RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

Réactivité aux événements d'entrées/sorties dans les environnements multithreads

Vincent Danjean

LIP, ENS Lyon, 46 allée d'Italie, F-69364 Lyon Cedex 07, France

Résumé
La réactivité aux événements d'entrées/sorties est un facteur de performance crucial pour les systèmes multithreadés distribués modernes. Nous proposons une nouvelle interface de gestion des événements d'E/S dont l'objectif est de fournir des mécanismes génériques permettant aux environnements d'appréhender la détection de ces événements de manière uniforme. Cette approche est fondée sur un support spécifique de l'ordonnanceur de threads, ce dernier pouvant être vu comme un *serveur* de détection d'événements du point de vue des environnements clients. Cette implantation au sein de l'ordonnanceur augmente les performances et permet un contrôle bien plus fin sur le niveau de réactivité obtenu.

Mots-clés : Réactivité, Multithreading, Entrées/Sorties, Scrutation, Interruptions.

La conception d'environnements d'exécution parallèle pour architectures distribuées est une activité de recherche qui connaît un important regain d'intérêt ces dernières années [9, 4, 8]. Les principales raisons tiennent bien sûr à l'augmentation des exigences des applications parallèles, notamment en matière de portabilité. Par exemple, l'exploitation à la fois transparente et performante des réseaux rapides que la diversité des technologies existantes (Myrinet, SCI, etc.) rend très complexe [7, 13] illustre bien cette tendance. Dans ce domaine, beaucoup de progrès ont été faits et l'on sait aujourd'hui construire des bibliothèques de communication permettant d'approcher les capacités du matériel en termes de latence et de bande passante [15, 3]. Toutefois, il demeure une propriété importante du point de vue des applications qui est souvent négligée par les supports exécutifs : il s'agit de la réactivité aux événements d'entrée/sortie relatifs aux communications, c'est-à-dire la capacité, pour une application, de réagir à des événements asynchrones extérieurs aussitôt que possible durant son déroulement normal. Chercher à minimiser – ou au moins à borner – ce temps de réaction est en fait un facteur crucial puisqu'il conditionne la latence réseau effectivement observée[1] au sein des applications mettant en œuvre des schémas de communication irréguliers. En pratique, de nombreux programmes verront leur exécution pénalisée si la signalisation des événements extérieurs n'est pas prise en compte au plus tôt (messages de contrôle, requêtes d'annulation, interrogations à distance, etc.) [2].

Si ce problème de réactivité se rencontre naturellement en contexte monothread, comme en attestent notamment les nombreuses implantations de MPI souffrant de ce problème [16], il n'en concerne pas moins les applications multithreads au sein desquelles il est encore plus difficile de proposer des solutions génériques en raison des disparités entre les ordonnanceurs de threads. Paradoxalement, c'est pourtant l'utilisation de processus légers dédiés aux traitement des entrées/sorties qui est généralement proposée pour améliorer la réactivité des applications monothread [16]. Pour toutes ces raisons, et aussi parce que les environnements multithreads sont aujourd'hui incontournables sur les architectures répandues que sont les grappes de PC multiprocesseurs [14], nous avons focalisé nos efforts sur l'étude de la réactivité dans le cadre multithreads. En outre, bien que ces travaux s'appliquent aux entrées/sorties au sens large, nous restreindrons les illustrations utilisées dans cet article au traitement des événements d'E/S relatifs au réseau.

[1] Si l'application est peu réactive aux événements réseaux, cette latence peut être arbitrairement éloignée de la latence promise par la bibliothèque de communication sous-jacente.

