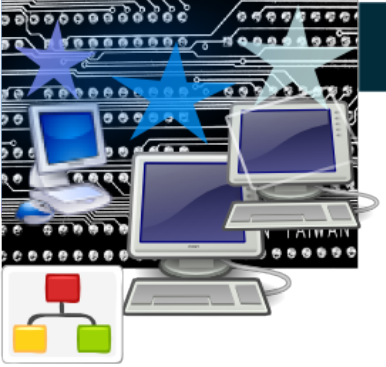


Programmation parallèle sur mémoire distribuée 3ème partie

– MASTER MIHPS –

Camille Coti[1]
camille.coti@lipn.univ-paris13.fr

[1] Université de Paris XIII, CNRS UMR 7030, France

Plan du cours

Performances d'une machine

Qu'est-ce qui fait les performances d'une machine ?

Une machine parallèle est composé de nombreux paramètres

- Des CPU
- Une quantité de mémoire par CPU, avec une certaine vitesse d'accès
- Des réseaux

Comment mesurer la performance ?

Performance globale

- Utilisation de benchmarks
  - Représentatifs d'un certain type d'utilisation

Performances de chaque caractéristique

- Ensemble de benchmarks spécialisés
  - Plusieurs résultats séparés

Performances globales de la machine

Application typique de calcul

- LINPACK : résolution de système
- NAS Parallel Benchmarks suite
  - FT : résolution d'équations aux dérivées partielles en 3D en utiliant une transformée de Fourier
  - CG : plus petite valeur propre en suivant la méthode du gradient conjugué
  - EP : génération indépendante de variables aléatoires
  - BT : mise sous forme tridiagonale par blocs
  - IS : tri d'un ensemble d'entiers

Caractéristiques mises en valeur

Chaque benchmark a certaines caractéristiques

- LINPACK
  - Communique peu, n^3 opération pour une occupation mémoire en n^2
- Les NAS permette de mettre en valeur une caractéristique
  - CG communique souvent avec des petits messages (latency-bound)
  - BT communique peu avec des gros messages (bandwidth-bound)
  - EP ne communique pas (embarrassingly parallel)

Performances réseaux

Caractéristiques hardware

Chaque machine dispose d'un ou plusieurs réseaux

- Qui ont des performances propres
  - Latence
  - Bande passante
- Une certaine topologie physique
- Un certain nombre de cartes réseaux
- Une certaine rapidité d'accès aux cartes réseaux
  - Rapidité du bus (PCI, PCI-eXpress...)
  - Concurrence d'accès (nb de processus par carte réseaux)

Mesure

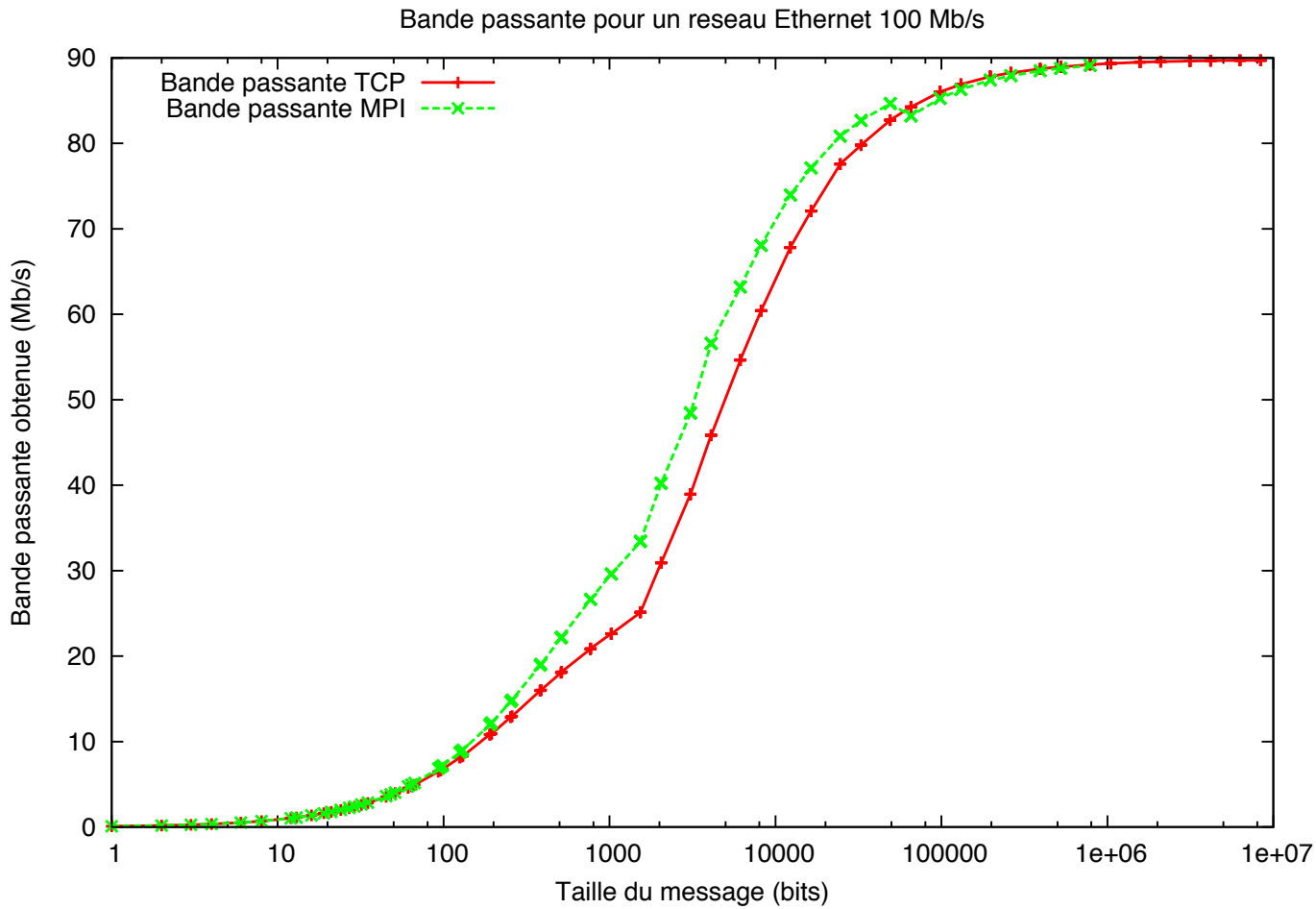
On mesure la latence et la bande passante

- Benchmark spécifique (exemple : NetPipe)

Dépend également du logiciel de communications

- Mesure "raw TCP", MPI

Mesure de performances réseau avec NetPipe



Mesure de performance CPU

Mesure de calcul séquentiel !

On calcule la puissance de chaque CPU individuellement

- Noyau de calcul intensif
  - Racine de 2, π, multiplication de matrices optimisée...

