Passage à l'Échelle Mémoire et Impact des Allocations Dynamiques dans l'Application GYSELA

Fabien Rozar

Passage à l'Échelle Mémoire et Impact des Allocations Dynamiques dans l'Application GYSELA

Fabien Rozar [1]

Maison de la simulation, CEA Saclay, FR-91191 Gif sur Yvette, FRANCE
fabien.rozar@cea.fr

RÉSUMÉ. *Les simulations gyrocinétiques nécessitent d'importants moyens de calcul. Jusqu'à présent, le code semi-Lagrangien* GYSELA *réalise des simulations sur quelques dizaines de milliers de cœurs de calcul (65k cœurs). Mais pour comprendre plus finement la nature de la turbulence des plasmas, nous devons raffiner la résolution de nos maillages, ce qui fera de* GYSELA *un candidat sérieux pour exploiter la puissance des futures machines de type Exascale. Le fait d'avoir moins de mémoire par cœur est une des difficultés majeures des machines envisagées pour l'Exascale. Cet article porte sur la réduction du pic mémoire. Il présente aussi une approche pour comprendre le comportement mémoire d'une application utilisant de très grands maillages. Ceci nous permet d'extrapoler dès maintenant le comportement de* GYSELA *sur des configurations de type Exascale.*

ABSTRACT. *Gyrokinetic simulations lead to huge computational needs. Up to now, the Semi-Lagrangian code* GYSELA *performs large simulations using up to 65k cores. To understand more accurately the nature of plasma turbulence, finer resolutions are necessary which make* GYSELA *a good candidate to exploit the computational power of future Extreme scale machines. Among the Exascale challenges, the less memory per core is one of the most critical issues. This paper deals with memory management in order to reduce the memory peak and presents a general method to understand the memory behavior of an application when dealing with very large meshes. This enables us to extrapolate the behavior of* GYSELA *for expected capabilities of Extreme scale machines.*

MOTS-CLÉS : *Exascale, Passage à l'échelle mémoire, Réduction de l'empreinte mémoire, Allocation dynamique, Physique des plasmas.*

KEYWORDS: *Exascale, Memory Scalability, Memory footprint, Dynamic allocation, Plasma physics.*

1. Introduction

Depuis plusieurs années, la fréquence des processeurs n'augmente plus. A défaut de doubler la vitesse des processeurs tous les 18–24 mois, le nombre de cœurs par nœud de calcul augmente maintenant à un rythme similaire. Les architectures parallèles récentes présentent un modèle de mémoire hiérarchique et une des tendances de ces machines est d'offrir de moins en moins de mémoire par cœur. Cette tendance est identifiée comme l'un des challenges pour l'utilisation de l'Exascale (Shalf *et al.*, 2011) et est l'une des raisons de cette étude.

Durant la dernière décennie, la simulation de la turbulence de plasmas de fusion dans les Tokamak a impliqué un nombre croissant de personnes venant des mathématiques appliquées et de la programmation parallèle (Åström *et al.*, 2013). Ce type d'application fait partie des candidats qui seront capables d'utiliser la première génération de machines Exascale. Le code GYSELA exploite déjà efficacement les capacités des super-calculateurs (Latu *et al.*, 2011). Dans cet article, nous nous intéressons en particulier à sa consommation mémoire. C'est un point crucial pour la simulation de cas physiques plus conséquents avec une quantité de mémoire toujours limitée par cœur.

Un module spécifique à l'application GYSELA a été développé pour permettre la génération d'une trace mémoire. Néanmoins, notre but final (pas encore atteint) est de définir une méthodologie et une bibliothèque portable pour aider le développeur à optimiser l'utilisation de la mémoire pour un certain type d'applications scientifiques parallèles.

Le but du travail présenté ici est de décomposer et de réduire l'empreinte mémoire de GYSELA pour améliorer son passage à l'échelle du point de vue mémoire. Nous présentons un outil qui permet de visualiser les allocations/désallocations mémoire de GYSELA dans un mode hors-ligne. Un autre outil nous permet de prédire le pic mémoire en fonction de certains paramètres d'entrée. Cela est utile pour vérifier si les besoins mémoire futurs des simulations pourront tenir dans l'espace mémoire d'une machine donnée. L'amélioration de l'extensibilité mémoire de GYSELA nécessite l'introduction d'allocations dynamiques. Une étude de leur impact sur le temps de calcul est menée. Le travail présenté par cet article fait suite à (Rozar *et al.*, 2014 ; 2015).

Cet article est organisé selon le plan suivant. La section 2 décrit le domaine d'application de GYSELA et donne une description succincte du code. La section 3 présente l'empreinte mémoire de GYSELA. La section 4 présente le module implémenté pour générer le fichier de traces des allocations/désallocations. Elle illustre aussi les capacités des outils de visualisation et de prédiction pour traiter les données du fichier de traces. La section 5 montre un exemple de réduction de l'empreinte mémoire et une étude de l'extensibilité mémoire grâce à l'outil de prédiction. La réduction de l'empreinte mémoire nécessitant l'utilisation d'allocations dynamiques pouvant potentiellement dégrader les performances de l'application, la section 6 est dédiée à l'étude de l'impact des allocations dynamiques dans GYSELA. La section 7 conclut et présente nos perspectives.

2. Domaine applicatif et vue d'ensemble de GYSELA

2.1. La fusion par confinement magnétique

Les besoins en énergie dans le monde sont de plus en plus importants et les énergies fossiles s'épuisent d'année en année alors qu'elles représentent actuelle la principale source énergétique. Dans ce contexte, la fusion nucléaire controlée est une solution possible. En effet, dans le cas où le processus de fusion serait maîtrisé, cela permettrait de produire de l'énergie de façon pérenne et plus sûre. Contrairement à la fission, la réaction de fusion ne peut pas s'emballer. Cette réaction est aussi plus propre car elle produit beaucoup moins de déchets radioactifs.

Néanmoins, le défi est de taille car les conditions de pression et de température qui permettent d'obtenir des plasmas adéquats mais aussi la réaction de fusion sont difficiles à mettre en œuvre. Un des moyens d'étudier la fusion dans les plasmas est d'utiliser un confinement par champ magnétique dans une enceinte fermée. Un type d'installation adapté est appelé *tokamak* (voir figure 1). Au CEA Cadarache, à l'Institut de Recherche sur la Fusion par confinement Magnétique (IRFM) [1], les scientifiques réalisent des expériences grâce au tokamak Tore-Supra. Ces expériences permettent de mieux connaître les mécanismes de fusion et les effets du confinement magnétique sur le plasma. Elles consistent principalement à tester différentes configurations de la machine, à collecter un grand nombre de grandeurs physiques, puis à les dépouiller et les interpréter. Le tokamak ITER [2], de plus grande taille, permettra bientôt d'optimiser la configuration et d'être à même de produire plus d'énergie que le tokamak n'en consomme.

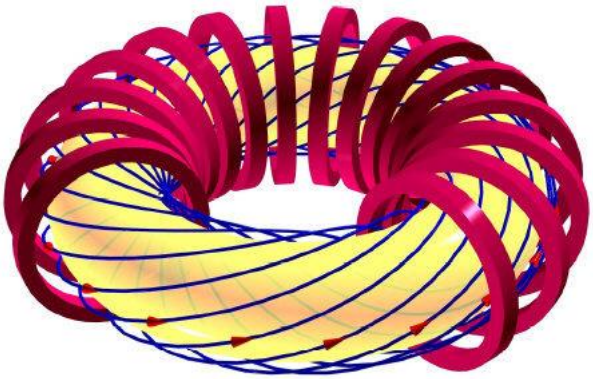

Figure 1. Schéma d'un tokamak

1. http://irfm.cea.fr/
2. http://www.iter.org/

La figure 1 donne une illustration de la forme torique d'un tokamak [3]. De façon usuelle pour se repérer, on distingue deux directions dans un tokamak. On parle de la *direction toroïdale* lorsqu'on considère la direction le long du grand rayon du tore et de la *direction poloïdale* lorsqu'on se restreint à une section circulaire perpendiculaire à la direction toroïdale. Les anneaux qui se succèdent à équidistance dans la direction toroïdale sont des bobines qui génèrent un champ de confinement magnétique le long de la direction toroïdale. Les plans qui contiennent les différentes bobines sont appelés *plans poloïdaux*. Les lignes qui s'enroulent autour du tore en suivant des trajectoires hélicoïdales représentent les lignes de champs magnétiques qui confinent efficacement les particules et retiennent le plasma éloigné des parois internes du tokamak (non représentées sur le schéma). Un des défis majeur du contrôle de la fusion par confinement magnétique est d'éviter l'endommagement de ces parois internes même si les conditions de température et de flux de puissance sont extrêmes. Des recherches sont toujours en cours pour trouver de bons matériaux pouvant faire face au plasma.

De façon complémentaire à l'expérimentation, la compréhension du comportement du plasma dans les tokamaks est abordée par l'approche théorique. La modélisation de la *physique des plasmas* permet d'appréhender l'évolution temporelle du plasma grâce aux équations (*i*) de Vlasov (Vlasov, 1945) et (*ii*) de Maxwell (Maxwell, 1865 ; Darrigol, 2005). Ces études amont cherchent à caractériser les phénomènes de transport turbulent et les instabilités qui ont lieu au sein du plasma. Les équations considérées n'admettent pas de solution analytique dans le cas général. Pour néanmoins les résoudre, de nombreux codes de simulation numérique ont été développés.