1. Problématique

En s'intéressant à la réactivité, la principale difficulté à laquelle se heurtent les concepteurs d'environnements d'exécution réside dans l'hétérogénéité des méthodes de détection des événements d'E/S. L'approche la plus utilisée est sans doute la *scrutation active* qui consiste à vérifier régulièrement si des événements d'E/S sont survenus en appelant une fonction appropriée du dispositif d'E/S. Un tel test est généralement peu coûteux, mais sa répétition un grand nombre de fois peut occuper les ressources de calcul de manière prohibitive. Par ailleurs, l'application peut opter pour une *stratégie passive* en utilisant des appels systèmes bloquants par exemple. Dans ce cas, les événements d'E/S sont notifiés au système d'exploitation par des interruptions matérielles du périphérique d'E/S, ce qui rend cette approche plus réactive. Malheureusement, récupérer une telle interruption est généralement coûteux[2], indépendamment de la latence additionnelle pour réordonnancer l'application.
Généralement, le choix de la stratégie de détection des E/S est fait dans l'application, ce qui ruine tout espoir de réutilisabilité. De plus, cette approche souffre de nombreux inconvénients :

Déterminer les méthodes disponibles. Le système (i.e. le dispositif d'E/S sous-jacent) n'offre souvent qu'un ensemble restreint de méthodes. Dans certains cas, seule la scrutation active est utilisable. Dans d'autres, la récupération d'interruptions peut être la seule manière de vérifier la terminaison d'une opération d'E/S. En outre, certains mécanismes peuvent être incompatibles avec les contraintes applicatives. Par exemple, la délivrance directe de signaux asynchrones impose de grandes contraintes de réentrance sur l'ensemble du code applicatif si la consistance de *toutes* les structures de données accédées doit être garantie.

Sélectionner la bonne méthode. Quand plusieurs méthodes sont disponibles au niveau du système d'-exploitation, sélectionner la plus appropriée dépend de nombreux facteurs. Un facteur important est le niveau auquel l'ordonnanceur de threads est implémenté. Actuellement, il existe de nombreuses variantes d'ordonnanceurs de threads (utilisateur, noyau, hybride) et chacune d'entre elles offre des caractéristiques propres. Par exemple, avec un ordonnanceur purement utilisateur, des opérations telles que les appels systèmes bloquants sont à proscrire, sauf si certaines extensions évoluées (comme les *Scheduler Activations* [1, 5, 6]) sont disponibles dans le système d'exploitation.

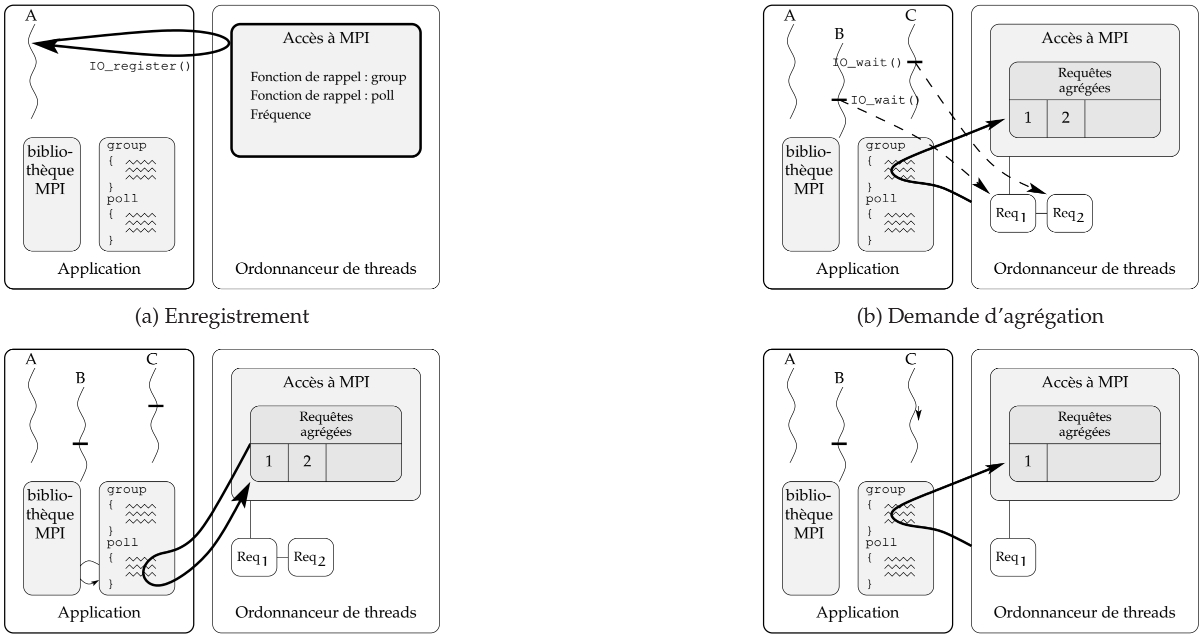
Adopter une stratégie adaptative. La plupart des dispositifs d'E/S (par exemple les gestionnaires de périphériques) offrent de façon native un mécanisme de scrutation peu coûteux. Cependant, utiliser efficacement un tel mécanisme est un défi difficile dans un contexte multithread [10, 11]. En particulier, la fréquence de scrutation a un impact décisif sur les performances globales de l'application. Si le dispositif d'E/S n'est pas scruté assez fréquemment, alors la réactivité de l'application peut fortement se dégrader. À l'inverse, une politique de scrutation trop agressive conduit à de nombreuses scrutations improductives, ce qui gaspille des ressources de calcul. Même dans les cas où l'on peut prédéterminer la fréquence « optimale », la conception d'un support exécutif capable de garantir une fréquence de scrutation reste un problème difficile.

