Projet NUMASIS
ANR-05-CIGC-002

Tâche T1.3 Traçage et visualisation

Rapport d'étude et d'évaluation des outils de traçage et de visualisation de programmes sur architectures NUMA.

Livrable T0+18

Partenaires : ID-IMAG/LIG (MESCAL, MOAIS)

31 juillet 2007

Objectifs de la tâche T1.3

Déterminer les goulots d'étranglement, vérifier le bon comportement d'une application dans des circonstances très précises ou encore comprendre la portée réelle d'une optimisation nécessitent une analyse extrêmement fine de l'exécution d'une application. L'objectif de cette tâche est de définir et réaliser des outils de traçage et visualisation de programmes (multi-threadés ou non) sur architectures NUMA.

La visualisation des traces pourra être basée sur le logiciel PAJÉ[13]. PAJÉ est un environnement de visualisation interactif, « scalable » et extensible, conçu pour la visualisation « post mortem » de traces d'exécution. Cet outil offre plusieurs visualisations : soit une visualisation détaillée de tout ce qui est susceptible de varier dans le temps, combinant diagramme espace-temps et diagramme de Gantt avec une représentation de l'état des objets de synchronisations (verrous, variables conditionnelles et sémaphores), soit des visualisations synthétiques. Pour ces dernières il est possible à l'utilisateur de choisir les données à visualiser : occupation des noeuds de calcul, un comptage des messages, le trafic de données, etc. ainsi que la forme de cette visualisation : graphe de colonnes, camembert, etc..

1 Introduction

1.1 Techniques pour le recueil des performances

Pour analyser l'exécution d'une application, on utilise des outils de *profilage* ; ces outils peuvent être séparés en deux catégories suivant le point de vue (noyau ou applicatif) qu'ils adoptent pour observer l'exécution de l'application.

Le point de vue *noyau* est obtenu essentiellement par une instrumentation légère du système d'exploitation. Cette instrumentation est souvent intégrée au code source du noyau par le programmeur (LINUX TRACE TOOLKIT [15], FAST KERNEL TRACE [12]), mais elle peut l'être aussi dynamiquement (KERNINST [14]). Ces outils permettent une analyse très fine des applications : dans [12], une analyse des performances de la pile TCP est menée, afin de comprendre le manque d'efficacité relatif d'un circuit de calcul des sommes de contrôle.

Le point de vue *applicatif* est obtenu par l'instrumentation du programme exécutable, souvent au moyen d'outils intervenant à la génération de code. De nos jours, toutes les plates-formes de développement sont pourvues de ce type d'outils ; parmi ceux-ci, on peut citer les utilitaires GNU `gcov` et `gprof` [5] ou encore l'outil d'INTEL `VTune` qui est capable de tirer partie des compteurs de performance des processeurs INTEL et de suivre le cheminement des processus légers ordonnancés par le système d'exploitation.

2 Traçage des bibliothèques de threads

La bibliothèque de threads retenue pour le projet NUMASIS (la bibliothèque PM^2) est une bibliothèque efficace et portable qui fait intervenir un ordonnancement à deux niveaux sur les machines NUMA. Cette dernière caractéristique a pour conséquence de rendre complexe à prévoir l'ordonnancement final des processus légers de l'application.

En outre, de nombreux outils d'analyse proposent des résultats par processus ou, au mieux, par threads noyaux (occupation mémoire, statistiques sur les défauts de cache, etc.). Il est donc nécessaire de concevoir et d'utiliser des outils adaptés à notre bibliothèque afin d'obtenir les informations souhaitées pour chacun des threads utilisateurs de nos applications.

2.1 Observer l'ordonnancement des threads

Obtenir des informations sur le déroulement interne d'une bibliothèque de thread a toujours été une tâche difficile lorsque les threads s'exécutent réellement en concurrence, ce qui est bien sûr le cas sur des machines NUMA. Cela s'explique par le fait que le système de trace a besoin de sérialiser, ou au moins de synchroniser, certaines de ses opérations telles que l'utilisation de ses tampons internes pour stocker les événements. Or les primitives de synchronisation au sein d'une bibliothèque sont généralement fournies par cette bibliothèque. Cela oblige donc le système de trace à utiliser les mécanismes de la bibliothèque, au prix de ne pouvoir tracer ces mécanismes eux-mêmes, ou bien de développer des mécanismes externes qui risquent de modifier l'ordonnancement de la bibliothèque (et donc de cacher des comportements que l'on voudrait observer).

L'environnement PM^2 possède déjà un support pour enregistrer et observer les événements d'ordonnancement en rapport avec son exécution. Ainsi, pour un programme multithreadé s'exécutant dans l'environnement PM^2, il est possible d'obtenir de manière efficace une trace de l'exécution détaillée[3]. Cette trace peut être convertie en une trace Pajé[13] que l'on peut visualiser graphiquement avec le logiciel du même nom. Elle peut également être convertie en une animation qui montre de manière interactive la succession des événements d'ordonnancement et les décisions de placement pendant une période donnée de l'exécution.

Pour réaliser cette prise de trace, PM^2 utilise la bibliothèque FxT composée d'une partie en espace noyau[1] (FKT) et une partie en espace utilisateur (FUT). Le principe de la bibliothèque FxT est d'instrumenter le code grâce à quelques macros. Ces macros, très courtes, perturbent très peu l'application en enregistrant tout un ensemble d'informations qui seront exploitées par les outils d'analyse post-mortem fournis. Sa synchronisation interne est assurée par des opérations atomiques du processeur ce qui permet d'observer le comportement interne de la bibliothèque (fonctionnement des verrous, des signaux, etc.) sans perturber l'ordonnancement normal de PM^2. Tous ces travaux sont décris en détails dans [12, 11, 2, 4] et un exemple de visualisation graphique des traces obtenues est donné dans la figure 1.

2.2 Utilisation des compteurs matériels

Être capable de suivre l'ordonnancement des différents threads d'une application est important mais cela n'est pas toujours suffisant. Sur une machine NUMA, les processeurs sont, certes, une ressources importante mais la mémoire l'est tout autant. Et dans de nombreux cas, l'interaction entre les allocations des processeurs pour les threads d'une part et les allocations mémoires d'autre part va conditionner les bonnes performances ou non du programme. Le développeur d'une application pour machine NUMA sera donc grandement intéressé par des informations sur ces interactions afin d'optimiser au mieux le placement en mémoire de ses données.