2.2. Vue d'ensemble de GYSELA

Dans le cadre de la physique des plasmas de tokamak, l'équation de Vlasov s'écrit

$$\frac{\partial f}{\partial t} + \mathbf{v}\frac{\partial f}{\partial \mathbf{x}} + \frac{q}{m}(\mathbf{E} + \mathbf{v} \wedge \mathbf{B})\frac{\partial f}{\partial \mathbf{v}} = 0$$

où q, m, $\mathbf{E}$ et $\mathbf{B}$ représentent respectivement la charge d'une particule du plasma, la masse d'une particule, les champs électrique et magnétique ressentis par le plasma. La variable f désigne la *fonction de distribution* qui est l'inconnue de l'équation. Cette fonction de distribution est à 7 dimensions : 3 pour la position en espace $\mathbf{x}$, 3 pour les coordonnées de vitesse $\mathbf{v}$ et 1 pour le temps t. Les coordonnées de position et de vitesse donnent une position dans l'*espace des phases*. A un temps t donné, la valeur $f(\mathbf{x}, \mathbf{v}, t)$ correspond au nombre moyen de particules dont la position et la vitesse se trouvent respectivement dans un élément de volume de taille d$\mathbf{x}$ dans l'espace ordinaire et de taille d$\mathbf{v}$ dans l'espace des vitesses centrés respectivement autour de la position ($\mathbf{x}$, $\mathbf{v}$).

Dans sa formulation générale, la résolution de l'équation de Vlasov est coûteuse en termes de temps de calcul pour des domaines de calcul de taille réaliste. Pour que les

3. Cette illustration provient du site : http://fusionnucleaire.e-monsite.com/pages/tpe/fonctionnement-d-une-centrale-iter/confinement-magnetique.html

simulations soient réalisables dans un temps raisonnable, des modèles qui réduisent la taille du problème ont été développés. Le travail effectué dans cet article concerne le code de simulation GYSELA qui résout le couplage non linéaire des équations Vlasov et Maxwell dans la modélisation *gyrocinétique*. L'hypothèse gyrocinétique permet, notamment à l'aide d'un changement de variables lentes/rapides, de supprimer une des dimensions de vitesse et de réduire ainsi d'une dimension le problème. A l'échelle de temps simulé par GYSELA, le champ magnétique **B** est considéré comme statique, ce qui permet de n'avoir à calculer uniquement que le champ électrique **E** à chaque itération. Cette approche est qualifiée d'*électrostatique*.

Dans le cadre de l'application GYSELA (Grandgirard *et al.*, 2006) le solveur de Vlasov modélise le mouvement des ions dans un tokamak; il utilise une méthode semi-Lagrangienne. Le solveur champs (Maxwell) calcule le champ électrostatique qui se traduit en l'application d'une force sur les ions; il se réduit sous les hypothèses considérées, à la résolution numérique d'une équation de type Poisson (Grandgirard *et al.*, 2008).

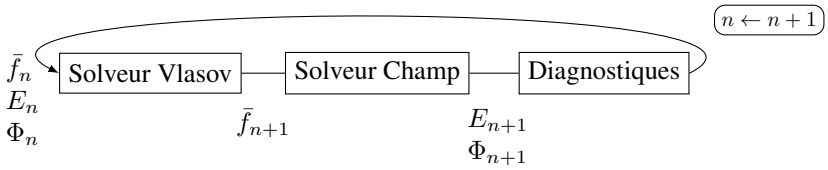

Figure 2. Schéma numérique d'un pas de temps de GYSELA

Dans le modèle gyrocinétique, l'inconnue principale est une fonction de distribution $\bar{f}$ qui représente la densité des centres guides à une position donnée de l'espace des phases. Dans la théorie gyrocinétique, les centres guides désignent les positions autours desquelles les particules du plasma sont en rotation rapide dans le plan poloïdale comparé à leur vitesse de déplacement dans la direction toroïdale. L'exécution de GYSELA se décompose en (*i*) une phase d'initialisation, (*ii*) des itérations temporelles et (*iii*) une phase de sortie. La figure 2 illustre le schéma numérique utilisé durant une itération temporelle. Au n-ième pas de temps, $\bar{f}_n$ représente la fonction de distribution, Φ_n le potentiel électrique et E_n le champ électrique qui correspond à la dérivée spatiale de Φ_n. L'étape Vlasov réalise l'évolution de $\bar{f}_n$ sur un pas de temps et l'étape du solveur champ calcule E_n. Périodiquement, GYSELA exécute des diagnostiques afin d'extraire des quantités physiques à partir de $\bar{f}_n$, E_n et sauvegarde les résultats dans des fichiers HDF5 (Folk *et al.*, 1999).

La fonction de distribution des centres guides $\bar{f}$ est mise en œuvre grâce à une variable à 5 dimensions qui évolue avec le temps. Les 3 premières dimensions sont comme dans le cas général des dimensions d'espace. Dans notre cas, nous utilisons le système de coordonnées (r, θ, φ), où r et θ sont les coordonnées polaires dans la coupe poloïdale du tore et φ désigne l'angle toroïdal. Les deux dernières coordonnées modélisent l'espace des vitesses, $v_{\|}$ la vitesse le long d'une ligne de champ magnétique et μ le moment magnétique (équivalent à une vitesse moyenne dans le plan poloïdal).

Soient N_r, N_θ, N_φ, $\mathrm{N}_{v_\parallel}$ respectivement le nombre de points dans chaque dimension r, θ, φ, $v_\parallel$. Dans le solveur de Vlasov, chaque valeur de μ est associée à un ensemble de processus MPI (un communicateur MPI). Dans chaque ensemble, une décomposition en domaine 2D nous permet d'attribuer à chaque processus MPI un sous-domaine pris dans les dimensions (r, θ). Un processus MPI est donc responsable du stockage du sous-domaine défini par $\bar{f}(r = [i_{start}, i_{end}], \theta = [j_{start}, j_{end}], \varphi = *, v_\parallel = *$ [4] $, \mu = \mu_{value})$. La décomposition parallèle est caractérisée par les valeurs $i_{start}, i_{end}, j_{start}, j_{end}, \mu_{value}$ qui sont connues localement. Ces domaines 2D découlent d'une décomposition par bloc classique de la direction r en $\mathrm{p_r}$ sous-domaines, et de la direction θ en p_θ sous-domaines. Le nombre de processus MPI utilisés durant une exécution est égal à $\mathrm{p_r} \times \mathrm{p}_\theta \times \mathrm{N}_\mu$. Le paradigme OPENMP est aussi utilisé au sein de chaque processus MPI (#T threads dans chaque processus MPI) pour exploiter le parallélisme à grain-fin.

Bien que le cadre gyrocinétique réduise d'une dimension la fonction de distribution, les ressources mémoire nécessaires pour une exécution de GYSELA restent très importantes. La section suivante présente l'application GYSELA du point de vue de sa consommation mémoire.

3. Analyse et stratégie de réduction de la consommation mémoire

3.1. Analyse de la consommation mémoire

Une exécution de GYSELA nécessite une quantité de mémoire importante. Durant son exécution, chaque processus MPI est associé avec une valeur de μ (voir section 2) et manipule la fonction de distribution comme un tableau 4D et le champ électrique comme un tableau 3D. Le reste de la consommation mémoire vient en grande partie des tableaux utilisés pour stocker des valeurs pré-calculées, des tampons mémoires MPI pour concaténer des données à l'envoi ou à la réception et des tampons mémoires utilisateurs OPENMP pour calculer des résultats temporaires. Quasiment tous ces tableaux sont alloués durant l'initialisation de GYSELA.

Afin de mieux comprendre le comportement mémoire de GYSELA, on enregistre à chaque allocation (au niveau des instructions `allocate()`) : le nom du tableau, son type et sa taille. En utilisant ces données nous avons réalisé un *strong scaling* illustré par le Tableau 1 (16 threads par processus MPI). D'un point de vue de la mémoire, le *strong scaling* est l'étude de la consommation mémoire d'un programme sur une instance de taille fixe alors que le nombre de nœud de calcul augmente. Sur le cas avec le plus grand nombre de nœuds, la consommation mémoire des structures qui bénéficient d'une décomposition de domaine représente généralement une faible portion de la consommation mémoire globale. La majeure partie de la mémoire est occupée par des structures non distribuées, ce qui n'est généralement pas favorable à l'extensibilité de l'application d'un point de vue mémoire. Si pour une simulation

4. La notation $*$ représente toutes les valeurs d'une dimension donnée.

donnée avec n processus on utilise x Go de mémoire par processus, on peut espérer dans le cas idéal qu'en faisant la même simulation avec $2n$ processus on obtienne une consommation mémoire de $\frac{x}{2}$ Go. Dans ce cas, on dit que l'*extensibilité* mémoire est parfaite. Mais en pratique, ce n'est généralement pas le cas à cause des surcoûts mémoires dus à la parallélisation.

Dans l'application GYSELA, un processus MPI correspond généralement à un nœud de calcul. La consommation mémoire par processus MPI dépend essentiellement du nombre de processus et de la taille du maillage considéré. Pour obtenir le comportement de la consommation mémoire en fonction du nombre de processus, nous avons choisi un maillage assez grand en entrée et nous avons fait varier le nombre de processus MPI comme le montre le Tableau 1.

Tableau 1. Strong scaling: taille des allocations statiques en Go (par processus MPI) et pourcentage de chaque type de donnée par rapport au total alloué

Nombre de cœurs Nombre de processus MPI	$2k$ 128	$4k$ 256	$8k$ 512	$16k$ 1024	$32k$ 2048
structures 4D	209.2 67.1%	107.1 59.6%	56.5 49.5%	28.4 34.2%	14.4 21.3%
structures 3D	62.7 20.1%	36.0 20.0%	22.6 19.8%	19.7 23.7%	18.3 27.1%
structures 2D	33.1 10.6%	33.1 18.4%	33.1 28.9%	33.1 39.9%	33.1 49.0%
structures 1D	6.6 2.1%	3.4 1.9%	2.0 1.7%	1.7 2.0%	1.6 2.3%
Total par processus MPI en Go	311.5	179.6	114.2	83.0	67.5

Le Tableau 1 montre l'évolution de la consommation mémoire en fonction du nombre de cœurs et de processus MPI. Le pourcentage de la consommation mémoire par rapport au total de la mémoire consommée est donné pour chaque type de structure de données. Les dimensions du maillage sont les suivantes : $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$. Ce maillage est plus conséquent que ceux utilisés en production actuellement, mais il correspond aux besoins futurs, spécialement à ceux de la physique en configuration multi-espèces ou avec électron cinétique. Le dernier cas avec 2048 processus requiert 67.5 Go de mémoire par processus MPI. Nous associons habituellement un processus MPI à chaque nœud de calcul. On peut remarquer que la mémoire requise est supérieure au 64 Go d'un nœud d'Helios [5] ou encore au 16 Go d'un nœud d'une Blue Gene/Q. Le Tableau 1 illustre aussi le fait que les structures $2D$ et les structures $1D$ ne bénéficient pas du parallélisme spatial induit par la décomposition de domaine. En fait, le coût mémoire des structures $2D$ ne dépend pas du tout du nombre de processus, mais plutôt de la taille du maillage et du nombre de

5. http://www.top500.org/system/177449

threads. Dans le dernier cas à 32k cœurs, le coût des structures $2D$ est le principal goulot d'étranglement. Il prend 49 % de l'empreinte mémoire totale.

