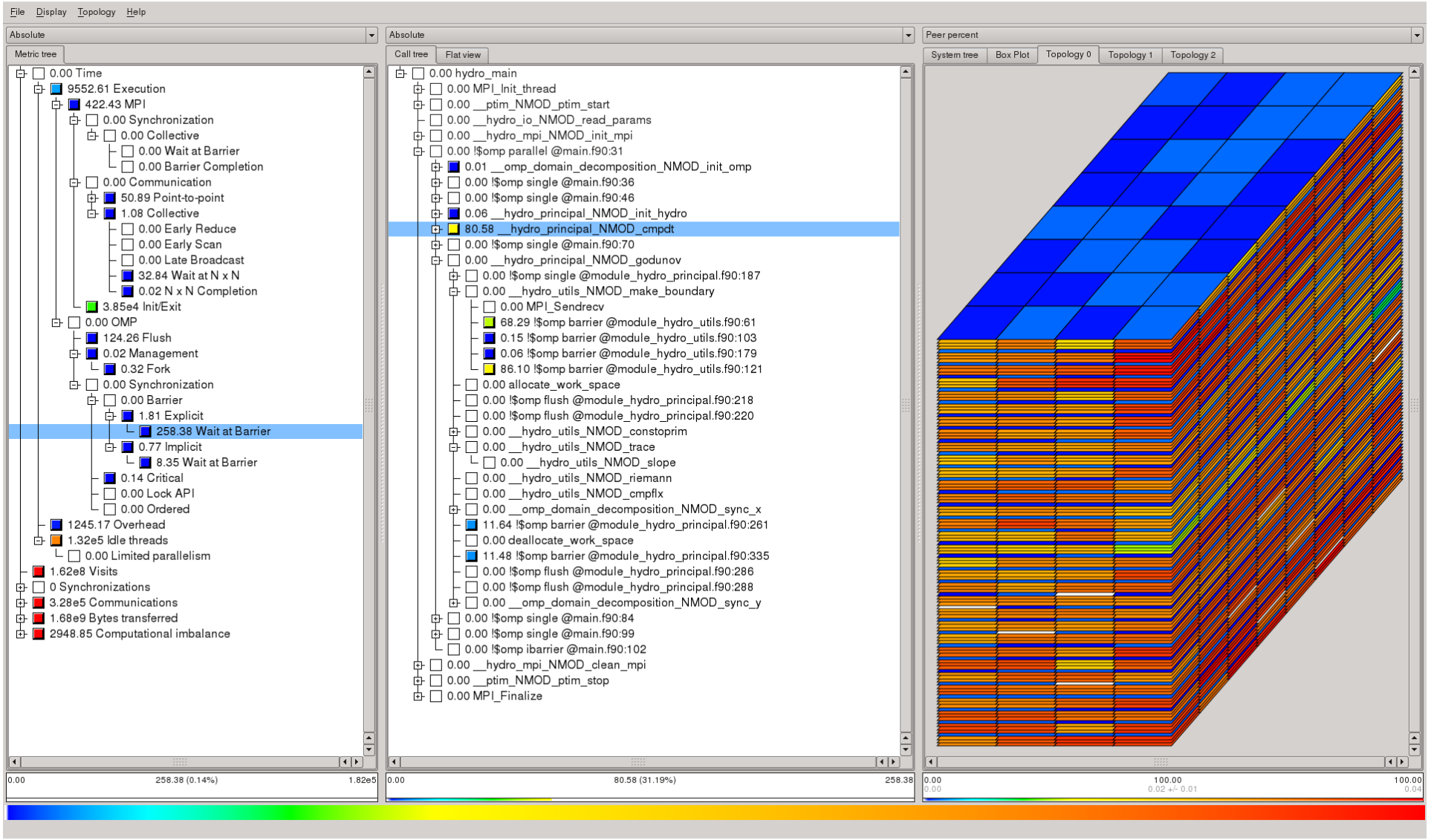
SCALASCA est un outil graphique d'analyse de performances pour applications parallèles. Principales caractéristiques :