Il est illusoire d'essayer de tracer chaque accès mémoire. Cela serait beaucoup trop coûteux et générerait des volumes de données à analyser beaucoup trop important. Toutefois, l'intérêt pour ce genre d'informations a amené les concepteurs de processeurs à intégrer dans ces dernier des mécanismes permettant de collecter automatiquement certaines informations. Cela se fait grâce à des registres spéciaux (compteurs matériels) du processeur que l'on peut pro-

[1] Sur une machine dédiée, FKT n'est pas strictement nécessaire : sans cette partie, on perd la possibilité de tracer automatiquement les appels systèmes et les interruptions de la machine. Mais on évite de devoir modifier le noyau de la machine utilisée.

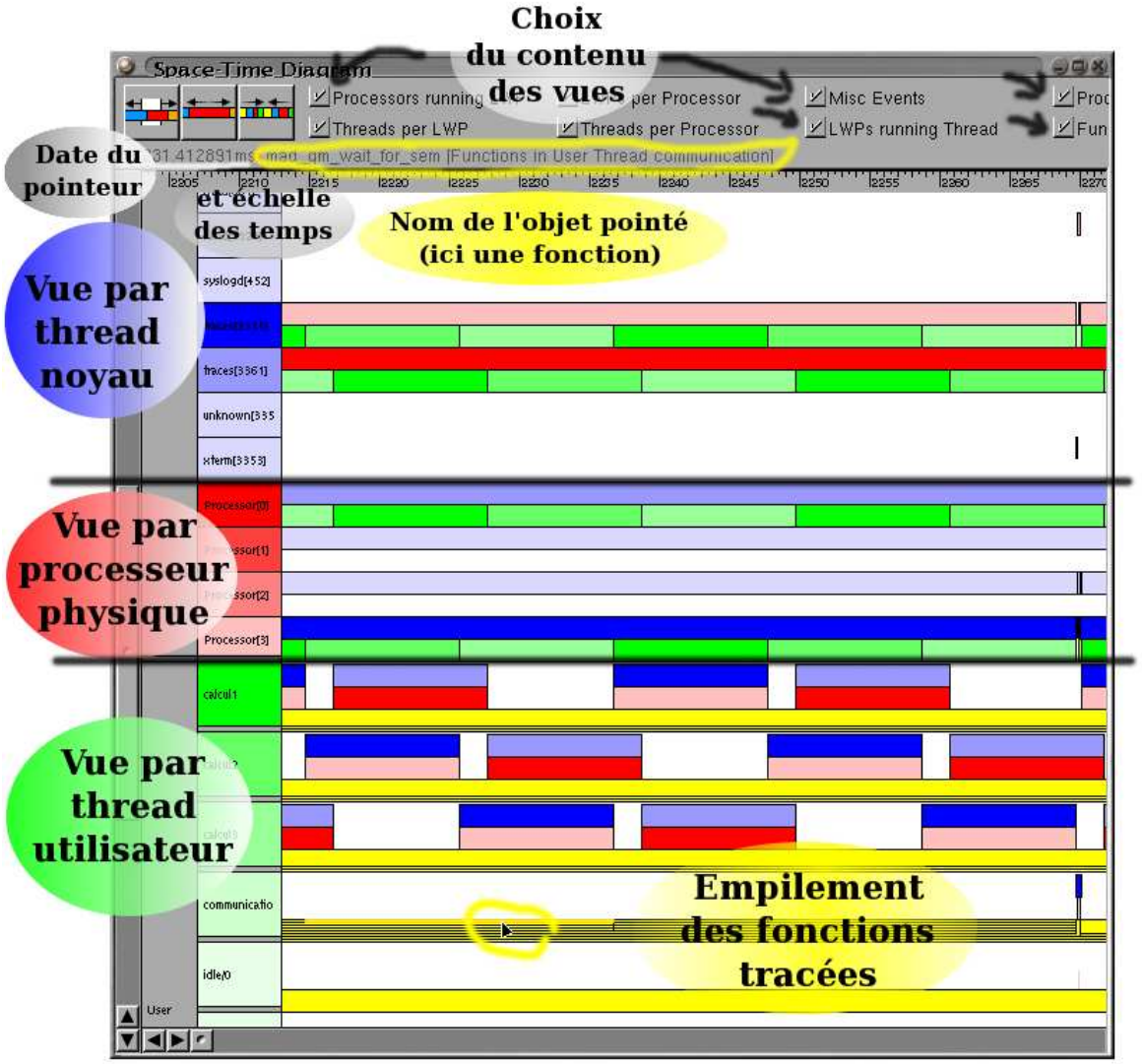

FIG. 1 – Visualisation d'une trace avec PAJÉ

grammer pour collecter des statistiques particulières telles que le nombre de défaut de cache, le nombre d'accès mémoire, le nombre de défaut de TLB, etc. Bien évidemment, ces registres et les opérations possibles dépendent du type exacte de processeur que l'on utilise. Certaines opérations sont présentes sur quasiment tous les modèles de processeurs (qui intègrent ces registres). D'autres sont spécifiques à un modèle spécifique de processeur.

Pour accéder aux informations qui peuvent être obtenues grâce à ces compteurs matériels, plusieurs bibliothèques ont été développées. L'une des plus connues et utilisées est la bibliothèque PAPI. La fin de cette partie (Section 2.2) va maintenant présenter cette bibliothèque et la suivante (Section 2.3) détaillera les adaptations qui ont été faites pour la faire fonctionner au sein de l'environnement PM^2.

2.2.1 La bibliothèque PAPI

Le projet PAPI[1] (*Performance Application Programming Interface*) définie une interface de programmation qui permet d'accéder aux compteurs matériels disponibles dans la plupart des processeurs moderne de manière portable et efficace. Grâce à la collecte de ces données, il sera parfois possible de mettre en évidence des corrélations entre d'une part la structure du code source et des données manipulées et d'autre part l'efficacité du placement de ces objets sur l'architecture cible.

2.2.2 Réalisation

Ces corrélations pourront par la suite être exploitée pour répondre à différents problèmes comme l'optimisation de code, le débogage, le benchmarking, la modélisation de performance, etc. Cela est particulièrement important pour les architectures NUMA où il set crucial de gérer correctement le placement des données en mémoire vis-à-vis de l'ordonnancement des flots d'exécution sur les processeurs. Les observations conduites grâce aux données récoltées par PAPI conduiront peut-être par la suite à la mise en place de nouvelles stratégies de placement mieux adaptées à ce contexte matériel.