La contribution de cet article est d'introduire une nouvelle approche du problème de la réactivité aux événements d'E/S dans les environnements multithreadés. Nous proposons une interface uniforme de gestion des événements d'E/S centrée sur l'ordonnanceur de threads. La sélection effective de la stratégie (scrutation active ou attente passive) est laissée à l'appréciation de l'ordonnanceur. Ceci permet d'encapsuler dans ce dernier tous les mécanismes de gestion de la réactivité, et donc de soulager le programmeur de cette tâche difficile. En outre, ceci permet à l'ordonnanceur d'ajuster sa stratégie d'ordonnancement avec le niveau de réactivité requis par les applications, indépendamment de la charge du système. Enfin, cela permet d'agréger les multiples requêtes des différentes tâches concurrentes pour un même dispositif d'E/S. Nous montrons la faisabilité de cette nouvelle approche dans le contexte de l'ordonnanceur de threads de PM^2 [14]. Des gains de performance significatifs ont été observés.

2. Notre proposition : une approche fondée sur *un support dans l'ordonnanceur*

Nous proposons de *centraliser* la gestion des événements d'E/S, dans l'ordonnanceur, de manière à fournir à l'application un mécanisme *uniforme* pour attendre la terminaison des événements d'E/S. Au

[2] de l'ordre de la dizaine de micro-secondes



(a) Enregistrement

(b) Demande d'agrégation

(c) Scrutation

(d) Événement

1(a) L'application enregistre les paramètres qui seront utilisés pour MPI dans l'ordonnanceur : les fonctions de rappel à utiliser (pour scruter, agréger les requêtes, etc.), et la fréquence de scrutation (le nombre de quanta de temps entre chaque scrutation).

1(b) Deux threads utilisateurs B et C ont soumis une requête de détection d'événement à l'ordonnanceur. Ils sont mis dans l'état endormi et enlevés de la liste des threads prêts. L'ordonnanceur utilise la fonction de rappel enregistrée *group* pour agréger les requêtes.

1(c) L'ordonnanceur scrute le réseau tous les n changements de contexte en utilisant la fonction de rappel *poll*. Comme les deux requêtes sont agrégées, un seul appel à MPI est nécessaire.

1(d) L'événement attendu est survenu. L'ordonnanceur réveille le thread correspondant (ici le thread C). Il utilise la fonction de rappel *group* pour agréger les requêtes restantes (ici, seulement une).

FIG. 1 – Un scénario MPI avec scrutation.

lieu que la détection de terminaison des E/S soit une partie explicite du code applicatif, nous voyons une telle action comme un *service de détection d'événement* demandé par l'application à un *serveur externe* : l'ordonnanceur. Le thread client est enlevé de la liste des threads prêts en attendant la terminaison du service. C'est la tâche de l'ordonnanceur que de déterminer la meilleure manière de servir la demande : par scrutation, gestion d'interruption, ou toute sorte de compromis adaptatif entre de ces méthodes, et de réveiller le thread ayant effectué la requête. Cette idée est illustrée dans la Figure 1 dans le cas où les opérations d'E/S sont en fait des communications réalisées avec MPI [12].

Pourquoi privilégier l'ordonnanceur de threads ?

