

Mémoire de stage Master 2

Stage du 1er février 2023 au 31 juillet 2023

Intitulé : Interruptions en espace utilisateur pour le réseau BXI

Charles GOEDEFROIT
Encadré par Alexandre DENIS (Inria), Grégoire PICHON (Atos) et Mathieu BARBE (Atos)

7 août 2023

Table des matières

1 Introduction

J'ai effectué mon stage de fin de master de six mois dans l'équipe-projet *TADaaM* du Centre Inria de l'université de Bordeaux. Le sujet du stage a été proposé par Alexandre Denis (Inria) et Grégoire Pichon (Atos). Ils ont aussi encadré le stage avec Mathieu BARBE (Atos).

1.1 Présentation Inria

Inria est l'institut national français de recherche en sciences et technologies du numérique. Il compte plus de 3 900 chercheurs et ingénieurs au sein de 215 équipes-projets. La plupart des centres sont communs avec de grandes universités.

1.2 Présentation Atos (Eviden)

Atos est l'un des leaders internationaux de la transformation numérique. Elle couvre un large éventail d'activités, notamment le cloud, la cybersécurité, les services transactionnels, le conseil, l'infogérance, le Big Data, les supercalculateurs, etc. *Atos* compte 112 000 collaborateurs. Actuellement, *Atos* est en restructuration afin de se séparer en deux entités. L'entité qui nous intéresse pour ce stage est Eviden, qui englobe notamment les activités liées aux supercalculateurs et au HPC. Cette restructuration est récente, donc lorsque nous évoquons *Atos* dans ce document, nous parlons de la partie Eviden.

Atos est le seul grand constructeur européen de supercalculateurs. Elle est donc le leader européen et est bien positionnée mondialement. Elle est présente dans le secteur du HPC depuis l'acquisition de la société *Bull* en 2014. *Bull* était déjà impliquée dans le HPC depuis les années 2000 et en avait fait son coeur de métier.

La machine *Leonardo* en Italie, construite par *Atos*, se classe quatrième au dernier *Top500*. Ce classement évalue les machines les plus puissantes pour le HPC et est publié deux fois par an.

1.3 Environnement de travail

Mon environnement de travail se trouve dans les locaux du Centre Inria de l'université de Bordeaux. Un bureau dans l'open space de l'équipe-projet *TADaaM* a été mis à ma disposition. Je peux participer et assister à des activités scientifiques intéressantes, telles que des séminaires, des soutenances de thèse, une soutenance HDR et d'autres activités diverses. Nous avons également accès aux salles de visioconférence, notamment pour les réunions de suivi de stage hebdomadaires. En plus, il y a un Baby-foot, une cafétéria, une petite médiathèque, etc.

1.4 Le cadre

Du côté de l'équipe *TADaaM* d'Inria, l'objectif de l'équipe est de mener des recherches sur les sujets suivants :

— Gestion des I/O (ordonnancement, bande passante...)

— Placement de processus

— Partitionnement de maillage (i.e. SCOTCH)

— Localité matérielle (i.e. hwloc)

— Optimisation des communications pour les réseaux haute performance, MPI (i.e. NewMadeleine)

Elle est composée de chercheurs, d'ingénieurs de recherche, de post-doc, de doctorants et de stagiaires. Sa culture informatique est l'utilisation d'environnements Linux, de logiciels open source, de clusters de calcul HPC, le traitement des données et le systèmes. L'équipe a mis à ma disposition un ordinateur portable avec une station de travail reliée à un écran, à Internet, à un clavier et une souris. Inria donne également l'accès à un ensemble de services, notamment une boîte e-mail, un service de communication (Mattermost), un service de visio (Webex) et un intranet avec le menu de la cafétéria, entre autres. Nous sommes libres d'installer le système d'exploitation et d'utiliser les outils informatiques de notre choix.

Du côté d'*Atos* l'équipe BXI-LL (BXI Low Level) a pour objectif de fournir le support logiciel bas niveau pour les cartes réseaux *BXI*. Cela comprend le développement des pilotes de la carte, le support Lustre (un système de fichiers parallèle et distribué utilisé pour l'I/O, qui ne fait pas partie de ce stage), ainsi qu'une implémentation de l'interface logicielle *Portal4*. *Portal4* est une interface logicielle qui permet une abstraction des opérations possibles sur les réseaux HPC. Son but est de faciliter l'utilisation de ces opérations réseau avec les meilleures performances possibles pour les implémentations MPI. /* je suis pas sur de ma présentation de Portal4 */ L'implémentation d'*Atos* permet aux implémentations MPI d'utiliser le réseau *BXI*. Le personnel de l'équipe est composé d'ingénieurs, de doctorants et de stagiaires. La culture informatique de l'équipe est l'utilisation de clusters HPC, la programmation système et l'utilisation d'un système de sécurité PKI pour accéder aux ordinateurs et aux services internes. *Atos* nous a donné accès à une machine avec deux CPU Intel® Sapphire Rapids. Pour accéder à cette machine, ils nous ont fourni un accès SSH et un compte VPN. Nous avons eu tous les droits d'accès sur cette machine pour changer le noyau du système.

1.5 Présentation du plan

Dans ce mémoire, nous commencerons par une présentation du contexte dans lequel le stage se place. Ensuite, nous aborderons les objectifs à long terme ainsi que les objectifs du stage. Nous continuerons en décrivant le

nouveau mécanisme d'interruption en espace utilisateur et ce que nous en avons fait. Ensuite, nous verrons l'intégration de ce mécanisme dans la bibliothèque de communication NewMadeleine, ainsi que les tests que nous avons effectués. Pour finir, nous ferons un bilan du stage et évoquerons les travaux qui suivront.

2 Contexte

2.1 HPC

Le stage s'est déroulé dans le contexte du Calcul Haute Performance (HPC, High-Performance Computing en anglais). Le HPC utilise des supercalculateurs pour la simulation numérique et le pré-apprentissage d'intelligences artificielles. Ces simulations simulent la dynamique des fluides, la résistance structurelle, les interactions moléculaires, les flux d'air...

Elles couvrent différents domaines :

- L'industrie : médical, automobile, aviation, construction, aérospatiale...
- La défense : simulation atomique...
- La recherche scientifique : création de galaxies, fusion nucléaire, climat...
- La météo

De nos jours, les supercalculateurs sont composés de plusieurs ordinateurs que l'on appelle noeuds de calcul. Ceux-ci sont regroupés en grappes (clusters) et sont tous vus et utilisés comme une seule grande machine. Ce fonctionnement soulève des questions concernant la répartition du calcul, la distribution de la mémoire et la communication entre les différents noeuds. Dans le contexte du stage, nous nous sommes concentrés sur les communications entre noeuds. Les noeuds sont connectés entre eux par un réseau haute performance dédié aux communications, qui n'est pas forcément de type Ethernet. La gestion des noeuds est généralement effectuée par un second réseau plus traditionnel (Ethernet ; TCP/IP). Les réseaux haute performance ont une latence de l'ordre de quelques microsecondes ce qui n'est pas le cas des réseaux classiques qui ont une plus grande latence. Avec ce type de réseaux, le débit est aussi bien plus élevé. Chaque noeud possède une ou plusieurs cartes réseau et il faut les programmer.

2.2 OS bypass

Le stage se passe donc aussi dans un contexte système. Chaque noeud a son propre système d'exploitation qui permet la gestion des ressources, des processus, des fichiers, des périphériques. Pour cela, le système a 2 espaces :

— Un espace noyau (kernel en anglais) où seul le code du système peut s'exécuter. Le code du système peut donc modifier n'importe quel endroit de la mémoire, exécuter n'importe quelle instruction...

— Un espace utilisateur où le code de l'utilisateur est exécuté. Cet espace est limité par le noyau qui contrôle ce que fait l'utilisateur.

En temps normal, les périphériques sont programmés directement depuis le noyau du système d'exploitation, cela est fait pour des raisons de sécurité et de standardisation des accès. Cependant, lorsque l'on passe par le noyau, il y a un surcoût qui n'est pas négligeable dans notre cas. En effet, pour passer par le noyau, on utilise généralement des appels système qui prennent la forme d'une fonction. Un appel système effectue un changement de contexte (context switch) pour passer de l'espace utilisateur à l'espace noyau. Ce changement de contexte est coûteux car il sauvegarde les états du code de l'utilisateur avant d'exécuter celui du noyau. save états du process (registres, registre vectoriel, le pc, esp, flags...), Une fois le changement de contexte effectué, le code noyau de l'appel système s'exécute avant de refaire un changement de contexte pour, cette fois, passer de l'espace noyau à l'espace utilisateur et ainsi restaurer l'état du code de l'utilisateur. Le noyau peut aussi être amené à mettre à jour ses structures internes, exécuter des tâches qu'il avait mises à faire plus tard, etc. Donc, le fait de passer par un appel système coûte plusieurs microsecondes. On voit donc qu'il n'est pas préférable d'utiliser les appels système, car leur durée est du même ordre de grandeur qu'une communication entre noeuds. En HPC, on programme donc directement la carte réseau à partir de l'espace utilisateur (OS bypass en anglais). Pour cela, on initialise toujours la carte à partir du noyau, mais on fait une projection de la mémoire et des registres de la carte réseau dans l'espace d'adressage virtuel du processus utilisateur.

Pour transmettre des événements à l'utilisateur, les périphériques utilisent généralement les interruptions ordinaires, qui passent par le système donc on les utilise très peu en HPC. En HPC on privilégie donc le polling que nous verrons en section 2.4.

2.3 Les interruptions matérielles

Les interruptions matérielles, également appelées IRQ pour "Interrupt ReQuest", existent depuis longtemps et servent à remonter des événements extérieurs, par exemple d'un clavier, d'un port de communication, d'un port IDE, etc. Elles ont ensuite été utilisées pour signaler des événements en provenance de périphériques plus variés, comme des cartes réseau. Dans ce document, nous utiliserons le terme "interruption ordinaire" ou "IRQ" pour les désigner.

2.4 Polling

Il faut donc, peut importe la technique, régulièrement scruté pour faire progressé les communications.

Le *polling* consiste à scruter (poll) régulièrement si un événement a été reçu. Concrètement, cela consiste à lire une zone mémoire modifiée par la carte réseau et vérifier si un bit est passé à 1. Pour cela, il est possible de dédier un thread qui effectuera une attente active, scrutant sans cesse si un événement a été reçu. Cependant, cette technique entraîne la perte d'une unité de calcul quand le thread est ordonnancer, ce qui réduit la puissance de calcul, donc elle est utilisée dans certaine cas comme expliqué dans cette article [4]. Une autre approche consiste à entrelacer le code de l'utilisateur avec les scrutations, ce qui est fréquemment utilisé, mais cela oblige l'utilisateur à prendre en compte la progression. Une troisième solution, utilisée par *Pioman* dans *NewMadeleine* (nous en parlerons plus tard), consiste à effectuer ces scrutations de manière opportuniste dans les threads qui ont fini leur calcul, mais pour cela, il faut déjà utiliser plusieurs threads et avoir une application avec des calculs de durée hétérogène.

2.4.1 Inconvénients du polling

Dans le cas d'un thread dédié qui effectue une attente active, la réactivité est excellente, sauf lorsque le nombre de threads dépasse le nombre d'unités de calcul et lorsque le thread n'est pas ordonnancé.

Dans le cas où l'on entrelace le calcul et les scrutations, la réactivité est moins bonne car il faut attendre que le calcul soit terminé pour effectuer un poll. C'est ce qui est habituellement fait.

Dans le cas où l'on utilise les threads de façon opportuniste pour effectuer des scrutations, la réactivité est moins bonne car il faut qu'il y ait un thread disponible.

Un choix doit donc être fait entre perdre de la capacité de calcul ou perdre en réactivité. De plus, un peu de temps de calcul est perdu à effectuer du polling.

Dans le cas où l'on effectue trop de scrutations, on peut rencontrer des problèmes de contention mémoire. En effet, le CPU manipule la mémoire par lignes de 8 mots mémoire appelés lignes de cache. Ces lignes de cache font 512 bits sur un CPU 64 bits (car $8 * 64 = 512$), et 256 bits pour des CPU 32 bits (car $8 * 32 = 256$). Pour garantir la cohérence d'accès à la mémoire, ces lignes de cache sont invalidée lorsque quelqu'un d'autre les manipule. Ainsi, si un thread scrute trop souvent une zone mémoire ou que plusieurs threads cherchent à scruter la même zone mémoire, on aboutit à des problématiques de contention de la mémoire, car la ligne de cache est invalidée plus souvent, ce qui entraîne un surcoût. Avec le polling nous somme donc contrains à respecter une granularité lié à la taille d'une ligne

de cache. Utiliser les interruptions permettrait de ne plus avoir ce problème, car seul le thread concerné par une zone mémoire serait prévenu donc plus de contention à ce niveau.

En plus, utiliser des interruptions permettrait de ne plus faire des scrutations pour rien, mais seulement scruter lorsqu'il y a un événement à traiter. Le nombre de scrutations serait donc grandement réduit.

2.5 BXI

BXI [6] pour "Bull eXascale Interconnect" est un type de réseau d'interconnexion (réseau haute performance) développé par *Atos*. Historiquement développé par *Bull* qui a été racheté par *Atos*, ce réseau est dédié aux communications entre noeuds. Il est composé de cartes réseau *BXI* et de switches *BXI*. Les cartes *BXI* sont capables de faire progresser les communications réseau sans aucune intervention du CPU (offload des communications réseau). Le CPU a juste à soumettre une commande dans une file sur la carte, et elle s'occupe de tout. Le CPU peut ensuite récupérer une file d'événements pour savoir ce qu'il s'est passé, en somme faire un poll. Les cartes sont également capables de déclencher des interruptions. Les cartes implémente donc l'API de communication *Portal4* [25]. Les utilisateurs utilisent donc cette API pour envoyer et recevoir des paquets réseau.

2.6 MPI

MPI, pour "Message Passing Interface", est un standard permettant de fournir une interface pour effectuer des communications entre plusieurs processus, souvent situés sur des noeuds différents, que l'on appelle *processus MPI*. Cette interface fournit une abstraction pour transmettre des données entre ces différents processus, masquant ainsi la complexité des communications. L'interface permet d'envoyer et de recevoir des messages. Pour cela, il existe deux modes de communications :

2.6.1 Les communications point à point

C'est-à-dire entre deux processus MPI, également appelé One-to-One. Pour ce faire, le processus MPI récepteur va appeler la fonction `MPI_Recv` qui est bloquante et va attendre la réception d'un message. L'émetteur va lui faire un appel à la fonction `MPI_Send` qui est également bloquante et va envoyer un message, puis attendre que la communication soit terminée. L'utilisation des fonctions `MPI_Send` / `MPI_Recv` bloque le code, ce qui nous fait perdre du temps à attendre. Le standard MPI propose également une version non bloquante de ces fonctions, qui sont `MPI_Isend` et `MPI_Irecv`. Cette version se contente de poster la communication et rend immédiatement la main. Pour la progression et vérifier la terminaison, il faut donc utiliser d'autres fonctions comme `MPI_Test`, qui vérifie la progression

et la fait si nécessaire, et la fonction `MPI_Wait`, qui attend activement la terminaison et s'occupe de la progression si nécessaire. Il est important de noter que le standard MPI ne précise pas si la progression se fait en tâche de fond ou non, c'est aux implémentations de la norme MPI de choisir. C'est pour cela que la progression peut se faire au niveau des fonctions `MPI_Wait` et `MPI_Test`, ou être faite avant, et donc l'appel aux fonctions s'occupe juste de la terminaison. L'envoi des messages peut donc être asynchrone. /* gardé cette phrase ? */

2.6.2 Les communications collective

Les communications collectives se font entre plusieurs processus. Il en existe différents types :

- Un processus vers plusieurs (One-to-All), par exemple un broadcast d'un message.
- De plusieurs processus vers un seul (All-to-One), par exemple une réduction (e.g. un processus reçoit la somme des valeurs des autres processus).
- De plusieurs processus vers plusieurs (All-to-All), par exemple lorsque tous les processus ont un message pour les autres.

Pour les communications collectives, il existe également deux versions, bloquante et non bloquante, qui fonctionnent de la même façon que les communications point à point.