Rpeak

Performance maximale *théorique* de la machine

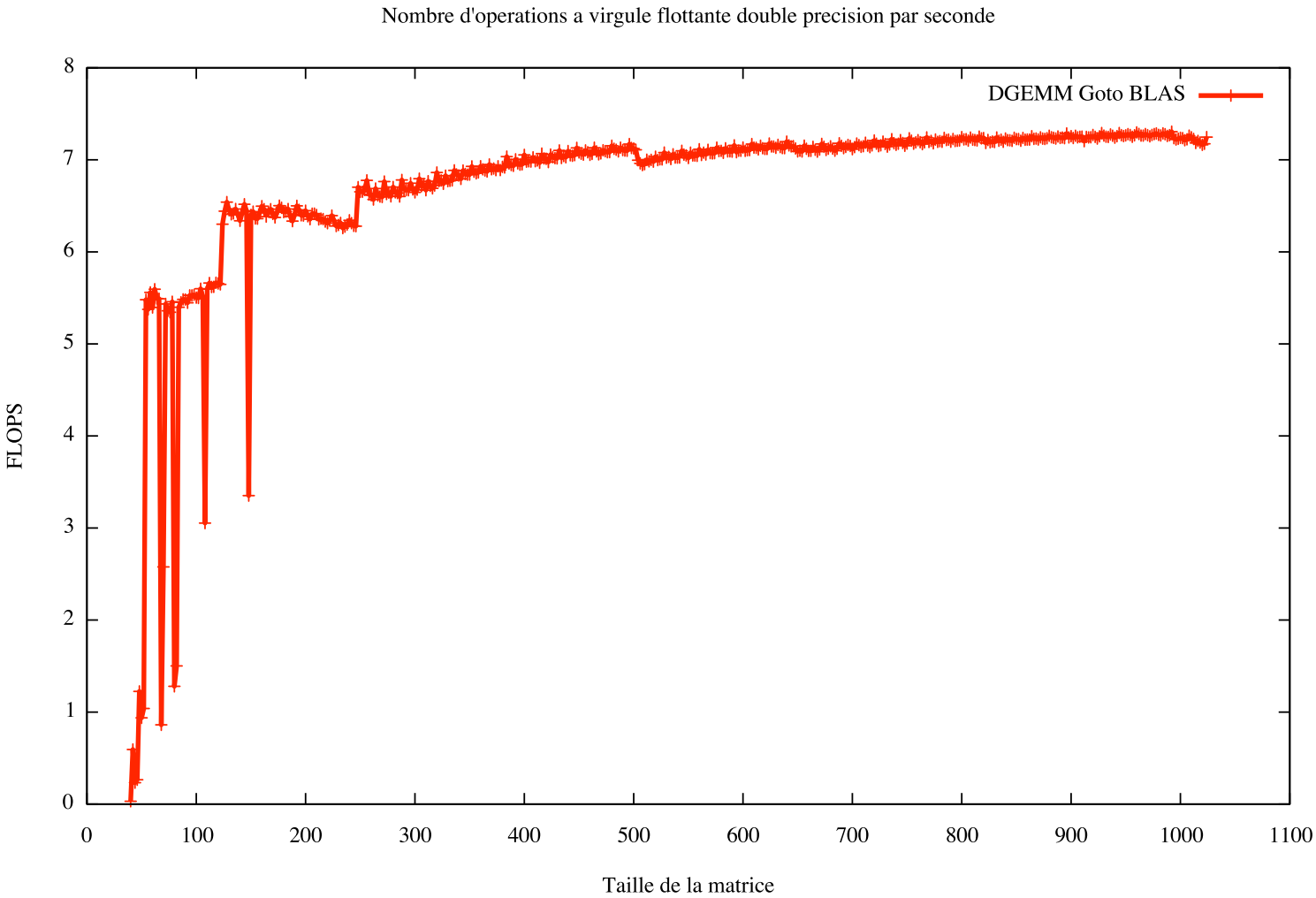
- Nombre total d'opérations par unité de temps
  - Nombre d'opérations effectuées par un processeur par cycle
  - $\times$ la fréquence
  - $\times$ le nombre de CPU total

Performance maxi de la machine

Sur un calcul qui ne communique pas, si tous les CPU sont exploités à fond :

$$Perf_{tot} = \sum_{i=1}^{N} Perf(CPU_i)$$

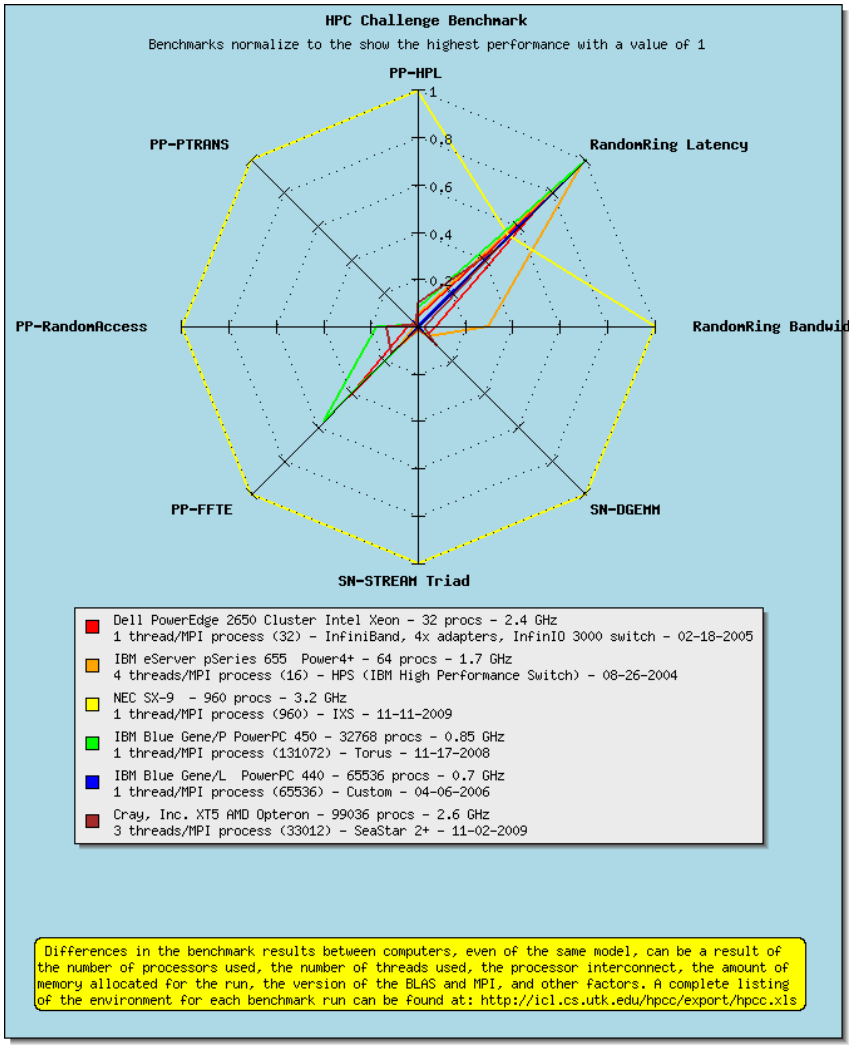
Mesure de performance CPU avec DGEMM



HPCC

Un ensemble standardisé de benchmarks spécifiques

- HPL : rapidité de calcul en virgule flottante
- DGEMM : rapidité de calcul en en virgule flottante double précision
- FFT : rapidité de calcul en en virgule flottante double précision
- STREAM : bande passante mémoire
- RandomAccess : rapidité d'accès à des zones aléatoires de mémoire
- PTRANS : communications par paires de processus
- latence et bande passante effectives



Quels benchmarks choisir ?