Dans GYSELA, le surcoût mémoire pour les simulations sur un grand nombre de cœurs s'explique pour diverses raisons. De la mémoire est nécessaire par exemple pour stocker des coefficients durant une phase d'interpolation (pour le solveur Semi-Lagrangien de l'équation de Vlasov). Les tampons mémoires MPI constituent aussi des surcoûts mémoires. Les appels aux routines MPI impliquent souvent une copie des données qu'on veut envoyer ou recevoir dans le format approprié. Nous avons réduit certains de ces surcoûts mémoires. Cela a amélioré l'extensibilité mémoire et nous a permis d'exécuter de plus grands cas physiques.

3.2. Stratégie pour réduire l'empreinte mémoire

Il y a au moins deux approches pour réduire l'empreinte mémoire associée à un processus d'une application parallèle. Premièrement on peut augmenter le nombre de nœuds utilisés pour la simulation. La taille des structures qui bénéficient de la décomposition de domaine diminue lorsque le nombre de processus MPI augmente. Deuxièmement, on peut gérer plus finement les allocations des tableaux afin de réduire spécifiquement les coûts mémoire qui ne passent pas l'échelle avec le nombre de threads/processus MPI et aussi limiter leur impact sur le pic mémoire.

Pour parvenir à réduire l'empreinte mémoire et pour repousser le goulot d'étranglement mémoire, nous allons ici nous concentrer sur la seconde approche.

Dans la version originale du code, la majeure partie des variables est allouée durant la phase d'initialisation. Cette approche est justifiée pour les structures qui sont des *variables persistantes* en opposition aux *variables temporaires* qui peuvent être allouées dynamiquement. En se basant sur le Tableau 1, on peut identifier les structures $3D$ et $4D$ comme des variables persistantes, celles qui bénéficient de la décomposition de domaine, alors que les structures $1D$ et $2D$ sont le plus souvent de nature temporaire. La consommation mémoire totale du plus grand cas avec 2048 processus MPI s'éléve à 135 To.

Dans cette configuration on peut, tout d'abord, déterminer rapidement l'espace mémoire requis en exécutant uniquement la phase d'initialisation. Cela permet à l'utilisateur de savoir rapidement si le cas "tient en mémoire" ou pas. Deuxièmement, cela évite les surcoûts en temps d'exécution dus à la gestion dynamique des allocations. Mais dans cette approche les variables utilisées localement dans certaines sous-routines consomment leur espace mémoire durant toute l'exécution de la simulation. Comme l'espace mémoire devient un point critique quand un grand nombre de cœurs est utilisé, nous avons alloué une grande partie de ces variables temporaires avec des allocations dynamiques. Ceci a réduit le pic mémoire et de fait repoussé le goulot d'étranglement mémoire. Néanmoins, l'inconvénient des allocations dynamiques est que nous perdons les deux avantages des allocations statiques, et en particulier la possibilité de déterminer à l'initialisation l'espace mémoire requis pour exécuter une

simulation. La section 6 propose une étude des surcoûts dus à la gestion dynamique de la mémoire.

La section suivante introduit les outils que nous avons développés au sein de GYSELA pour avoir un suivi de la consommation mémoire mais aussi pour nous permettre de faire de la prédiction sur cette consommation mémoire.

4. Outils de modélisation de la consommation mémoire

Pour suivre la consommation mémoire de GYSELA et pour mesurer la réduction de l'empreinte mémoire, des outils complémentaires ont été développés : un module FORTRAN d'instrumentation pour générer un fichier de traces des allocations/désallocations et un script PYTHON de visualisation & prédiction qui exploite ce fichier de traces. Les informations collectées durant l'exécution de GYSELA grâce au module d'instrumentation est un composant clé de notre analyse mémoire.

Une vue d'ensemble des outils d'analyse de performances et de profilage mémoire est donnée en section 4.1. Les sections 4.2–4.4 détaillent en partie l'implémentation des outils que nous utilisons pour notre analyse. La dernière section 4.5 discute de la généralisation et des limitations de notre approche concernant la possibilité de prédire la consommation mémoire d'une application autre que GYSELA.

4.1. Travaux relatifs

Dans la communauté des chercheurs travaillant sur les outils d'analyse de performances d'applications parallèles, différentes approches existent. Souvent ces approches s'appuient sur des *fichiers de traces*. Un fichier de traces collecte des informations durant l'exécution de l'application concernant un ou plusieurs de ses aspects : le temps d'exécution, le nombre de messages MPI envoyés, le temps inoccupé (idle), la consommation mémoire, et plus encore. Pour obtenir ces informations, l'application doit être instrumentée. L'instrumentation peut être faite à 4 niveaux : dans le code source, à la compilation, à l'édition de lien ou durant l'exécution (*just in time*).

Différents outils génériques sont disponibles pour analyser les aspects performance d'un programme. L'outil de performance SCALASCA (Geimer *et al.*, 2010) est capable d'instrumenter le code source à la compilation. Cette approche a l'avantage de couvrir toutes les parties du code et elle permet la personnalisation des informations à récupérer. Cette approche systématique donne une trace complète mais la récupération d'informations dans toutes les sous-routines du code peut induire un surcoût en exécution important.

L'ensemble d'outils EZTRACE (Aulagnon *et al.*, 2013) offre la possibilité d'étudier, entre autres choses, la consommation mémoire en interceptant les appels à l'ensemble des fonctions C qui gère les allocations mémoires (p. ex. `malloc()`, `free()`). Cet outil peut rapidement instrumenter une application grâce à un lien avec des biblio-

thèques de tierce-partie lors de l'édition de lien. Contrairement à notre approche, celle-ci ne nécessite pas une instrumentation du code source.

Les outils PIN (Luk *et al*., 2005), DYNAMORIO (Bruening *et al*., 2003) ou MAQAO (Djoudi *et al*., 2005) produisent une instrumentation durant l'exécution. L'avantage est ici l'aspect générique de la méthode. Tous les programmes peuvent être instrumentés de cette façon, mais contrairement à notre approche, l'instrumentation dynamique introduit souvent un surcoût assez important en temps d'exécution.

Un autre logiciel couramment utilisé pour détecter les fuites mémoires est VALGRIND (Nethercote, Seward, 2007). Ce logiciel utilise une instrumentation lourde au niveau binaire. Cette instrumentation lui permet de récupérer l'ensemble des accès mémoires effectués lors de l'exécution d'un programme. Parmi les outils développés à partir de VALGRIND, l'outil MASSIF (Carrez, 2013) permet de représenter graphiquement la consommation mémoire d'un programme. La Fig. 3 présentée dans la section 4.3 remplit le même objectif, à savoir avoir une vision d'ensemble de la consommation mémoire et situer le pic mémoire. Néanmoins, le réel désavantage de l'instrumentation binaire est le surcoût par rapport au temps d'exécution qu'il introduit, comme cela a déjà été mentionné plus haut.

Concernant plus particulièrement l'étude de la consommation mémoire dans le tas, des travaux similaires existent. Le programme MPROF (Zorn, Hilfinger, 1988) qui s'inspire de GPROF est potentiellement le premier projet à s'intéresser au suivi de la consommation mémoire dans le tas. Cette problématique est néanmoins toujours d'actualité comme on peut le voir avec (Milian, 2014 ; *Gperftool - Heap profiler*, 2009). Le projet PURIFY (Hastings, Joyce, 1991) est plus orienté vers la détection des fuites mémoires et des erreurs d'accès, mais il permet tout de même d'avoir accès au suivi de la consommation. Ces outils fournissent une bibliothèque qui permet d'intercepter les appels aux fonctions d'allocation. Ils se basent donc sur une instrumentation au niveau de l'édition de lien. Le but des outils de suivi mémoire étant de fournir des informations au développeur afin qu'il puisse minimiser la consommation mémoire de son application, des études sur des méthodes de minimisation de l'utilisation du tas ont été réalisées dans (Runciman, Röjemo, 1997) et ses références. Un brevet concernant un processus d'optimisation de la consommation mémoire a même été déposé (Gupta, Rangarajan, 2012).

Parmi toutes les approches ci-dessus, aucune d'entre elle ne proposent d'étude sur l'extensibilité mémoire d'une application en fonction de ses paramètres d'entrées. L'approche de l'instrumentation au niveau binaire demande le moins d'effort de la part de l'utilisateur final, mais cette approche s'appuie sur un programme déjà compilé. Après compilation, il est difficile de faire le rapprochement entre les valeurs que contient la pile d'exécution lors d'une allocation et les variables du code source responsables de cette allocation. C'est un aspect qui est essentiel pour notre outil de prédiction de la consommation mémoire. Le but de l'outil de prédiction développé dans GYSELA est de prédire la consommation mémoire du programme si la valeur de certains paramètres change (p. ex. les dimensions du maillage). La prédiction de la consommation mémoire est inaccessible si on ne peut pas faire le lien entre la taille des

structures allouées et le nom des variables qui donnent les dimensions des allocations. Afin de récupérer les informations nécessaires à la prédiction, notre approche se base sur une instrumentation au niveau du code source qui est à la charge du développeur.