Premièrement, l'ordonnanceur de threads connaît les caractéristiques du système, par opposition à une application dont le code doit être portable sur de nombreux systèmes. Aussi, l'ordonnanceur a l'entière liberté d'utiliser de manière sûre tous les mécanismes fournis par le système d'exploitation, y compris les plus sophistiqués. Par exemple, un ordonnanceur de threads purement utilisateur "sait" qu'il est dangereux d'effectuer un appel système qui peut éventuellement bloquer le processus, sauf quand il n'y a qu'un seul thread actif dans le système. De plus, si certains mécanismes asynchrones sont disponibles, alors l'ordonnanceur de threads peut fournir des primitives de synchronisation fonctionnant avec les signaux pour les threads attendant des événements d'E/S tout en fournissant des primitives normales et rapides pour les autres threads.

Deuxièmement, l'ordonnanceur est certainement la meilleure place où une scrutation efficace peut être faite. En particulier, la fréquence spécifique de scrutation pour chaque thread demandeur peut être précisément observée par l'ordonnanceur puisqu'il connaît toutes les informations utiles ; ainsi une décision optimale peut-elle être prise à chaque changement de contexte. De plus, l'ordonnanceur peut maintenir pour chaque type de requête un historique des précédentes requêtes dans l'optique de sélectionner le mécanisme le plus efficace. Une stratégie possible serait de commencer par faire de la scrutation active puis de basculer vers l'attente passive d'une interruption après un certain temps. En outre, l'ordonnanceur peut utiliser les périodes d'oisiveté pour effectuer des scrutations intensives si une haute priorité leur a été donnée. Finalement, l'ordonnanceur a une grande latitude quant au prochain thread à ordonnancer. Il peut alors ordonnancer un thread aussitôt que son événement d'E/S a été détecté.

Troisièmement, l'ordonnanceur apparaît comme le seul point d'entrée pour les requêtes de détection d'événements. Ceci fournit une opportunité intéressante pour *agréger* les requêtes émises par les différents threads. Par exemple, si plusieurs threads sont en attente d'un message sur la même interface réseau, alors il n'y a aucunement besoin que chacun d'entre eux scrute l'interface : l'ordonnanceur peut agréger toutes les requêtes et scruter l'interface avec l'ensemble des requêtes ; une fois qu'un événement a été détecté, il peut parcourir ses tables internes de requêtes pour déterminer quel thread doit être servi. Observons que cette capacité d'agrégation est totalement compatible avec les autres aspects énumérés auparavant : on peut très bien utiliser un mélange de scrutation active et d'attente passive pour détecter les événements communs pour de multiples requêtes d'E/S ! Cette idée généralise la fonctionnalité `MPI_testany()` de MPI à toute sorte de requêtes d'événements, utilisant n'importe quel type d'interface de communication.

Une interface de programmation uniforme.

Nous avons conçu l'interface de programmation de manière à isoler l'application des particularités des différentes interfaces réseaux et des événements spécifiques lors des détections. L'idée générale est que l'application cliente, la plupart du temps une bibliothèque de communication, doit enregistrer les fonctions spécifiques de rappel qui seront utilisées par l'ordonnanceur pour effectuer ses requêtes.

L'application doit d'abord enregistrer auprès de l'ordonnanceur quel type d'événements elle a l'intention de détecter et comment. Ceci est fait en remplissant les champs d'une structure `params` avec un certain nombre de paramètres : fonctions de rappel pour scruter les événements et grouper les requêtes ensemble, fréquence objective de scrutation, etc. La primitive `thread_IO_register` retourne alors un *identifiant* qui sera utilisé pour toute requête futur de ce type d'événement. Seules les requêtes avec le même identifiant pourront être agrégées ensemble. Les threads clients ont alors à leur disposition une *unique* primitive pour attendre la venu d'un événement d'E/S :

```
thread_IO_t  thread_IO_register  (thread_IO_registration_t  params);
void   thread_IO_wait  (thread_IO_t  IO_handle,  void   *arg);
```