Benchmark global

Un seul chiffre

- Facile à appréhender
- Mais est-ce représentatif ?
  - Machines optimisées pour LINPACK...

Benchmarks spécifiques

Plusieurs chiffres

- Plus précis
- Mais plus difficile à appréhender

Le débat est ouvert !

Architecture des machines actuelles

Tendance architecturale : tout hybride

Clusters de multi-cœurs

- On rassemble plusieurs puces par carte et plusieurs cœurs par puce
  - Cray XT5m : 12 cœurs par nœud...
- Gain de place, d'énergie, de chaleur...
  - Pour avoir 128 cœurs:
    - 6 blades de 16 cœurs
    - ou 128 blades (4 armoires) de 1 cœur?

Architecture de la mémoire

Les accès mémoire sont inégaux

- Mémoire partagée entre les cœurs d'un nœud
- Mémoire distribuée entre les nœuds

Programmation hybride

Un type de memoire $\leftrightarrow$ un modèle de programmation

Mémoire partagée entre les cœurs

- Utilisation d'OpenMP

Mémoire distribuée entre les nœuds

- Utilisation de MPI

Répartition du calcul

Un processus par nœud

- Plusieurs threads par processus
- Une seule entité sur le communicateur MPI_COMM_WORLD
- Attention à la thread-safety !

Exemple

Une machine

- 32 nœuds
- 16 cœurs par nœud

Exécution de programmes hybrides

Écriture de programmes hybrides

Utilisation des `#pragma omp`

Compilation

Compiler l'application MPI en activant OpenMP avec l'option `-fopenmp` (pour gcc, sinon `-xopenmp`, `-mp`...)

- `mpicc -o toto toto.c -fopenmp`

Exécution

Attention au nombre de processus exécutés par machine

- Écriture du machinefile en conséquence
  - `host.domain.fr slots=1`

Exemple : Hello World hybride

Travaux pratiques 1

Écriture d'un hello world multi-threadé

- Chaque thread doit afficher son rang MPI, le nombre de processus MPI, son rang dans les threads OpenMP et le nombre de threads OpenMP
- Exemple :
  `hello world from thread 2/4 on process 1 among 3`

Indications

- Attention à la variable d'environnement OMP_NUM_THREADS
  - Propagration avec Open MPI : `mpiexec -x OMP_NUM_THREADS=4`
- Attention à la cohérence de la mémoire partagée
  - Quelles variables sont privées, quelles variables sont publiques ?
  - Barrière OpenMP : `#pragma omp barrier`

Gestion des communications

Qu'en dit la norme MPI

Pas grand chose !

- Tous les threads peuvent communiquer en MPI
- Attention aux collectives : il n'est pas permis que plusieurs threads effectuent en même temps une communication collective sur un même communicateur

Donc : agir avec précaution

- Attendre que tous les threads aient terminé leur calcul avant de commencer la communication
  - Barrières OpenMP
- Ne pas oublier qu'on a plusieurs threads par processus
  - Tout le monde ne communique pas !
  - `if( 0 == thread_id )`

Exemple : calcul d'une somme globale

Travaux pratiques 2

Implémenter un programme hybride OpenMP / MPI qui calcule la somme globale des carrés d'un ensemble d'entiers situés dans un tableau. Les entiers du tableau sont générés aléatoirement et compris entre 0 et 100. Tous les threads de tous les processus doivent disposer du résultat final.

Indications

- Procédez par étapes : d'abord l'allocation, puis le remplissage du tableau, puis les sommes locales des carrés...
  - Attention à la cohérence de la mémoire
  - Il n'y a rien de trivial en environnement multithreadé
- Utilisation de OpenMP pour calculer la somme *locale* (*i.e.*, sur le nœud)
- Utilisation d'une communication collective MPI pour calculer la somme des sommes locales
  - Laquelle ?
  - Attention à la façon dont elle est appelée (par qui ?)
- Attention à la cohérence de la mémoire !
  - `#pragma omp critical`

Exécution d'une tâche par un seul thread

Directives OpenMP

On dispose pour cela de deux directives :

- `SINGLE` : un seul thread exécute la tâche, généralement le premier arrivé à cet endroit du code
- `MASTER` : seul le thread maître exécute la tâche, généralement celui de rang 0

Exemple `SINGLE`

```
#pragma omp parallel shared( tab )
{
#pragma omp single
  tab = (int*) malloc( tabsize * sizeof( int ));
}
```

Exemple `MASTER`

```
#pragma omp parallel
{
#pragma omp master
    printf("coucou c'est moi\n");
}
```

Question

Quand va-t-on utiliser `MASTER` pour synchroniser les threads en vue d'une communication ? Quand va-t-on utiliser `SINGLE` ?

Avantages de la programmation hybride

Alternatives

On pourrait utiliser du MPI pur

- MPI est capable de communiquer entre deux cœurs
  - En utilisant un segment de mémoire partagée
  - En utilisant TCP sur la boucle locale
- MPI permet de contrôler ses schémas de communication
  - Où est la donnée ? Dans un cache distant ?
    - Placement optimal avec MPI, pas forcément avec OpenMP
    - Politiques "first hit", "first touch"... pas forcément optimales

OpenMP permet de hiérarchiser ses schémas de communication

- Utilisation d'un bus mémoire
  - Concurrence d'accès au médium de communication entre les cœurs
- Concurrence d'accès à la carte réseau
  - 1 processus vs n processus en concurrence : évitement de l'engorgement
- La hiérarchisation est parfois indispensable en milieu hiérarchique
  - MPI Applications on Grids: a Topology Aware Approach, *C. Coti, T. Herault, F. Cappello*, EuroPar'09
  - QR Factorization of Tall and Skinny Matrices in a Grid Computing Environment, *E. Agullo, C. Coti, J. Dongarra, T. Herault, J. Langou*, IPDPS'10

Pourquoi tolérer les défaillances ?

Comportement de l'application en cas de défaillance

Si un processus d'une application MPI termine son exécution avant le MPI_Finalize()

- Terminaison de l'application
- Les processus s'envoient un signal de terminaison et quittent leur exécution

Conséquences :

- Perte du calcul !