- 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

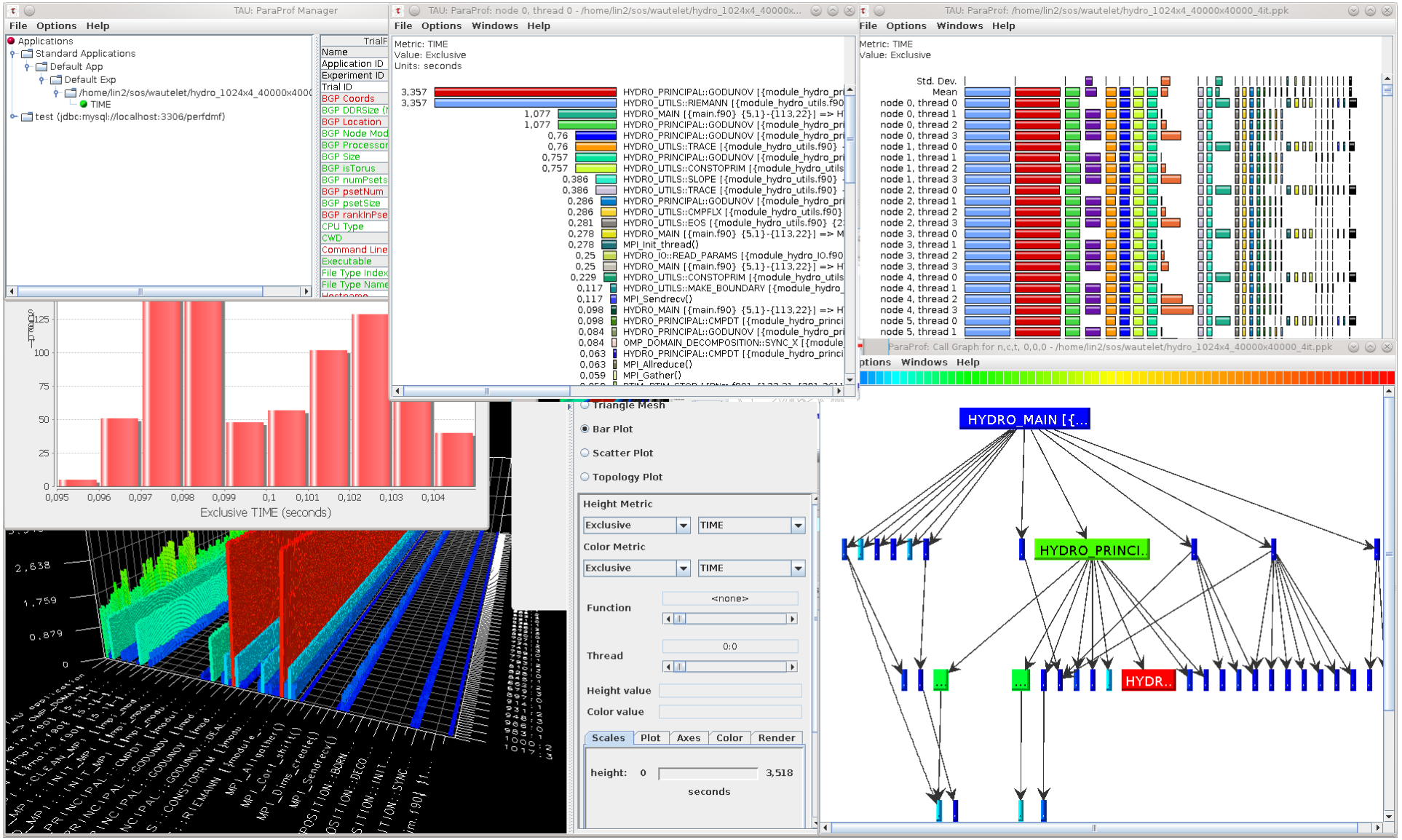
TAU

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
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
TAU: ParaProf: /home/lin2/sos/wautelet/hydro_1024x4_40000x40000_4it.ppk
File Options Windows Help
Metric: TIME
Value: Exclusive
Std. Dev.
Mean
Exclusive TIME (seconds)
Triangle Mesh
Bar Plot
Scatter Plot
Topology Plot
Height Metric
Exclusive
TIME
Color Metric
Function
<none>
Thread
0:0
Height value
Color value
Scales
Plot
Axes
Color
Render
height: 0
3,518
seconds
ParaProf: Call Graph for n,c,t, 0,0,0 - /home/lin2/sos/wautelet/hydro_1024x4_40000x40000_4it.ppk
HYDRO_MAIN [{...
HYDRO_PRINCI.
HYDR..