La fonction `thread_IO_wait` est une fonction bloquante (pour le thread appelant). Il est très facile de faire des communications asynchrones avec cette primitive : il suffit de créer un nouveau thread pour s'occuper de la communication (rappelons que nous nous plaçons en contexte multithread). L'argument `arg` de la primitive `thread_IO_wait` sera transmis aux fonctions de rappel précédemment enregistrées. L'ordonnanceur, lui, ne connaît rien de ces fonctions. La primitive doit revenir de l'ordonnanceur aussitôt que possible une fois que l'événement est prêt à être traité. Par exemple, enregistrer une *routine de scrutation* pour MPI et recevoir un message MPI asynchrone ressembleraient à :

```
thread_IO_registration_t MPI_params;
thread_IO_t MPI_handle;
...
MPI_params.blocking_system_call = NULL;
MPI_params.group = &MPI_group();
MPI_params.poll = &MPI_poll();
MPI_params.frequency = 1;

MPI_handle=thread_IO_register(&MPI_params);
```

```
MPI_Request request;
MPI_IO_info_t MPI_IO_info;
...
MPI_Irecv(buf, size, ..., &request);

MPI_IO_info.request = request;
thread_IO_wait(MPI_handle,
               (void *) &MPI_IO_info);
```

3. Détails d'implantation

Nous avons implémenté notre mécanisme générique dans l'ordonnanceur "caméléon" de l'environnement multithreadé PM^2 [14] qui peut utiliser sur demande les différents types d'ordonnancement suivants : niveau utilisateur ou hybride. Notre mécanisme est virtuellement capable de traiter avec un très grand nombre de combinaisons de *types d'ordonnancement/capacité des gestionnaires de périphérique*. Nous nous intéressons par la suite aux situations les plus communes.

Scrutation active

Un certain nombre de fonctions de rappel sont nécessaires à l'ordonnanceur pour une scrutation efficace. Elles sont passées à l'ordonnanceur à travers la structure `params`. Si l'interface du dispositif d'E/S le permet, alors la fonction assignée au champ `group` doit être capable d'agréger toutes les requêtes pour ce dispositif. Sinon, un pointer `NULL` doit être spécifié pour ce champ. Cette fonction est appelée chaque fois qu'une requête est ajoutée ou enlevée relativement à l'identifiant donné. Le champ `poll` référence la fonction qui réalise effectivement la scrutation. Cette fonction doit retourner `-1` si aucun événement n'est survenu, ou bien l'index de la requête prête s'il y en a une. En outre, quelques autres paramètres doivent être spécifiés dans la structure `param`, dont un entier `frequency` qui contient le nombre de quanta de temps entre chaque action de scrutation. De cette façon, divers dispositifs d'E/S peuvent être scrutés avec des fréquences différentes, même s'ils sont tous utilisés à travers la même interface. La Figure 2 montre le squelette de la fonction de rappel `poll` pour l'interface de communication MPI qui, en fait, généralise la primitive `MPI_Testany` de MPI.

```
MPI_Request MPI_requests[MAX_MPI_REQUEST];
int MPI_count;

typedef struct { MPI_Request request; } MPI_IO_info_t;

void MPI_group(void) {
  MPI_IO_info_t *MPI_info;
  MPI_count=0;
  thread_IO_for_each_request(MPI_info) {
    MPI_requests[MPI_count++] = MPI_info->request;
  }
}

int MPI_poll(void) {
  int index, flag;

  MPI_Testany(MPI_count, MPI_requests, &index, &flag, ...);
  if (!flag) return -1;
  return index;
}
```

FIG. 2 – Fonction de rappel de scrutation dans le cas d'opération de communication basée sur MPI.

Attente passive

La plupart des cartes d'interface sont capables de générer une interruption pour le processeur quand un événement survient. Comme le processeur récupère les interruptions dans un mode spécial avec accès niveau noyau, l'application ne peut pas être directement notifiée par la carte réseau et donc une certaine forme de support de la part du système d'exploitation est nécessaire. La manière la plus simple pour attendre une interruption depuis l'espace utilisateur est d'utiliser un appel système bloquant. C'est-à-dire que l'application appelle le système d'exploitation qui la suspend jusqu'à ce qu'une interruption survienne. Quand de tels appels systèmes bloquants sont fournis par l'interface d'E/S, alors il est très simple de les rendre utilisables par l'ordonnanceur : le champs `blocking_system_call` de la structure `param` doit référencer une fonction intermédiaire de l'application qui appellera effectivement la routine bloquante.