Tolérance aux défaillances

Tolérer les défaillances, c'est

- Continuer son exécution malgré l'apparition de défaillances
  - On ne perd pas l'intégralité du calcul effectué jusque là
  - On continue d'avancer, au moins au bout d'un certain temps
- Utiliser une implémentation MPI spécifique
  - Comportement qui n'est pas dans la norme MPI

Pourquoi il est indispensable de tenir compte des défaillances

Fréquence des défaillances

Chaque composant du système a un *temps moyen avant défaillance* (MTBF) qui lui est propre. Causes des défaillances :

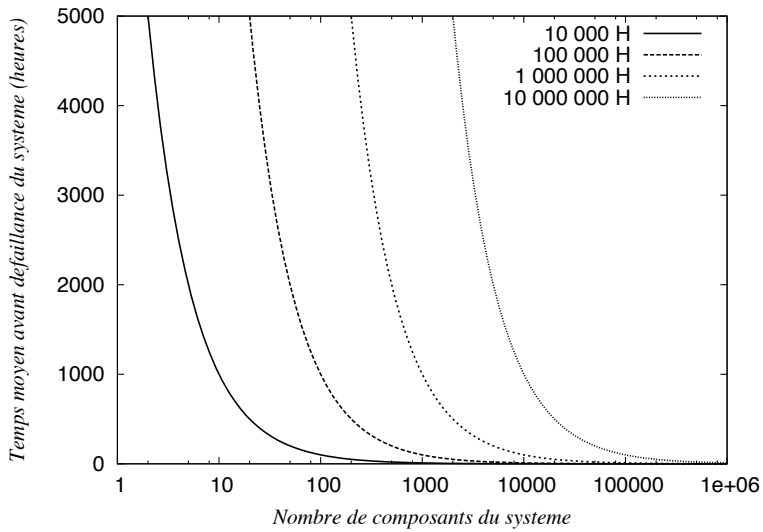
- Disque dur
- Mémoire vive
- Surchauffe : panne ventilateur, capteur, surchauffe d'un voisin...
- Bug logiciel (scheduler...), etc

Le MTBF d'un système est égal au plus petit MTBF de ses composants.

Un peu de statistiques

Chaque composant i du système a un MTBF noté $MTBF_i$

$$MTBF_{total} = (\sum_{i=0}^{n-1} \frac{1}{MTBF_i})^{-1}$$



Approches de la tolérance aux défaillances

Approche transparente

La bibliothèque MPI prend en charge la tolérance aux défaillances

- L'application n'est pas modifiée → *transparence*
- L'application MPI est traitée comme n'importe quel système distribué
  - Application des principes de l'algorithmique répartie

Approche dirigée par l'application

L'application prend en charge la tolérance aux pannes

- Utilisation d'un environnement d'exécution et de communication supportant ces fonctionnalités
- Non transparent : l'application doit s'adapter, rétablir son calcul
- Ad-hoc : adapté à une application donnée, donc plus efficace

Retour sur point de reprise

Le point de reprise

Enregistrement de l'état d'un processus

- Mémoire virtuelle du processus
- Compteur de programme
- Compteur d'instructions
- Registres

Enregistrement de l'état d'un processus séquentiel

Enregistrement de son état à un instant donné (checkpoint)

- Rétablissement de cet état plus tard
  - Par exemple si le processus a été interrompu

Exemple : Berkeley Lab Checkpoint/Restart

Bibliothèque pour checkpointer à partir du processus ou outils externes en ligne de commande

- `cr_checkpoint PID`
- `cr_restart fichier`

Retour sur points de reprise d'une application distribuée

Modèle de système distribué

Un système distribué est constitué

- D'un ensemble de processus $\{P_0, ..., P_{N-1}\}$
- De canaux de communications reliant ces processus

État d'un système distribué

L'état d'un système distribué est constitué de :

- L'ensemble des états de ses processus
- Du contenu de ses canaux de communications
- Du contenu des files d'attente de ses canaux de communications

Bibliographie

- Introduction to Distributed Algorithms, *Gerard Tel*, Cambridge University Press, ISBN 0521794838
- Self Stabilization, *Shlomi Dolev*, MIT Press, ISBN 0262041782

Restaurer l'état d'un système distribué

De programme séquentiel au système distribué

Un système distribué n'est pas simplement un ensemble de processus séquentiel

- Canaux de communications entre les processus
- Communications (*i.e.*, interactions) entre les processus

On ne peut pas se contenter de prendre des checkpoints : il faut tenir compte des interactions entre les processus.

Définition : État cohérent

Un état d'un système distribué est dit *cohérent* si il s'agit d'un état dans lequel il peut arriver au cours d'un calcul n'étant pas touché par une faute.

Restauration de l'état d'un système distribué

On restaure alors l'état de façon à ce que le système se retrouve, à la fin de l'exécution du protocole de restauration d'état, dans un état cohérent.

Coupe cohérente

Définition : coupe d'un système distribué

Une *coupe* d'un système distribué est la collection d'un ensemble d'états des processus le constituant.

Définition : coupe cohérente d'un système distribué

Une *coupe cohérente* d'un système distribué est une coupe telle que l'ensemble des états constitue un état cohérent de ce système.

Concrètement

Si on trace une ligne entre les états pris dans la coupe

- Aucun message ne traverse la ligne
- La coupure respecte les dépendances causales entre les processus

Retour arrière sur points de reprise coordonnés

Points de reprise sur la coupe cohérente

La coupe cohérente forme une *ligne de recouvrement*

- Si les processus reviennent en arrière dans l'état de la coupe cohérente, alors le système est remis dans un état cohérent

Prise de points de reprise sur cette coupe cohérente

- Algorithme de Chandy et Lamport, 1985 : vague de checkpoints

Retour en arrière lorsqu'une défaillance survient

- Tous les processus retournent en arrière sur le dernier checkpoint
  - L'ensemble du système revient dans un état cohérent

Avantages et inconvénients

Simple à mettre en œuvre

- Au niveau de l'implémentation
- Faible complexification de l'exécution
  - Le protocole intervient peu dans l'exécution

Mais retour arrière global

- Perte du calcul effectué depuis la dernière vague de checkpoints
- Pas scalable, inadapté à des fréquences de défaillances élevées

Implémentation de l'algorithme de Chandy-Lamport

Prise de la vague de checkpoints

Matérialisation avec la circulation d'un marqueur

- Sur réception du marqueur :
  - Prise du checkpoint local
  - Transmission du marqueur
- Fin de la vague quand on a reçu tous les marqueurs des autres processus

Et les messages qui traversent la vague ?