Un intérêt important de PAPI est sa portabilité vis-à-vis de nombreuses architectures. Il permet d'accéder avec les mêmes fonctions et les mêmes arguments aux différents compteurs matériels de chacune des architectures supportées. Bien évidemment, tous les types de compteurs ne sont pas disponibles sur toutes les architectures. Mais PAPI uniformise tout ce qui est possible et renvoie des codes d'erreur appropriés lorsque cela est nécessaire. De plus, les opérations de collecte d'information qu'il propose restent très efficaces grâce à une implémentation optimisée.

2.2.3 Interface de programmation de PAPI

PAPI propose deux interfaces de programmation pour accéder aux compteurs matériels. La première, de haut niveau, est destinée obtenir facilement des événements simple. La seconde, de bas niveau, permet d'exploiter pleinement PAPI et tous les compteurs matériels disponibles sur l'architecture. Elle est cependant un peu plus délicate à manier.

Interface PAPI de haut niveau. Cette interface est réduite et ne comporte que 8 fonctions (voir figure 2).

Cette interface permet de choisir facilement des compteurs matériels à observer, de démarrer, arrêter les observations et de récupérer les événements observés. Il est de la responsabilité du programmeur de ne pas chercher à observer plus d'événements que le matériel ne le supporte. En cas d'erreur, ces fonctions retourneront un code d'erreur ou termineront l'application.

Ces fonctions peuvent être mélangées avec des appels aux fonctions de bas niveau, là encore à condition de ne pas demander des choses non supportées par le matériel (suivre des types d'événements incompatibles, etc.).

Interface PAPI de bas niveau. Cette interface comporte un nombre beaucoup plus conséquent de fonctions qui ne seront pas toutes présentées ici (se référer à la documentation de PAPI pour plus d'information). Ces fonctions permettent de manipuler plusieurs concepts

```
/* obtient le nombre de compteurs matériels disponibles */
int PAPI_num_counters(void);
/* obtient le nombre d'instructions flottantes et de temps réel et processeur */
int PAPI_flips(float *rtime, float *ptime, long_long *flpins, float *mflips);
/* obtient le nombre d'opérations flottantes et de temps réel et processeur */
int PAPI_flops(float *rtime, float *ptime, long_long *flpops, float *mflops);
/* obtient le nombre d'instructions par cycle et de temps réel et processeur */
int PAPI_ipc(float *rtime, float *ptime, long_long *ins, float *ipc);
/* lit les compteurs dans un tableau en les remettant à zéro */
int PAPI_read_counters(long_long *values, int array_len);
/* accumule les compteurs dans un tableau en les remettant à zéro */
int PAPI_accum_counters(long_long *values, int array_len);
/* démarre le comptage d'événements matériels */
int PAPI_start_counters(int *events, int array_len);
/* arrête les comptage d'événements matériels */
int PAPI_stop_counters(long_long *values, int array_len);
```

FIG. 2 – Interface de haut niveau de PAPI

introduits par PAPI qui servent à assurer une portabilité entre les différentes architectures supportées.

L'interface de bas niveau introduit le concept d'*EventSets* (ensembles d'événements). Ces ensembles sont dynamiques : ils sont créés par le programmeur. Cela permet d'observer de manière synchronisée un ensemble de compteurs corrélés. Par exemple, on peut compter les accès mémoires en même temps que les accès manqué à travers le cache de niveau L1. Une forte corrélation peut indiquer une mauvaise localité des données.

2.3 Intégration de PAPI dans PM^2

La bibliothèque de threads PM^2 utilisée au sein du projet NUMASIS est une bibliothèque à deux niveaux ordonnançant des threads utilisateurs. C'est une bibliothèque non standard (même si son interface l'est) que les concepteur de PAPI n'ont bien évidemment pas pris en compte. Pour utiliser simultanément PAPI et PM^2, un certain nombre d'adaptations ont dues être menées à bien. Nous verrons tout d'abord le support multithread qu'offre PAPI puis les modifications que nous avons dû apporter pour que PM^2 puisse profiter de PAPI.

2.3.1 PAPI et les threads

La plupart des machines récentes étant multiprocesseurs (ou au moins multicore), il aurait été très étonnant que PAPI ne supporte pas les programmes multithreadés. Cependant, le support multithread de PAPI est limité. En effet, PAPI protège correctement ses quelques variables globales grâce à des primitives de synchronisation (mutex, etc.). Cependant, du point de vue des compteurs de performance, il traite chaque thread comme s'il s'agissait de processus indépendants : chaque thread est responsable de créer, démarrer, arrêter et lire ses propres compteurs. En outre, cela suppose que les threads de l'application sont des threads noyau.

PAPI "supporte" également les threads purement utilisateurs mais, dans ce cas, les valeurs obtenues concernent systématiquement le processus en entier et pas les différents threads du

processus : il est alors impossible par exemple de comparer les défauts de cache de deux threads différents.

En résumé, PAPI compte les événements par flot d'exécution ordonnancé par le noyau, que ce soit un processus monothreadé, un thread d'une application multithreadé par une bibliothèque de niveau noyau ou encore l'ensemble des threads d'une application multithreadé par une bibliothèque de niveau utilisateur. Cela s'explique par le fait que PAPI programme et lit les compteurs matériels pour le flot d'exécution de niveau noyau courant. Lorsqu'il y a un changement de flot d'exécution de niveau noyau par l'ordonnanceur système, PAPI laisse au système d'exploitation la charge de reprogrammer si nécessaire les compteurs matériels pour qu'ils correspondent au nouveau flot d'exécution de niveau noyau.

2.3.2 PAPI et PM^2

PM^2 est une bibliothèque de threads à deux niveaux qui n'est pas une situation que PAPI sait gérer. Un support particulier doit donc être mis en place.