Notons que les événements d'E/S peuvent aussi être propagés en espace utilisateur en utilisant des *signaux* Unix comme proposé par l'interface asynchrone POSIX par exemple. Quand une telle stratégie est possible, notre mécanisme gère les signaux d'E/S en utilisant simplement les routines de scrutation précédemment mentionnées pour détecter quel thread est concerné lorsqu'un tel signal est reçu. Les threads attendant les événements d'E/S sont bloqués en utilisant des verrous internes *signal-safe* sans que cela n'influe sur les opérations de synchronisation régulière effectuées par les autres parts de l'application.

Stratégies de l'ordonnanceur

Comme mentionné précédemment, une caractéristique importante de l'extension de notre ordonnanceur consiste à sélectionner de manière transparente la méthode appropriée pour détecter les événe-

ments d'E/S. Actuellement, cette sélection est faite en fonction de deux paramètres : le type d'ordonnanceur de threads et l'ensemble des méthodes enregistrées par l'application.
Quand l'ordonnanceur de threads est entièrement implémenté au niveau utilisateur, la méthode de scrutation active est généralement sélectionnée, à moins que des extensions du système d'exploitation (comme les *Scheduler Activations* [1]) permettent aux threads de niveau utilisateur d'effectuer des appels bloquants. En effet, cette dernière méthode est préférée car les threads sont garantis d'être réveillés dans un temps très court après la détection des interruptions. La même remarque s'applique à la méthode de détection basée sur les signaux qui est aussi préférée à l'attente active.
Un ordonnanceur de threads hybride à deux niveaux, qui est généralement un ordonnanceur de niveau utilisateur fonctionnant dans un ensemble fixé de threads noyaux, empêche également l'utilisation directe d'appels bloquants par les threads de l'application. En remplacement, nous utilisons une technique qui met en jeu des threads noyaux qui sont dédiés aux opérations d'E/S. Quand un thread utilisateur de l'application est sur le point d'effectuer une opération d'E/S, notre mécanisme alloue un nouveau thread noyau dans lequel le thread utilisateur effectue son appel. Les autres threads de l'application ne seront pas dérangés, même si ce thread bloque. Notons que ces threads noyaux spécifiques sont endormis la plupart du temps, attendant un événement d'E/S, de sorte que peu de surcoût est généré. De plus, observons que la capacité d'agréger les requêtes de détection d'événements a un impact très favorable : cela diminue le nombre de threads noyaux, et donc allège le travail du système d'exploitation.
Enfin, comme chacune de ces trois méthodes (scrutation active, appels bloquants et gestion de signaux) est compatible avec un ordonnanceur de threads de niveau noyau, il n'y a pas grand chose à dire sur ce dernier type d'ordonnancement.

4. Évaluation expérimentale

Une grande partie des idées de cet article ont été implémentées dans notre environnement de programmation distribué multithreadé appelé PM^2 [14]. D'une part, nous avons ajouté dans notre ordonnanceur de threads le support permettant aux applications d'enregistrer des types d'événements quelconques dont la signalisation se fait par appels systèmes bloquants et/ou par scrutation active (la signalisation par signaux n'a pas encore été implémentée). D'autre part, nous avons modifié notre bibliothèque de communication, de sorte qu'elle utilise les fonctionnalités nouvelles de l'ordonnanceur pour ses communications. Ainsi, les réseaux MPI, TCP, UDP et BIP peuvent être accédés à travers cette nouvelle interface.
Les tests décrits par la suite ont été réalisés avec deux noeuds (bi-pentium II 450 MHz) reliés par un lien Ethernet 100 Mb/s. PM^2 nous fournit à la fois un ordonnanceur de threads de niveau utilisateur ainsi qu'un ordonnanceur hybride à deux niveaux sur le système d'exploitation Linux, ce qui permet alors d'utiliser les appels systèmes bloquants. Le but de ces tests est de démontrer l'intérêt de déléguer les scrutations (quand elles sont nécessaires) à l'ordonnanceur ainsi que de valider l'intérêt de l'agrégation lorsqu'elle est possible.