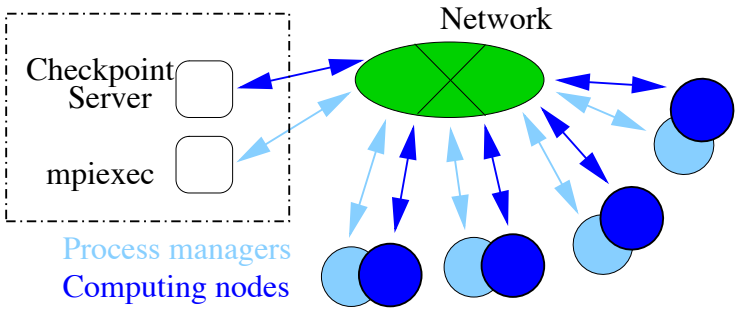
Deux solutions :

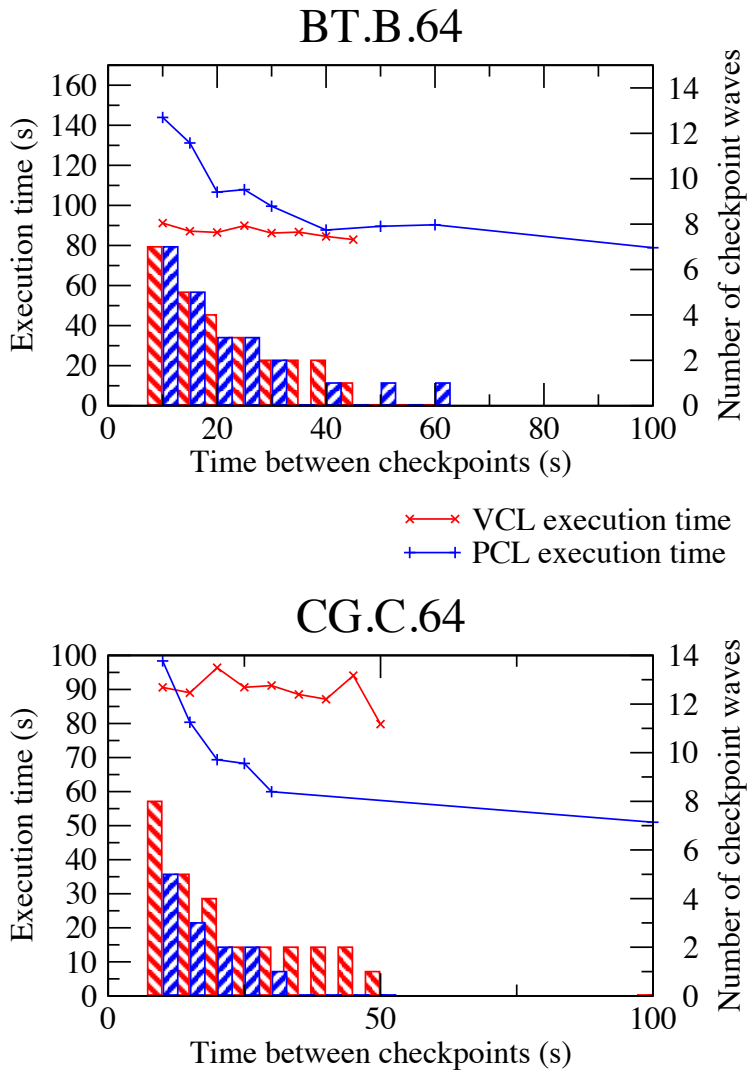
- Implémentation bloquante : on les bloque jusqu'à la fin de la vague
- Implémentation non-bloquante : on les sauvegarde

Comparaison des deux implémentations

Implémentation dans MPICH-V

- Ajout d'un serveur de checkpoints pour stocker les checkpoints
- La vague est initiée par le mpiexec
- Expérimentation sur réseaux rapides





Conclusions

Coût de la sauvegarde des messages vs coût du blocage

Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI, *C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, F. Cappello*, SC'06.

Déterminisme d'un système distribué

Définition : Déterminisme

Un programme est dit déterministe si, à partir d'un état initial donné, on arrive toujours au même état final.

Évenements non-déterministes

Les évenements non-déterministes pouvant survenir auprès d'un processus d'une application distribuée sont les interactions avec le reste du monde

- Entrées-sorties

Piecewise Deterministic Asumption

Si on rejoue les évenements non-déterministes d'un programme déterministe par morceaux (*piecewise deterministic*), alors à partir d'un état initial donné, on arrive toujours au même état final.

Prise des checkpoints non-coordonnée

Traitement indépendant de chaque processus

Chaque processus prend ses checkpoints de façon indépendante

- Les évenements non-déterministes survenant entre deux checkpoints sont enregistrés
  - On parle de *message-logging*
  - Effectué par l'émetteur : sender-based message-logging
  - Plus rarement, receiver-based message-logging

Retour en arrière après une défaillance

Seul le processus touché par la défaillance retourne en arrière

- Les autres continuent leur exécution
- Les messages reçus par ce processus entre la prise du checkpoint et la défaillance sont rejoués

Il finit par retrouver l'état dans lequel il etait quand la défaillance est survenue

- Sans influence sur les autres processus
- Le système retrouve un *état cohérent*

Sauvegarde et rejeu des messages

Mémoire de canal

Implémentation naïve d'une sauvegarde de messages

- Ne passe pas à l'échelle
- Performances non satisfaisantes : nécessité d'une mémoire de canal par processus

Sauvegarde des messages dans les processus

Alternative : on sauvegarde localement les messages et on ne sauvegarde à distance que les informations pour rejouer les messages.

- Sender-based : l'émetteur conserve le message dans sa mémoire
  - C'est lui qui enregistre les informations de causalité entre les *émissions* de messages
- Receiver-based : le receveur conserve le message dans sa mémoire
- Les protocoles sender-based sont les plus courants

Les messages en mémoire sont enregistrés avec les checkpoints.

Causalité des évenements

Enregistrement des informations de causalité des évenements

Le rejeu des messages reçus est conditionné par l'ordre causal de ceux-ci. On sauvegarde donc les informations de causalité entre les messages

- Sauvegarde optimiste
  - Les informations sont sauvegardées sur le support distant et on envoie les message avant d'avoir reçu l'acquittement
  - On fait l'hypothèse qu'il n'y aura pas de défaillance entre temps
- Sauvegarde pessimiste
  - Les informations sont sauvegardées sur le support distant et on n'envoie le message qu'une fois l'acquittement reçu
  - Le protocole assure que lorsqu'un message est émis, ses informations de causalité sont enregistrées
- Sauvegarde causale
  - Tant que la sauvegarde des informations de causalité n'a pas été acquitée, celles-ci sont conservées dans le système en étant envoyées avec les messsages

Comparaison des trois familles de protocoles

Comparaison théorique

- Sauvegarde optimiste
  - N'assure pas la cohérence de l'état en cas de panne au mauvais moment
- Sauvegarde pessimiste
  - Ajoute une latence supplémentaire
- Sauvegarde causale
  - Augmente la taille des messages → diminue la bande passante disponible