2.6.3 La progression en tâche de fond

Pour faire progresser les communications en tâche de fond, il est possible de :

- Faire régulièrement des appels à `MPI_test` et effectuer des calculs entre chaque appel. Cela permet de recouvrir la latence des communications par du calcul. La progression peut également se faire dans d'autres appels aux fonctions MPI. Lorsqu'il n'y a plus de calcul à effectuer, on repasse à une progression bloquante par un appel à `MPI_Wait`, sauf si la communication est déjà terminée.
- Utiliser un thread dédié aux progressions. Dans ce cas, c'est la bibliothèque MPI ou l'application qui s'occupe de la progression des communications en tâche de fond grâce à un thread dédié. Il faut donc faire attention au placement des threads et prendre en compte qu'un thread est déjà utilisé pour les communications. Il faut aussi éviter d'appeler trop souvent `MPI_Test` car cela peut créer de la contention.
- Utilisation des threads de façon opportuniste, c'est-à-dire qu'un des threads, une fois son calcul terminé, fera progresser les communications. C'est ce qui est fait par *Pioman* dans *NewMadeleine*.

2.7 NewMadeleine

NewMadeleine est une bibliothèque de communications qui prend en charge le RPC pour "Remote Procedure Call" et implémente également une interface MPI. On peut donc la considérer comme une implémentation du standard MPI avec des fonctionnalités supplémentaires. Elle est développée au Centre Inria de l'université de Bordeaux. Elle est basée sur un système de progression événementielle, ce qui lui permet d'être asynchrone. Elle est composée de modules, ce qui lui permet de charger dynamiquement des stratégies d'optimisation sur les paquets. Ces stratégies sont l'agrégation de paquets, l'utilisation de priorités et le multi-rail. /* TODO : précision multi-rail/split_balance */. Elle possède également un système de drivers pour supporter différents types de communications, tels que des réseaux (e.g. portals4 pour BXI, ibverbs pour InfiniBand, psm2 pour OmniPath, ofi pour "Open Fabrics Interfaces", TCP) et des communications locales (shm pour "mémoire partagée", socket, self).

2.8 Les signaux

Les signaux[19] sont l'un des mécanismes permettant l'interface entre un processus et le système. Pour cela, le processus utilisateur effectue des appels système pour définir les signaux pour lesquels il souhaite être notifié. Pour être notifié, l'utilisateur peut définir un handler pour chaque signal auprès du système. Ces handler sont déclenchés par le noyau qui prend en charge toutes les opérations nécessaires (sauvegarde de l'état du processus, changement de contexte, etc.).

Les signaux que l'utilisateur peut recevoir correspondent à des exceptions système (e.g. *SIGFPE* pour une division par zéro ou *SIGSEGV* pour une erreur de segmentation), à des informations (e.g. *SIGTERM* pour demander au processus de se fermer) ou à des notifications d'autres processus (i.e. *SIGUSR1* et *SIGUSR2*). Il est possible de masquer la réception de certains signaux en utilisant des appels système de masquage.

2.9 Travaux antérieur

TODO : Amélioré ? le faire pour tous ce qui existe

Les origines de *NewMadeleine* sont décrites dans cet article [1]. Le système de progression dans *NewMadeleine* fait l'objet de plusieurs travaux. le fonctionnement du système de progression opportuniste *Pioman* est décrit dans cet article[3]. Pour l'impact de l'overlap et la manière de le mesurer, l'article [5] explique tout en détail. Concernant l'utilisation des interruptions pour transmettre des événements réseau, les travaux de Mathieu Barbe pendant son stage en 2019 sont disponibles ici [2]. Ces travaux visent à réduire la latence due au passage par un driver noyau. Ils abordent également les perspectives de traitement direct des interruptions depuis l'espace utilisateur, ce

qui a conduit à ce stage.

2.10 Définitions

Il est important d'avoir les définitions suivantes en tête :

— *CPU* désigne la puce dans sa totalité.

— *core* ou *processeur* désigne l'un des coeur du *CPU*.

— *core logique*, *processeur logique* ou *unité de calcul* désigne une unité de calcul au sein d'un coeur. Il y en a deux par core dans les CPUs Intel® Sapphire Rapids fourni par *Atos*.

/* TODO : besoin d'expliquer ? : Un processus peut avoir plusieurs threads, un thread correspond à un fil d'execution de code. Et tous les threads au sien d'un même processus partage le même espace d'adressage virtuel. */

3 Problématiques / Objectifs

3.1 Sujet

Le sujet est *Interruptions en espace utilisateur pour le réseau BXI* [16].

3.1.1 Nouveau mécanisme d'interruption en espace utilisateur

Ce nouveau mécanisme qui permet de dérouler une interruption à partir de l'espace utilisateur est très récent. Il est seulement disponible sur les CPU Intel® Sapphire Rapids qui sont officiellement sortis le 10 janvier 2023. [1] La plupart des CPU ont été disponibles à la vente le 14 mars. [2] *AMD* n'a pas encore annoncé de support pour les interruptions en espace utilisateur.

3.2 Projet global

Dans la plupart des autres domaines, les périphériques envoient une interruption ordinaire à l'application, par le biais des signaux ou d'un appel système bloquant, pour avertir d'un changement. L'idée serait de faire la même chose en HPC grâce aux interruptions en espace utilisateur. Le projet global vise donc à faire progresser les communications entre plusieurs nœuds du réseau *BXI* sans faire de polling et en utilisant les interruptions en espace utilisateur. Cela permettrait de réduire globalement le temps de calcul d'une application. Pour ce faire, la carte réseau *BXI* devra être capable de lever des interruptions en espace utilisateur. Le fait de supprimer le polling

1. https://www.datacenterknowledge.com/intel/intel-launches-sapphire-rapids-after-4-delays-it-worth
2. https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-intel-xeon-sprints-into-market.html

et de réduire le temps de calcul permettra de diminuer la consommation électrique.

3.2.1 Les objectifs

Il y a plusieurs objectifs, les principaux sont les suivants :

— La réduction du temps de calcul.

— Utiliser des interruptions en espace utilisateur pour remplacer le polling.

— Simplifier pour l'utilisateur le recouvrement des communications par du calcul, afin qu'il n'ait plus besoin d'ajouter des `MPI_Test` en plein milieu des calculs.

— Améliorer la réactivité des communications sans avoir besoin d'une unité de calcul dédiée à l'attente active.

Ce stage est donc une première étape de ce projet global.

3.3 Objectifs du stage

Le premier objectif est de défricher le fonctionnement des interruptions en espace utilisateur à partir des éléments suivants :

— Le manuel Intel® de l'architecture 64 et IA-32 pour les développeurs logiciels [13].

— La présentation du mécanisme de *Sohil Mehta*, ingénieur chez Intel®, qui a développé le patch noyau. Cette présentation est une diapositive associée à des discussions sur LWN.net [29].

— Le patch du noyau Linux [14] avec ses manuels [15].

Le second objectif est de connaître les propriétés du mécanisme et de mesurer sa performance. Le troisième objectif est de ne plus avoir à faire du polling, que ce soit en dédiant un thread ou en utilisant les threads de façon opportuniste, afin de ne plus perdre de temps de calcul à poll et de résoudre les problèmes de réactivité.

Le troisième objectif est de montrer que l'utilisation des *Uintr* dans les communications HPC est possible. Pour cela, on se place dans un cadre simplifié :

— On se concentre sur les communications entre processus (IPC, Inter-Process Communication en anglais). Pour cela, on va utiliser la mémoire partagée (shm) pour communiquer au lieu du réseau *BXI*.

— On se limite aux cas généraux des communications (donc pas de gros messages, pas de communications multi-thread, etc).

On envisage donc l'intégration de ces interruptions dans le driver de mémoire partagée (shm) de NewMadeleine, ce qui nous permet une utilisation réelle pour voir ce que donnerait une version complète.

Pour intégrer les interruptions dans les drivers de *NewMadeleine*, il faut également permettre la progression des communications à partir d'un handler d'interruption. Le dernier objectif est de montrer que l'utilisation d'interruption permet bien d'améliorer le recouvrement des communications par du calcul.

3.4 La suite

Les objectifs suivants du projet global seront traités dans une thèse qui fait suite au stage.

4 Exploration des interruptions en espace utilisateur.

Pour cette partie, j'ai utilisé mes connaissances personnelles autour du système, du développement noyau, de Linux, ainsi que ce que j'ai appris en cours de *programmation système*, de *système d'exploitation*, d'*architecture des ordinateurs* et *Programmation des Architectures Parallèles*.

4.1 Prérequis et accès

Pour utiliser le mécanisme d'interruption en espace utilisateur, que nous allons abréger en *Uintr* dans la suite de ce document, il est nécessaire d'avoir accès à un CPU Intel® Sapphire Rapids. Du fait qu'ils étaient sortis récemment, ils étaient assez difficiles d'accès. *Atos* nous a donné accès, non sans difficulté, à une machine qui possède 2 CPU Intel® Sapphire Rapids, lesquels sont des Intel® Xeon® Platinum 8470. Les difficultés étaient liées à la disponibilité d'une machine, à trouver un endroit où l'installer et à s'assurer qu'elle soit connectée à un réseau accessible depuis l'Inria. Nous avons donc obtenu un accès VPN qui utiliser l'ancien système VPN, car le nouveau ne fonctionnait pas. Nous avons eu accès à la machine environ deux mois et demi après le début du stage. La machine était déjà configurée avec le système d'exploitation Red Hat Enterprise Linux (RHEL) version 9.1 avec un noyau Linux version *5.14.0-162.6.1*. Elle possède également au moins une carte *BXI* v2 que nous n'avons pas utilisées pendant le stage.

Il faut également avoir une version patchée du noyau Linux prenant en charge le nouveau mécanisme. Cette version patchée n'est pas encore disponible dans la branche principale du noyau. Cependant, elle est accessible sur le GitHub d'Intel® [14]. Elle est basée sur la version *6.0.0*. Nous avons donc téléchargé cette version patchée, puis nous l'avons compilée et installée sur la machine, toutes les commandes nécessaires pour cela sont disponibles

en annexe. Lors de la compilation, il est nécessaire d'activer le support des *Uintr* (Voir la figure 12 en annexe). Il est également possible d'activer le support permettant à un thread bloqué, c'est-à-dire non ordonnancé ou dans un appel système interruptible, de recevoir une *Uintr*.

Le mécanisme utilise de nouvelles instructions, donc une version récente du compilateur *GCC* est nécessaire pour compiler les programmes utilisateurs qui utiliseront les *Uintr*. Il faut donc la version *11.3.0* ou plus récente de *GCC*, et sur RHEL il faut la version *12.1.1* ou supérieure. Le support n'est pas encore disponible dans d'autres compilateurs comme *LLVM-Clang* ou *ICC*. Pour compiler un programme utilisateur, il faut spécifier le flag de compilation `-muintr` pour les fichiers qui définissent un handler d'interruption ou qui utilisent les nouvelles instructions.

4.2 Fonctionnement des interruptions en espace utilisateur

Dans le cadre de ce stage, nous avons étudié en détail le fonctionnement des interruptions ordinaires ainsi que le fonctionnement des interruptions en espace utilisateur. Les *Uintr* n'était pas connus des équipes *Inria* et *Atos*.

4.2.1 Les interruptions

Pour commencer, nous allons voire le fonctionnement des interruptions matérielles. Nous nous concentrerons sur l'envoi d'interruptions entre deux processus fixés sur deux unités de calcul. Pour ces interruptions les CPU disposent d'une unité dédiée à leur traitement, l'*APIC* pour "Advanced Programmable Interrupt Controller". Cette *APIC* permet au système d'enregistrer un handler pour chaque interruption. Le noyau définit un tableau appelé *IDT* pour "Interrupt Descriptor Table" qui contient 256 entrées, correspondant à 256 interruptions possibles. Les indices de l'*IDT* sont des valeurs comprises entre 0 et 255, que l'on appelle également **vecteur**s d'interruption. Parmi eux, les vecteurs entre 0 et 31 sont réservés pour les exceptions et les interruptions système, les vecteurs entre 32 et 127 sont dédiés aux interruptions liées aux périphériques, le vecteur 128 est réservé pour les appels système, et les vecteurs entre 129 et 255 sont destinés à des utilisations diverses.

Il est important de savoir que chaque unité de calcul (processeur logique) possède un *APIC ID* physique. À titre informatif, le *core ID* est un sous-ensemble de l'*APIC ID*.

Pour déclencher une interruption, il y a quatre possibilités :

1. Une exception déclenchée par un processeur (e.g. une division par zéro, un défaut de segmentation...).
2. Une instruction comme `INT80 numSysCall` pour déclencher un appel système, ou bien `INT3` pour définir un point d'arrêt, ou encore `INTO`, `BOUND` et `INT n`.

3. Des broches du CPU dédiées à la réception d'interruptions lancées à partir d'un périphérique.
4. Demander à l'*APIC* elle-même grâce à un registre *ICR* pour "Interrupt Command Register". Il existe un *ICR* par vecteur, donc il faut écrire l'*APIC ID* du destinataire dans le *ICR* du vecteur que l'on veut déclencher. Seul le CPU et le noyau peuvent modifier les *ICR*.

On voit bien que les *IRQ* fonctionnent au niveau du noyau et du CPU.

Nous allons voir un exemple d'envoi d'*IRQ* entre deux unités de calcul en cours d'exécution. Tout d'abord, l'initialisation des *IRQ* se fait au démarrage du système et consiste principalement à définir les handlers noyau dans l'*IDT*. Il faut aussi définir quel vecteur nous voulons utiliser, le code du handler noyau qui sera invoqué, comment l'utilisateur va contacter le système et comment faire l'identification du récepteur. Pour cela, on peut appliquer un patch au noyau ou faire un module noyau.

Dans notre exemple, nous allons supposer que nous avons déjà patché le noyau en ajoutant des appels système et que nous avons choisi un vecteur.

Nous allons maintenant voir les étapes de l'envoi d'une *IRQ*, illustrées sur la figure 1 :

(1) Le récepteur fait un appel système pour indiquer au noyau comment il veut être averti d'une interruption (e.g. un descripteur de fichiers qu'il va lire, un appel système bloquant, une zone mémoire où lire, un handler utilisateur, etc.).

(2) L'émetteur peut donc avertir le noyau qu'il faut envoyer une interruption. Pour cela, il peut utiliser un appel système ou écrire dans un descripteur de fichiers, par exemple.

(3) Le noyau détermine l'unité de calcul où se trouve le récepteur. Pour cela, il peut utiliser par exemple un *PID* pour "Processus ID" donné par l'émetteur ou autre. Ainsi, il peut déterminer l'*APIC ID* de l'unité de calcul à interrompre.

(4) Le noyau écrit donc l'*APIC ID* dans le *ICR* d'un vecteur déterminé à l'avance. L'émetteur reprend alors la main après un autre changement de contexte.

(5) L'*APIC* va donc interrompre le récepteur qui va alors stopper son exécution et passer dans le noyau. Une fois dans le noyau, le handler va se déclencher et exécuter le code prévu au préalable (écriture dans un descripteur de fichiers, écriture dans une zone mémoire, déclenchement d'un handler utilisateur, etc.).

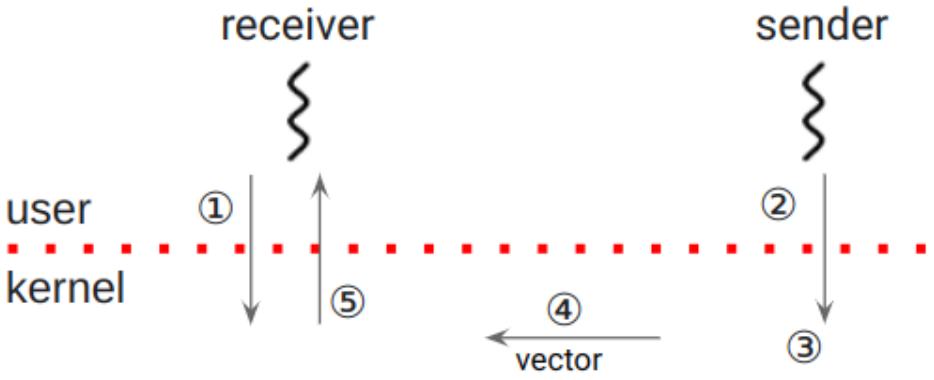


FIGURE 1 – L'envoi d'une interruption ordinaire

Lors du déclenchement du handler noyau, certains registres actuels sont sauvegardés, tels que le pointeur de pile `RSP`, le registre d'états `RFLAGS`, le registre `CS` et le registre de pointeur d'instruction *RIP*. Cette sauvegarde est réalisée en les empilant dans une nouvelle pile. Le vecteur de l'interruption est aussi empilé en tant que code erreur (*errorCode*).