TotalView

Description

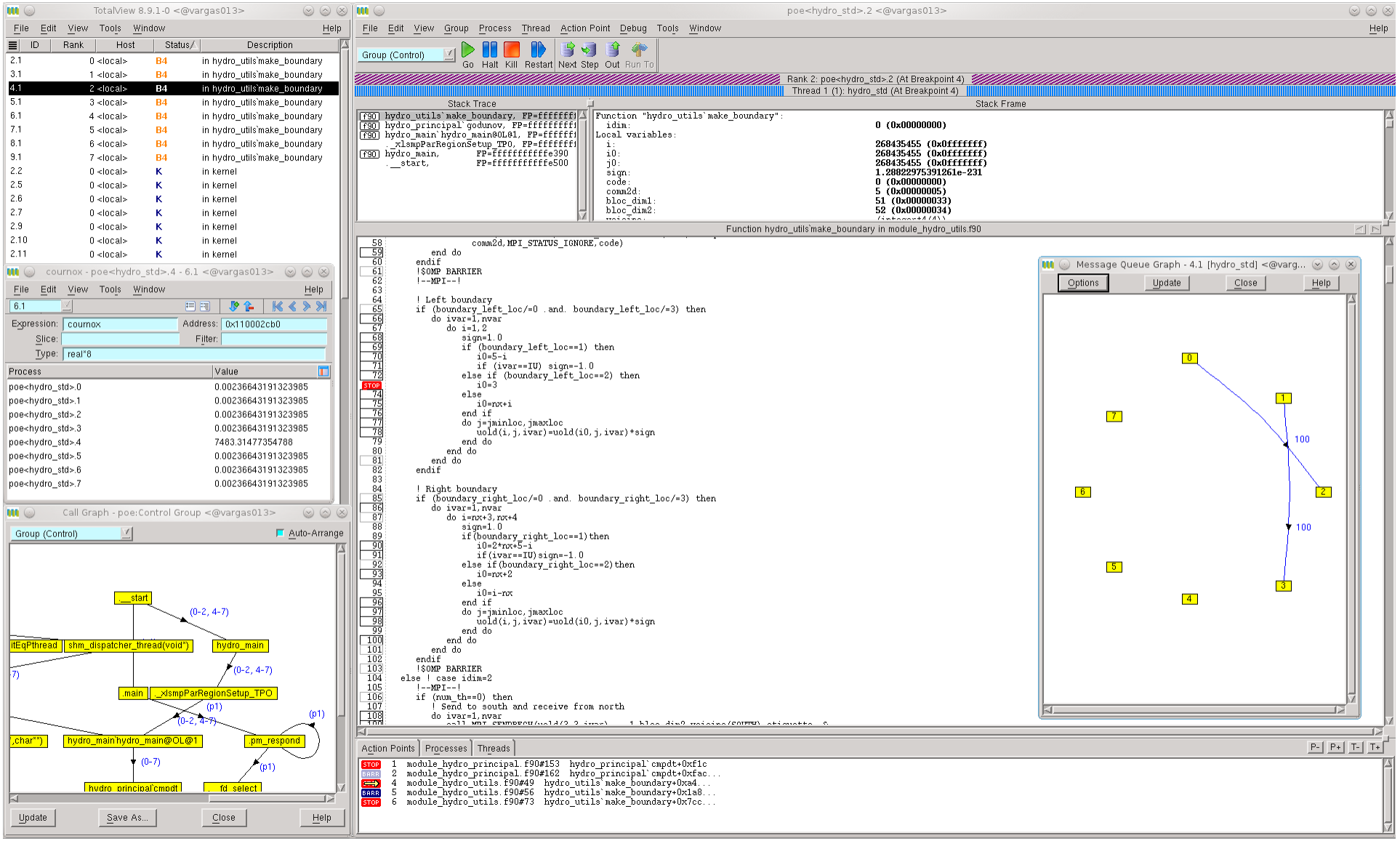
TotalView est un outil de débogage graphique d'applications parallèles. Principales caractéristiques :

- 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

Annexes

Facteurs influençant les performances MPI

Nœuds de la machine

- Cœurs (fréquence, nombre par nœud...)
- Mémoire (débits, latences, caches, nombre de canaux, partage entre les différents cœurs...)
- Cartes réseaux (débits, latences, type, connexion au nœud, nombre de liens...)
- Présence ou non du moteur RDMA pour les communications
- OS (*light kernel*...)
- Configuration/*tuning* de la machine