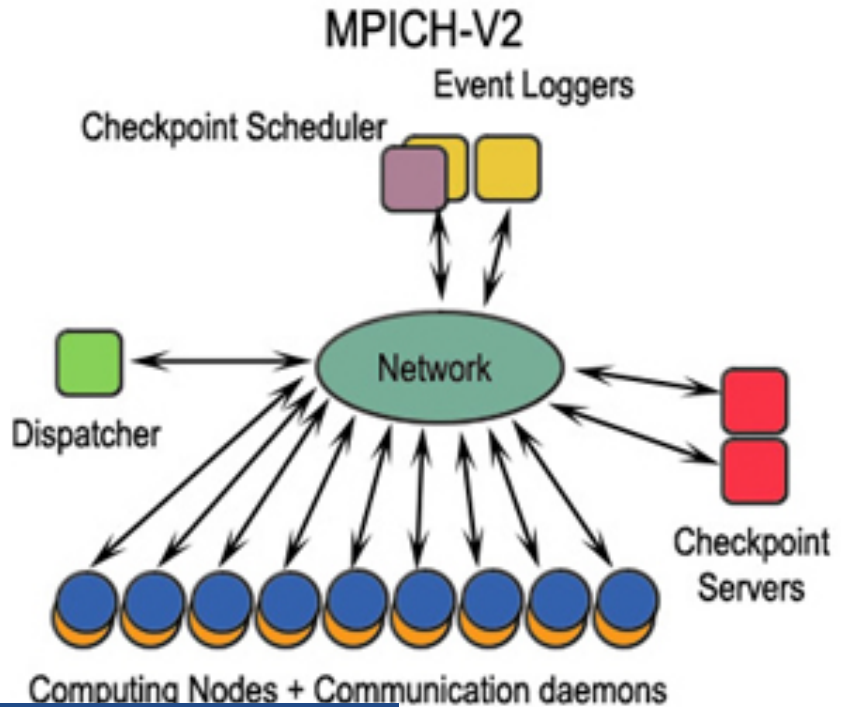
Implémentation dans MPICH-V

Utilisation de composants supplémentaires

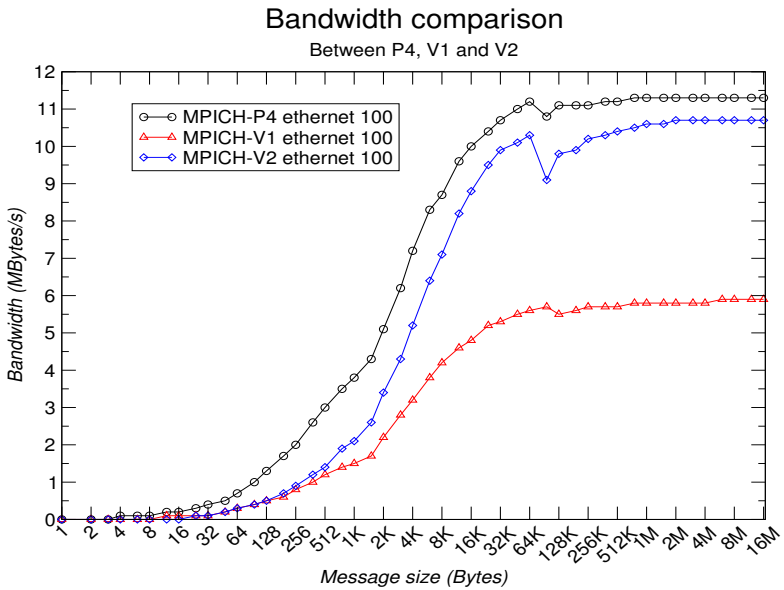
- Serveur de checkpoints
- Enregistreur d'évennements