L'outil que nous avons développé nous permet de mesurer les performances de GYSELA, du point de vue mémoire. Un outil de visualisation a été développé pour permettre l'analyse de ce fichier de traces. Il offre une vision globale de la consommation mémoire et une vision précise autour du pic mémoire pour aider le développeur à réduire l'empreinte mémoire. La sortie sur terminal du script de post-traitement donne des informations précises au sujet des tableaux alloués au moment du pic mémoire. Etant donné un fichier de traces, il est possible de prédire exactement la consommation mémoire en fonction des paramètres d'entrée. Ce nouvel outil nous permet d'investiguer l'extensibilité mémoire. Nous ne connaissons pas d'outil équivalent pour modéliser le comportement mémoire.

4.2. Format du fichier de traces

GYSELA fait intervenir différents types de structures de données et afin de gérer leurs allocations/désallocations, un module FORTRAN dédié a été développé pour les enregistrer dans un fichier : la *trace mémoire dynamique*. Une trace est générée par processus MPI. Le processus `0` étant généralement le plus gourmand, notre analyse se concentre généralement sur sa trace.

Le fichier de trace mémoire obtenu grâce au module d'instrumentation est analysé de façon post-mortem. Le module d'instrumentation propose dans son interface les fonctions `take()` et `drop()`, qui enveloppe les appels à `allocate()` et `deallocate()`. Les sous-routines `take()` et `drop()` réalisent respectivement l'allocation et la désallocation d'un tableau manipulé et elles enregistrent cette action dans le fichier de traces.

Pour reconstituer un historique plus précis des allocations mémoire, l'entrée/la sortie de sous-routines que nous avons choisies est enregistrée par l'interface `write_begin_sub()` et `write_end_sub()`. Ceci nous permet de localiser où se produisent les allocations/désallocations, ce qui est un aspect essentiel pour l'étape de visualisation.

Pour chaque allocation et désallocation, le module enregistre le nom du tableau, son type, sa taille et l'expression du nombre d'éléments. L'expression est nécessaire pour pouvoir faire de la prédiction. Par exemple, l'expression associée à cette allocation :

```
integer, dimension(:,:), pointer  :: array
integer                           :: a0, a1, b0, b1
... (intialisation des variables a0, a1, b0, b1)
allocate(array(a0:a1, b0:b1))
```

est

$$(a1 - a0 + 1) \times (b1 - b0 + 1). \tag{1}$$

Pour être capable d'évaluer ces expressions d'allocation en dehors de l'application, les variables qui interviennent doivent être renseignées. Soit la valeur de la variable est fixée ou soit une expression arithmétique dépendant d'autres variables est renseignée. Ces actions sont réalisées respectivement par les sous-routines `write_param()` et `write_expr()` . L'écriture des expressions permet de restituer les liens qui existent entre les différents paramètres. Ils sont essentiels pour que la prédiction soit correcte (cf. section 4.4). L'extrait de code suivant est un exemple de l'enregistrement des paramètres $a0, a1, b0, b1$:

```
call write_param('a0', 1); call write_param('a1', 10)
call write_param('b0', 1); call write_expr('b1', '2*(a1-a0+1)')
```

Le module d'instrumentation se limite actuellement aux tableaux qui contiennent des types de base de FORTRAN : entier, float et complexe. Une des extensions possible du module serait de pouvoir gérer des tableaux de structures.

4.3. Visualisation de la consommation mémoire

Afin d'analyser finement la consommation mémoire, notre outil permet d'identifier les parties du code où l'usage de la mémoire atteint son pic. Le fichier de traces peut être volumineux, plusieurs Méga octets. Un script PYTHON a été développé pour visualiser ces traces. Cet outil aide le développeur à comprendre le coût mémoire des algorithmes qu'il manipule et lui donne des indices sur comment et où il peut appliquer des modifications pour diminuer l'empreinte mémoire. Ces informations sont restituées par deux types de graphiques.

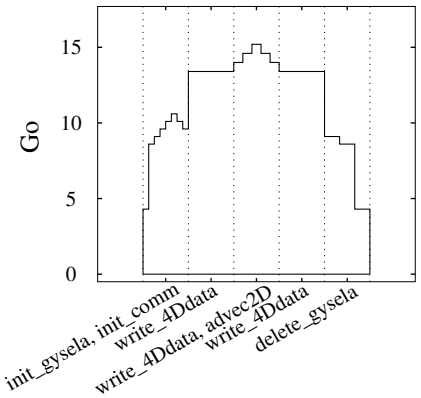

FIGURE *3. L'évolution de la consommation mémoire dynamique durant une exécution très courte de* GYSELA

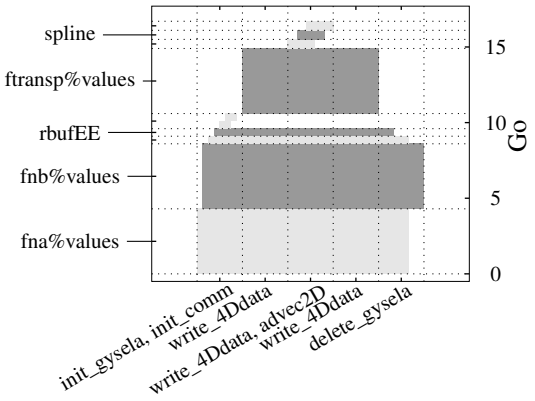

FIGURE *4. Allocations et désallocations des tableaux utilisés dans diverses sous-routines de* GYSELA

La figure 3 trace la consommation de mémoire *dynamique* en Go en fonction du temps. L'axe des abscisses représente les entrées/sorties chronologiques des sous-

routines instrumentées. L'axe des ordonnées donne la consommation en Go. La figure 4 montre dans quelles sous-routines les tableaux sont alloués. L'axe des abscisses reste identique au graphique précédent et l'axe des ordonnées montre les noms des tableaux. L'allocation d'un tableau correspond à un rectangle en gris clair ou en gris foncé dans sa ligne correspondante. La largeur des rectangles dépend de la durée de l'allocation.

Sur la figure 3, on peut localiser dans quelle sous-routine le pic mémoire est atteint. Sur la figure 4, on peut ensuite identifier les tableaux qui sont effectivement alloués quand le pic mémoire est atteint. Grâce à ces informations, nous savons exactement où modifier le code pour réduire le pic mémoire.

4.4. Prédiction du pic mémoire

Pour anticiper nos besoins mémoire avant de lancer une simulation donnée, il est utile de prédire la consommation mémoire pour un jeu de paramètres d'entrée. Grâce aux expressions des tailles de tableau et la valeur ou l'expression des paramètres numériques contenue dans le fichier trace, nous pouvons modéliser le comportement mémoire hors ligne. L'idée ici est de reproduire les allocations avec n'importe quel ensemble de paramètres d'entrée.

Il est possible qu'une valeur d'un paramètre ne puisse pas être exprimée en une ligne d'expression arithmétique (p. ex. une boucle d'optimisation multicritère pour déterminer cette valeur). Pour gérer ce cas, la partie de code FORTRAN qui retourne la valeur dans GYSELA est appelée depuis le script PYTHON. Cela est possible grâce à une compilation du code FORTRAN avec `f2py` (Peterson, 2009).

En changeant la valeur d'un paramètre d'entrée, notre outil de prédiction PYTHON offre la possibilité d'extrapoler précisément la consommation mémoire de GYSELA sur de plus grands maillages ou sur des configurations de super-calculateurs qui n'existent pas encore, p. ex. pour faire une projection à l'Exascale.

Lorsqu'on fait de la prédiction, on peut se demander naturellement quel est le degré de précision atteint. Dans notre approche, pour chaque allocation la trace mémoire renseigne l'expression du nombre d'éléments du tableau alloué mais aussi sa taille effective en octets lors de l'exécution. Grâce à ces deux informations, le script de post-traitement détecte les différences potentielles entre la taille d'un tableau calculée à partir de son expression et sa taille effective enregistrée lors de l'exécution.

Pour valider l'outil de prédiction, nous vérifions les résultats sur différents cas tests. La procédure de vérification est la suivante :

1. fixer un jeu de paramètres d'entrée et lancer la simulation correspondante;
2. récupérer la trace mémoire, *tm1*, associée à ces paramètres;
3. doubler la valeur d'un des paramètres d'entrée (une dimension du maillage);
4. lancer la simulation correspondant à ce nouveau jeu de paramètres;

5. comparer la prédiction du pic mémoire sur la trace *tm1* avec le paramètre d'entrée modifié et la valeur du pic mémoire réel obtenu sur la seconde simulation.

Ces tests nous ont permis de valider l'outil de prédiction dans de nombreuses configurations. La prédiction du pic mémoire correspond exactement au pic mémoire effectif d'une simulation. Les résultats de cet outil sont présentés dans la section 5.2.

4.5. Extensions

L'objet de cet article donc porte sur l'étude de l'extensibilité mémoire de l'application GYSELA. Cependant, les outils qui ont été développés pour y parvenir sont utilisables dans d'autres applications.

Nous travaillons actuellement sur l'implémentation d'une bibliothèque qui regroupe les outils décrits dans cette section et qui permettra surtout d'externaliser les fonctionnalités nécessaires pour modéliser le comportement mémoire d'un programme. Notre approche se base sur l'analyse post-mortem de traces. Le script qui s'occupe du post-traitement ne nécessite pas d'adaptation particulière pour être utilisé sur des traces générées par une autre application. Par contre, le module d'instrumentation nécessite une refonte pour être utilisable sous forme de bibliothèque. Nous souhaitons aussi que cette bibliothèque d'instrumentation soit utilisable par des programmes en langage C ou en FORTRAN, ce qui recouvre de nombreuses applications scientifiques. La création de cette bibliothèque représente un effort d'implémentation qui est toujours en cours.

Notre approche ne permet pas de mesurer toutes les consommations mémoires liées à l'exécution d'une application. Les zones mémoires qui sont allouées typiquement sur la pile (p. ex. les variables locales aux fonctions, les piles locales aux threads) ou les zones allouées par des bibliothèques de tierce-partie utilisées par l'application ne sont pas prises en compte. Seuls les tableaux de certains types primitifs alloués dans le tas grâce à la fonction `allocate()` en FORTRAN sont pris en compte. Néanmoins, ce sont ces allocations qui représentent, pour une grande classe d'applications, la plus grande portion de l'utilisation de la mémoire. Aussi les consommations non prises en compte par le module d'instrumentation restent mesurables de façon indirecte en consultant le champ `Rss` (Resident in size) du fichier système */proc/<pid>/stat* lié au programme considéré (sur les systèmes LINUX).

