Contribution à l'amélioration de méthodes d'optimisation de la gestion mémoire dans le cadre du HPC

Sébastien Valat[1,2] sous la responsabilité de Marc Pérache[1] et William jalby[2]

[1] *CEA,DAM,DIF F-91927 Arpajon, FRANCE*

[2] *Université de Versailles Saint-Quentin en Yvelines, Versailles, France*

1 mars 2012

energie atomique • energies alternatives

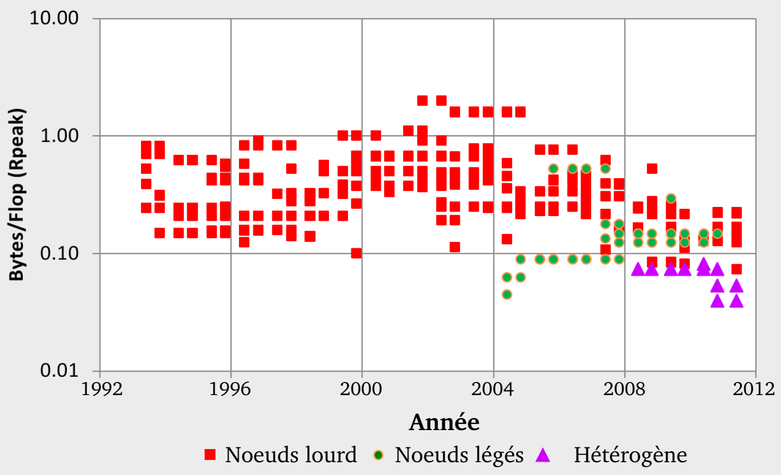
UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

1 Le contexte : parallélisme croissant

Depuis 2005, les gains de **performances** ne se font plus par des augmentations de **fréquence**, mais par multiplication du nombre de **coeurs** présents dans les nouveaux processeurs.

Pour raison de **coût** et de **consommation énergétique**, la quantité de **mémoire** ne peut croitre plus rapidement que le nombre de coeurs. La proportion de **mémoire par coeur** va donc tendre à la **stabilité**, voir à la **baisse**. D'où un besoin de gestion efficace de la ressource mémoire.

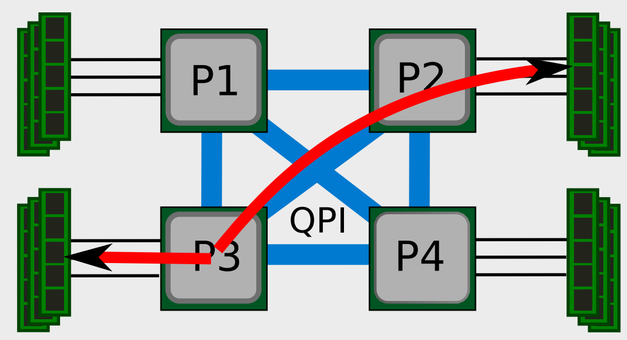
Tendance du ratio mémoire / performance [1]



2 Du mode UMA au NUMA

On ne peut accéder efficacement à une mémoire commune symétrique (**UMA**) avec un nombre trop important de processeurs / coeurs.

- **UMA** : Uniform Memory Access
- **NUMA** : Non Uniform Memory Access



D'où une importance de la **localisation des données**. Les accès distants peuvent causer un ralentissement important.

En mode **multi-thread** le support du **NUMA** remonte de l'OS vers **l'allocateur mémoire** de l'application et vers le **développeur.**

L'interface de l'allocateur ne permet toutefois pas de transmettre des informations explicites quant à ce qui est attendu, il doit donc le deviner au mieux ou étendre cette interface.