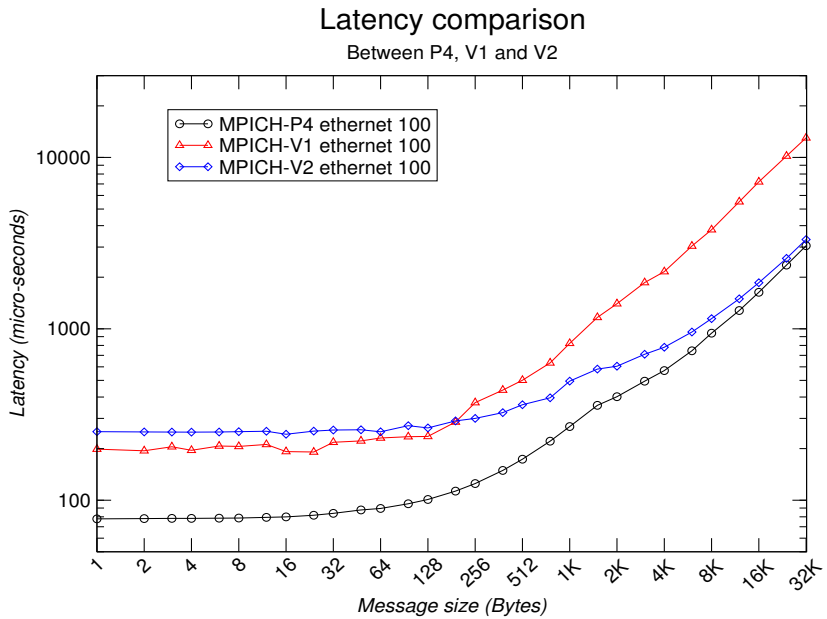
Comparaison des performances

- En l'absence de défaillances
- En cas de défaillances



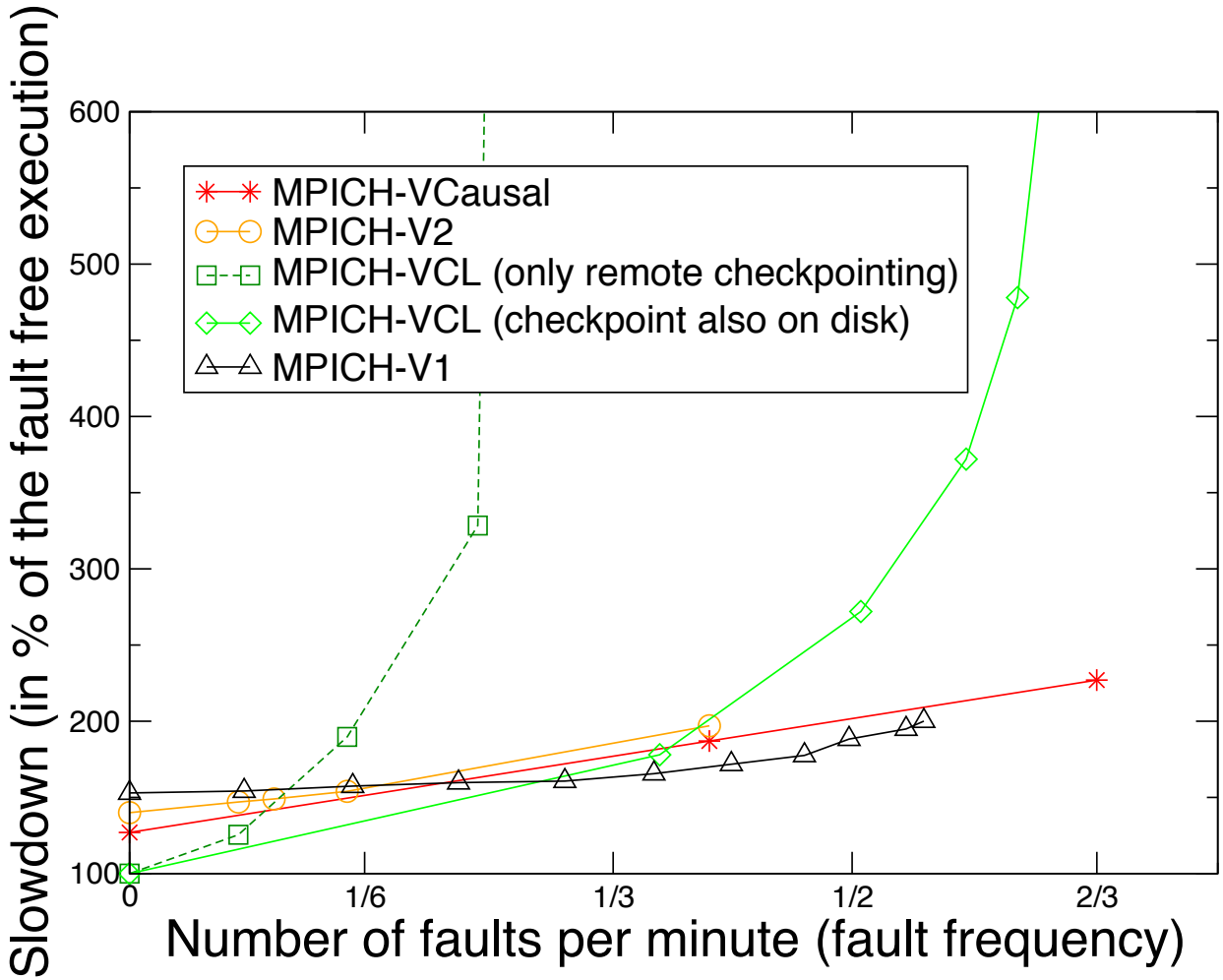
Impact des protocoles sur la bande passante et la latence





MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging, *A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, F. Magniette*, SC'03

Comparaison des protocoles



MPICH-V Project: a Multiprotocol Automatic Fault Tolerant MPI, *A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, F. Cappello*, IJHPCA 2005

Comparaison entre les procoles

En l'absence de défaillances

Les protocoles coordonnés donnent de meilleures performances à petite échelle

- Mise en œuvre *simple*
- Peu d'influence sur le chemin critique de l'exécution
  - Le protocole intervient uniquement au moment de la prise de checkpoints
  - Les protocoles non-coordonnés ont un surcoût permanent sur les communications (latence, bande passante)
- Mais la coordination a son prix à grande échelle

En présence de défaillances

Les protocoles non-coordonnés font perdre moins de temps de calcul au moment du retour en arrière

- Seul le processus victime de la défaillance retourne en arrière
- Attention cependant aux synchronisations ultérieures présentes dans l'application (communications...)

Tolérance aux défaillances guidée par l'application

Approche non-transparente

L'application définit le comportement à suivre en cas de défaillance

- Relancer un nouveau processus pour remplacer le processus mort ?
  - Comment rétablir son état, retrouver sa place dans le calcul parallèle ?
- Continuer avec un processus en moins ?

Nécessité d'une implémentation MPI spécifique

Comportements non prévus dans la norme MPI (au moins 1.x et 2.x)

- Nécessité de fournir cette interface à l'application
- Nécessité pour le middleware de survivre à la panne
  - Et éventuellement de permettre au système de cicatriser (mise en place d'algorithmes self-healing)

Exemple

FT-MPI

- Basé sur HARNESS
  - Environnement d'exécution tolérant aux défaillances
  - Infrastructure self-healing
- La bibliothèque fournit une interface permettant à l'application de définir son comportement en cas de défaillance

Sémantique

Ce que dit la norme MPI

En cas de défaillance d'un ou plusieurs processus, le communicateur MPI_COMM_WORLD devient *invalide*.

Extension fournie par FT-MPI

États du communicateur MPI_COMM_WORLD

- Les deux états prévus par la norme MPI {valide, invalide} deviennent {FT_OK, FT_DETECTED, FT_RECOVER, FT_RECOVERED, FT_FAILED}

États de chaque processus

- Chaque processus peut alors être dans un état parmi {OK, Unavailable, Joining, Failed}

Manipulation du communicateur MPI_COMM_WORLD

Comportement défini en cas de défaillance

Choix d'un comportement parmi quatre :

- *SHRINK* : on réduit la taille du MPI_COMM_WORLD pour avoir N-1 processus avec un nommage continu
- *BLANK* : on laisse un vide dans le MPI_COMM_WORLD ; on a alors N-1 processus avec un nommage qui n'est plus continu
- *REBUILD* : on relance le processus victime de la défaillance et il reprend sa place dans le MPI_COMM_WORLD
- *ABORT* : terminaison de l'application (comportement défini par la norme MPI)

Implémentation

Utilisation d'un *error handler* qui définit ce comportement

- `MPI_Errhandler_create ( recover, &errh );`
- `MPI_Errhandler_set ( MPI_COMM_WORLD, errh );`

ou vérification de la valeur retournée par chaque fonction MPI

- `if( MPI_ERR_OTHER == MPI_Recv( ...) )`

Récupération après une défaillance

État des processus

L'état de chaque processus est dans un attribut du communicateur

- Obtenu avec `MPI_Comm_get_attr`
- Deux attributs
  - `FTMPI_ERROR_FAILURE`
  - `FTMPI_NUM_FAILED_PROCS`

Reconstruction du communicateur MPI_COMM_WORLD en fonction du comportement souhaité.

Relancement d'un processus

Vérification de la valeur retournée par `MPI_Init`

- `if( MPI_INIT_RESTARTED_NODE = MPI_Init( &argc, &argv ) )`

Suivant ce qui est retourné, on sait si on est exécuté pour la première fois ou on vient d'être relancé.

Exemple d'algorithme itératif tolérant aux défaillances

Ajout de processus

On a une grille de processus $m \times n$.

- Ajout d'une ligne de processus supplémentaires

Calcul de chaque itération

Le calcul est effectué sur les $m \times n$ originaux.

- On calcule un CRC sur les processus de chaque ligne et on met le résultat dans le processus supplémentaire de cette ligne

En cas de défaillance

Si un processus meurt, on peut rétablir l'état dans lequel il était à la fin de l'itération précédente

- En utilisant le CRC de ligne et les processus de sa ligne toujours vivants

Robustesse

On tolère *une défaillance* par ligne en utilisant cet algorithme.