Le principe va consister à introduire dans l'ordonnanceur utilisateur de PM^2 une routine qui, à chaque changement de contexte, va sauvegarder les valeurs de compteurs utilisés par le thread courant pour que le thread suivant ne soit pas crédité des événements qui ne le concernent pas. Idéalement, la valeur anciennement sauvegardée devrait être restaurée lorsque le thread reprend la main. Malheureusement, l'interface de PAPI ne permet pas cela : on peut lire les compteurs matériels et les remettre à zéro mais pas leur affecter une valeur quelconque. Heureusement, on peut facilement contourner ce problème en faisant en sorte que la fonction retournant la valeur d'un compteur va maintenant ajouter la valeur sauvegardée. L'algorithme simplifié de ce comportement est représenté dans la figure 3.

Bien évidemment, cela ne fonctionne que si l'ensemble des threads noyaux sont en train de mesurer le même ensemble d'événements. Dans le cas contraire, un thread utilisateur pourrait surveiller un ensemble d'événements dans son thread noyau, être désordonnancé (en accumulant les valeurs des compteurs dans son espace de sauvegarde) puis réordonnancé sur un autre thread noyau. Si ce dernier ne surveille pas les bons événements, la lecture des compteurs par le thread utilisateur donnera des résultats complètement aberrants.

La première version de l'intégration de PAPI et PM^2 choisissait les événements à observer tout au début de son exécution et tous les threads noyau de PM^2 (donc tous les threads utilisateurs également) observaient les mêmes types d'événements. Nous travaillons actuellement pour proposer une solution plus souple où les threads utilisateurs peuvent choisir indépendemment les événements qu'ils souhaitent observer. Cela nécessite alors lors des changements de contexte entre threads utilisateurs de reprogrammer les compteurs matériels si les événements suivis par le thread précédent et le suivant ne sont pas les mêmes. Bien évidemment, ces reprogrammations des compteurs matériels vont ralentir le programme lorsqu'elles sont nécessaires.

Le rapport [8] décrit ces travaux. Il montre également un exemple d'utilisation de ce support : grâce aux mesures fournies par PAPI sur PM^2, nous parvenons à mettre en évidence un mauvais placement de données sur une application synthétique.

3 Traçage et visualisation de l'occupation mémoire

Le traçage et la visualisation de comportement d'application sur une architecture mémoire NUMA permet de corréler les performances applicatives à l'exploitation physique de la ma-

```
struct thread {
  ...champs originaux...
  long long papi_counters[PAPI_MAX_COUNTERS];
} *marcel_t;

void marcel_switchto(marcel_t old, marcel_t new) {
  __marcel_papi_save(old);
  ...code original de la fonction...
}


void  __marcel_papi_save(marcel_t old) {
  papi_accu_counters(old->papi_counters, PAPI_MAX_COUNTERS);
}

int marcel_PAPI_read_counters(long long *values, int size) {
  papi_read_counters(values, size);
  for(int i=0; i<size; i++) {
    values[i]+=get_self()->papi_counters[i];
  }
}

int marcel_PAPI_accu_counters(long long *values, int size) {
  papi_accu_counters(values, size);
  for(int i=0; i<size; i++) {
    values[i]+=get_self()->papi_counters[i];
  }
}
```

FIG. 3 – Schéma simplifié de l'intégration de PAPI et PM^2

chine. En effet, par exemple une mauvaise distribution des données entre les nœuds NUMA peut-être source de contention et donc de perte de performances. L'objectif de cette section est de proposer des outils indiquant à l'utilisateur le comportement mémoire des applications scientifiques. De ce fait, ces outils apportent une meilleure compréhension du comportement applicatif et des performances obtenues.

Cette section présente deux outils de traçage et de visualisation de l'exploitation mémoire d'une architecture NUMA. Le premier outil, présenté dans la sous-section 3.1, trace les appels à la librairie d'allocation par tas. Les traces obtenues sont visualisées par l'outil PAJÉ[13]. Le second outil, sous-section 3.2, permet de visualiser dynamiquement l'allocation physique des pages mémoires associées à une variable d'un programme. Cette visualisation dynamique reprend une interface similaire à celle de l'outil TOP[9] sous Linux.

3.1 Traçage des allocateurs de mémoire dynamique sur machine NUMA

3.1.1 Contexte et objectifs

La librairie d'allocation par tas, nommée librairie *heapalloc*, a été développée dans le délivrable T1.2 du projet NUMASIS. Cette librairie a pour objectif de définir des méthodes d'allocation mémoire dynamique (du type *malloc/free*) dans des zones d'adressage préalablement réservées. Ces zones réservées sont appelées des tas et contiennent toutes les données nécessaires aux allocations dynamiques. Ainsi, aucune variable globale est nécessaire pour la gestion des différents tas. De plus, des méthodes de placement mémoire sur les nœuds physiques sont ajoutées à la librairie permettant au concepteur d'application de préciser les nœuds mémoires pour les allocations dynamiques des variables du programme. Les tas peuvent également être déplacés entre différents nœuds NUMA impliquant une migration des pages physiques. Toutefois, il est intéressant de visualiser le comportement d'une application utilisant cette librairie en précisant sur quels nœuds NUMA les allocations dynamiques sont effectivement réalisées.

L'objectif principal est d'obtenir une trace des appels aux méthodes d'allocation par tas et de visualisés ces appels de manière temporelle grâce à l'outil PAJÉ. Les méthodes de traçage sont incorporées à la librairie HEAPALLOC. Les méthodes tracées sont les méthodes de création et de destruction de tas ainsi que les méthodes d'allocation et de libération de la mémoire. En fonction des méthodes tracées différentes informations sont ajoutées au fichier de traçage. Par exemple, dans le cas d'une allocation `ma_amalloc`, les informations relatives à l'identification du thread appelant la méthode, de la taille de données allouées, la date et le masque de bits représentant les nœuds NUMA sont stockées dans la trace.

3.1.2 Visualisation

La visualisation d'une trace est réalisée par l'outil PAJÉ. PAJÉ propose une représentation temporelle en forme de diagramme de Gantt. La figure 6 visualise une trace obtenue lors de l'exécution d'une application calculant un jacobi. Les détails de la trace sont présentés dans la figure 4. Le pseudo code de la figure 5 décrit le code du jacobi et l'utilisation de la librairie HEAPALLOC. Cette application, utilisant la librairie HEAPALLOC, crée 16 threads s'exécutant sur une architecture NUMA à 8 nœuds NUMA.