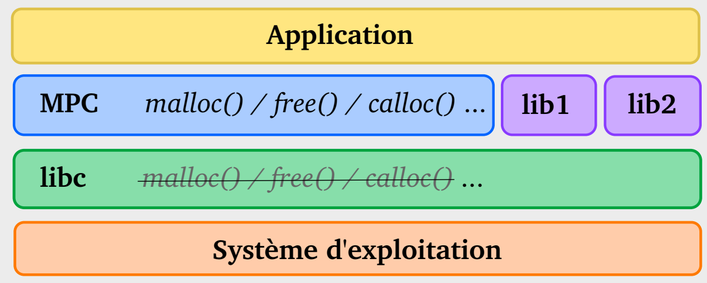
3 Le project MPC : Multi-Processor Computing

MPC[2][3] est un framework visant à supporter les différents modèles de programmation **parallèles** actuels sur un unique runtime. Unification qui permet une meillure collaboration entre ces derniers. MPC offre déjà en standard **MPI** 1.3 et **OpenMP** 2.5.

4 Objectif : l'allocateur mémoire de MPC

La mémoire devient un point critique en terme de **quantité** et de **performance**. Or actuellement l'allocateur de Linux est par exemple non **parallèle** et non **NUMA aware**.

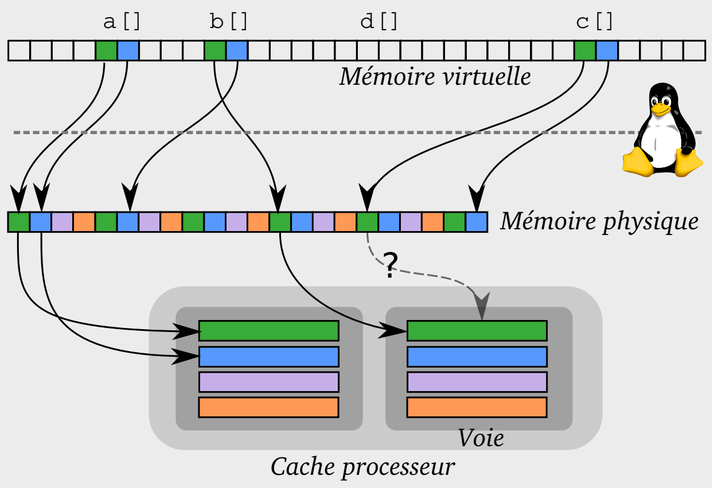
L'objectif est de ré-implémenter un allocateur mémoire **parallèle**, **NUMA aware**, et faisant des choix de **compromis consommation/ performance** compatible avec les contraintes des nouveaux calculateurs.



5 Le cadre technique

Une application manipule des adresses **virtuelles** qui n'ont pas de correspondance directe avec les adresses dans la mémoire **physique**. L'OS et le matériel gèrent l'association entre ces deux espaces d'adressages à la granularité d'une **page** (4 Ko).

C'est justement le rôle de **malloc** de **demander** des pages au système d'exploitation et d'y **placer** les segments de l'utilisateur de tailles rarement multiples de 4 Ko.

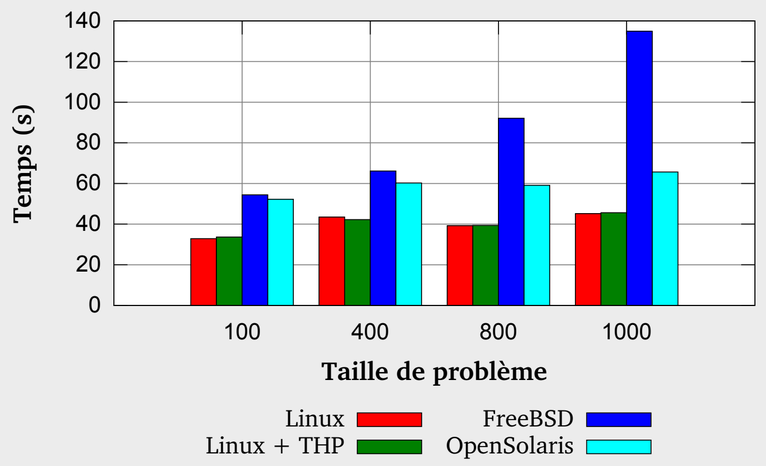


6 Etude préliminaire : interaction des composants

Lors d'une étude préliminaire, nous avons observé des cas de **mauvaise interaction** entre la politique de placement de **malloc** et l'algorithme de **sélection des pages** de l'**OS**. Ce cas de figure est par exemple observé sous FreeBSD avec son allocateur par défaut, ici sur les grosses pages (2M) utilisé par ce système.