Mis à part l'effort de développement dû à la création d'une bibliothèque, le suivi de consommation mémoire des tableaux alloués dans le tas serait accessible pour toutes applications en langage C ou en FORTRAN. Par contre, pour bénéficier de la prédiction de consommation mémoire en mode off-line, certaines conditions doivent être satisfaites par les applications cibles. De façon globale, les bornes des tableaux instrumentés dont on souhaite prédire le comportement en mode off-line ne doivent pas dépendre de valeurs produites dynamiquement par l'application (c.-à-d. qui ne peuvent pas être calculée simplement à partir des paramètres d'entrée de l'application).

Nous pouvons capturer efficacement le comportement mémoire des programmes qui utilisent un maillage statique avec une décomposition de domaine. Par statique, nous entendons plus précisément que le nombre de points du maillage ne varie pas en fonction des valeurs calculées durant l'exécution. Par opposition, la prédiction de consommation mémoire des applications qui utilisent des méthodes de raffinement dynamique de maillage n'est, par exemple, pas envisageable dans notre approche.

Dans les cas où des dépendances entre les valeurs liées à une exécution et la consommation mémoire de l'application sont fortes, la prédiction off-line de la consommation mémoire est délicate. Une prédiction du pic mémoire dans le pire des cas sur ces applications pourrait permettre de savoir si l'application pourra tenir en mémoire.

5. Minimisation de l'empreinte mémoire : résultats expérimentaux

Dans cette section nous présentons deux cas d'utilisation des outils que nous avons mis en place pour le suivi de la consommation mémoire. Le premier cas illustre une réduction de l'empreinte mémoire que nous avons obtenue grâce à l'outil de visualisation. La seconde utilisation montre une évaluation de l'extensibilité mémoire de GYSELA accessible grâce à l'outil de prédiction.

5.1. Réduction de l'empreinte mémoire

Réduire l'empreinte mémoire équivaut à diminuer le pic mémoire. Les figures 5 et 6 montrent l'impact sur la mémoire de quelques modifications du code. Après analyse du code, nous avons remarqué qu'au moment du pic mémoire, la variable `ftransp%values` contient les mêmes valeurs que la fonction de distribution `fnb%values` organisées différemment. Nous obtenons le fichier de traces de la figure 6 en désallouant `fnb%values` au moment du pic mémoire. Dans GYSELA, la fonction de distribution est la variable la plus gourmande en ressource mémoire. L'optimisation illustrée sur les figures 5 et 6 nous a permis à elle seule de réduire le pic mémoire de 10% sur certaines simulations sur le plus grand cas à 32k cœurs. Ce type d'optimisation réduit efficacement le pic mémoire mais détériore un peu la lisibilité du code.

L'utilisation de cet outil sur différentes simulations nous a montré que le pic mémoire n'est pas toujours atteint au même endroit de l'application. En fonction de la taille du maillage, du nombre de processus MPI et de threads OPENMP, le pic mémoire se déplace. Ce comportement peut être expliqué par la dépendance entre la taille de certains tableaux caractéristiques et la valeur de certains paramètres d'entrée. Par exemple, la taille des buffers MPI est sensible aux paramètres de parallélisation. Dans GYSELA, les tailles des buffers temporaires sont sensibles au nombre de points dans les directions r et θ.

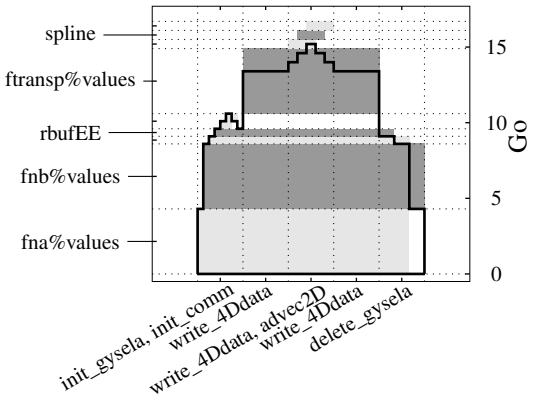

FIGURE *5. Visualisation simplifiée d'une trace mémoire*

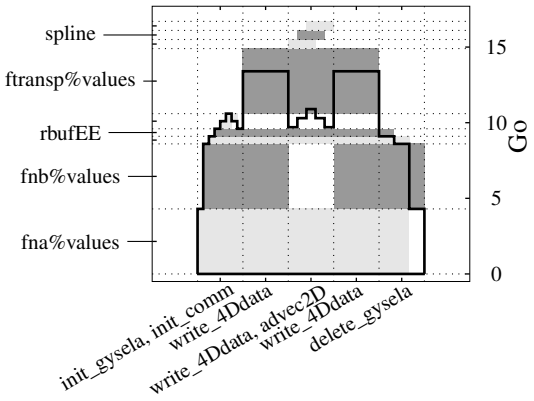

FIGURE *6. Visualisation après optimisation des allocations*

L'outil de visualisation donne un nouveau point de vue sur le code source. Cet outil nous a aidés à réduire en plusieurs passes les surcoûts mémoire et donc à améliorer l'extensibilité mémoire du code GYSELA.

5.2. Étude de l'extensibilité grâce à la prédiction

Le tableau 2 présente le test en *strong scaling* sur une version de GYSELA où un nombre conséquent de tableaux sont alloués dynamiquement, au plus proche de leur utilisation, mais sans toutefois être dans les zones OPENMP. Aussi cette version comporte plusieurs améliorations algorithmiques réalisées grâce à l'outil de visualisation (non détaillées ici). L'outil de prédiction nous permet de reproduire le tableau 1 sur le même maillage, c.-à-d. $N_r = 1024$, $N_\theta = 4096$, $N_\varphi = 1024$, $N_{v_\parallel} = 128$, $N_\mu = 2$.

Tableau 2. Strong scaling : taille des allocations mémoires et pourcentage par rapport au total de chaque type de donnée au pic mémoire

Nombre de cœurs Nombre de processus MPI	$2k$ 128	$4k$ 256	$8k$ 512	$16k$ 1024	$32k$ 2048
structures 4D	207.2 79.2%	104.4 71.5%	53.7 65.6%	27.3 52.2%	14.4 42.0%
structures 3D	42.0 16.1%	31.1 21.3%	18.6 22.7%	15.9 30.4%	11.0 32.1%
structures 2D	7.1 2.7%	7.1 4.9%	7.1 8.7%	7.1 13.6%	7.1 20.8%
structures 1D	5.2 2.0%	3.3 2.3%	2.4 3.0%	2.0 3.8%	1.7 5.1%
Total par processus MPI en Go	261.5	145.9	81.9	52.3	34.3

Le tableau 2 donne la consommation mémoire de différentes simulations au pic mémoire. Pour toutes les simulations, la taille du maillage est fixe. Le nombre de processus MPI est doublé entre deux simulations. L'ensemble des chiffres du tableau 2 ont été obtenu en utilisant l'outil de prédiction. Comme on peut le voir, sur le plus gros cas (32k cœurs), la consommation des structures 2D a été réduite à 20.8%. Aussi, la consommation mémoire sur ce cas a été réduite de **50.8%** par rapport à la version originale (voir tableau 1). Les structures $4D$ contiennent les données les plus pertinentes utilisées durant le calcul et elles consomment la plus grande partie de la mémoire comme on le souhaitait naturellement. Malgré le fait que les chiffres du tableau 2 aient été obtenus uniquement grâce à l'outil de prédiction, nous avons vérifié en pratique que la configuration du cas à 32k cœurs pouvait effectivement s'exécuter sur la machine Helios avec une estimation parfaite de la consommation mémoire effective.

Les surcoûts mémoire de GYSELA ont été globalement réduits grâce à l'utilisation des allocations dynamiques. Cela améliore l'extensibilité mémoire de GYSELA et permet l'exécution de plus grandes simulations. L'utilisation d'allocations dynamiques peut augmenter le nombre d'appels système effectués par l'application, et donc éventuellement la ralentir. La section suivante fait l'étude du potentiel surcoût en temps d'exécution associé aux allocations dynamiques dans GYSELA.

6. L'impact des allocations dynamiques

Comme nous avons pu le voir dans la Section 5, afin de réduire l'empreinte mémoire de l'application GYSELA, nous avons rapproché l'allocation des données de leur utilisation. Pour que cette approche soit la plus efficace possible, nous nous sommes d'abord concentrés sur les allocations les plus pénalisantes concernant l'extensibilité mémoire. Les figures 7 et 8 illustrent les modifications que nous avons opérées dans la gestion des allocations mémoire de GYSELA. La figure 7 montre que l'ensemble des tableaux utilisés par l'application est alloué dans la phase d'initialisation et désalloué à la fin, alors que sur la figure 8 une portion importante des allocations se fait de façon dynamique durant chaque itération.

Bien que cette approche minimise l'empreinte mémoire, elle implique des appels fréquents aux primitives d'allocation (`allocate()` et `deallocate()` en FORTRAN), ce qui peut se révéler pénalisant pour les performances de l'application.

L'allocation de la mémoire sur les machines actuelles est gérée à deux niveaux, (*i*) par le système (espace noyau) et (*ii*) par des bibliothèques (espace utilisateur). Le système fourni une abstraction du matériel pour les applications en espace utilisateur. Les applications interagissent avec le matériel grâce à des appels systèmes. Les appels du système LINUX qui permettent de demander de la mémoire sur le tas sont `mmap()` et `brk/sbrk()` (Gorman, 2004). Les applications qui utilisent directement ces appels systèmes sont rares car ces fonctions, dites de bas niveau, proposent une interface où les aspects système sont pris en considération. D'autre part, des appels systèmes fréquents peuvent être pénalisants pour les performances de l'application.