Sur cette figure, les couleurs représentent les différents nœuds NUMA au moyen d'un masque de bits[2]. Pour chaque appel aux méthodes, les informations collectées sont inscrites à

[2]Un masque de bits correspond à un nombre binaire où le numéro des digits correspond au numéro des

l'intérieur de l'événement affiché. Dans cet exemple, il est intéressant de noter que les threads (ou Process dans la trace) allouent uniquement dans des tas placés suivant le même masque de bits. Ainsi, l'utilisateur peut, pour favoriser la localité des threads et des données, fixer les threads sur les processeurs appartenant aux nœuds du masque de bits des tas qu'ils utilisent.

Le traçage de la librairie HEAPALLOC utilise une technique de visualisation post-mortem. Une autre approche pour visualiser et étudier le comportement mémoire d'une application, serait d'observer le placement mémoire des variables de l'application de manière dynamique.

```
Evénement: type id, date, catégorie, numéro du processus, description

10 0.002230 Etat P12 Create_Tas_mask=36
10 0.002121 Etat P8 Create_Tas_mask=32
10 0.002359 Etat P14 Create_Tas_mask=38
10 0.002387 Etat P14 Malloc_size=544_node=38
10 0.002395 Etat P3 Malloc_size=544_node=26
10 0.002405 Etat P3 Malloc_size=544_node=26
```

FIG. 4 – Exemple d'une trace PAJÉ des allocateurs de la librairie HEAPALLOC

3.2 Traçage de l'allocation physique des pages mémoires

3.2.1 Objectif

Lors de l'exécution d'un programme plusieurs données sont allouées dans un ensemble de pages qui sont potentiellement distribuées physiquement sur plusieurs nœuds NUMA. Il devient alors intéressant de visualiser cette distribution pour un ensemble de variables du programme. En effet, cela permet de mettre en évidence un bon ou un mauvais usage de la ressource mémoire et d'y associer les variables du programme qui en sont responsables. Un tel outil apporte d'une part des informations supplémentaires pour décider d'un bon placement d'une variable et d'autre part de vérifier un comportement applicatif.

Dans le cadre du délivrable T1.3, nous avons développé un outil qui atteint ces différents objectifs. Cet outil, nommé MEMTOP, permet grâce à un interface de choisir des zones mémoires d'un programme pour en visualiser dynamiquement la distribution sur les nœuds NUMA.

3.2.2 Outil : MEMTOP

L'outil MEMTOP se divise en deux parties indépendantes. La première partie est une librairie qui est intégrée au code à étudier. Elle permet de choisir des variables à observer et d'en collecter les informations relatives au placement des pages. Elle se compose d'une interface de programmation regroupant les différentes méthodes pour sélectionner et de retirer des zones mémoires à observer. Cette interface est implémentée dans un librairie appelée MEMON. Elle est principalement composée de 4 méthodes présentées dans la figure 7.

Les deux premières méthodes permettent d'initier et finaliser la librairie de collecte des données sur les pages touchées. Les deux autres méthodes permettent de sélectionner une zone mémoire et d'y associer un identifiant (une chaîne de caractères) utilisé pour l'affichage. Ces

nœuds NUMA. Par exemple le nombre binaire 0010 indique que seul le nœud 2 est choisi sur 4 nœuds.

```
int main(int argc, char *argv[])  {
  ...

  /* initialisation du traçage */
  ma_init_trace();
  ...

  /* création des threads de calcul */
  for (i = 0; i < numWorkers; i++)
      pthread_create(&workerid[i], &attr, Worker, (void *) i);
  ...

}

void *Worker(void *arg) {
  /* déclaration d'un tas */
  heap_t heap;

  /* création d'un tas avec un placement représenté par le masque de bits: mask */
  /* l'appel à cette méthode est automatiquement tracé */
  heap = ma_acreatenuma(size, HEAP_DYN_ALLOC, CYCLIC, LOW_WEIGHT, &mask, maxnode);
  ...

  /* allocation de la zone mémoire nécessaire aux calculs */
  /* les appels aux méthodes ma_amalloc sont automatiquement tracés */

  grid1 = (double**)ma_amalloc(size_grid*sizeof(double*), heap);
  grid2 = (double**)ma_amalloc(size_grid*sizeof(double*), heap);

  for(i = 0; i <= strip; i++) {
    grid1[i] = (double*)ma_amalloc(size_grid*sizeof(double), heap);
    grid2[i] = (double*)ma_amalloc(size_grid*sizeof(double), heap);
  }
  ...calcul du jacobi...

}
```