Réactivité en fonction du nombre de threads

Un programme synthétique lance un certain nombre de threads exécutant une tâche de calcul pendant qu'un autre thread attend des messages et renvoie ces derniers aussitôt qu'il les reçoit. Nous mesurons le temps nécessaire à ce programme pour renvoyer en écho un message réseau reçu en fonction du nombre de threads effectuant du calcul (voir Table 1).

	Nombre de threads de calcul				
Version de l'ordonnanceur	Aucun	1	2	5	10
Scrutation naïve (ms)	0.130	5.011	10.022	25.010	50.010
Scrutation améliorée (ms)	0.134	4.846	4.837	5.471	4.848
Appels systèmes bloquants (ms)	0.451	0.453	0.452	0.457	0.453

TAB. 1 – Temps de réaction à une requête d'E/S en fonction du nombre de threads de calcul.

Avec la bibliothèque de thread originelle de niveau utilisateur, sans support de la part de l'ordonnanceur, le thread à l'écoute du réseau teste si un événement est survenu chaque fois qu'il est ordonnancé. Si aucun événement n'est survenu, alors il repasse la main immédiatement. Si n threads sont en cours d'exécution, un événement réseau peut rester non détecté pendant une durée pouvant aller jusqu'à n quanta de temps. Le quantum de la bibliothèque est classiquement de 10 ms, donc $10 * n/2$ ms sont nécessaires en moyenne pour réagir comme le montre la première ligne de la Table 1.

Avec notre version améliorée de la bibliothèque de threads, le thread réseau délègue sa scrutation à l'ordonnanceur utilisateur. L'ordonnanceur peut alors contrôler le délai entre chaque scrutation, et ceci quelque soit le nombre de threads en exécution. Le temps de réponse aux requêtes réseaux devient plus ou moins constant. En moyenne, il doit être d'un demi quantum de temps, soit 5 ms, comme observé dans les résultats. Ce comportement peut être observé sur un exemple plus complexe comme le programme TSP (voyageur de commerce) sur la Figure 4 : plus le nombre de threads lancé est grand, moins l'ordonnanceur *originel* est réactif, et donc plus le temps nécessaire à propager aux autres noeuds les nouvelles bornes découvertes est important.

Utiliser des appels systèmes bloquants fournit de meilleures performances : un temps constant de 450 μs quelque soit le nombre de threads en exécution est observé. Cependant, un ordonnanceur de threads à deux niveaux est nécessaire pour gérer correctement les appels systèmes bloquants. Ceci introduit un surcoût significatif qui apparaît ici lorsqu'aucun thread de calcul n'est lancé ou, pour une application réelle, lorsqu'ils sont tous bloqués (colonne 1 de la Table 1).

Temps de réaction en fonction du nombre de requêtes

Un seul thread de calcul effectue du travail (calcul et primitives de synchronisation), alors que d'autres threads sont en attente de messages sur l'interface TCP. Tous les threads en attente utilisent le même identifiant pour les requêtes. La primitive `select` est utilisée pour détecter les événements. Nous observons le temps nécessaire pour effectuer la tâche alors que des événements surviennent au hasard (voir la Table 2 et la Figure 3).

	Nombre de threads en attente							
Version de l'ordonnanceur	1	2	3	4	5	6	7	8
Scrutation naïve (ms)	80.3	101.3	119.0	137.2	156.6	175.7	195.2	215.7
Scrutation améliorée (ms)	81.2	84.0	84.0	84.7	86.4	87.9	89.6	91.6

TAB. 2 – Temps de calcul d'une tâche en fonction du nombre de threads en attente d'E/S.

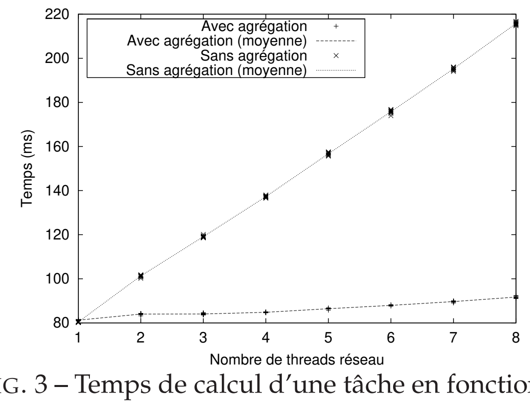


FIG. 3 – Temps de calcul d'une tâche en fonction du nombre de threads en attente d'événement.