Réseau

- Type de carte (propriétaire, InfiniBand, Ethernet...)
- Topologie réseau (en étoile, tore 3D ou plus, *fat tree*, hypercube...)
- Protocole (bas niveau, TCP/IP...)
- Contention (entre processus du même *job* ou entre différents *jobs*)
- Configuration/*tuning* du réseau

Facteurs influençant les performances MPI

Application

- Algorithmes, accès mémoire...
- Rapport calculs/communications, granularité
- Méthode de partage des données
- Equilibrage de charge
- Placement des processus
- *Binding* des processus sur les cœurs
- Entrées/sorties
- Taille des messages
- Types de communications, utilisation de communicateurs...

Facteurs influençant les performances MPI

Implémentation MPI

- Adaptation à l'architecture de la machine (implémentation constructeur, *open source*...)
- Utilisation des *buffers*
- Protocoles de communications (*eager*, rendez-vous...)
- Influence des variables d'environnement
- Qualité de l'implémentation et des algorithmes utilisés

Envois en mode *ready*

Envois en mode *ready*

Un envoi en mode *ready* se fait en appellant le sous-programme `MPI_Rsend` ou `MPI_Irsend`.

Attention : il est obligatoire de faire ces appels seulement lorsque la réception est déjà postée.

Leur utilisation est fortement déconseillée.

Avantages

- Légèrement plus performant que le mode synchrone car le protocole de synchronisation peut être simplifié

Inconvénients

- Erreurs si le récepteur n'est pas prêt lors de l'envoi

Communications persistantes

Caractéristiques

- Les communications persistantes permettent de répéter à volonté des communications portant sur le même espace mémoire, mais avec des valeurs différentes.
- L'avantage est d'éviter d'initialiser ces communications (et les structures de données associées) à chaque appel (théoriquement moins de surcoûts).
- Un canal de communication est ainsi créé pour l'envoi des messages.
- Ce sont des communications point à point non bloquantes.
- Les communications MPI persistantes n'apportent aucun gain significatif sur les machines actuelles. Les performances sont généralement très proches de celles des communications point à point standards non bloquantes.

L'utilisation des communications persistantes n'est pas conseillée car elles n'apportent (actuellement) aucun avantage.