FIG. 5 – Exemple d'un jacobi utilisant la librairie HEAPALLOC

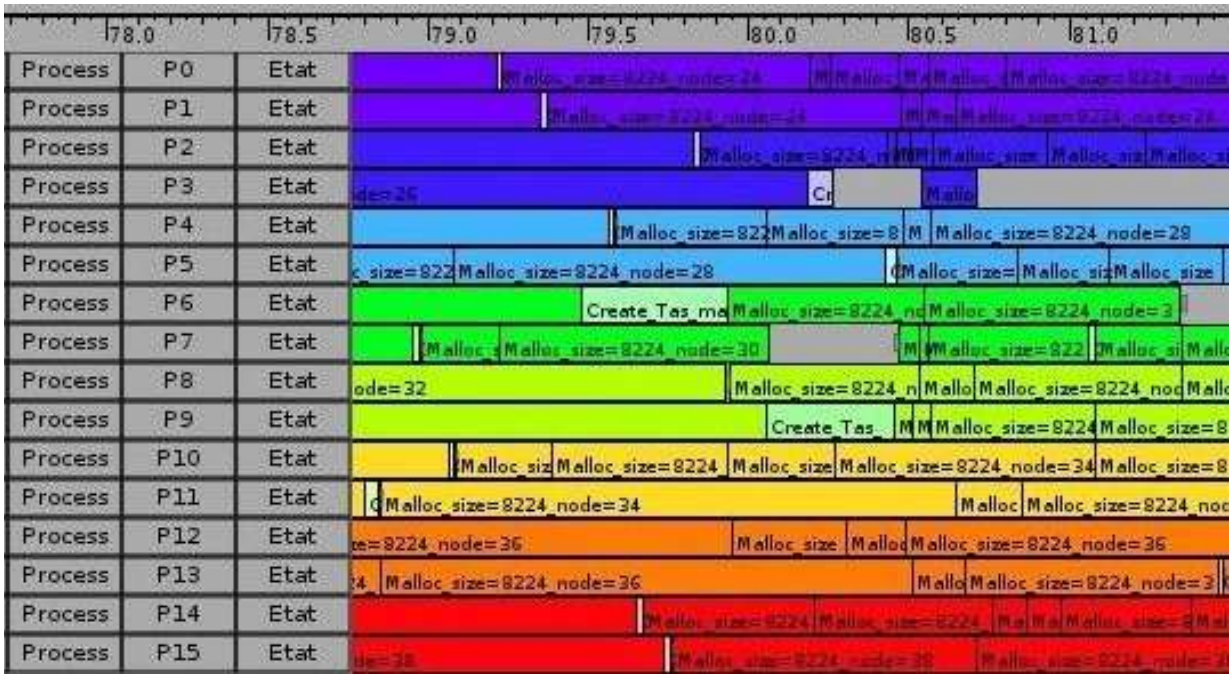

FIG. 6 – Exemple de visualisation des appels aux méthodes de la librairie HEAPALLOC

```
/* initialisation de la librairie de collecte des données */
int init_memon();

/* finalisation de la librairie de collecte des données */
int  finalize_memon();

/* attachement d'une zone mémoire à visualiser */
int  attach_memory(char *name, void *ptr, size_t size);

/* détachement d'une zone mémoire à visualiser */
int  detach_memory(char *name, void *ptr, size_t size);
```

FIG. 7 – Interface de MEMON

méthodes sont appelées dans le programme cible qui est lié avec la librairie MEMON durant la phase d'édition des liens, comme le montre la figure 8.

La deuxième partie consiste en un programme MEMTOP qui visualise les données collectées par MEMON. Ce programme utilise un interface graphique textuelle obtenue par la librairie NCURSES[10]. Les données visualisées consistent en la taille et le pourcentage des données par nœuds NUMA et pour chacune des variables. De manière similaire, un récapitulatif de l'occupation totale de l'ensemble des variables sélectionnées est également affiché. Chaque variable est identifiée par la chaîne de caractères utilisée dans les appels aux méthodes de sélection de MEMON.

Les outil MEMON et MEMTOP peuvent être utilisés avec des applications multithreadées. De plus, plusieurs applications liées à la librairie MEMON peuvent être utilisées simultanément avec une seul instance du programme MEMTOP. En effet, MEMTOP affiche également l'identificateur système du processus (PID) pour chacune des variables à observer.

La figure 10 est un exemple des données observées. Dans cet exemple, nous avons lancé deux applications instanciant chacune deux threads. Chacun des threads a attaché au système de collecte une variable ayant le nom "dyn grid". Les threads appartenant au processus de PID=6412 ont les pages de données de leurs variables exclusivement sur le nœud NUMA numéro 2, alors que les threads du processus de PID=6488 sont réparties de manières équitable sur chacun des nœuds.

Principe de fonctionnement :

Pour obtenir de manière dynamique les informations de placement des pages, la méthode **init_memon** initie un thread chargé de la collecte des données. Ce thread exécute de manière périodique l'appel système **move_pages** sur l'ensemble des pages des variables attachées grâce à la méthode **attach_memory**. Cet appel système permet d'obtenir le numéro du nœud NUMA pour une page donnée.

La communication des données collectés entre MEMON et MEMTOP est réalisée à travers un pipe nommé, qui correspond à un fichier ayant des accès du type FIFO (First-In First-Out). Ce fichier est ouvert en écriture par le thread instancié par l'appel à la méthode **init_memon**. Il est ouvert en lecture par MEMTOP. Ainsi, MEMTOP peut rafraîchir de manière périodique les informations collectes dans le pipe nommé par MEMON. La figure 9 résume le principe de fonctionnement de MEMTOP.

4 Visualisation des communications sur grappe de machines NUMA

Les applications scientifiques parallèles sont conçues pour exploiter les machines multiprocesseurs et en particulier lorsqu'elles sont regroupées entre elles au travers de grappes de calcul. Pour pouvoir utiliser efficacement des grappes de calcul, les applications sont programmées suivant un modèle de programmation par passage de message. L'analyse de performance qu'il est possible d'obtenir, grâce en particulier à des outils de visualisation, est plus complexe à cause de l'aspect distribué des telles architectures.

Dans cette section, nous proposons un outil permettant de visualiser le comportement des communications d'une application distribuée sur une grappe. Cet outil se base sur l'interface de communication MPI. Il utilise PAJÉ[13] pour le rendu final et offre à l'utilisateur la possibilité de mettre en évidence certains comportements tels que ceux responsables de contention réseau.

```
int main(int argc, char *argv[])  {
  ...

  /* initialisation de memon */
  /* cette méthode crée le thread collectant les données de placement */
  init_memon();
  ...

  /* création des threads de calcul */
  for (i = 0; i < numWorkers; i++)
      pthread_create(&workerid[i], &attr, Worker, (void *) i);
  ...

}

void *Worker(void *arg) {

  /* allocation de la zone mémoire nécessaire aux calculs */

  grid1 = (double**)malloc(size_grid*sizeof(double*));
  grid2 = (double**)malloc(size_grid*sizeof(double*));

  for(i = 0; i <= strip; i++) {
    grid1[i] = (double*)malloc(size_grid*sizeof(double));
    grid2[i] = (double*)malloc(size_grid*sizeof(double));
  }

  /* attachement d'une zone mémoire afin d'en visualiser son placement */
  /* cette variable est identifiée dans memtop par ``dyn grid'' */
  attach_memory(``dyn grid'',grid1,size_grid*size_grid*sizeof(double));

  ...calcul du jacobi...

}
```

FIG. 8 – Exemple d'un jacobi utilisant la librairie MEMON

4.1 Contexte

La ressource réseau est un élément important à prendre en considération pour exploiter au mieux les architectures distribuées. Le réseau fait partie des éléments pénalisant le plus le temps d'une exécution parallèle. Lors du développement d'algorithme parallèle, il est fréquent de choisir un schéma de communication prédéfini ou communément utilisé. Par exemple, les schémas en anneau ou en grille sont des schémas de communication largement répandus. Toutefois certaines applications non régulières en terme de communication ne peuvent pas profiter de ce type de schéma de communication. C'est en particulier le cas de solveur de matrices creuses qui génère des communications irrégulières en fonction de la matrice à résoudre. La tâche 2.3 du WP2 tâche 2.3 du projet NUMASIS est consacrée à l'étude d'algorithmes