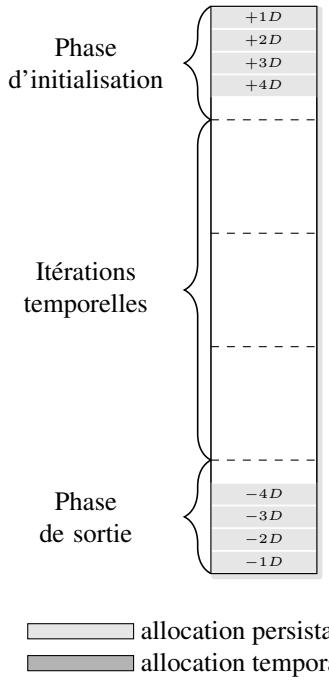

FIGURE *7. Allocation persistante des structures*

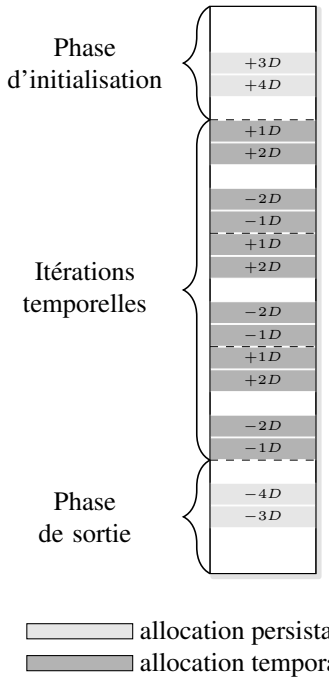

FIGURE *8. Allocation temporaire de certaines structures*

Les bibliothèques d'allocation dans l'espace utilisateur proposent deux avantages majeurs : (*i*) une interface plus simple pour manipuler les allocations que les appels systèmes et (*ii*) leur fonctionnement interne tente de limiter les interactions entre l'espace utilisateur et le système. La bibliothèque standard *glibc* fournit les fonctions `malloc()` et `free()` avec une interface simple pour gérer les allocations dynamiques. Dans notre étude, nous nous sommes intéressés à l'influence de différentes bibliothèques d'allocation sur les performances de l'application GYSELA.

Différentes études de performances des bibliothèques d'allocation dans l'espace utilisateur existent (Lever, Boreham, 2000 ; Ferreira *et al.*, 2011 ; Barootkoob *et al.*, 2011 ; Elias *et al.*, 2014 ; Widmer *et al.*, 2013), mais elles s'intéressent principalement à des applications qui font énormément d'allocations de petites tailles ce qui n'est pas le cas de GYSELA. Dans cette section, nous comparons l'impact de différentes façons d'allouer et d'initialiser la mémoire en utilisant la bibliothèque standard dans un premier temps. Dans un second temps, des tests avec les allocateurs JEMALLOC (Evans, 2006) et TCMALLOC (Ghemawat, Menage, 2009) ont été réalisés pour tenter de limiter les surcoûts dus aux opérations mémoire.

6.1. Exploration de différentes stratégies d'allocation + initialisation

L'application GYSELA bénéficie d'une parallélisation MPI + OPENMP. Des tableaux temporaires sont généralement nécessaires à l'intérieur des zones OPENMP pour stocker des résultats intermédiaires. Ces tableaux sont pour la plupart alloués juste avant l'entrée de la zone OPENMP et désalloués à la sortie. Dans une application plus réduite de GYSELA, un prototype où des algorithmes alternatifs du solveur de Vlasov sont testés, nous avons mis en place des allocations dynamiques autour de la section de calcul du solveur de VLASOV et nous l'avons comparé à la version avec allocations persistantes (figure 7 versus figure 8).

Nous avons constaté dans un premier temps que le coût associé à l'opération d'allocation + initialisation représente près de 17 % du temps d'exécution (1 nœud, 16 cœurs) dans la version dynamique alors que pour la version persistante cette opération ne représente que 2 % du temps d'exécution.

Afin d'identifier la cause de ce surcoût important, 8 approches d'allocation mémoire et initialisation éventuelle ont été mises en œuvre :

allocate allocation avec la fonction FORTRAN `allocate()` sans initialisation des données allouées.

allocate + init (1) allocation avec la fonction FORTRAN `allocate()` avec initialisation des données à l'aide de boucles. Une fois le tableau alloué, on parcourt l'ensemble de ces éléments pour les mettre à zéro.

allocate + init (2) allocation avec à la fonction FORTRAN `allocate()` avec initialisation à l'aide d'une affection à 0. Les tableaux sont être initialisés de façon globale. Par exemple l'écriture "`tab = 0.0`" initialise à `0.0` l'ensemble des éléments du tableau.

allocate + init (3) allocation avec la fonction FORTRAN `allocate()` avec initialisation à l'aide de boucles `for` parallélisées en OPENMP. L'ensemble des cœurs disponibles sera utilisé pour l'initialisation.

malloc allocation avec à l'appel de la fonction C `malloc()` sans initialisation.

malloc + init (1) allocation avec l'appel de la fonction C `malloc()` avec initialisation à l'aide de boucles `for` en C. Une fois le tableau alloué, on parcourt l'ensemble de ces éléments pour les mettre à zéro.

malloc + init (2) allocation avec l'appel de la fonction C `malloc()` avec initialisation à l'aide de la fonction `memset()`.

calloc allocation avec l'appel de la fonction C `calloc()` qui assure à l'utilisateur que la mémoire obtenue est mise à zéro.

Parmi ces 8 approches, on peut regrouper les 4 premières comme étant des approches purement FORTRAN, par opposition aux 4 dernières qui consistent à appeler des fonctions C à partir d'un code FORTRAN. Ceci est possible de façon portable à partir de FORTRAN 2003 grâce au module `ISO C BINDING`. Ce module natif à FORTRAN 2003 a été rétro-porté dans la plupart des compilateurs FORTRAN 95. On peut aussi noter que la fonction `calloc()` assure que la zone mémoire retournée

est mise à 0, dans ce cas l'initialisation à zéro de la part de l'utilisateur n'est pas nécessaire. Il n'y a pas d'équivalent FORTRAN pour cette fonction.

Le tableau 3 nous permet de comparer ces 8 approches du point de vue temps d'exécution. Les simulations ont été réalisées avec 1 processus MPI et 16 threads OPENMP. Ces temps ont été obtenus sur un nœud de la machine Poincare (IDRIS, Orsay) équipée de Sandy Bridge E5-2670 (2.60GHz, 8 cœurs par processeur, soit 16 cœurs par nœud). Les différentes simulations du tableau 3 ont été faites sur le maillage suivant :

$$N_r = 128,\ N_\theta = 256,\ N_\varphi = 64,\ N_{v_\parallel} = 64,\ N_\mu = 1 \tag{2}$$

Tableau 3. Comparaison de l'impact des différentes stratégies d'allocation + initialisation sur les temps d'exécution en seconde

	allocate	**allocate + init(1)**	**allocate + init(2)**	**allocate + init(3)**
allocation (+ init.)	0.01 0.02%	13.5 19.5%	11.1 16.0%	1.1 1.8%
calcul du solveur VLASOV	46.8 83.5%	46.5 67.1%	48.6 70.2%	46.8 81.8%
autres calculs	9.2 16.5%	9.3 13.4%	9.3 16.3%	9.6 13.8%
Total	56	69	69	57

	malloc	**malloc + init (1)**	**malloc + init (2)**	**calloc**
allocation (+ init.)	0.01 0.0%	12.4 17.7%	14.8 20.6%	0.01 0.0%
calcul du solveur VLASOV	46.5 83.3%	47.4 67.9%	47.0 65.3%	48.8 82.6%
autres calculs	9.3 16.7%	10.0 14.4%	10.1 14.0%	10.3 17.4%
Total	56	70	72	59

Le tableau 3 présente les temps d'allocations + éventuelles initialisations précédant la zone OPENMP qui calcule le solveur de VLASOV, leur pourcentage par rapport au temps d'exécution total, le temps de calcul du solveur VLASOV, son pourcentage par rapport à l'exécution total, le temps d'exécution complémentaire des deux étapes précédentes, son pourcentage par rapport à l'exécution totale et le temps d'exécution total des simulations. Dans les cas sans initialisation, les résultats du solveur de VLASOV restent valides (l'initialisation est superflue) mais ce n'est pas le cas de tous les algorithmes. L'opération la plus coûteuse en temps de calcul de notre programme est le calcul du solveur de VLASOV qui nécessite de plus l'utilisation de grandes zones

mémoires. Pour récupérer les temps d'allocation, des compteurs ont été placés juste autour de l'appel à la fonction d'allocation et de l'initialisation si elle est faite.

A la vue de ce tableau, on peut séparer les simulations en deux catégories, celles où l'initialisation à zéro est faite côté application et les autres. Dans le cas où l'initialisation se fait dans le code utilisateur, le temps de l'opération d'allocation et initialisation occupe une portion non négligeable du temps d'exécution total (jusque 16 %), néanmoins ces temps sont moindres dans le cas **allocate + init (3)**. On peut constater aussi qu'une initialisation en pur FORTRAN est aussi efficace qu'une initialisation avec la fonction `memset()` du C, le temps pris dépend directement de la bande passante mémoire.

Dans les cas **allocate**, **allocate + init (3)**, **malloc** et **calloc** on voit que les temps de simulation totaux sont nettement inférieurs aux autres cas. Le cas **calloc** est intéressant car il présente un temps de calcul total comparable au cas **allocate** et **malloc** tout en assurant à l'utilisateur que la mémoire obtenue est initialisée à zéro. Le cas **allocate + init (3)** est aussi remarquable car il présente aussi un temps de calcul total comparable au cas **allocate** et **malloc** avec une initialisation à zéro (ou éventuellement à une autre valeur) des structures allouées dans l'espace utilisateur. Cette amélioration découle uniquement de la parallélisation en OPENMP de l'initialisation.