Une fois que le handler noyau a fini de s'exécuter, il doit exécuter l'instruction `iret` qui a pour effet de dépiler les registres sauvegardés et de les restaurer.

Il est possible de masquer les interruptions grâce à deux instructions utilisables seulement par le noyau, qui sont `cli` et `sti`. Ces instructions permettent de modifier le flag *IF* pour "Interrupt Flag" qui se trouve dans le registre d'états de l'unité de calcul, `RFLAGS` (aussi nommé `EFLAGS` sur les architectures 32 bits). La liste des instructions pour les *IRQ* se trouve dans le tableau 1.

Comme nous l'avons vu, ce mécanisme fonctionne totalement dans le noyau du système. Dans notre exemple, il faut au minimum deux changements de contexte (context switch) pour le récepteur et l'émetteur, et peut-être même plus si le récepteur doit déclencher un handler côté utilisateur.

4.2.2 Les uintr

Le mécanisme d'interruption en espace utilisateur utilise cinq nouvelles instructions qui sont listées dans le tableau suivant 1. Les deux premières, `clui` et `stui` qui sont analogues à `cli` et `sti`, permettent le masquage des interruptions. En effet, tout comme les interruptions ordinaires qui ont un flag *IF* pour activer ou désactiver les interruptions, les *Uintr* ont un flag *UIF* pour "User Interrupt Flag". L'instruction suivante, `testui`, permet à l'utilisateur de savoir si les interruptions sont masquées ou non. Cette instruction existe car l'utilisateur n'a pas accès directement au *UIF*, contrai-

rement aux interruptions ordinaires qui ont un accès direct à *IF* puisqu'elles fonctionnent dans le noyau.

L'instruction suivante, `uiret`, fonctionne de manière similaire à celle des interruptions ordinaires (`iret`), mais elle n'utilise pas les mêmes registres et surtout elle est utilisable en espace utilisateur. Enfin, la dernière instruction permet d'envoyer une *Uintr* en utilisant un indice, que nous allons voir dans la section 4.2.4.

Interruption (IRQ)	**Interruption utilisateur (*Uintr*)**
cli (**CL**ear **I**F)	clui (**CL**ear **UI**F)
sti (**S**e**T** **I**F)	stui (**S**e**T** **UI**F)
	testui (Read **UI**F)
iret (**I**nterrupt **RET**urn)	uiret (**U**ser **I**nterrupt **RET**urn)
APIC pins, APIC ICR, `INT n`, `INT3`, `INTO`, `BOUND` et `INT80 n`	sendipi <uipi_index>

Table 1 – Instructions des interruptions et des interruptions en espace utilisateur

Le mécanisme est également accompagné de six nouveaux *registres d'états*, appelés registres *MSR* pour "Model-Specific Registers". Ces registres sont modifiés par le noyau grâce à des appels système que l'utilisateur effectue pour initialiser les *Uintr* et sont utilisés par le CPU. Le tableau 2 décrit ces registres, et nous expliquerons leur utilité par la suite.

Nom du registre	**Description**
IA32_UINTR_STACKADJUST	Utilisé par le récepteur pour définir l'adresse de la pile alternative
IA32_UINTR_HANDLER	Utilisé par le récepteur pour définir l'adresse du handler *Uintr*
IA32_UINTR_MISC	Utilisé par l'émetteur pour définir la taille du *UITT* et par le récepteur pour que l'*APIC* connaisse le vecteur d'interruption ordinaire qu'il dois reconnaître pour déclencher le handler *Uintr* et le dernier bit, pour le flag de masquage *UIF*
IA32_UINTR_PD	Utilisé par le récepteur pour définir l'adresse du *UPID*
IA32_UINTR_RR	Utilisé par l'*APIC* pour *lister* les vecteurs *Uintr* qu'il doit envoyer au récepteur. Correspond aux derniers 64 bits du *UPID*
IA32_UINTR_TT	Utilisé par l'émetteur pour définir l'adresse du *UITT*

Table 2 – Liste des six registres d'états des *Uintr*

4.2.3 Capacités présentes et futures

Le mécanisme d'interruption en espace utilisateur a une interface pour l'utilisateur similaire aux signaux 2.8. Nous allons donc voir les capacités des *Uintr* en les comparant à celles des signaux. Tout d'abord, le fonctionnement des *Uintr* se fait au niveau des threads, tandis que celui des signaux se fait au niveau du processus. Il est possible d'avoir un fonctionnement qui se rapproche d'un fonctionnement par threads en utilisant plusieurs options.

Avec les *Uintr*, il est possible de définir un handler différent pour chaque thread d'un processus, alors que pour les signaux, on peut définir un seul handler pour tous les threads d'un processus. Par contre, avec les signaux, il est possible de définir un handler différent pour chaque signal, ce qui n'est pas possible avec les *Uintr*. Pour les *Uintr*, il faut gérer cette différenciation manuellement en appelant la fonction correspondant au vecteur reçu.

Il existe 64 signaux possibles, parmi lesquels les 32 premiers ont une signification particulière. En revanche, pour les *Uintr*, il existe également 64 vecteurs possibles, entre 0 à 63, qui n'ont aucune signification particulière par défaut.

Pour les *Uintr*, le masquage se fait via une instruction, tandis que pour les signaux, il faut effectuer un appel système pour les masquer.

L'envoi d'un signal se fait par le noyau suite à une exception, une décision du noyau ou la demande d'un processus grâce à un appel système (`kill(signum)` ou `tgkill(signum)`). Pour les *Uintr*, l'envoi peut se faire depuis un autre processus ou depuis le noyau, et dans le futur, il pourra également se faire depuis un périphérique.

Avec les signaux, le handler peut être déclenché que le processus cible soit endormi ou non, tandis que pour les *Uintr*, c'est différent. Il faut que le thread soit en espace utilisateur pour recevoir une interruption. Sinon, l'interruption sera reçue quand le thread revient en espace utilisateur. On a vu précédemment, dans les prérequis, section 4.1, qu'une fonctionnalité existe lors de la compilation du noyau pour autoriser l'interruption d'un thread bloqué. Si la fonctionnalité est activée, il est donc possible d'interrompre un thread qui n'est pas ordonnancé ou qui est en train de faire un appel système interruptible, et ainsi le passer en espace utilisateur pour qu'il puisse recevoir l'interruption. Pour utiliser cette fonctionnalité, l'utilisateur doit renseigner un flag au moment de définir le handler.

Il existe donc trois flags :

— `UINTR_HANDLER_FLAG_WAITING_ANY` qui active la fonctionnalité,

— `UINTR_HANDLER_FLAG_WAITING_RECEIVER` et `UINTR_HANDLER_FLAG_WAITING_SENDER` qui s'ajoutent au précédent pour préciser si c'est l'émetteur ou le récepteur qui prendra en charge le surcoût du passage par le noyau.

Pour les signaux, le déclenchement du handler est géré par le noyau qui sauvegarde l'état du processus, définit une pile alternative si besoin, change

de contexte et appelle le handler utilisateur. Pour les *Uintr*, le déclenchement du handler est effectué par le CPU, il est donc très sommaire :

— changer la pile si une pile alternative est disponible dans le registre `IA32_UINTR_STACKADJUST`,

— empiler l'ancien pointeur de pile, le registre d'états de l'unité de calcul, le registre de pointeur d'instruction `RIP` et le vecteur *Uintr*

— aller à l'adresse du handler utilisateur disponible dans le registre `IA32_UINTR_HANDLER`.

C'est donc à l'utilisateur qu'appartient la responsabilité de sauvegarder les registres généraux, les registres vectoriels (SIMD)... et de les restaurer à la sortie du handler. Le compilateur permet déjà de sauvegarder les registres généraux avec le flag `general-regs-only`. Cependant, pour les registres vectoriels, il faut les sauvegarder avant de les utiliser. Il faut faire attention avec les opérations sur les chaînes de caractères de la *libc* car les fonctions `memcpy`, `memmove`, `memset` et `memcmp` utilisent des registres vectoriels par défaut. Le compilateur fournit le flag `-minline-all-stringops` qui permet de les remplacer par des versions inline de ces opérations afin de ne plus utiliser de registres vectoriels.

Une fois que le handler a fini son exécution, il faut s'occuper du retour. Pour les signaux, c'est le noyau qui s'en occupe, tandis que pour les *Uintr*, il incombe à l'utilisateur d'utiliser l'instruction `uiret`. Donc, l'unité de calcul va dépiler le vecteur et les registres qui suivent pour les restaurer, ce qui permettra au code de continuer là où il en était.

Que ce soit dans un handler de signal ou dans un handler d'interruption utilisateur, on a la même contrainte : on ne peut pas faire d'attente, donc on peut seulement appeler des fonctions et des appels système dits "async safe".

4.2.4 Exemple de fonctionnement

Nous allons maintenant voir un exemple d'initialisation des *Uintr* illustré sur la figure 2 :

(1) Le récepteur enregistre auprès du noyau un handler d'interruption qu'il a défini à l'aide de l'appel système `uintr_register_handler(ui_handler)`. Le noyau va enregistrer ce handler dans le registre `IA32_UINTR_HANDLER` et va initialiser une zone mémoire nommée *UPID* pour "User Posted Interrupt Descriptor". Ce *UPID* permet au mécanisme des *Uintr* de manipuler des informations propres à ce thread, essentielles pour l'envoi d'*Uintr*. L'adresse du *UPID* est enregistré dans le registre `IA32_UINTR_PD`.

(2) Le récepteur donne au noyau un vecteur *Uintr* entre 0 et 63 qu'il souhaite recevoir (8 dans la figure), à l'aide de l'appel système

`uvec_fd <- uintr_vector_fd(8)`. Le noyau lui renvoie un descripteur de fichier qui pointe vers une structure contenant à la fois le vecteur et l'adresse du *UPID*.

(3) Le récepteur envoie ce descripteur de fichier aux émetteurs potentiels, un seul dans notre cas. Nous verrons comment partager ce descripteur de fichier dans la section 4.2.5.

(4) L'émetteur va s'enregistrer auprès du noyau grâce au descripteur de fichier en appelant l'appel système

`uipi_index <- uintr_register_sender(uvec_fd)`. Pour ce faire, le noyau possède un tableau *UITT* pour "User Interrupt Target Table" qui fait une taille de 256 entrées par défaut. L'adresse de ce tableau doit être enregistrée dans le registre `IA32_UINTR_TT`, et sa taille doit être placée dans les quatre premiers octets du registre `IA32_UINTR_MISC`. Ainsi, la taille du *UITT* peut varier en fonction des besoins. Chaque entrée de ce tableau *UITT* correspond à une zone mémoire nommée *UITTE* pour "User Interrupt Target Table Entry". Le noyau va rechercher une entrée libre dans le *UITT* et remplir l'*UITTE* avec le vecteur *Uintr* et l'adresse du *UPID* obtenus grâce au descripteur de fichier. Il va également mettre un vecteur d'interruption ordinaire dans le cinquième octet du registre `IA32_UINTR_MISC`, ce vecteur "ordinaire" étant dédié aux *Uintr* et a pour valeur 236. Pour finir il retourne l'indice de l'entré à l'émetteur.

(5) L'utilisateur peut maintenant envoyer autant d'interruptions que nécessaire, totalement depuis l'espace utilisateur.

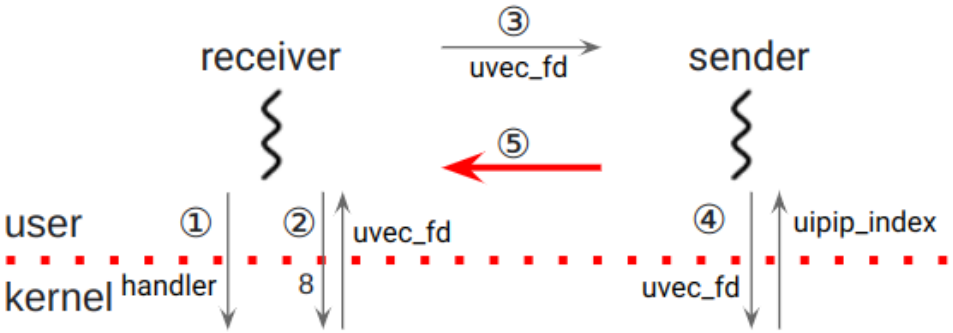


FIGURE 2 – Phase d'initialisation des *Uintr*

Maintenant que nous avons vu l'initialisation, nous allons pouvoir voir comment l'envoi d'interruption est réalisé, comme illustré dans la figure 3 :

(1) L'émetteur utilise l'instruction `senduipi` avec l'indice récupéré lors de l'initialisation. L'unité de calcul peut accéder au tableau *UITT* qui se trouve dans le registre `IA32_UINTR_TT`. Elle peut vérifie que

l'indice se trouve bien dans le tableau, en utilisant la taille qui se trouve dans le registre `IA32_UINTR_MISC`. De plus, il s'assure que l'entrée dans le *UITT* est valide. À partir de l'indice donné à l'instruction, l'émetteur peut récupérer l'adresse de l'*UPID* et le vecteur *Uintr* à envoyer au récepteur. Dans la zone mémoire de l'*UPID*, il écrit le vecteur *Uintr* à envoyer et détermine s'il faut également envoyer une interruption ordinaire. En effet, les interruptions en espace utilisateur utilisent les interruptions ordinaires, et l'envoi de celles-ci dépend de si une interruption ordinaire n'a pas déjà été envoyée. Dans notre cas, nous considérons que c'est le premier envoi.

2. Donc, l'émetteur va récupérer le vecteur d'interruption ordinaire dans le registre `IA32_UINTR_MISC` et sélectionner le registre *ICR* correspondant au vecteur "ordinaire". Ensuite, il va récupérer l'*APIC ID* dans l'*UPID* et l'écrire dans le registre *ICR*. Cela permettra d'envoyer l'interruption ordinaire à l'unité de calcul correspondante.
3. L'*APIC* va recevoir l'interruption, puis elle va comparer le vecteur d'interruption ordinaire reçu avec celui qui se trouve dans l'*UPID*, qu'il connaît grâce au registre `IA32_UINTR_PD`. Si les vecteurs sont identiques, l'*APIC* va pouvoir déclencher le système de traitement des *Uintr*, sinon, elle va utiliser le mécanisme habituel des *IRQ*.
4. Le système de traitement des *Uintr* va donc indiquer dans l'*UPID* que l'interruption ordinaire a déjà été envoyée, puis va commencer le déclenchement des handlers pour tous les vecteurs *Uintr* reçus, en partant du plus grand, 63, jusqu'au plus petit, 0. `IA32_UINTR_RR` est utilisé à cette étape. Dans notre exemple, nous en avons un seul qui est 8. L'*APIC* va donc ordonner à l'unité de calcul de changer de pile si une pile alternative est disponible dans le registre `IA32_UINTR_STACKADJUST`, empiler les registres nécessaires, empiler le vecteur *Uintr* (donc 8 dans l'exemple) et aller à l'adresse du handler qui est disponible dans le registre `IA32_UINTR_HANDLER`.

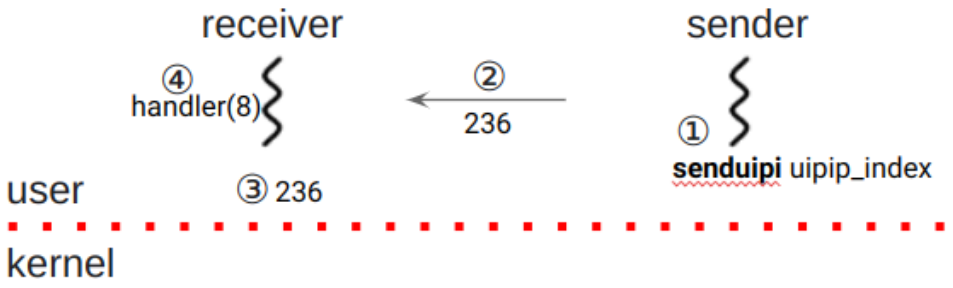


FIGURE 3 – L'envoi d'une *Uintr*

Le mécanisme habituel des *IRQ* peut être utilisé dans le cas où l'*Uintr*

est déclenchée depuis le noyau et dans le cas où le thread destinataire n'est pas ordonnancé ou est dans un appel système interruptible.

4.2.5 Partage du descripteur de fichier

Pour partager le descripteur de fichier, il y a plusieurs façons possibles.