- On peut améliorer sa robustesse en utilisant également une colonne de processus qui recevront un CRC

FT-LA

Fault Tolerant Linear Algebra

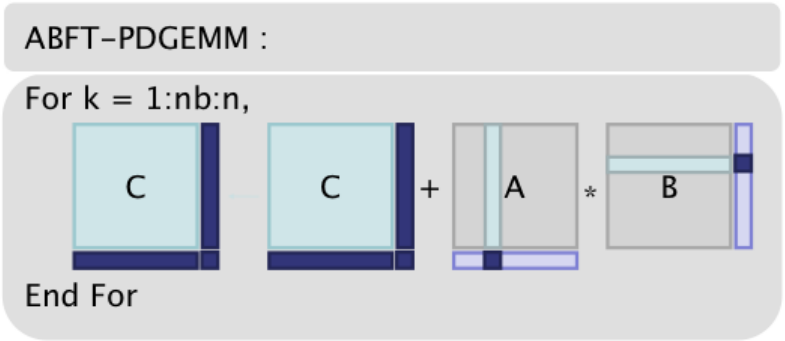
Ensemble de noyaux de calcul implémentant la tolérance aux défaillances dirigée par l'application

- Utilise FT-MPI
- Implémentation de la plupart des noyaux de calcul d'algèbre linéaire dense
- Utilisations de propriétés des algorithmes de calcul

Exemple : multiplication matrice-matrice

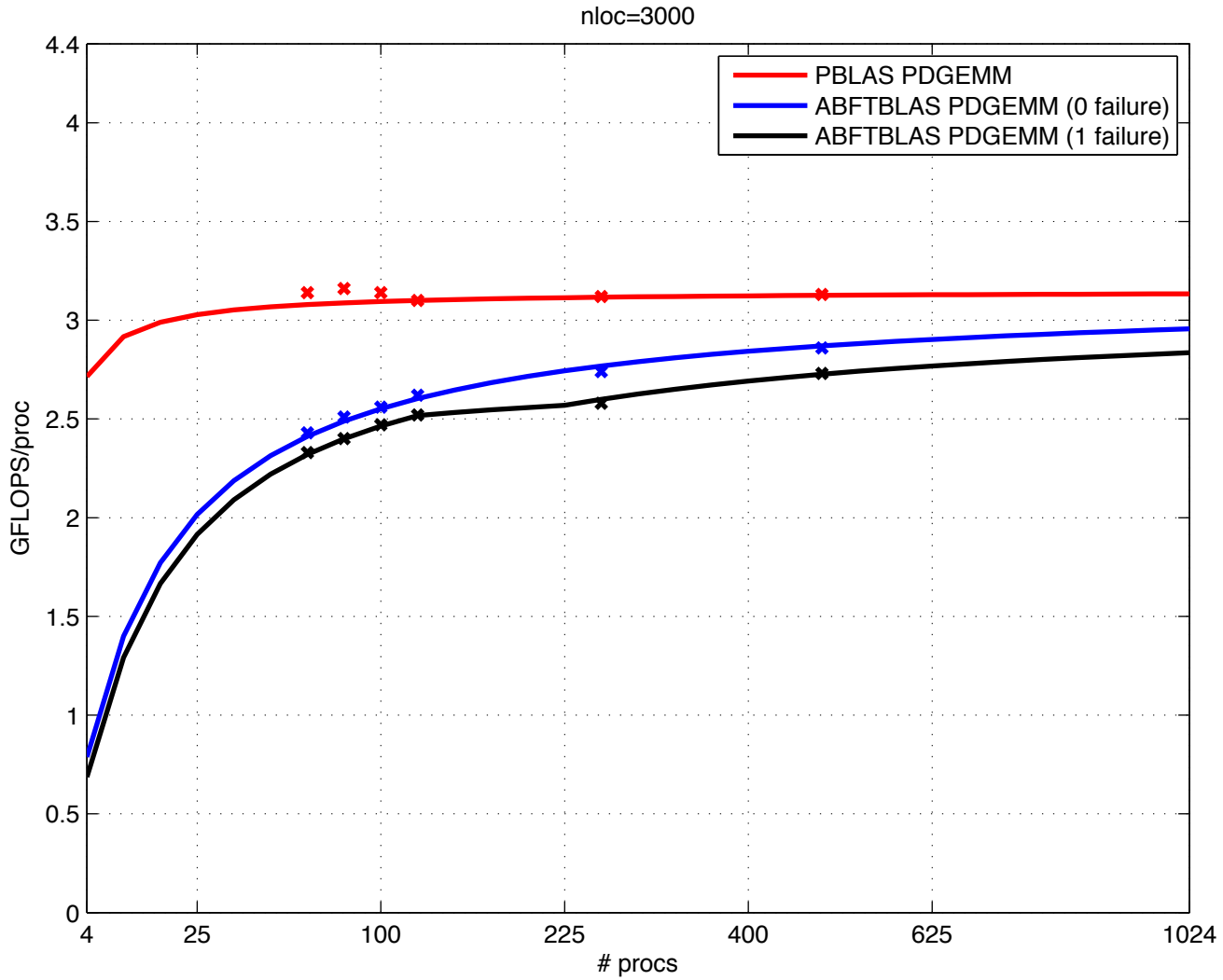
Préconditionnement de la matrice pour utiliser des données redondantes sur des processus supplémentaires

- Multiplication utilisant l'algorithme "classique"
- Réduction utilisant les données redondantes de ligne et de colonne en cas de défaillance



Algorithm-based fault tolerance applied to high performance computing, *G. Bosilca, R. Delmas, J. Dongarra, J. Langou*, JPDC 2009

Performance de ABFT : multiplication matrice-matrice



Algorithm-based fault tolerance applied to high performance computing, *G. Bosilca, R. Delmas, J. Dongarra, J. Langou*, JPDC 2009

Comparaison

Plusieurs protocoles et approches à choisir suivant la situation

Échelle du système, fréquence des pannes, communications (fréquentes ou pas, grosses ou petites)

- L'approche coordonnée a moins d'impact sur l'exécution en absence de défaillances
- L'approche non-coordonnée ralenti moins l'application après une défaillance
- L'approche guidée par l'application est gagnante dans les deux dans

Transparence

L'approche transparente est *portable*

- Pas de nécessité de transformer l'application
- Suit la norme MPI

Une application écrite pour FT-MPI ne pourra pas être exécutée par une autre implémentation

Prise en compte de la tolérance aux défaillance

Élément incontournable aux échelles actuelles

Ce n'est pas simplement un objet d'étude marginal

À venir

Prise en compte de la tolérance aux pannes dans MPI 3

- Groupe de travail sur la normalisation de l'interface pour l'application
- Nécessité d'implémentations la supportant
  - Infrastructures auto-cicatrisantes
  - Bibliothèques de communications mettant à jour l'état des processus