Les temps d'exécution des cas étudiés peuvent en partie s'expliquer du fait que les allocations se font de façon paresseuse sur le système LINUX [6]. Le principe de l'allocation paresseuse signifie qu'au moment des appels aux fonctions d'allocation, de la mémoire virtuelle est allouée mais pas les pages associées en mémoire physique. La manipulation de la mémoire virtuelle est bien moins coûteuse que la manipulation des pages physiques. L'allocation de la mémoire physique est retardée au moment du premier accès. Lors du premier accès à une page virtuelle, elle n'est pas encore associée à une page physique. Une exception de type *faute de page* est levée et le système traite l'événement. Dans notre étude, les temps relevés sur le tableau 3 pour les cas sans initialisation ne correspondent pas au coût d'allocation des pages physiques, mais uniquement au coût d'allocation des pages virtuelles qui est négligeable par rapport au temps d'exécution total. Dans les cas sans initialisation, le coût des allocations des pages physiques est inclus dans le temps d'exécution du solveur de VLASOV qui accèdera pour la première fois aux pages virtuelles lors des calculs. A la vue du temps d'allocation dans le cas **calloc**, il s'avère que la fonction `calloc()` agit aussi de façon paresseuse. Cela implique que l'initialisation à zéro des pages physiques est retardée au moment du premier accès.

L'allocation paresseuse a aussi la bonne propriété de ne pas perturber la politique de *first-touch* (Terboven *et al*., 2008). Cette politique consiste à allouer la mémoire physique sur le banc mémoire le plus proche du cœur exécutant le thread qui y accède pour la première fois. Dans les cas où l'initialisation se fait de manière explicite par l'utilisateur avec un seul thread, des effets NUMA (Non Uniform Memory Access)

6. http://landley.net/writing/memory-faq.txt

peuvent dégrader les performances. Dans le cadre de notre application prototype, le motif d'accès à la mémoire est tel qu'il ne permet pas de mettre en évidence ces effets.

L'utilisation de `calloc()` qui assure l'initialisation à zéro des tableaux nous semble être un bon compris entre simplicité d'utilisation et efficacité. L'utilisation de `calloc()` est moins contraignante comparée à la mise en œuvre d'une zone OPENMP dédiée à l'initialisation de tableaux à chaque allocation mais ne permet que les initialisations à zéro.

Bien que l'utilisation de `calloc()` assure l'obtention d'une zone mémoire initialisée à zéro, cette méthode est plus efficace que les cas où l'initialisation se fait de façon explicite par l'utilisateur. Pour expliquer l'efficacité de la fonction `calloc()`, nous nous sommes intéressés au fonctionnement interne de l'allocateur standard.

6.2. Aperçu du fonctionnement interne de l'allocateur standard

Dans l'objectif d'améliorer les performances de l'application GYSELA qui est programmée en FORTRAN, l'étude de l'allocateur fourni par défaut, par la bibliothèque standard GNU C LIBRARY est pertinente car l'utilisation de la fonction `allocate()` est généralement redirigée vers un appel à la fonction C `malloc()`. Actuellement l'allocateur par défaut de la GNU C LIBRARY est PTMALLOC (Gloger, 2006) qui se base sur l'allocateur DLMALLOC (Lea, Gloger, 1996).

L'explication du fonctionnement interne de l'allocateur standard est largement détaillée dans (Kaempf, 2001). La fonction `malloc()` y est discuté. On peut distinguer deux types de comportements de cette fonction. Pour les petites allocations, la mémoire est allouée avec l'appel système `sbrk()`, et pour les plus grosses allocations, au-delà d'un seuil, la mémoire est obtenue via l'appel système `mmap()`. Le seuil qui différencie les deux comportements est défini à l'aide un paramètre nommé `M_MMAP_THRESHOLD`. Ce paramètre est ajustable par l'utilisateur grâce à la fonction `mallopt()`. Par défaut dans la GNU C LIBRARY, ce paramètre vaut 256 kilo octets.

Les allocations de grande taille se font grâce à l'appel système `mmap()` avec l'argument `MAP_ANONYMOUS`. Cet argument indique qu'il n'y a pas de fichier à projeter sur la zone mémoire demandée. Deux comportements particuliers à l'utilisation de la fonction `mmap()` doivent être mentionnés : *(i)* avec l'argument `MAP_ANONYMOUS`, la mémoire retournée par le noyau LINUX est initialisée à zéro pour des raisons de sécurité [7]; *(ii)* les pages physiques associées aux zones mémoires allouées via `mmap()` sont rendues au système lors d'une désallocation. Pour les allocations de plus petites tailles qui se font via l'appel système `sbrk()`, lors d'une désallocation, les zones mémoires associées sont gardées dans l'espace utilisateur pour une

7. Plus de détails à : http://stackoverflow.com/questions/2688466/why-mallocmemset-is-slower-than-calloc#answers-header

éventuelle réutilisation et ce afin de limiter le recyclage des pages par le système et les coûts associés.

Dans toutes les simulations qui ont été réalisées, l'ensemble des tableaux dont le temps d'allocation est résumé dans le tableau 3 sont de taille bien supérieure au seuil `M_MMAP_THRESHOLD`. Un appel à la fonction `malloc()` pour ces structures correspond donc un appel à `mmap()`. La mémoire retournée par le système est donc déjà initialisée à zéro sur la plupart des systèmes. Néanmoins, la sémantique de la fonction `malloc()` ne garantit pas l'initialisation à zéro, contrairement à la fonction `calloc()`.

Dans notre cas l'allocation des structures gourmandes en ressource mémoire se fait donc via l'appel à `mmap()` derrière l'appel à `malloc()`. Comme il a été dit plus haut, la partie la plus coûteuse des allocations est l'obtention des pages physiques. Notre approche pour réduire l'empreinte mémoire d'une application consiste principalement à introduire des allocations mémoires dynamiques ce qui implique des interactions entre l'application et le système pour les grosses structures. Cette pratique augmente substantiellement le temps d'exécution. Pour pallier à cette difficulté, nous avons testé d'autres allocateurs dans l'espace utilisateur. La section suivante les présente brièvement et montre les résultats que nous avons obtenus.

6.3. Limiter les interactions entre application et système

Afin de limiter les surcoûts, nous avons testé d'autres allocateurs que celui proposé par défaut. Dans notre étude nous nous sommes concentrés sur les allocateurs TCMALLOC et JEMALLOC. Ils se différencient par rapport à l'allocateur par défaut dans leur gestion des zones mémoires. Leur fonctionnement ne sera pas détaillé ici, mais une analyse poussée de nombreux allocateurs a été réalisée dans la thèse de S. Valat (Valat, 2014). Pour résumer brièvement l'esprit de l'allocateur JEMALLOC, des tas locaux contenant des pages mémoires réutilisables sont associés aux threads de l'application utilisant la bibliothèque. Cet allocateur maintient un équilibre des tailles des tas locaux entre tous les threads. Il assure aussi une consommation mémoire faible en retournant régulièrement les zones mémoires libérées au système. Dans l'approche de TCMALLOC, un tas global est partagé entre tous les threads. Les pages mémoires de ce tas sont retournées au système uniquement si nécessaire, sinon elles sont conservées. Bien que les approches de ces deux allocateurs soient différentes, elles ont toutes les deux pour objectif de limiter les interactions fréquentes avec le système.

L'utilisation des bibliothèques TCMALLOC et JEMALLOC est aisée sur les systèmes UNIX. Il suffit de renseigner la variable d'environnement `LD_PRELOAD` avec le chemin de la bibliothèque qu'on souhaite utiliser avant de lancer une application. Les fonctions disponibles dans la bibliothèque renseignée par `LD_PRELOAD` seront appelées en priorité à l'exécution. Dans notre cas, cela nous permet de court-circuiter les fonctions d'allocation de la GNU C LIBRARY.

Nous avons identifié le cas **calloc** comme étant le plus attrayant pour allouer et initialiser de la mémoire. Nous avons testé les allocateurs TCMALLOC et JEMALLOC sur les cas **allocate** et **calloc** du tableau 3 afin de pouvoir comparer les différents temps d'exécution globaux avec et sans initialisation.

Tableau 4. Comparaison de l'impact des allocateurs TCMALLOC *et* JEMALLOC *sur le temps d'exécution en seconde*

	allocate		**calloc**	
	TCMALLOC	JEMALLOC	TCMALLOC	JEMALLOC
allocation (+ init.)	0.001 0.0%	0.0002 0.0%	8.7 13.2%	0.0003 0.0%
calcul du solveur VLASOV	44.2 82.3%	45.9 82.3%	46.9 71.1%	46.4 80.0%
autres calculs	9.5 17.7%	9.9 17.7%	10.3 15.6%	11.6 20.0%
Total	53	56	66	58

Les temps d'exécution exposés dans le tableau 4 ont été obtenus dans les mêmes conditions que pour les simulations du tableau 3. On peut voir une différence notable sur le temps d'allocation entre les cas **allocate** et **calloc** avec l'utilisation de TCMALLOC. Dans le premier cas, le temps d'allocation est négligeable alors qu'il ne l'est pas avec l'utilisation de `calloc()`. En comparant avec le tableau 3 on peut conclure que l'implémentation de `calloc()` de la bibliothèque TCMALLOC n'est pas paresseuse. Cela peut s'expliquer par le fait que les pages mémoires obtenues par TCMALLOC sont gardées dans l'espace utilisateur. Pour assurer que la sémantique de la fonction `calloc()` soit respectée, l'initialisation à zéro se fait certainement de façon séquentielle au niveau de l'appel dans l'espace utilisateur. Pour limiter au maximum les interactions avec le système d'exploitation, TCMALLOC rend des pages mémoire au système uniquement lorsque c'est nécessaire. De fait, cette bibliothèque ne peut pas bénéficier de la remise à zéro liée à l'appel système `mmap()`.

Ces bibliothèques qui limitent les interactions avec le système offrent de faibles gains de performance sur l'application prototype dans certaines configurations d'utilisation.

6.4. Mise en œuvre d'initialisation performante dans GYSELA

Les résultats que montre le tableau 3 encouragent l'utilisation la primitive d'allocation `calloc()` au sein de l'application GYSELA. Durant le processus de réduction de l'empreinte mémoire, les allocations ont été déplacées et nous avons évité, autant que possible, d'initialiser les structures lorsque cela n'était pas nécessaire. Nous avons identifié des situations où l'initialisation est requise. Par exemple dans le cas où des sommes partielles sont cumulées dans un tableau initialisé à zéro. D'autre part, nous

initialisons aussi les tableaux pour des raisons de débogage. En utilisant la valeur *sNaN* (signal Not a Number) pour initialiser les tableaux, lors de l'exécution si cette valeur est utilisée, une exception est levée et interrompt l'exécution du programme. Cela nous permet de repérer rapidement des bugs difficiles à détecter autrement. Ces initialisations ne peuvent pas être effectuées avec un simple à `calloc()`.