Dans nos tests du mécanisme entre processus, nous avons utilisé l'héritage des descripteurs de fichier. Dans nos tests du mécanisme entre threads, nous avons utilisé un appel système qui permet d'enregistrer tous les threads du processus comme émetteurs, sans utiliser le descripteur de fichier, mais il est possible d'utiliser une variable globale pour y mettre le descripteur de fichier.

Dans le cas où les deux processus sont indépendants, on peut utiliser l'appel système `pidfd_getfd` qui permet de dupliquer le descripteur de fichier d'un autre processus s'il a le même propriétaire et si l'on connaît son *PID* et le numéro du descripteur de fichier. Nous avons donc testé en partageant le numéro de descripteur de fichier avec un pipe. On aurait aussi pu passer par un fichier ou par des sockets.

Par la suite, dans *NewMadeleine*, nous avons utilisé le système d'URL à la connexion qui permet l'envoi de paramètres. Donc nous avons ajouté un paramètre avec la valeur du descripteur de fichier.

4.3 Tests du mécanisme

Nous avons donc effectué des tests du mécanisme avec des exemples minimaux de communication entre processus. Nous examinerons les plus pertinents dans cette section.

Tout d'abord, des tests pour mesurer le temps d'envoi d'interruption en espace utilisateur entre deux threads et deux processus.

Pour le test d'envoi entre deux threads, il commence par créer un nouveau thread qui va commencer par se "bind" à une unité de calcul. Nous le verrons en détail dans la section 4.5.

Il va ensuite enregistrer un handler d'interruption, démasquer les interruptions, enregistrer tous les threads du processus comme émetteurs avec l'appel système `uipi_index <- uintr_register_self(vector)`,
(uipi_index est global au processus) et attendre grâce à une boucle.

Pendant ce temps-là, le thread principal attend une seconde. Ce temps est arbitraire et laisse le temps au thread d'enregistrer un handler d'interruption. Après son attente, il va se "bind" à une unité de calcul puis envoyer une interruption. Pour cela, nous commençons par enregistrer le temps processeur actuel, puis nous utilisons l'instruction `senduipi` avec la variable globale `uipi_index`. Une fois que le handler d'interruption se déclenche, on enregistre le temps actuel du processeur pour ensuite calculer la différence. Ce test est capable de faire cette mesure plusieurs fois. Il finit par imprimer

les différences de temps dans la console et par désallouer et de-enregistrer les *Uintr* et les autres structures.

Pour le teste d'envoi entre deux processus, il commence par créer un pipe pour l'envoi du descripteur de fichier et une zone de mémoire partagée pour stocker les mesures de temps. Puis le processus se fork en deux, le premier devient l'émetteur et le second le récepteur. Comme pour la version avec threads, ils se "bind" à une unité de calcul. Le récepteur enregistre un handler d'interruption et récupère un descripteur de fichier avec l'appel système `uvec_fd <- uintr_vector_fd(vector)` qu'il envoie dans le pipe avant d'attendre grâce à une boucle. L'émetteur reçoit le numéro du descripteur de fichier, il connaît déjà le *PID* du processus grâce au fork, il peut donc utiliser `pidfd_getfd` pour dupliquer le descripteur de fichier. Avec ce descripteur, il s'enregistre en tant qu'émetteur d'*Uintr*. La mesure du temps et l'envoi d'interruption fonctionnent de la même manière que pour la version avec threads. On peut aussi effectuer la mesure plusieurs fois et on imprime et termine proprement.

On verra les résultats de ces mesures dans la section 4.6.

Un test où le thread s'auto-interrompt tout simplement en faisant un `uipi_index <- uintr_register_self(vector)`,
puis un `senduipi uipi_index`, et on fait la mesure de la même façon que pour les autres tests.

Pour les tests avec la pile alternative, nous avons un test très simple qui est basé sur celui qui s'auto-interrompt, et nous modifions les tests d'envoi entre deux processus ou threads pour mesurer l'impact les performances. Il est bien sûr possible d'activer ou non l'utilisation de la pile alternative. Pour définir cette nouvelle pile, on le fait juste après avoir enregistré le handler d'interruption et avant le démasquage.

Un test de démasquage des *Uintr* dans un handler d'interruption. Il nous permet de voir que c'est tous à fait possible et cela pose des problématiques similaires à celles des signaux.

Un test consiste à envoyer plusieurs interruptions d'affilée. Il nous permet de voir qu'il y a une différence de comportement par rapport aux signaux. Quand on reçoit une interruption et que le handler d'interruption est déclenché, le comportement est le même, c'est-à-dire que les interruptions vont s'écraser et le handler d'interruption se déclenchera à nouveau une fois. La différence réside dans le fait que si l'on effectue plusieurs interruptions avant que le handler d'interruption ne se déclenche, les interruptions s'écrasent également à ce moment-là. Ainsi, le handler d'interruption ne se déclenchera qu'une seule fois lorsque l'on effectue deux interruptions successives. Avec les signaux, en revanche, le handler de signal se déclencherait deux fois pour deux émissions de signal successives car le handler est déjà déclenché dès le premier signal.

Grâce à un test, nous avons constaté qu'actuellement on peut enregistrer plusieurs fois le même descripteur de fichier, ce qui mène à des doublons

dans le tableau *UITT* avec plusieurs *UITTE* pour le même couple vecteur / *UPID*. Cette limitation de l'implémentation noyau est documentée dans le patch et accompagnée d'un commentaire *TODO*.

4.4 Correction d'un bug dans le patch noyau

En manipulant le mécanisme, nous sommes tombés sur un bug qui concerne l'utilisation d'une pile alternative. Comme pour les signaux, il est possible de définir une pile alternative. Cette nouvelle pile est utilisée au moment où le handler est déclenché. L'interface pour définir la pile alternative est la même pour les signaux et les *Uintr*. Dans les manuels (des signaux et des *Uintr*), il est bien indiqué que l'utilisateur doit lui-même allouer une zone mémoire consacrée à la nouvelle pile, et il a également la responsabilité de libérer la mémoire une fois le handler dé-enregistré. Il est bien indiqué que l'utilisateur doit donner l'adresse de début (*base address*) de la zone mémoire, ainsi que la taille, à l'appel système qui définit la pile alternative. Pour les *Uintr*, l'appel système est `uintr_alt_stack(spAddress, size)`. Il faut noter qu'une pile empile les éléments vers le haut, c'est-à-dire que l'adresse du pointeur de pile décroît à l'ajout d'un élément. Donc, pour utiliser la zone mémoire dédiée à la pile, il faut partir de la fin. Du côté du noyau, le mécanisme des signaux garde en mémoire l'adresse et la taille de la pile pour le moment où le handler doit être déclenché. Le calcul du pointeur de pile de la nouvelle pile se fait donc juste avant de déclencher le handler. Pour les *Uintr*, le noyau se contente juste d'enregistrer l'adresse dans le registre dédié `IA32_UINTR_STACKADJUST`, et le processeur utilise l'adresse telle quelle comme pointeur de pile. On est donc confronté à un bug de débordement mémoire car on part du début de la zone mémoire. Il y a bien un test dans le patch du noyau qui vérifie ce cas, mais mal. Nous sommes tombés sur ce bug au moment d'utiliser les *Uintr* dans *NewMadeleine*, qui manipule bien plus la mémoire qu'un simple test. Nous nous sommes retrouvés avec des problèmes de corruption mémoire, des défauts de segmentation et des "double free detected". Nous avons donc corrigé le test du patch du noyau ainsi que l'appel système. Pour ce faire, on ajoute la taille de la zone mémoire à l'adresse avant de l'écrire dans le registre `IA32_UINTR_STACKADJUST`. Nous avons donc effectué une *Pull request* [26] sur le dépôt GitHub du patch, et à l'heure où nous écrivons ce document, il n'a toujours pas été appliqué par Intel®.

4.5 Mesure de la latence

Pour mesurer la latence entre le moment où on envoie une interruption et où l'interruption est reçue par le handler, on fait deux mesures de temps. Pour faire les mesures de temps, on utilise `clock_gettime` qui utilise l'instruction `rdtsc` et retourne le temps actuel du processeur. On fait une

première mesure juste avant d'envoyer une interruption et une seconde au tout début du handler. Pour obtenir la latence, on a juste à soustraire la première mesure à la seconde.

Nous avons regardé le code assembleur pour nous assurer de la mesure. Pour l'envoi, que l'on peut voir en listing 1, on peut constater qu'entre la mesure et l'instruction d'envoi, il y a seulement une lecture mémoire qui n'est pas très coûteuse. Du côté de la réception, que l'on peut voir en listing 2, on voit la sauvegarde des registres généraux au début du handler. Cette sauvegarde ajoute un petit surcoût, mais il est obligatoire. Le compilateur *GCC* nous force à mettre le flag `general-regs-only` pour compiler un handler d'interruption et donc sauvegarder les registres généraux. On peut voir la déclaration d'un handler avec la mesure du temps en listing 3.

```
    call    clock_gettime@PLT
    movq    uipi_index(%rip), %rax
    senduipi    %rax
```

Listing 1 – Code assembleur de l'envoi d'*Uintr*

```
handler:
.LFB203:
    .cfi_startproc
    .cfi_def_cfa_offset 16
    pushq   %r11
    .cfi_def_cfa_offset 24
    .cfi_offset 11, -24
    pushq   %r10
    .cfi_def_cfa_offset 32
    .cfi_offset 10, -32
    pushq   %r9
    .cfi_def_cfa_offset 40
    .cfi_offset 9, -40
    pushq   %r8
    .cfi_def_cfa_offset 48
    .cfi_offset 8, -48
    pushq   %rdi
    .cfi_def_cfa_offset 56
    .cfi_offset 5, -56
    movl    $1, %edi
    pushq   %rsi
    .cfi_def_cfa_offset 64
    .cfi_offset 4, -64
    leaq    ts2(%rip), %rsi
    pushq   %rcx
    .cfi_def_cfa_offset 72
    .cfi_offset 2, -72
    pushq   %rdx
    .cfi_def_cfa_offset 80
    .cfi_offset 1, -80
    pushq   %rax
    .cfi_def_cfa_offset 88
    .cfi_offset 0, -88
```

```
34      subq    $8, %rsp
35      .cfi_def_cfa_offset 96
36      cld
37      call    clock_gettime@PLT
```

Listing 2 – Code assembleur de l'handler *Uintr*

```
1   __attribute__((target("general-regs-only")))
2   __attribute__((interrupt))
3   void handler(struct __uintr_frame* ui_frame, u64 vector) {
4     clock_gettime(CLOCK_MONOTONIC, &ts2);
```

Listing 3 – Déclaration du handler *Uintr*

Nous faisons la mesure de la même façon pour les signaux.

Pour ne pas perturber la mesure, il faut "bind" les threads à une unité de calcul. "bind" consiste à demander au noyau de toujours ordonnancer le thread sur la même unité de calcul. Pour ce faire nous utilisons la bibliothèque *hwloc* [12]. Il est donc important de "bind" les threads pendant les mesures car dans le cas contraire, le noyau va changer le thread d'unité de calcul, ce qui va amener à un surcoût non négligeable. En effet, le changement d'unité est un peu coûteux et invalide certains caches. "bind" les threads nous permet aussi de contrôler le placement de ceux-ci, c'est-à-dire si on met le thread émetteur et le thread récepteur sur deux unités de calcul proches ou distantes. Comme nous le verrons dans la section 4.6, le placement a un impact sur la latence des *Uintr*.

Il est aussi important de fixer la fréquence de tous les core du CPU pour avoir des mesures reproductibles.

4.6 Performances

Dans cette section, nous allons voir des mesures de la latence des *Uintr* dans différents contextes. La fréquence des unités de calcul est fixée à 2 GHz lors des mesures. Les mesures faites avec le turbo boost activé montent à 3.8 GHz. Les mesures de latence sont en nanosecondes. Il est intéressant de noter que sur une unité de calcul cadencée à une fréquence de 2 GHz, elle exécute environ deux instructions par nanoseconde. Certaines mesures, que ce soit avec les *Uintr* ou les signaux, sont très élevées mais sont très peu nombreuses. Elles sont certainement liées au système. Dans les graphiques, nous avons donc coupé les valeurs qui dépassent les 8000 nanosecondes. Nous avons défini trois placements à partir de la topologie de la machine fournie par *Atos*. Vous pouvez trouver la topologie de la machine sur la figure 13.

Les trois placements sont les suivants :

— Le placement "proche" consiste à placer les threads sur deux unités de calcul proches mais pas dans le même core.

— Le placement "éloigné" consiste à placer les threads sur deux unités de calcul qui sont dans le même CPU et qui sont éloignées.

— Le placement "très éloigné" consiste à placer les threads sur deux unités de calcul qui se trouvent sur deux CPU différents.

Les mesures que nous allons présenter ont été faites sans que la pile alternative ne soit activée. Nous avons bien fait des mesures avec la pile alternative des *Uintr* activée et nous n'avons vu aucune différence, car l'utilisation de cette pile amène seulement à une copie de la mémoire d'un registre à un autre si le registre contient une adresse, ce qui est très peu coûteux.

Dans nos graphiques, nous appelons une mesure une itération. Nous faisons donc un million d'itérations et nous n'affichons pas la première, car cette itération est énormément perturbée notamment par le coût de chargement des caches.

Dans les graphiques, nous avons représenté la mesure de la latence par des points bleus pour les signaux et par des points rouges pour les *Uintr*. Nous ne cherchons pas à expliquer les mesures des signaux, elles sont juste là pour comparer les *Uintr* avec un mécanisme qui passe par le noyau.

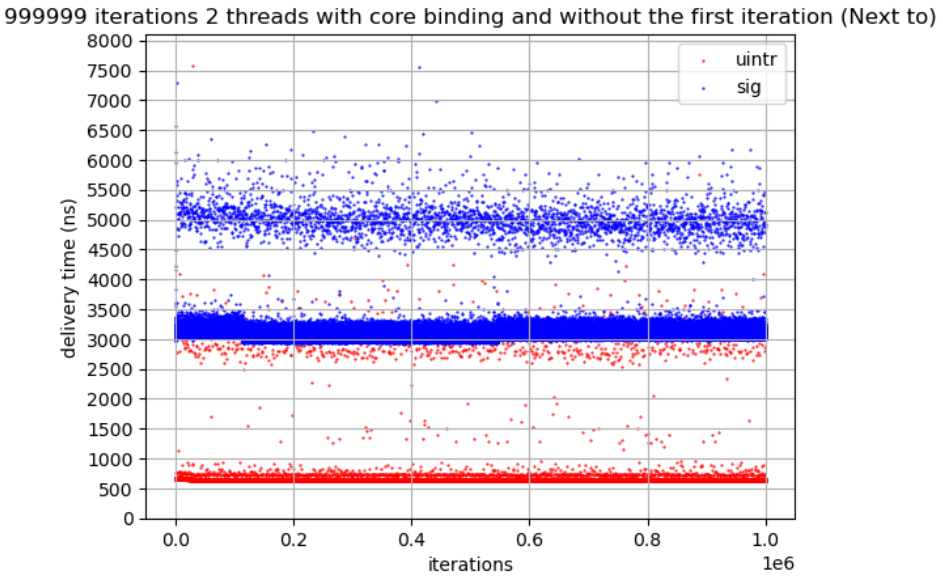
Sur le graphique 4a, les itérations sont représentées sur l'axe des abscisses et la latence sur l'axe des ordonnées. Bien sûr, plus la latence est basse, mieux c'est. Nous observons que le mécanisme a une latence d'environ 642 nanosecondes, ce qui est environ 4.75 fois plus rapide que les signaux.

On voit qu'il y a deux groupes de mesures :

— Un autour des 642 nanosecondes qui comporte la majorité des mesures. On le voit bien dans le tableau 4b qui se trouve juste en dessous du graphique. En effet, entre la mesure minimum et la mesure à 95%, il y a une différence de 19 nanosecondes. On peut voir cette distribution aussi dans l'histogramme sur la figure 5. Cet histogramme porte sur des plages de 300 nanosecondes. On voit bien pour les *Uintr*, en orange, que la majorité se trouve sur la bande entre 300 et 600 nanosecondes.

— Un autre autour de 2700 nanosecondes qui correspond certainement au moment où l'interruption n'a pas pu être reçue car le thread était dans le noyau.

Pour un mécanisme qui fonctionne au niveau des instructions, on pourrait s'attendre à une latence moins grande, mais le mécanisme est bien plus rapide que le fait de passer par le système.