Dans les cas observés, un changement de **politique de placement** de l'**allocateur mémoire** permet de résoudre le problème.

EulerMHD, 8 processus MPI , allocateur de l'OS



7 Extention de l'interface : segments utilisateurs

Les librairies et applications peuvent avoir besoin d'allouer des **segments** avec des **propriétés particulières** (mémoire partagée, mémoire verouillée, mémoire allouée par la librairie CUDA...).

Une fois le segment mis en place le développeur ne peut pas demander à malloc() d'y placer des données. Il doit donc le faire lui même et donc **ré-implémenter** un **allocateur**.

Une **extention** est offerte permettant de gérer des **segments utilisateurs**. Ceci peut à terme permettre de garder une **politique cohérente** et centralisée de **consommation mémoire** au sein de l'application. Ou bien **isoler** certaines allocations.

```
struct sctk_alloc_chain chain;

void * buffer = mmap(...);
sctk_alloc_chain_user_init(&chain,buffer,BUF_SIZE);
sctk_alloc_set_default_chain(&chain);

void * ptr = malloc(42);
MyObject * obj = new MyObject;

sctk_alloc_restore_default_chain();

free(ptr);
delete obj;
```

8 Structure interne générale

La structure générale de l'allocateur contient deux composants : une **source mémoire** et des **gestionnaires locaux**. Une réduction des contentions s'obtient en créant un gestionnaire local pour chaque thread. L'avantage de cette approche est de pouvoir ré-utiliser les gestionnaires locaux pour gérer les **segments utilisateurs**.

Pour éviter une dispertion de la mémoire, la source mémoire reste commune au sein d'un noeud **NUMA** en partageant les macros blocs libres (blocs de 2Mo). A ce niveau on pourra également appliquer des **politique dynamiques de libération** en fonction du taux d'utilisation de la mémoire de la machine et mettre en place des **compromis consommation / performance**.

9 Résultat actuel sur une grosse simulation C++

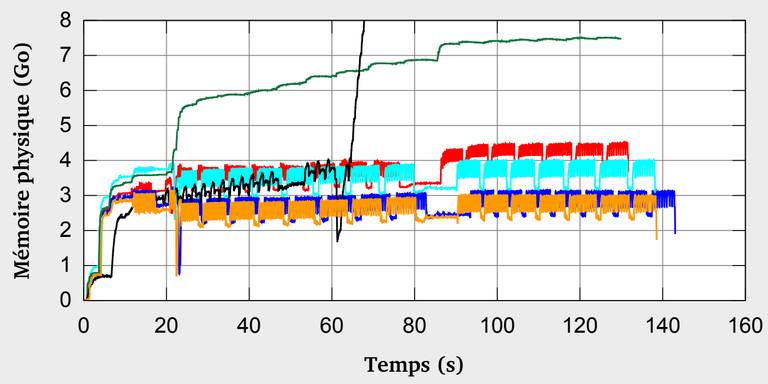
Simulation avec maillage adaptatif, 2 **minutes** de calcule sur **16 threads**, correspondant à **75 millions d'allocations** mémoire.