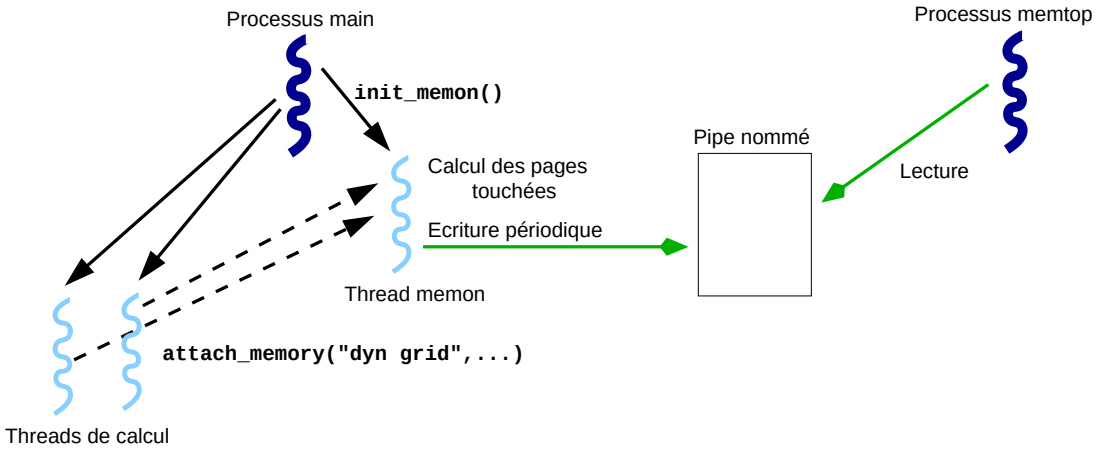

FIG. 9 – Architecture et principe de fonctionnement de MEMTOP

FIG. 10 – Exemple d'une exécution de MEMTOP

scientifiques efficaces sur des ensembles de machines NUMA. L'application PASTIX[6] est une application de référence pour ces travaux. PASTIX est un solveur de matrices creuses utilisant l'interface MPI. Nous nous proposons d'utiliser PASTIX à titre d'exemple d'utilisation de l'outil de traçage et de visualisation d'applications MPI.

En outre, des travaux ont déjà étudié la contention réseau de manière fine [7]. Cette étude a pour objectif de modéliser les phénomènes liés à l'apparition de communications en concurrence en vue de déterminer les performances d'applications tracées sur de nouveaux types de réseau. Cette prédiction est réalisée par simulation. L'outil de traçage que nous proposons s'interface également avec cette simulation proposant des traces compatibles.

4.2 Traçage

Le traçage d'une application MPI est réalisée en capturant les appels aux différentes méthodes MPI. Lorsqu'une méthode de communication est appelée par l'application une première méthode capture l'appel est construit un nouvel événement dans la trace puis exécute la méthode effective de la librairie MPI. Ce mécanisme de capture est inclus dans une bibliothèque générant la trace MPI qu'il convient de lier avec l'application à tracer. Le format de la trace est un format binaire contenant les données des événements. Un outil a été développé pour manipuler ce fichier de trace et le convertir soit en fichier textuel soit en format de trace PAJÉ. Cet outil reconstruit également l'ordre logique des appels en corrélant des événements tels que par exemple un envoi et une réception grâce à une système d'identifiant. Cette reconstruction est capable de gérer des envois et des réceptions asynchrones ainsi que des propriétés de la librairie MPI telle que l'utilisation de `MPI_ANY_SOURCE`.

4.3 Visualisation

La figure 12 est une trace obtenue pour l'application PASTIX. Cette trace a été convertie au format PAJÉ. Elle permet de mettre en évidence une zone de congestion (en rouge sur la figure). En effet, à ce moment de l'exécution, un grand nombre de tâches MPI communiquent simultanément sur le réseau.

Il devient intéressant de comprendre les impacts de ces phénomènes de congestion, afin de pouvoir les réduire. Nous pouvons, par exemple, supposer qu'un placement des tâches MPI différent sur les machines de la grappe pourrait tendre à réduire ces phénomènes de congestion.

5 Conclusion et travaux futurs

5.1 Résumé des développement

La tâche 1.3 du WP1 est focalisée sur la réalisation d'outils logiciels permettant de visualiser le comportement d'applications parallèle sur des architectures NUMA. Le but des ces outils est de proposer aux concepteurs d'applications un plus grande compréhension de l'exploitation de ces architectures NUMA par leurs applications.

Pour pourvoir visualiser de manière dynamique ou post mortem le comportement applicatif, une première étape se basant sur la collecte d'information ou le traçage doit-être réalisée. Dans ce document, les outils que nous avons proposé sont principalement positionnés dans cette première étape. La partie visualisation est obtenue par le logiciel PAJÉ.

```
Evénement calcul: type id, date, catégorie, numéro du processus, description
Evénement com: type id, date, catégorie, numéro du processus source,
description, numéro du processus destination, taille

10 19.253586308 Etat P4 Appli
10 19.267428382 Etat P3 Appli
10 19.318789253 Etat P3 MPI_Recv
10 19.318843162 Etat P3 Appli
17 19.318843162 Comm R1 SEND(17416) P3 2317416
10 19.339199314 Etat P3 MPI_Recv
10 19.339692409 Etat P3 Appli
17 19.339692409 Comm R1 SEND(37848) P3 2337848
10 19.351962020 Etat P3 MPI_Recv
17 19.354440181 Comm R1 SEND(70496) P3 2370496
10 19.358932875 Etat P3 MPI_Recv
17 19.358964656 Comm R1 SEND(7360) P3 237360
```