(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	3065	160	2920	3001	3054	3157	68532
uintr	643	90	629	638	642	648	65973

(b)

FIGURE 4 – Mesures de latence entre deux threads avec un placement proche

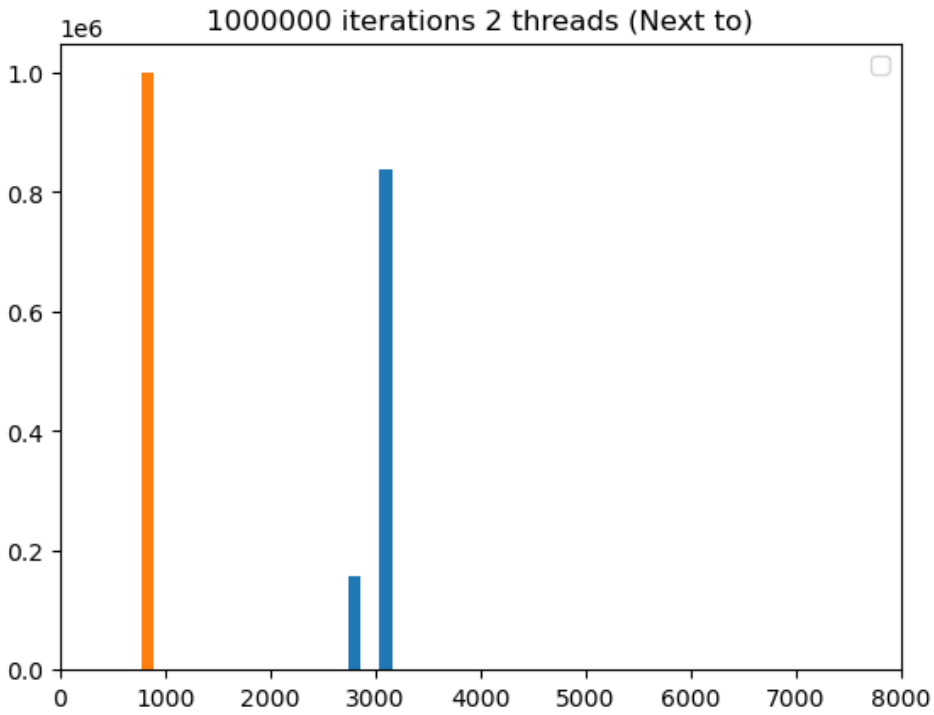
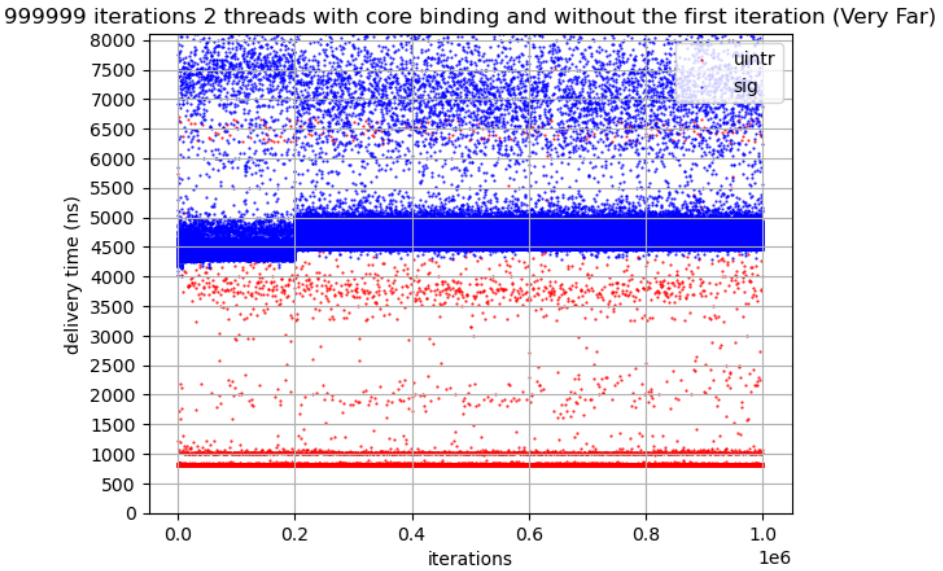


FIGURE 5 – Histogramme de distribution des mesures de la latence entre deux threads avec un placement proche

Nous avons fait les mêmes mesures entre deux processus et les valeurs sont très similaires, peu importe le placement, car le mécanisme fonctionne au niveau des threads. On peut retrouver ces mesures en annexe sur la figure 15.

En comparant les mesures de latences entre le placement proche et éloigné on voit une petite différence de 10 nanosecondes de plus, on peut voir ça sur la figure 16.

Quand on compare entre le placement proche et très éloigné on voit une grande différence qui est due notamment au fait de passer d'un noeud mémoire NUMA à un autre. On a donc une différence d'environ 172 nanosecondes de plus. Le graphique nous montre également une plus grande dispersion de la latence avec la majorité qui est toujours en-dessous de 1000 nanosecondes. On retrouve ces mesures sur la figure 6 juste en-dessous.



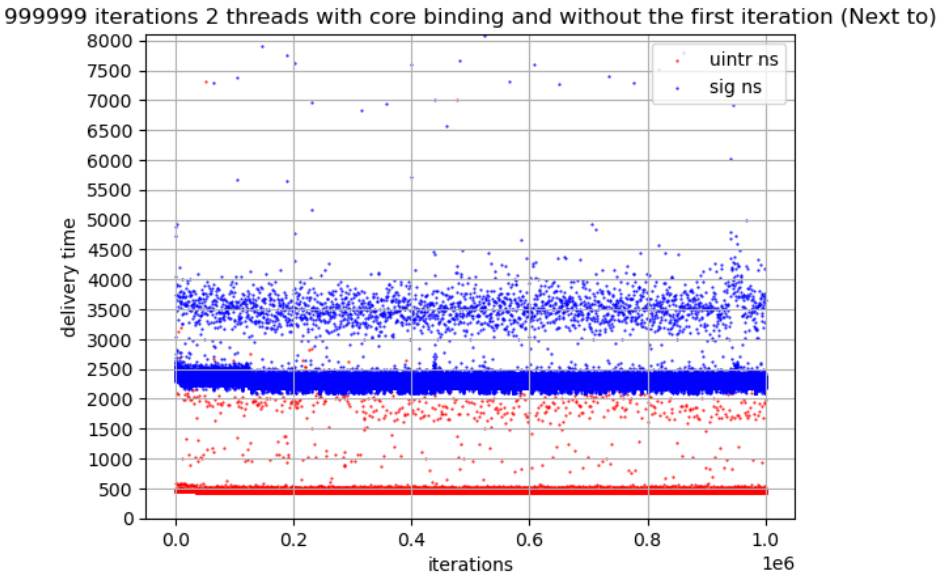
(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	4601	403	4009	4396	4561	4796	58200
uintr	818	136	801	810	814	820	58072

(b)

FIGURE 6 – Mesures de latence entre deux threads avec un placement très éloigné

Quand on augmente la fréquence des unités de calcul la latence diminue. Pour se faire on active le turbo boost du CPU. On le voit sur les mesures de la figure 7 pour le placement proche mais c'est également le cas pour le placement éloigné et le très éloigné en figure 17 et figure 18.



(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	2262	294	2081	2216	2251	2357	283171
uintr	442	313	426	437	440	450	311486

(b)

FIGURE 7 – Mesures de latence entre deux threads avec un placement proche et le turbo boost activé

Nous n'avons fait aucune mesure de latence pour les threads bloqués et interruptibles.

5 Intégration dans NewMadeleine

Pour cette partie, j'ai utilisé ce que j'ai appris en cours :

5 Intégration dans NewMadeleine 5.1 Présentation des détails de New-Madeleine

- De *Programmation des Architectures Parallèles* (PAP) et de *Langages du Parallélisme* pour la compréhension du fonctionnement interne des bibliothèques *MPI* et pour aborder différentes problématiques liées à la programmation parallèle.
- De *Projet de Programmation* (PdP) et de *Lecture d'article et documentation scientifique* pour effectuer les recherches bibliographiques.
- D'*Analyse des données* pour traiter les courbes.
- D'*algo des structure de données* pour m'aider à comprendre les file *lock-free* et *wait-free*.

5.1 Présentation détaillée de NewMadeleine

La bibliothèque de communications *NewMadeleine* propose plusieurs interfaces disponibles pour l'utilisateur. Parmi les interfaces les plus importantes, on trouve *sendrecv* pour envoyer et recevoir des données, *coll* pour les opérations collectives, *MPI* qui implémente le standard *MPI* en utilisant les interfaces *sendrecv* et *coll*, et *rpc* qui permet le support des *RPC*.

Pour ce stage, nous avons principalement utilisé l'interface *sendrecv*.

Cette interface permet à l'utilisateur de manipuler des requêtes de communication en soumettant des tâches au coeur de la bibliothèque *nm_core*.

Ces tâches, nommé *core_task*, peuvent être de différents types, tels que la soumission d'une requête, la complétion des paquets, l'exécution de handlers, etc. Les tâches du coeur fonctionnent dans des sections critiques en utilisant un verrou nommé *core_lock*.

La bibliothèque s'occupe automatiquement de la génération des requêtes et des paquets, du découpage des données en tronçons et de l'application des stratégies d'optimisation.

Le coeur de la bibliothèque s'occupe de l'exécution des tâches et de la progression des communications grâce à la fonction *nm_schedule*. Cette fonction peut être déclenchée à partir d'une interface (e.g. la fonction *MPI_Test*).

Pour effectuer les communications, il est possible d'utiliser différents réseaux ou autres technologies qui utilisent une même interface, appelés *drivers*. Au cours de ce stage, nous nous sommes concentrés sur les parties qui s'occupent de la progression des communications et des drivers. Nous ne rentrerons donc pas dans les détails concernant les différentes interfaces, les stratégies, *Pioman*, etc.

Dans les parties que nous avons manipulées lors du stage, les paquets sont représentés par des "paquet wrappers", que nous abrégerons par "*pw*" par la suite. Les *pw* possèdent toutes les informations nécessaires pour l'envoi ou la réception de tronçons de données et pour que la tâche de complétion de la requête puisse remonter les données à l'utilisateur.

La progression se fait donc grâce à la fonction *nm_schedule*, qui effectue les étapes suivantes :

1. Commence par exécuter les tâches en utilisant le verrou *core_lock*,
2. Puis elle fait progresser les *pw* de réception,
3. Puis elle fait progresser les *pw* d'envois,
4. Elle déclenche le prefetch si nécessaire (nous ne rentrerons pas dans le détail pour le prefetch),
5. Puis elle exécute de nouveau les tâches dans le cas où de nouvelles tâches ont été soumises par la progression des *pw*,
6. Enfin, elle envoie, si nécessaire, des événements aux interfaces supérieures.

Tous les drivers suivent une interface qui permet :

— D'instancier le driver grâce à une fonction *init* qui donne une URL qui faut partagée avec les autres processus pour initier une connexion.

— D'instancier une connexion grâce à une fonction *connect* qui prend en paramètre l'URL d'un autre processus avec lequel on souhaite communiquer. La connexion est bien sûr essentielle pour communiquer et une instance d'un driver peut avoir plusieurs connexions simultanées.

— De poster une réception ou un envoi pour les trois types de représentation de données (iovec, buffer, data). Seul un des types peut être implémenté dans le driver final. Les fonctions sont *send_XX_post* et *recv_XX_post*, où "XX" est remplacé par les trois types *iov*, *buf* et *data*. Nous utiliserons les termes "poster une réception" et "poster un envoi" par la suite.

— De scruter une réception ou un envoi avec *send_poll*, *recv_poll_one* et *recv_buf_poll*. Nous utiliserons les termes "scruter une réception" et "scruter un envoi" par la suite.

— De scruter une réception de n'importe quelle source avec *recv_probe_any*.

— De libérer une connexion et le driver en entier avec toutes les connexions respectivement avec les fonctions *disconnect* et *close*.

— De réaliser diverses actions, comme *send_buf_get* pour récupérer l'adresse d'un nouveau buffer, *recv_buf_release* pour le libérer, *get_rdv_data* et *set_rdv_data* pour les rendez-vous, etc.

Les trois types différents représentent des données de manière distincte :

— Le type "data" représente une zone mémoire contiguë allouée par l'utilisateur.

— Le type "buffer" représente également une zone mémoire contiguë, mais elle est allouée par le driver.

— Le type "iovec" représente un vecteur de plusieurs zones mémoire qui peuvent être de tailles différentes, donc non continues.

Les paquets sont distingués en deux catégories : les petits qui rentrent entièrement dans l'espace mémoire utilisé pour la communication, et les gros qui ne rentrent pas entièrement et sont donc coupés en tronçons pour être envoyés un par un. Les drivers ont une implémentation différente pour les petits et les gros paquets, donc deux instances sont manipulées.

Pour manipuler les drivers, la bibliothèque utilise une représentation nommée *nm_drv* que nous abrégeons *drv*. Ce *drv* est donc instancié et va s'occuper d'instancier un driver pour les petits paquets et un pour les gros paquets. Il va les paramétrer et les stocker avec leur URL dans son instance. Par la suite, nous allons modifier sa structure pour qu'il puisse stocker des listes en plus. Un *drv* permet donc de faire la jonction entre les drivers et

le coeur de la bibliothèque. La bibliothèque peut accéder au URL pour les partager avec d'autres processus.

Le core possède deux listes de *pw* pour la réception et pour l'envoi, qui sont *pending_send_list* et *pending_recv_list*.

Ces listes possèdent :

— Un *pw* pour chaque connexion de chaque instance de driver.

— Un *pw* supplémentaire pour la liste de réception pour chaque instance de driver qui supporte les petits paquets et qui implémente la fonction *recv_probe_any*. En effet, seuls les petits paquets peuvent avoir cette fonction. Le *pw* supplémentaire est donc posté dans le cas où l'on n'a pas posté de réception, mais qu'on en reçoit une.

Le *pw* connaît l'instance du driver à laquelle il est rattaché. Dans la fonction *nm_schedule*, faire progresser les *pw* de réception consiste à parcourir chaque *pw* de la liste de réception pour scruter une réception ou pour poster une réception dans le cas du *pw* dédié à la réception des paquets non posté. Si la scrutation est positive, une tâche du coeur (*core_task*) de complétion sera soumise. Lorsque le *nm_schedule* fait progresser les *pw* d'envoi, le fonctionnement est similaire à la réception, mais sans la gestion des paquets non posté qui n'existe que pour la reception des petit paquets.

Le polling se fait donc par des appels réguliers à *nm_schedule* depuis *Pioman*, ou par des appels à *MPI_Test* ou *MPI_Wait*.

J'ai eu des difficultés de compréhension pour le cycle de vie des *pw* qui est un peu complexe car ils passent à plusieurs endroits (les listes, les core tasks, ils sont réutilisés quand la tâche est traitée...).

5.2 Modifications

5.2.1 Ajout de drivers

Lors de ce stage, nous nous concentrons donc sur le driver qui effectue les communications en mémoire partagée, que nous appellerons *driver shm*.

Nous avons donc ajouté deux drivers basés sur le driver *shm*. Le premier est le driver *sig_shm*, qui utilise les signaux comme abstraction des uintr. Ce driver *sig_shm* a été très utile lorsque nous n'avions pas accès à la machine qui supporte les *Uintr*, ainsi que pour faciliter le débogage. Le second est le driver *uintr_shm*, qui utilise les *Uintr*.

Ces deux drivers initialisent leur handler lors de l'initialisation (*init*) et libèrent la mémoire à la terminaison (*close*). Chaque driver utilise son propre handler, soit de signal, soit d'interruption, qui permet de vérifier s'il existe un *handler de driver* défini, et le cas échéant, de l'appeler.

Un *handler de driver* est un nouveau type de handler qui est défini par une fonction *set_handler*, que nous avons ajoutée à l'interface des drivers. Cette fonction est appelée lors de l'instanciation du *drv*, juste après avoir instancié le driver, et uniquement si le driver supporte les *handler de driver*.

Le *handler de driver* est fourni par le *drv* lui-même et défini dans le *drv*. Il permet de gérer la progression des tâches, comme nous le verrons dans la section 5.2.2.

Ces deux drivers ont deux variantes, une pour les petits paquets et une pour les gros paquets.

Pour les signaux, on utilise *SIGUSR1* pour les petits paquets et *SIGUSR2* pour les gros paquets, avec un handler de signal distinct pour chacun.