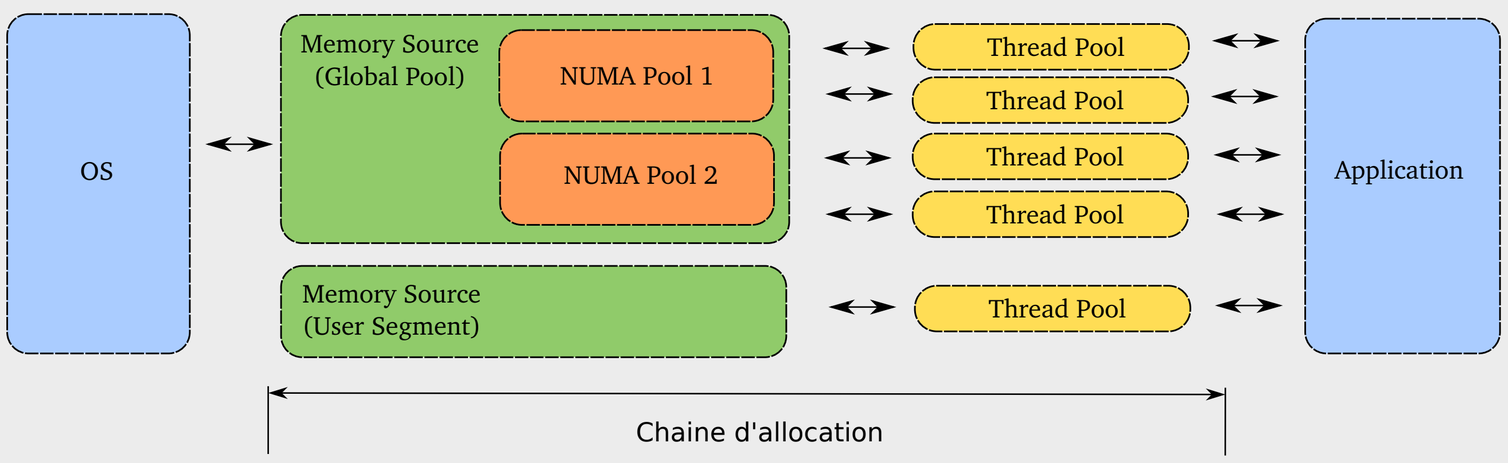
Plusieurs allocateurs sont comparés sur ce cas test :

- **jemalloc** : l'allocateur parallèle de FreeBSD [4].
- **HOARD** : un allocateur parallèle de recherche [5].
- **Glibc** : l'allocateur par défaut de Linux, non parallèle.
- **TCMalloc** : l'allocateur parallèle de google.
- **MPC-2.3.0** : l'allocateur actuel de MPC.
- **MPC-new** : Notre nouvel allocateur.

Parmi ces allocateurs, seul MPC propose un support explicite du **NUMA,** celui de la glibc est le seul non parallèle.

Mémoire physique utilisée sur 8 coeurs





10 Avancement

Faits

- **Etude** des **interactions OS / allocateur / matériel**
- **Implémentation** fonctionnelle de base
- Support des **segments utilisateurs**
- Fonctionnement avec **MPC.**

En cours

- **Optimisation** des routines internes
- Support **NUMA**
- Intégration complète dans **MPC**

A faire

- Politique **adaptative consommation / performance**
- Gestion des **libérations groupées / mémoire partagée**
- Mécanismes d'aide au **débuggage / profiling mémoire**

[1] Using the TOP500 to Trace and Project Technology and Architecture. *Trends, P. M. Kogge, T. J. Dysart.* SC 2011

[2] MPC: A Unified Parallel Runtime for Cluster of NUMA Machines. *M. Pérache, H. Jourdren, R. Namyst*. EUROPAR'08

[3] MPC-MPI; An MPI Implementation Reducinf the Overall Memory Consumption. MPC-MPI: *M. Pérache, P. Carribault, H. Jourdren*. EUROPVM/MPI'09

[4] Hoard : a scalable memory allocator for multithreaded applications. *E. D. Berger, K. S. McKinley, R. D. Blumofe, Paul R. Wilson*. ASPLOS-IX

[5] A Scalable Concurrent malloc(3) Implementation for freeBSD. *J. Evans*.