FIG. 11 – Exemple d'une trace PAJÉ de PASTIX

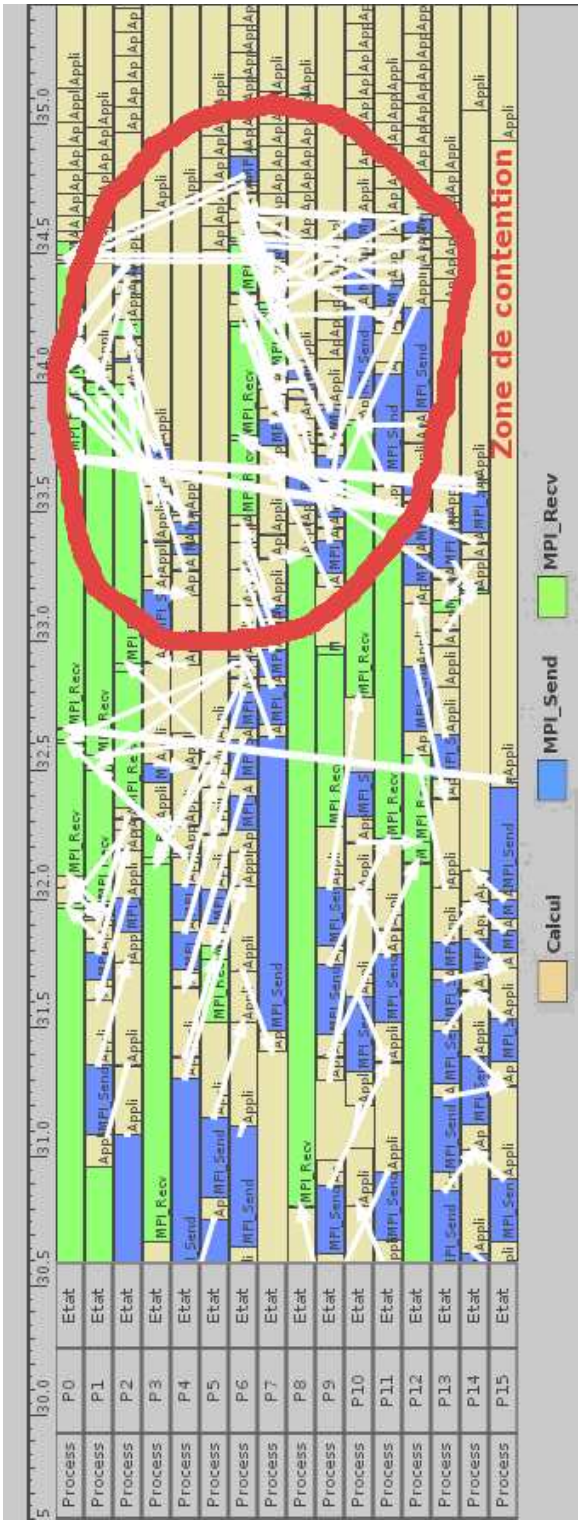

FIG. 12 – Exemple de visualisation des appels MPI de PASTIX

Un premier outil de récolte d'information concerne le traçage des valeurs de compteurs matériels au niveau des threads de calcul. Ce traçage permet d'indiquer à l'utilisateur des pertes de performance, par exemple, quels sont les threads qui provoquent le plus de défauts de cache. Cet outil utilise l'interface PAPI pour accéder aux compteurs matériels.

Le deuxième outil présenté permet de tracer les allocations dynamique sur les architectures NUMA. Il est intégré à la bibliothèque HEAPALLOC est indique, de manière temporelle, à l'utilisateur les paramètres d'appel des méthodes d'allocation dynamique (en particulier le placement choisi sur les nœuds NUMA).

Le troisième outil est un outil qui indique de manière dynamique l'utilisation de l'architecture mémoire NUMA par des variables d'un programme. Cet outil se compose d'une librairie définissant des méthodes pour attachées des zones mémoires à scruter ainsi que d'un programme visualisant dynamiquement le placement des pages mémoires de ces variables.

Finalement le dernier outil est un outil de traçage des appels aux méthodes MPI d'une application scientifique distribuée sur une grappe de calcul. Il permet de mettre en évidence des comportements de congestion réseaux entre communications concurrentes.

5.2 Travaux futurs

Les travaux futurs se portent essentiellement sur la continuation et l'amélioration des outils proposés. En outre, ces outils vont être mis à disposition de la communauté et du projet NUMASIS au travers la gforge INRIA[3]. Les développements futurs concernent principalement le logiciel MEMTOP. Pour cet outil, nous prévoyons d'intégrer une interface Fortran et d'améliorer les performances de la collecte des données.

La rédaction de la documentation détaillée du fonctionnement et des possibilité qu'offre ces outils est également une étape prévue dans la continuation de cette tâche.

[3] `http://gforge.inria.fr/projects/numasis/`

Bibliographie

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3) :189–204, 2000.

[2] Vincent Danjean. Mécanismes de traces efficaces pour programmes multithreadés. In Michel Auguin, Françoise Baude, Dominique Lavenier, and Michel Riveill, editors, *Actes de RenPar'15, CFSE'3, SympAAA'2003*, pages 92–100, La Colle sur Loup, France, October 2003. INRIA. ISBN 2-7261-1264-1.

[3] Vincent Danjean. *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs.* PhD thesis, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, December 2004. 156 pages.

[4] Vincent Danjean and Pierre-André Wacrenier. Mécanismes de traces efficaces pour programmes multithreadés. *TSI*, 24, May 2005. version longue de [2].

[5] S. Graham, P. Kessler, and M. McKusick. gprof : A call graph execution profiler. In *SIGPLAN Notices*, 1982.

[6] P. Hénon, P. Ramet, and J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, January 2002.

[7] Maxime Martinasso. *Analyse et modélisation des communications concurrentes dans les réseaux haute performance.* PhD thesis, Université Joseph Fourier, 2007.

[8] Aurélien Pralong. Optimisation de l'affinité mémoire sur de machines multiprocesseurs NUMA. Rapport de master 2, LIG, 2007.

[9] GNU Project. Core utilities. `http://www.gnu.org/software/coreutils/`.

[10] GNU Project. Ncurses. `http://www.gnu.org/software/ncurses/`.

[11] Robert D. Russell. FKT : Fast Kernel Tracing. Technical Report 00-02, University of New Hampshire, 2002.

[12] Robert D. Russell and Mrinalini Chavan. Fast Kernel Tracing : a Performance Evaluation Tool for Linux. In *Proc. 19th IASTED International Conference on Applied Informatics (AI 2001)*. IASTED, February 2001.

[13] B. De Oliveira Stein. *Visualisation interactive et extensible de programmes parallèles à base de processus légers.* PhD thesis, Université Joseph-Fourier Grenoble I, October 1999. 148 pages.

[14] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3) :263–276, Fall 1999.

[15] Karim Yaghmour and Michel R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceeding of the 2000 USENIX Annual Technical Conference*, June 2000.