Communications persistantes

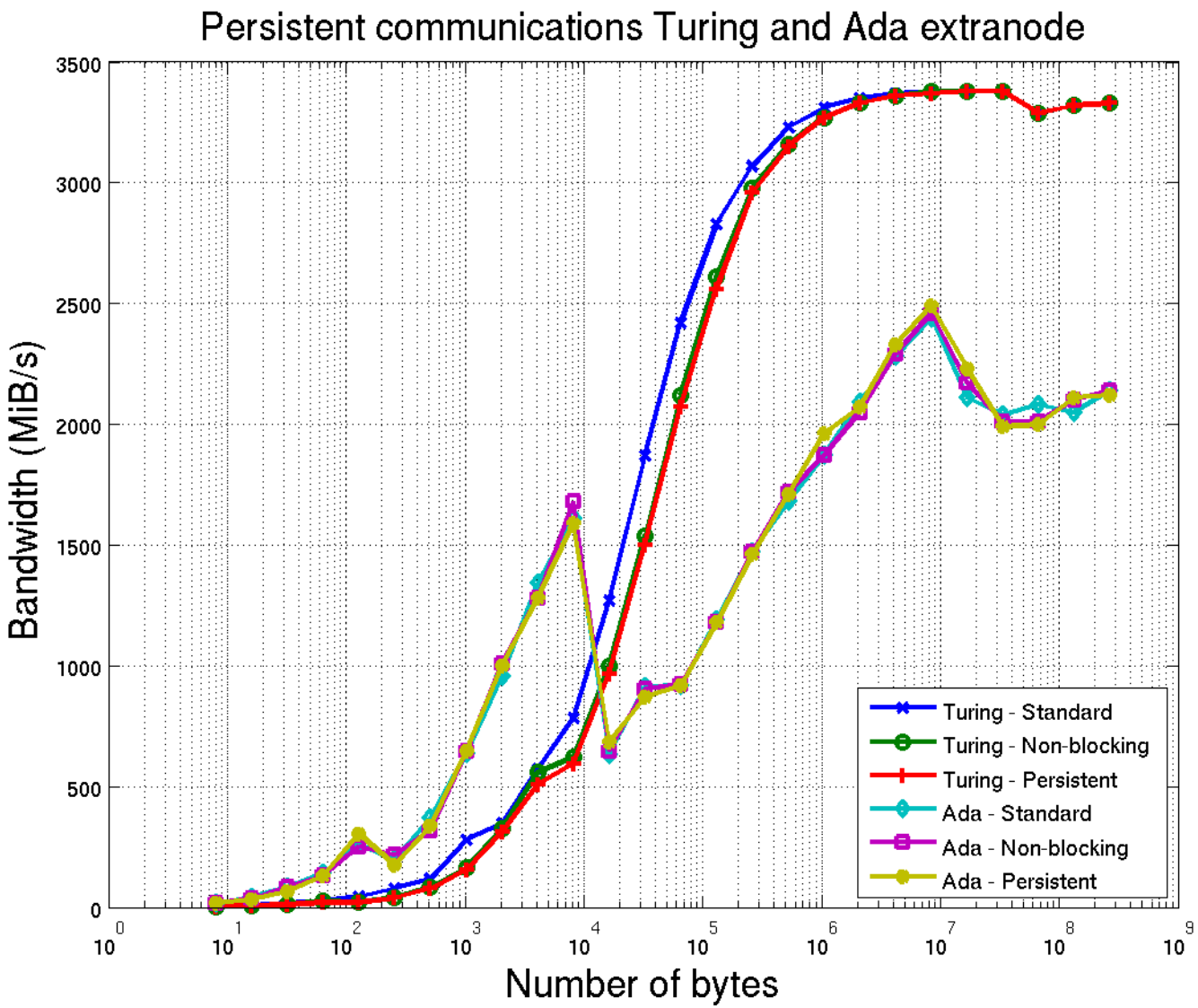
Utilisation

- Initialisation de la communication persistante avec `MPI_Send_init`/`MPI_Recv_init` (ou variantes)
- Répétition de la séquence de communication (dans une boucle par exemple) :
  - Début de la communication avec appel à `MPI_Start`
  - Fin de la communication (non bloquante) avec `MPI_Wait`
- Libération des ressources avec `MPI_Request_free`

Exemple

```
call MPI_Send_init(data,sz,MPI_REAL,dest,tag,comm,req,ierr)
do i=1,niter
  call MPI_Start(req,ierr)
  call kernel()
  call MPI_Wait(req,MPI_STATUS_IGNORE,ierr)
end do
call MPI_Request_free(req,ierr)
```

Communications persistantes



Introduction à l'optimisation de codes

Définition

- Optimiser un code de calcul consiste à réduire ses besoins en ressources.
- Ces dernières sont diverses mais généralement on parle du temps de restitution.
- La consommation de mémoire ou d'espace disque entrent également dans cette catégorie.

Introduction à l'optimisation de codes

Pourquoi optimiser ?

Optimiser une application peut apporter un certain nombre d'avantages :

- Obtention plus rapide des résultats par une diminution du temps de restitution ;
- Possibilité d'obtenir plus de résultats avec les heures qui vous ont été attribuées ;
- Possibilité d'effectuer de plus gros calculs ;
- Meilleure compréhension du code, de l'architecture de la machine et de leurs interactions ;
- Avantage compétitif par rapport à d'autres équipes ;
- Découverte et correction de bogues grâce à la relecture des sources du code.

L'amélioration des performances d'une application permet également :

- Une réduction de la consommation d'énergie de vos calculs ;
- Une meilleure utilisation de la machine (le coût d'achat, de maintenance et d'utilisation d'un supercalculateur n'est pas négligeable). A l'IDRIS, chaque heure qui vous est attribuée représente une dépense pour toute la communauté scientifique ;
- De libérer des ressources pour d'autres groupes de recherche.

Introduction à l'optimisation de codes

Pourquoi ne pas optimiser ?

- Insuffisance de ressources ou de moyens pour le faire (manque de personnel, de temps, de compétences...) ;
- Diminution de la portabilité du code (beaucoup d'optimisations sont spécifiques à l'architecture de la machine) ;
- Risque de pertes de performances sur d'autres machines ;
- Diminution de la lisibilité des sources et maintenance plus délicate ;
- Risque d'introduction de bogues ;
- Code non pérenne ;
- Code suffisamment rapide (il ne sert à rien d'optimiser un code qui fournit déjà des résultats dans un temps acceptable) ;
- Code déjà optimisé.