Pour les *Uintr*, on utilise le vecteur 6 pour les petits paquets et le vecteur 8 pour les gros paquets. Comme nous ne pouvons définir qu'un seul handler pour les deux vecteurs, nous avons ajouté un mécanisme qui permet d'appeler un handler local au driver à partir d'un handler global. Ainsi, chaque driver possède son propre handler, que ce soit de signal ou d'interruption, capable d'appeler le *handler de driver*.

Pour ajouter le driver avec les *Uintr*, nous avons ajouté l'argument *–enable-uintr* dans le fichier *configure.ac* de *NewMadeleine*. Le fichier *configure.ac* est utilisé par *Autotools* pour la génération d'un *Makefile*. L'argument permet d'activer le support des *Uintr* uniquement lorsque le compilateur et la machine les supportent. De plus, nous avons modifié le fichier *Makefile.in* en ajoutant les bons flags de compilation nécessaires pour le support des *Uintr*.

L'envoi des signaux ou des interruptions se fait au moment de scruter une réception ou un envoi. L'objectif final est que l'émetteur envoie une interruption au récepteur pour l'avertir qu'il peut faire une scrutation. Le récepteur fait sa scrutation et envoie une interruption à l'émetteur pour le prévenir qu'il peut recevoir la suite.

Dans le cas des petits paquets, l'émetteur ne renvoie pas d'interruption car la communication est terminée. En revanche, dans le cas des gros paquets, les allers-retours continuent jusqu'à la fin de la communication. Ainsi, on ne réalise des scrutations que lorsque cela est nécessaire.

Actuellement, nous envoyons une interruption ou un signal seulement au moment de scruter l'envoi car le support des gros paquets ne fonctionne pas encore.

On effectue l'envoi de l'interruption au moment de la scrutation car la bibliothèque fait toujours une scrutation après avoir posté une communication.

5.2.2 Progression à partir des handlers

Pour faire progresser les communications à partir des handlers et non par des scrutations répétées, nous commençons par ne plus ajouter les *pw* qui sont liés à un driver possédant un *handler* dans les listes *pending_send_list* et *pending_recv_list*.

Nous avons donc ajouté deux nouvelles listes de *pw* pour la réception

et pour l'envoi dans le *drv*. Ces listes seront utilisées spécifiquement par le *handler de driver*.

Le *handler de driver* est défini dans le *drv* et est nommé *drv_handler*. Il est déclenché par le handler local des drivers lorsqu'un signal ou une interruption est reçu.

Lors du déclenchement, celui-ci connaît l'instance du *drv* et effectue une progression. Il commence par tenter de faire progresser les tâches du coeur si le verrou est *core_lock*. Pour ce faire, il essaie de prendre le verrou *core_lock* avec la fonction *try_lock*. Si le verrou est pris, alors les tâches du coeur sont exécutés. Sinon, on passe à l'étape suivante.

Il est important de rappeler qu'on ne peut pas attendre dans un handler de signal ou d'interruption, et c'est précisément ce genre de handler qui déclenche le *handler de driver*. Donc, plutôt que d'attendre, le *handler de driver*.

Ensuite, nous essayons de faire progresser tous les éléments des nouvelles liste de *pw*. Pour cela, nous appelons les mêmes fonctions que celles utilisées dans le *nm_schedule* principal. Ces fonctions ont été modifiées pour utiliser la liste appropriée en fonction de si le *pw* prend en charge les handlers ou non. Avant de passer à la progression de l'élément suivant, nous essayons de faire progresser les tâches du coeur si possible. Si ce n'est pas possible, nous ajoutons l'instance du *drv* dans une nouvelle file de *drv* en attente (*pending_drv*) qui se trouve dans le *nm_core*.

Une fois que nous avons fait progresser tous les éléments une fois, nous pouvons déterminer s'il faut le faire une nouvelle fois. Pour cela, il faut que les tâches du coeur aient pu progresser pour que les *pw* soient disponibles et qu'il soit nécessaire de faire une seconde progression. Il est important de noter que nous sommes limités en nombre de *pw* car nous ne pouvons pas effectuer d'allocation dans un handler de signal ou d'interruption. Une allocation, par exemple avec *malloc*, n'est pas "async safe" et peut entraîner une attente, ce qui n'est pas autorisé dans un handler.

Une seconde progression est nécessaire dans le cas où plusieurs interruptions ont été reçues en même temps. Étant donné que nous ne pouvons pas savoir combien d'interruptions ont été reçues car elles s'écrasent, nous utilisons la fonction *recv_probe_any* du driver pour déterminer s'il faut refaire une progression. Cependant, l'utilisation de *recv_probe_any* fonctionne uniquement pour les drivers des petits paquets. Pour les drivers des gros paquets, nous ne faisons qu'une seule progression pour le moment. Le support des gros paquets n'est pas encore terminé et présente encore des bugs.

Nous avons donc vu que nous utilisons une file de *drv* en attente, car une partie de la progression se fait dans une zone critique qui nécessite la prise d'un verrou. Comme nous ne pouvons pas attendre dans un handler, nous ne pouvons pas garantir la prise du verrou, donc nous remettons le traitement à plus tard grâce à cette file.

Les *drv* de cette file sont traités au moment où un thread relâche le

verrou du core ou lorsque nous recevons de nouveau une interruption. Le fait de traiter les progressions des *drv* peut entraîner des conflits si le handler est déclenché et fait également une progression, donc il faut protéger cette progression. Pour ce faire, nous avons utilisé l'opération atomique *compare and swap* pour nous assurer d'être seuls à faire la progression. Si le handler échoue le *compare and swap*, alors il ajoute l'instance du *drv* à la file de *drv* en attente.

5.2.3 File de *drv* en attente

La file de *drv* en attente est une file dont les opérations d'enfiler et de défiler (enqueue et dequeue) sont effectuées sans verrou (*lock-free*). Ces files *lock-free* sont donc très utilisées dans *NewMadeleine*. Le principe de fonctionnement consiste à déterminer si quelqu'un d'autre modifie la file en même temps, et si c'est le cas, on attend. Cependant, cela peut entraîner une problématique d'état bloquant (deadlock en anglais), que nous avons rencontrée lors de nos tests.

L'état bloquant survient car le thread qui est en train de modifier la file est interrompu, et le handler veut également modifier la file, donc il attend que le thread interrompu ait fini, ce qui n'arrive jamais. Pour résoudre ce problème, nous avons besoin d'une file sans attente (*wait-free*).

5.2.4 File *lock-free* et *wait-free*

J'ai donc effectué une recherche bibliographique pour trouver une implémentation dans la littérature.

L'origine des files *wait-free* provient des files *lock-free*. La première file *lock-free* est celle de Michael & Scott [20], qui est nommée *MSQueue* dans la littérature.

Une première file *wait-free*, basée sur le même fonctionnement que la *MSQueue*, est proposée par A. Kogan & E. Petrank [17] et est nommée *KPQueue*. Cependant, cette file présente des performances très limitées et ne passe pas à l'échelle. Plusieurs autres files ont été proposées pour améliorer les performances, comme la *FCQueue* de D. Hendler & I. Incze & N. Shavit & M. Tzafrir [11]. Cette file est *wait-free* grâce à l'utilisation de verrous et offre des performances raisonnables. Cependant, elle est basée sur des listes chaînées, ce qui signifie que les éléments sont alloués lors de l'insertion. Étant donné que nous ne pouvons pas utiliser d'allocateur dans un handler d'interruption, cette file ne convient pas à notre cas d'utilisation.

Les files *CCQueue* et *H-Queue* sont basées sur la même idée que la file *FCQueue*, mais elles peuvent passer à l'échelle. Elles ont été proposées par P. Fatourou & N. D. Kallimanis [8] et elles rencontrent les mêmes problèmes que la *FCQueue*.

La file *SimQueue* (également appelée *FKQueue*) est une file *lock-free* et

wait-free basée sur des listes chaînées, donc ne nous convient pas à notre cas et est moins performante que la *FCQueue*. Elle présente aussi des problèmes de libération de mémoire, comme décrit par P. Ramalhete & A. Correia [27]. Cette file a été proposée par P. Fatourou & N. D. Kallimanis [7].

Les mêmes auteurs ont également proposé *PSim* [10], qui est basé sur *SimQueue* mais avec des améliorations pour une meilleure efficacité. Cependant, il semblerait qu'il présente toujours les problèmes de libération de mémoire décrits par P. Ramalhete & A. Correia. L'accès à l'article est payant, mais selon des articles plus récents, *PSim* montre des performances relativement bonnes.

Une autre file *wait-free* qui présente de très bonnes performances est la *WFQueue* (aussi connue sous le nom de *YMC queue*), proposée par C. Yang & J. Mellor-Crummey [31]. Elle est basée sur des listes chaînées mais pourrait également rencontrer les problèmes de libération de mémoire décrits par P. Ramalhete & A. Correia.

P. Ramalhete & A. Correia présentent une file nommée *CRTurn* [27], qui est *wait-free* et cherche à résoudre les problèmes de libération de mémoire. Leur implémentation n'est pas très performante, mais ils discutent d'une grande partie des files précédentes et les regroupent dans un même dépôt *Github*, avec un moyen de mesurer leurs performances.

En 2019, R. Nikolaev propose une file *lock-free* uniquement, nommée *SCQ* [23]. Elle est inspirée de la file *CRQ*, mais elle est plus efficace en termes de mémoire et évite les problèmes de *livelocks*. La file *CRQ* est une file *lock-free* performante qui utilise l'opération atomique *compare and swap* sur 2 mots contigus en mémoire (*CAS2*), qui n'est pas disponible sur tous les CPU. *CRQ* est proposée par A. Morrison & Y. Afek [21]. *SCQ* n'utilise pas les *CAS2* et parvient à des performances similaires à *CRQ*. De plus, *SCQ* passe très bien à l'échelle.

En 2022, la file *wCQ* est proposée par R. Nikolaev & B. Ravindran [24]. Basée sur *SCQ*, elle est en plus *wait-free*. Cette file présente de bonnes performances et passe très bien à l'échelle. Les auteurs se sont appuyés sur le dépôt *GitHub* de P. Ramalhete & A. Correia pour y ajouter leur implémentation dans ce nouveau dépôt [30].

Pendant mes recherches, mon tuteur de stage, *Alexandre Denis*, a amélioré son implémentation de la *lfqueue* pour "Lock-Free Queue". Dans cette file, les éléments qui peuvent y être insérés sont des pointeurs. Il a utilisé une technique très astucieuse consistant à utiliser les 16 premiers bits non utilisés des pointeurs pour compter le nombre de modifications qu'a subi une entrée de la file. Ainsi, s'il y a eu un changement entre la lecture et l'écriture atomique, il sait qu'une autre opération modifie la liste et peut réagir en conséquence. Cette implémentation a plusieurs variantes et je ne suis pas sûr de pouvoir bien les expliquer. Dans tous les cas, cette amélioration fera l'objet d'une publication future.

Son implémentation serait meilleure que la *wCQ* avec peu de threads et

peu de contention, mais un peu moins performante dans les autres cas. Tout ceci est en cours de test.

L'amélioration de la *lfqueue* nous a permis de corriger les deadlocks.

5.3 Suite de tests

NewMadeleine possède une suite de tests très complete qui test toutes ses interfaces, la charge, etc. Nous avons donc executer la suite de tests après nos modification. Environ 87 tests passe sur les 93. Les tests qui ne passe pas sont généralement les collective qui engendre beaucoup de communications et certains test RPC, ces tests appel du code utilisateur qui peut utiliser des fonction non "async safe" et donc posé un problème avec le handler d'interruption. Le fait que ces tests ne passe pas est du un problèmes de modification concurrent des *pw*. Nous avons une solution temporaire où l'on masque les interruptions qui nous permet de passer tous les tests sauf certain RPC. Ce bug reste donc à corriger.

NewMadeleine possède une suite de tests très complète qui teste toutes ses interfaces, la charge, etc. Nous avons donc exécuté la suite de tests après nos modifications. Environ 87 tests passent sur les 93. Les tests qui ne passent pas sont généralement les collectives qui engendrent beaucoup de communications, ainsi que certains tests RPC. Ces tests appellent du code utilisateur qui peut utiliser des fonctions non "async safe" et poser ainsi un problème avec le handler d'interruption. Le fait que ces tests ne passent pas est dû à un problème de modification concurrente des *pw*. Nous avons une solution temporaire où l'on masque les interruptions, ce qui nous permet de passer tous les tests sauf certains RPC. Ce bug reste donc à corriger.

Lors de l'exécution de la suite de tests sur la machine fournie par *Atos*, nous avons été confrontés à des crashs. En effet, la machine est mal refroidie et elle plante lorsqu'elle est soumise à une charge élevée. Même en configurant le CPU en mode économie d'énergie, la machine finit par crasher. On peut voir sur la figure 14 en annexe les courbes de température au cours du temps, en bleu pour un CPU et en orange pour un autre. Les mesures sont faite par la *BMC*.

5.4 Reste à faire et les bugs

Il reste donc à corriger le bug de modification concurrente des *pw*, à finir le support des gros paquets en envoyant des interruptions au moment où l'on scrute la réception, et à implémenter le support de *Pioman* pour la progression multithread.

Une communication de gros paquets commence toujours par un petit paquet pour l'entête, puis passe aux gros paquets. Actuellement, le passage des petits paquets aux gros paquets ne fonctionne pas correctement avec les interruptions, et cela pourrait être dû au masquage des interruptions. Le bug

existe également avec les signaux, mais lorsque l'on force l'envoi d'un signal, la communication se déroule correctement, c'est-à-dire avec les échanges de signaux aller-retour, etc.

5.5 Performances

Nous avons donc mesuré les performances de nos drivers en utilisant les benchmarks de *NewMadeleine*.

Les résultats suivants se concentrent uniquement sur les tailles de paquets (ou messages) inférieurs à 16 ko, ce qui correspond à la taille des petits paquets.

5.5.1 Résultats avec attente active

Le premier benchmark est *nm_bench_sendrecv*, qui mesure notamment la latence et le débit d'un aller-retour (ping-pong) en attente active. Sur la figure 8, on peut voir les courbes de latence.

Sur l'axe des abscisses, nous voyons la taille des paquets, et sur l'axe des ordonnées, la latence en microsecondes. La courbe bleue représente les performances du driver *shm*, qui utilise uniquement l'attente active. La courbe verte montre les performances du driver *uintr_shm*, où l'on peut remarquer que la latence est la même que pour le driver *shm*, avec la latence des *Uintr* en plus.

Cela est conforme aux attentes, car dans ce cas, l'attente active a forcément de meilleures performances car elle se contente de vérifier si un paquet est reçu. Enfin, la courbe orange représente les performances du driver *sig_shm*, qui ont également la latence des signaux en plus et même un peut plus.

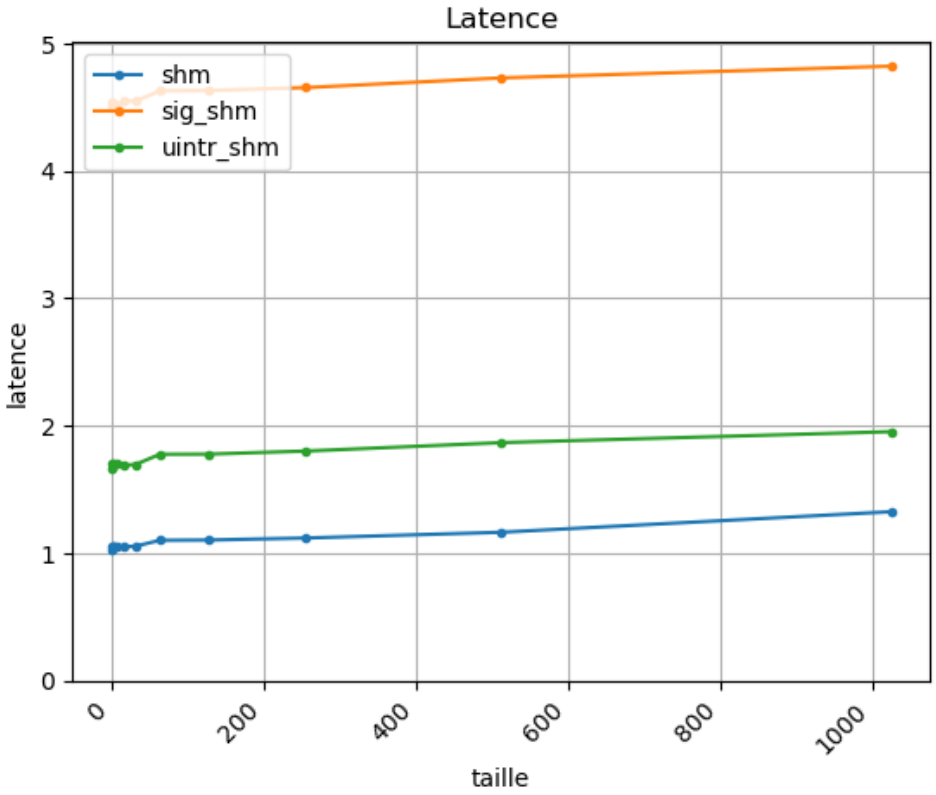


FIGURE 8 – Latence avec le benchmark *sendrecv*

Sur la figure 9, nous pouvons observer les courbes de débit du même benchmark. L'axe des abscisses représente la taille des paquets, et l'axe des ordonnées indique le débit en Mo par seconde. Grâce à ces courbes de débit, on remarque que le surcoût ajouté par l'utilisation des *Uintr* ou des signaux est constant.