Afin de limiter l'impact des initialisations à zéro, nous avons utilisé la fonction `calloc()` pour tous les tableaux nécessitant une initialisation. Pour se faire, nous avons étendu le module dédié à la gestion des allocations mémoires. Il permet dorénavant l'utilisation des fonctions d'allocation C grâce à l'`ISO C BINDING`. Dans la version de production de GYSELA, nous n'observons pas de gains significatifs de performance après la mise en place de l'appel à `calloc()`. Comme il est dit plus haut, durant la mise en place des allocations dynamiques, nous avons évité autant que possible l'initialisation à zéro des structures, ce qui limite l'impact des surcoûts dus aux initialisations.

7. Conclusion

L'extensibilité mémoire du code GYSELA a été améliorée grâce à une gestion dynamique des allocations/désallocations. Une réduction de 50.8% du pic mémoire a été réalisée sur certaines simulations à 32k cœurs.

Cet article décrit un module de modélisation et de trace mémoire et quelques outils de post-traitement qui permettent de réduire le pic mémoire. Avec ce jeu d'outils, la modélisation du comportement de l'empreinte mémoire au cours du temps de GYSELA est accessible. L'outil de prédiction permet d'extrapoler de façon exacte la consommation mémoire pour différents jeux de paramètres d'entrées en mode hors-ligne ; cet aspect est important aussi bien pour les utilisateurs finaux qui ont besoin de plus grandes résolutions ou de fonctionnalités gourmandes en mémoire, que pour les développeurs qui ajustent leurs algorithmes pour une machine spécifique, p. ex. de type Exascale.

Nous avons aussi montré que l'utilisation d'allocations dynamiques représente peu de surcoût par rapport aux allocations persistantes. Par contre, l'initialisation des zones allouées peut représenter un coût non négligeable. Notre étude sur un prototype de GYSELA nous a montré qu'en fonction de l'allocation, de l'initialisation éventuelle des tableaux et de la bibliothèque d'allocation, les temps d'exécution varient entre 53 à 72 secondes. Ces facteurs ont donc un impact visible sur les temps d'exécution. Face à ce bilan, nous tentons de limiter autant que possible l'initialisation à zéro des zones allouées.

Notre prochain objectif est d'implémenter une bibliothèque utilisable en C/FORTRAN qui permet de générer des traces mémoires, de les visualiser et de faire de la prédiction de consommation mémoire. Le travail présenté dans cet article est la première étape de la construction d'une méthodologie qui aide les développeurs à améliorer l'extensibilité mémoire de leur application parallèle.

Remerciements

Le travail restitué dans cet article est en partie financé par le projet NUFUSE G8-EXASCALE *(http://www.nu-fuse.com).*

Bibliographie

Åström J., Carter A., Hetherington J., Ioakimidis K., Lindahl E., Mozdzynski G. *et al*. (2013). Preparing scientific application software for exascale computing. In *Applied parallel and scientific computing*, p. 27–42. Springer.

Aulagnon C., Martin-Guillerez D., Rué F., Trahay F. (2013). Runtime function instrumentation with eztrace. In *Euro-par 2012: Parallel processing workshops*, p. 395–403. Consulté sur http://link.springer.com/chapter/10.1007/978-3-642-36949-0_45

Barootkoob G., Sharifi M., Khaneghah E. M., Mirtaheri S. L. (2011). Parameters affecting the functionality of memory allocators. In *Communication software and networks (iccsn), 2011 ieee 3rd international conference on*, p. 499–503.

Bruening D., Garnett T., Amarasinghe S. (2003). An infrastructure for adaptive dynamic optimization. In *[cgo 2003]*, p. 265–275. Consulté sur http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1191551

Carrez S. (2013). Optimization with valgrind massif and cachegrind.

Darrigol O. (2005). *Les équations de maxwell*. Belin, Paris.

Djoudi L., Barthou D., Carribault P., Lemuet C., Acquaviva J.-T., Jalby W. *et al*. (2005). Maqao: Modular assembler quality analyzer and optimizer for itanium 2. In *The 4th workshop on epic architectures and compiler technology, san jose.* Consulté sur http://www.labri.fr/perso/barthou/ps/maqao.pdf

Elias D., Matias R., Fernandes M., Borges L. (2014). Experimental and theoretical analyses of memory allocation algorithms. In *Proceedings of the 29th annual acm symposium on applied computing*, p. 1545–1546.

Evans J. (2006). A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada.*

Ferreira T. B., Matias R., Macedo A., Araujo L. B. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and distributed computing, applications and technologies (pdcat), 2011 12th international conference on*, p. 92–98.

Folk M., Cheng A., Yates K. (1999). Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of supercomputing*, vol. 99.

Geimer M., Wolf F., Wylie B. J., Ábrahám E., Becker D., Mohr B. (2010). The scalasca performance toolset architecture. *CCPE*, vol. 22, n° 6, p. 702–719. Consulté sur http://onlinelibrary.wiley.com/doi/10.1002/cpe.1556/full

Ghemawat S., Menage P. (2009). *Tcmalloc: Thread-caching malloc.* Consulté sur goog-perftools.sourceforge.net/doc/tcmalloc.html

Gloger W. (2006). *Ptmalloc.* Consulté sur http://www.malloc.de/en/

Gorman M. (2004). *Understanding the linux virtual memory manager*. Prentice Hall Upper Saddle River.

Gperftool - heap profiler. (2009). Consulté sur http://goog-perftools.sourceforge.net/doc/heap_profiler.html

Grandgirard V., Sarazin Y., Garbet X., Dif-Pradalier G., Ghendrih P., Crouseilles N. *et al*. (2006). Gysela, a full-f global gyrokinetic semi-lagrangian code for itg turbulence simulations. In *Aip conference proceedings*, vol. 871, p. 100.

Grandgirard V., Sarazin Y., Garbet X., Dif-Pradalier G., Ghendrih P., Crouseilles N. *et al*. (2008). Computing ITG turbulence with a full-f semi-Lagrangian code. *Communications in Nonlinear Science and Num. Sim.*, vol. 13, n° 1, p. 81 - 87.

Gupta S. C., Rangarajan K. (2012, may). *Optimizing heap memory usage.* Google Patents. Consulté sur http://www.google.com/patents/US20120278585 (US Patent App. 13/462,766)

Hastings R., Joyce B. (1991). Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference.*

Kaempf M. (2001). *Vudo malloc tricks. phrack magazine.* Consulté sur http://phrack.org/issues/57/8.html#article

Latu G., Grandgirard V., Crouseilles N., Dif-Pradalier G. (2011). Scalable quasineutral solver for gyrokinetic simulation. In *Ppam (2)*, p. 221-231. Springer.

Lea D., Gloger W. (1996). *A memory allocator.* Consulté sur http://gee.cs.oswego.edu/dl/html/malloc.html

Lever C., Boreham D. (2000). Malloc () performance in a multithreaded linux environment. *http://citi.umich.edu/projects/linux-scalability/reports/malloc.html*.

Luk C.-K., Cohn R., Muth R., Patil H., Klauser A., Lowney G. *et al*. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, vol. 40, p. 190–200. Consulté sur http://dl.acm.org/citation.cfm?id=1065034

Maxwell J. C. (1865). A dynamical theory of the electromagnetic field. *Philosophical Transactions of the Royal Society of London*, p. 459–512.

Milian W. (2014). Heaptrack - a heap memory profiler for linux. *http://milianw.de/blog/heaptrack-a-heap-memory-profiler-for-linux*.

Nethercote N., Seward J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Acm sigplan notices*, vol. 42, p. 89–100.

Peterson P. (2009). F2py: a tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, vol. 4, n° 4, p. 296–305. Consulté sur http://cens.ioc.ee/~pearu/papers/IJCSE4.4_Paper_8.pdf

Rozar F., Latu G., Roman J. (2014). Achieving memory scalability in the gysela code to fit exascale constraints. In R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), *Parallel processing and applied mathematics*, p. 185-195. Springer Berlin Heidelberg. Consulté sur http://dx.doi.org/10.1007/978-3-642-55195-6_17

Rozar F., Latu G., Roman J., Grandgirard V. (2015). Toward memory scalability of gysela code for extreme scale computers. *Concurrency and Computation: Practice and Experience*, vol. 27, n° 4, p. 994–1009. Consulté sur http://dx.doi.org/10.1002/cpe.3429

Runciman C., Röjemo N. (1997). Two-pass heap profiling: A matter of life and death. In *Implementation of functional languages*, p. 222–232. Springer.

Shalf J., Dosanjh S., Morrison J. (2011). Exascale computing technology challenges. In *High performance computing for computational science – vecpar 2010*, p. 1–25. Springer. Consulté sur https://www.nersc.gov/assets/NERSC-Staff-Publications/2010/ShalfVecpar2010.pdf

Terboven C., Mey D. an, Schmidl D., Jin H., Reichstein T. (2008). Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on memory access on future processors: A solved problem?*, p. 377–384. New York, NY, USA, ACM. Consulté sur http://doi.acm.org/10.1145/1366219.1366222

Valat S. (2014). *Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du calcul haute performance*. Thèse de doctorat non publiée, Versailles-St Quentin en Yvelines.

Vlasov A. (1945). On the kinetic theory of an assembly of particles with collective interaction. *Russ. Phys. J.*, vol. 9, p. 25–40.

Widmer S., Wodniok D., Weber N., Goesele M. (2013). Fast dynamic memory allocator for massively parallel architectures. In *Proceedings of the 6th workshop on general purpose processor using graphics processing units*, p. 120–126.

Zorn B., Hilfinger P. (1988). A memory allocation profiler for c and lisp programs. In *Proceedings of the summer usenix conference*, p. 223–237.