Introduction à l'optimisation de codes

Quand optimiser ?

- Une application ne doit être optimisée que lorsqu'elle fonctionne correctement car
  - risques de nouveaux bogues,
  - réduction lisibilité et compréhension du code et
  - risques d'optimisation de procédures qui seront abandonnées ou ne seront pas ou très peu utilisées ou qui seront totalement réécrites.
- Ne se lancer dans l'optimisation que si application est trop lente ou ne permet pas de faire tourner de gros calculs dans un temps acceptable.
- Si vous êtes pleinement satisfait des performances de votre application, l'investissement n'est peut être pas nécessaire.
- Avoir du temps devant soi.

Introduction à l'optimisation de codes

Comment optimiser ?

- Avant tout : faire un profilage séquentiel et parallèle avec un jeu de test réaliste pour identifier les zones critiques
- Optimiser là où l'on consomme le plus de ressources
- Vérifier chaque optimisation : résultats toujours corrects ? Performances réellement améliorées ?
- Question : si gain faible : garder l'optimisation ?

Quoi optimiser ?

- Les performances séquentielles
- Les communications MPI, les performances OpenMP et l'extensibilité
- Les entrées/sorties

Introduction à l'optimisation de codes

Optimisation séquentielle

- Algorithmes et conception
- Bibliothèques
- Compilateur
- Caches
- Unités de calcul spécialisées (SIMD, SSE, AVX, QPX...)
- Autres

Introduction à l'optimisation de codes

Optimisation des communications MPI, d'OpenMP et de l'extensibilité

- Algorithmes et conception
- Equilibrage de charge
- Recouvrement calculs/communications
- Placement des processus
- Programmation hybride
- Autres

Optimisation des entrées/sorties

- Ne lire et écrire que le nécessaire
- Réduire la précision (flottants simple précision au lieu de double)
- Entrées/sorties parallèles
- Bibilothèques (MPI-I/O, HDF5, NetCDF, Parallel-NetCDF...)
- *Hints* MPI-I/O
- Autres

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.

Travaux pratiques

TP1 — MPI — HYDRO

Objectif

Paralléliser une application à l'aide de MPI.

Enoncé

On vous demande de partir de l'application HYDRO version séquentielle. Elle devra être parallélisée à l'aide de MPI.

1. Dans un premier temps, la parallélisation peut se faire dans une seule direction (nord-sud en Fortran pour éviter de passer par des types dérivés).
2. Ensuite, une version parallélisée dans les 2 directions spatiales vous est demandée.

Quelques conseils

- Les mailles servant à imposer les conditions limites dans la version séquentielle peuvent servir de mailles fantômes pour les communications entre processus.
- Une frontière entre domaines peut être vue comme une condition limite particulière.
- L'utilisation d'un communicateur de type topologie cartésienne est vivement conseillé surtout pour la décomposition 2D.

TP2 — OpenMP — Nid de boucles à double dépendance

Objectif

Paralléliser le noyau de calcul suivant à l'aide d'OpenMP.

```
! Boucles avec double dependance
do j = 2, ny
   do i = 2, nx
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   end do
end do
```

Enoncé

On vous demande de partir du noyau de calcul en version séquentielle. Il devra être parallélisé à l'aide d'OpenMP.

1. En utilisant la méthode du pipeline.
2. *Optionnel :* En utilisant la méthode des hyperplans
3. Faites une courbe d'extensibilité de votre/vos version(s) parallèle(s).

TP3 — OpenMP — HYDRO

Objectif

Paralléliser une application à l'aide d'OpenMP.

Enoncé

On vous demande de partir de l'application HYDRO version séquentielle pour construire une version parallèle OpenMP *Fine-grain*, mais avec une seule et unique région parallèle.

TP4 — Hybride MPI et OpenMP — Barrière de synchronisation

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*.

TP5 — Hybride MPI et OpenMP — HYDRO

Objectif

Paralléliser une application à l'aide de MPI et d'OpenMP.

Enoncé

1. Intégrer les changements apportés à HYDRO dans les travaux pratiques précédents afin d'obtenir une version hybride du code.
2. Comparer les performances obtenues avec les différentes versions. L'extensibilité est-elle bonne ?
3. Quelles améliorations peuvent être apportées pour obtenir de meilleures performances ? Faites des essais et comparez.