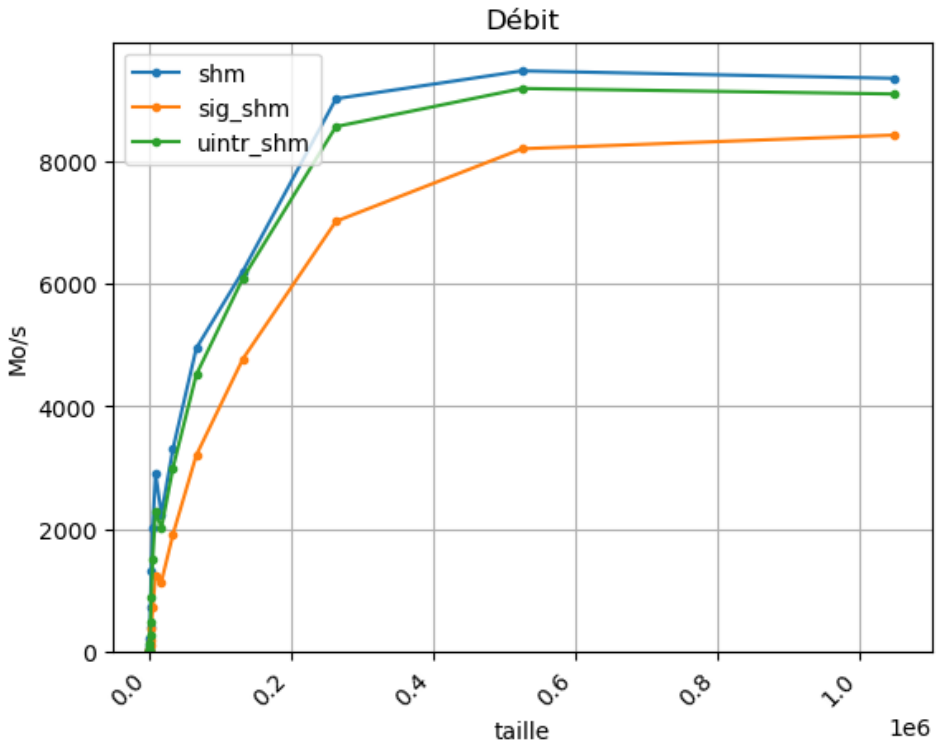


FIGURE 9 – Débit avec le benchmark *sendrecv*

Ce cas d'attente active est peu représentatif des vraies applications où l'on cherche à recouvrir la latence des communications avec du calcul. Par conséquent, on accepte de perdre un peu en performances en attente active pour gagner ailleurs.

5.5.2 Résultats du recouvrement des communications par du calcul

Dans la plupart des applications HPC, on cherche à recouvrir la latence des communications avec du calcul, c'est ce qu'on appelle l'*overlap* en anglais. Le comportement souhaité est donc que l'application effectue du calcul en continu, et qu'elle soit interrompue à certains moments pour effectuer les communications. Un exemple illustratif est présenté dans la figure 19 en annexe, où l'on compare une application utilisant le polling avec une autre application utilisant les *Uintr*.

Les figures suivantes mesurent l'*overlap* au niveau de la réception, car actuellement, seul le récepteur est interrompu. Nous nous concentrerons uniquement sur la partie de gauche de ces figures, qui concerne les tailles inférieures à 16 ko.

Sur la figure 10, on peut voir un graphique sous forme de tuile qui re-

présente la taille des messages sur l'axe des abscisses et le temps de calcul en microsecondes sur l'axe des ordonnées. Les valeurs dans les coins en haut à gauche et en bas à droite ont tendance à être erronées en raison de la précision de la mesure. Quand on envoie très peu de données avec un temps de calcul très long, il est compliqué de voir une différence de temps liée à l'envoi.

Les couleurs correspondent à un ratio du temps mesuré de l'exécution moins le plus grand temps entre le temps de calcul et le temps de communication, le tout normalisé à 1. La couleur noire correspond donc à une valeur de 0, ce qui signifie que l'*overlap* est parfait. La couleur rouge correspond à 1, donc il n'y a pas d'*overlap*, et la couleur jaune correspond à 2 ou plus, ce qui signifie que l'execution est plus lente que si on n'avait pas essayé de faire d'*overlap*. Toutes les valeurs entre 0 et 1 qui passent du noir au violet pour arriver au rouge correspondent donc à un *overlap* plus ou moins présent.

Donc, notre figure correspond au benchmark d'*overlap* à la réception avec le driver *shm*. On peut voir qu'il n'y a pas vraiment d'*overlap* car c'est un driver qui fait seulement de l'attente active.

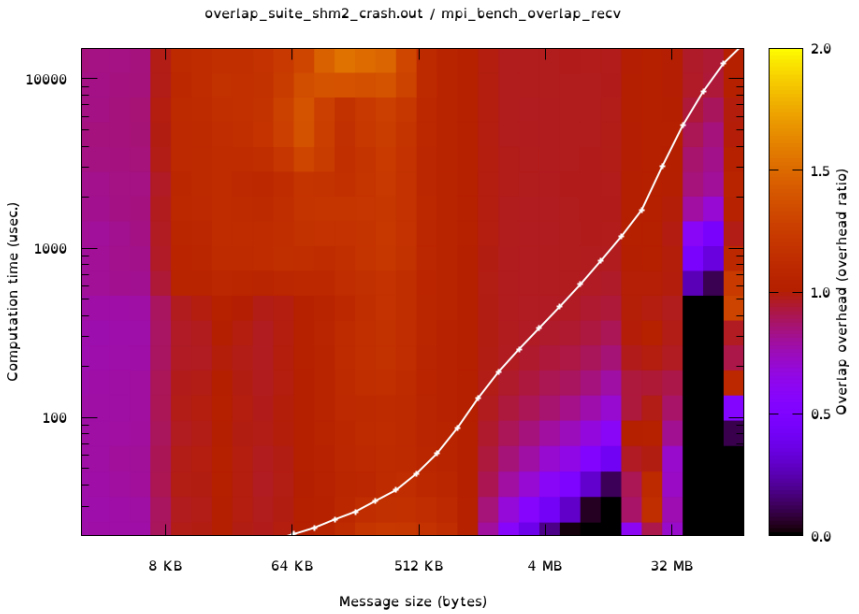


FIGURE 10 – Benchmark d'*overlap* pour la réception avec le driver *shm*

Sur la figure 11, on peut voir un graphique qui correspond au benchmark d'*overlap* à la réception avec le driver *uintr_shm*. On remarque que pour les petits paquets, les messages de moins de 16ko, l'*overlap* est presque parfait.

Même pour les plus gros paquets, on constate une légère amélioration car le premier paquet d'entête passe par les petits paquets.

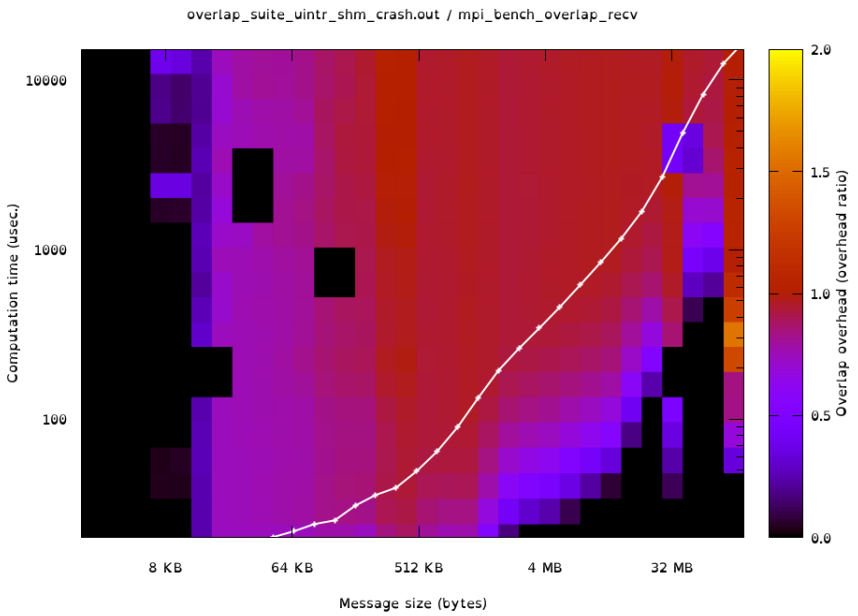


FIGURE 11 – Benchmark d'overlap pour la réception avec le driver *uintr_shm*

Ces résultats nous montrent que l'utilisation d'*Uintr* permet effectivement de faire progresser les communications sans recourir au polling, tout en offrant de bonnes performances.

6 Bilan

7 Remerciements

8 Annexes

```
git clone git@github.com:intel/uintr-linux-kernel.git
```

Listing 4 – Récupération du code source du noyau Linux patché

```
# Configurer le noyau sur la machine cible
cp -v /boot/config-$(uname -r) .config
make menuconfig

# Désactiver les clef de signature
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS

# Compiler
make -j 16
```

Listing 5 – Configurer et compiler le noyau

```
sudo make modules_install
sudo make install
```

Listing 6 – Installer le noyau

```
# À faire la premier fois
sudo grub2-mkconfig -o /boot/grub2/grub.cfg

sudo grubby --set-default /boot/vmlinuz-6.0.0+
```

Listing 7 – Utiliser le noyau installé

```
make -C tools/testing/selftests TARGETS=uintr run_tests
```

Listing 8 – Lancer les tests *Uintr* du noyau (le noyau doit être installé)

```
sudo grubby --remove-kernel /boot/vmlinuz-6.0.0+
```

Listing 9 – Supprimer le noyau installé

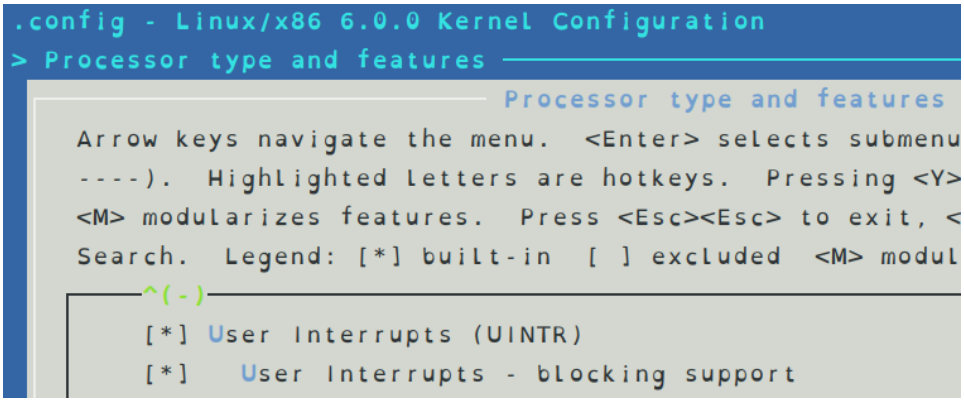


FIGURE 12 – Activer le support des *Uintr* à la compilation du noyau

```c
#include <x86gprintrin.h>

#define VECTOR 6
bool isOver = 0;

__attribute__((target("general-regs-only")))
__attribute__((interrupt))
void ui_handler(struct __uintr_frame* ui_frame, unsigned long
   vector) {
  /* Second clock_gettime() here */
  printf("handler invoked (vector: %lld)\n", vector);

  isOver = 1;
}

void receiver() {
  if (uintr_register_handler(ui_handler, 0)) {
    perror("interrupt handler register error");
    exit(1);
  }

  _stui();

  int uvec_fd = uintr_vector_fd(VECTOR, 0);
  if (uvec_fd < 0) {
    perror("vector fd error");
    exit(1);
  }

  send_FD_to_sender(uvec_fd);

  while (!isOver) continue;

  if (uintr_unregister_handler(0))
    perror("interrupt handler unregister error");
  close(uvec_fd);
}
```

Listing 10 – Code récepteur

```c
#include <x86gprintrin.h>

void sender(pid_t targetPid) {
  int uvec_fd = wait_to_receive_FD_from_receiver(targetPid);

  long uipi_index = uintr_register_sender(uvec_fd, 0);
  if (uipi_index < 0) {
    perror("sender register error");
    exit(1);
  }

  /* First clock_gettime() here */
  _senduipi(uipi_index);

```

```
15      if (uintr_unregister_sender(uipi_index, 0))
16        perror("sender unregister error");
17      close(uvec_fd);
18    }
```

Listing 11 – Code émetteur

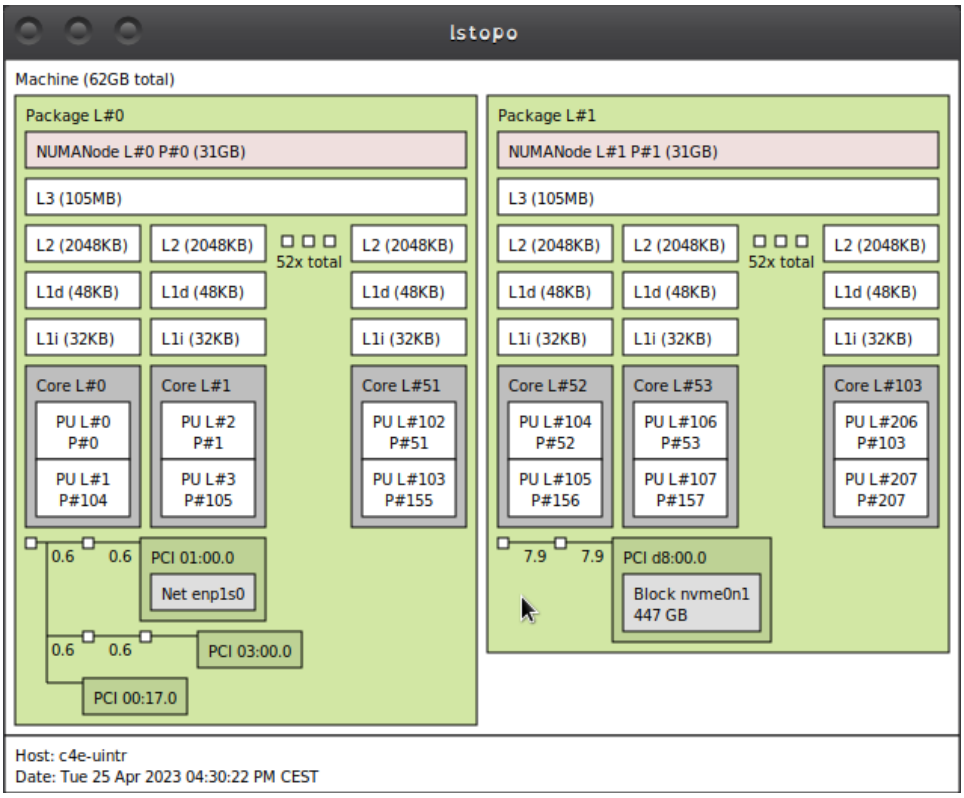


FIGURE 13 – Topologie de la machine fournie par *Atos* obtenue grâce à la commande *lstopo*

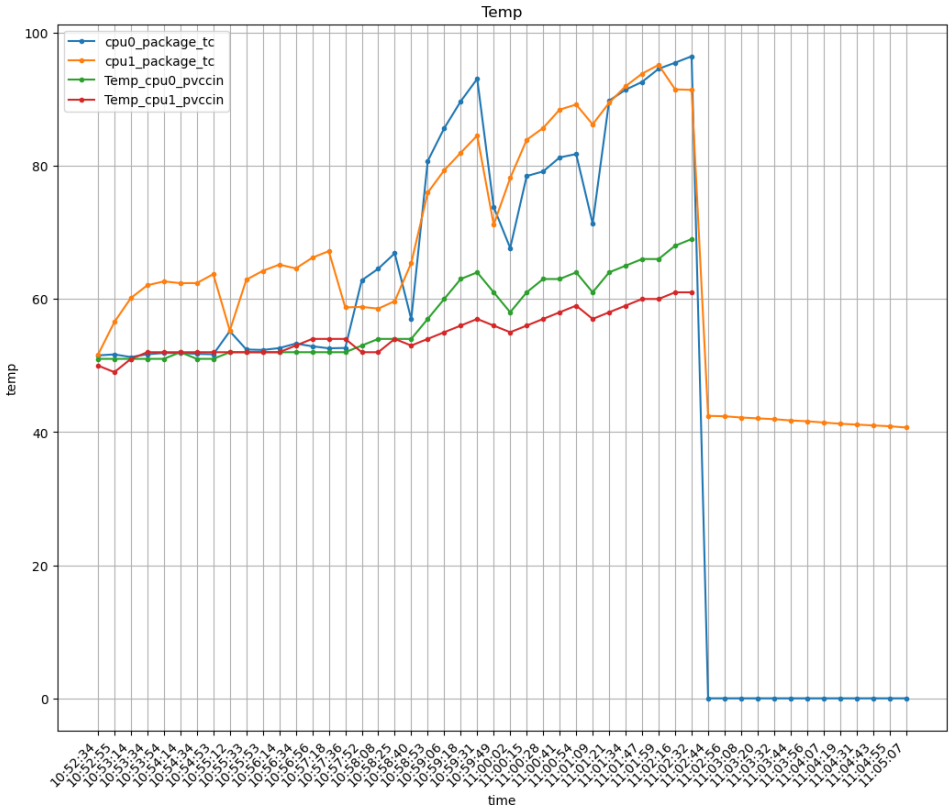
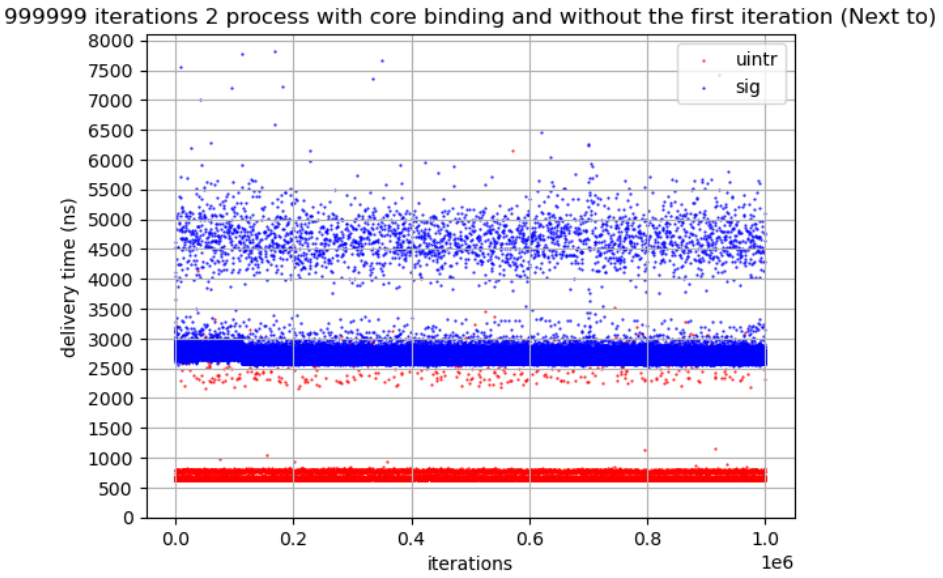


FIGURE 14 – Courbes de température pour la machine fourni par *Atos*

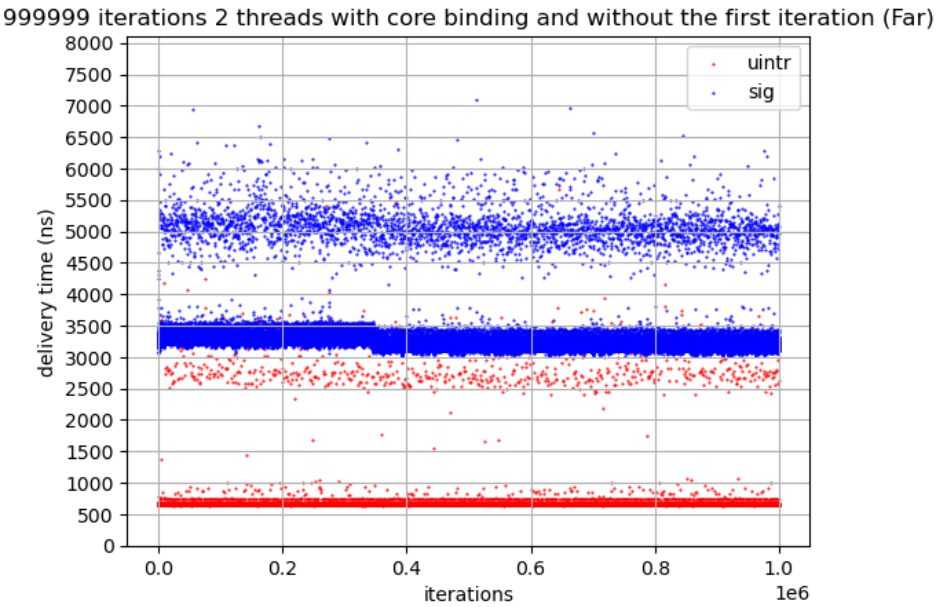


(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	2675	147	2520	2620	2658	2781	44311
uintr	651	72	631	642	649	659	46053

(b)

FIGURE 15 – Mesures de latence entre deux processus avec un placement proche

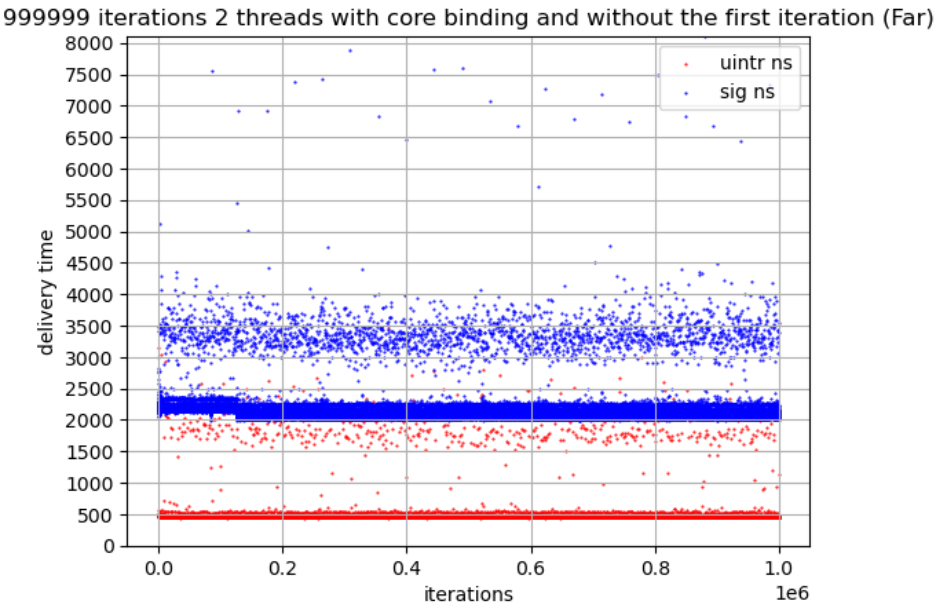


(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	3210	156	3014	3141	3177	3302	63690
uintr	654	329	638	648	653	659	325408

(b)

FIGURE 16 – Mesures de latence entre deux threads avec un placement éloigné

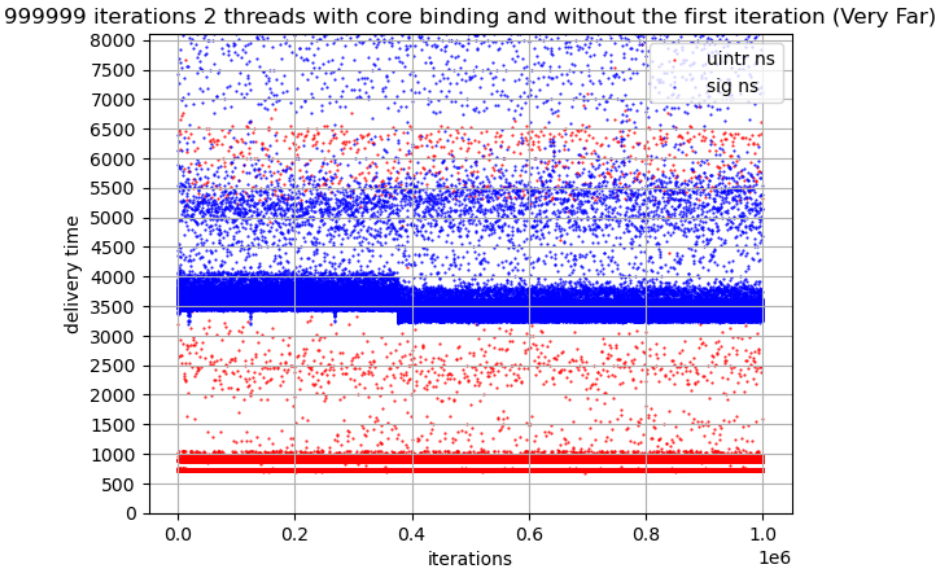


(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	2097	282	1981	2064	2082	2178	270625
uintr	455	389	440	450	454	460	388815

(b)

FIGURE 17 – Mesures de latence entre deux threads avec un placement éloigné et le turbo boost activé



(a)

	mean	**std**	**min**	**10%**	**50%**	**95%**	**max**
sig	3520	327	3189	3383	3481	3678	51798
uintr	736	159	683	721	725	735	46787

(b)

FIGURE 18 – Mesures de latence entre deux threads avec un placement très éloigné et le turbo boost activé

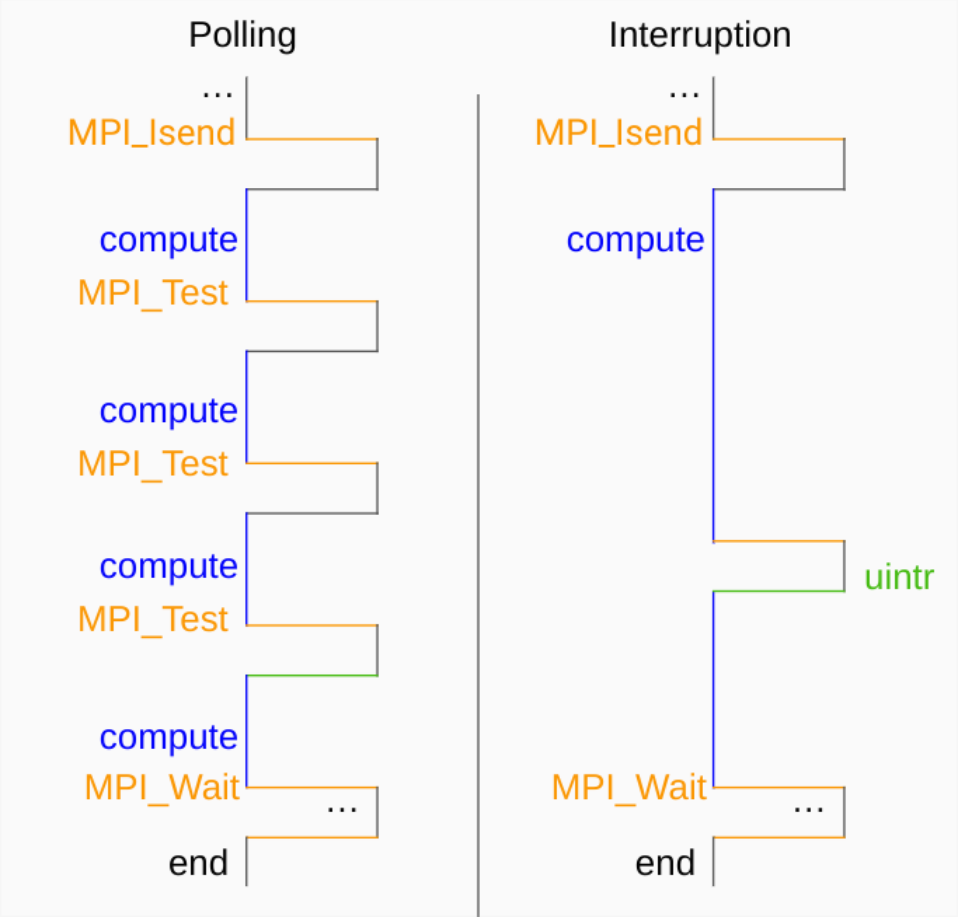


FIGURE 19 – Application MPI qui utilise du polling contre une qui utilise des *Uintr*

Références

[1] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks. In *Workshop on Communication Architecture for Clusters (CAC 2007), workshop held in conjunction with IPDPS 2007*, Long Beach, California, United States, March 2007.

[2] Mathieu Barbe. Rapport de projet de fin d'études : Design and implement interrupt-handling code of a low-latency network adapter linux driver, 18 mars 2019 - 27 septembre 2019. Chez Bull (Atos). Grenoble INP - ENSIMAG.

[3] Alexandre Denis. pioman : a pthread-based Multithreaded Communication Engine. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, March 2015.

[4] Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, and Florian Reynier. One core dedicated to MPI nonblocking communication progression ? A model to assess whether it is worth it. In *International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, Taormina, Italy, May 2022.

[5] Alexandre Denis and François Trahay. MPI Overlap : Benchmark and Analysis. In *International Conference on Parallel Processing*, 45th International Conference on Parallel Processing, Philadelphia, United States, August 2016.

[6] Saïd Derradji, Thibaut Palfer-Sollier, Jean-Pierre Panziera, Axel Poudes, and François Wellenreiter Atos. The bxi interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, 2015.

[7] Panagiota Fatourou and Nikolaos Kallimanis. A highly-efficient wait-free universal construction. pages 325–334, 06 2011.

[8] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. *SIGPLAN Not.*, 47(8) :257–266, feb 2012.

[9] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 257–266, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3) :475–520, Oct 2014.

[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Portable hardware locality (hwloc). https://www.open-mpi.org/projects/hwloc/.

[13] Intel® 64 and ia-32 architectures, software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[14] Intel linux kernel fork with uintr. https://github.com/intel/uintr-linux-kernel/tree/uintr-next.

[15] Intel uintr linux man pages. https://github.com/intel/uintr-linux-kernel/tree/uintr-next/tools/uintr/manpages.

[16] Sujet de stage. https://dept-info.labri.fr/~denis/Enseignement/Sujet_PFE_2023_uintr.html.

[17] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 223–234, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, 46(8) :223–234, feb 2011.

[19] Signal linux man pages. https://man7.org/linux/man-pages/man2/sigaction.2.html.

[20] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1) :1–26, 1998.

[21] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. *SIGPLAN Not.*, 48(8) :103–112, feb 2013.

[22] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 103–112, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free fifo queue. 2019.

[24] Ruslan Nikolaev and Binoy Ravindran. Wcq : A fast wait-free queue with bounded memory usage. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '22, page 307–319, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Portals 4.0 specification. https://www.sandia.gov/portals/portals-4-0-specification-clone-2/.

[26] Pull request for uintr_alt_stack() fix. https://github.com/intel/uintr-linux-kernel/pull/2.

[27] Pedro Ramalhete and Andreia Correia. Poster : A wait-free queue with wait-free memory reclamation. *SIGPLAN Not.*, 52(8) :453–454, jan 2017. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf.

[28] Pedro Ramalhete and Andreia Correia. Poster : A wait-free queue with wait-free memory reclamation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 453–454, New York, NY, USA, 2017. Association for Computing Machinery. https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf.

[29] Présentation du support des interruptions en espace utilisateur avec diapositive sur lwn.net. https://lwn.net/Articles/869140/.

[30] wcq, scq, wcq... benchmark. https://github.com/rusnikola/lfqueue.

[31] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Not.*, 51(8), feb 2016.

[32] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, New York, NY, USA, 2016. Association for Computing Machinery.