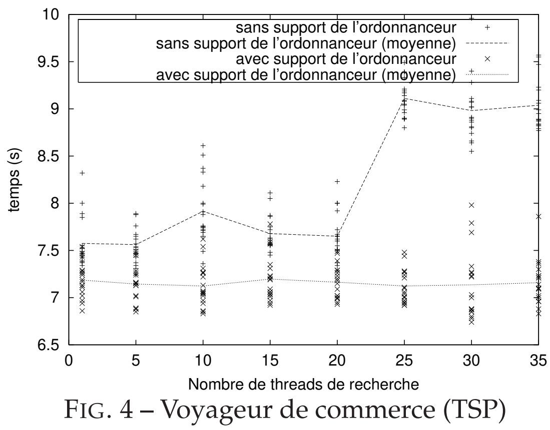


FIG. 4 – Voyageur de commerce (TSP)

Ceci montre qu'agréger les requêtes de détection d'événements dans l'ordonnanceur augmente de manière significative les performances. Sans agrégation, le temps d'exécution de la tâche principale augmente de manière importante avec le nombre de threads en attente d'E/S. Avec l'agrégation, ce temps devient presque constant ; la légère augmentation est due au temps nécessaire pour agréger les requêtes.

5. Conclusion et travaux futurs

Nous avons proposé un mécanisme générique fondé sur un support de l'ordonnanceur pour résoudre le délicat problème de la détection des événements d'E/S dans les applications multithreadées, et ce de manière *réactive*. Notre approche est basée sur une interface uniforme qui fournit une routine de détection d'événements synchrone aux applications. Au moment de l'initialisation, une application enregistre toutes les méthodes de détection qui sont fournies par le dispositif d'E/S sous-jacent (scrutation, appels bloquants, signaux). Puis, les threads doivent appeler une unique fonction synchrone pour attendre un événement d'E/S. Le choix de la méthode de détection appropriée dépend de divers et complexes facteurs. Ceci est entièrement pris en charge par l'implantation de manière transparente.

Nous avons montré que le bon endroit pour implémenter un tel mécanisme est l'ordonnanceur de threads car le comportement des mécanismes de notification d'événement d'E/S dépend fortement des capacités de l'ordonnanceur de threads. En outre, l'ordonnanceur a un contrôle complet sur les mécanismes de synchronisation et de changement de contexte, ainsi il peut effectuer des opérations complexes (scrutation régulière, verrous compatibles avec les signaux, etc.) beaucoup plus efficacement que l'application.

Nous avons implémenté notre approche fondée sur un support de l'ordonnanceur dans l'environnement multithreadé PM^2 et nous avons effectué un certain nombre de tests sur des applications à la fois synthétiques et réelles. Dans le cas d'une stratégie de scrutation active par exemple, les résultats montrent clairement une amélioration comparé à une approche où l'application doit gérer cela elle-même.

Dans un futur proche, nous avons l'intention d'étudier plus précisément l'utilisation de stratégies adaptatives dans l'ordonnanceur de threads. En particulier, nous projetons d'étendre le travail de Bal *et al.* [10] dans le contexte des ordonnanceurs de threads hybrides. Il serait également intéressant de voir s'il est souhaitable de fournir à l'utilisateur un contrôle lui permettant d'exprimer au moment de l'appel ses préférences quant au moyen de détection à utiliser (appel bloquant, scrutation, ...). Une scrutation peut, par exemple, être efficace si l'application sait qu'un message doit bientôt arriver.

Bibliographie

1. Anderson (T.E.), Bershad (B.N.), Lazowska (E.D.) et Levy (H.M.). – Scheduler activations : Efficient kernel support for the user-level managment of parallelism. *In : Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91)*, pp. 95–105.
2. Antoniu (Gabriel). – *DSM-PM2 : une plate-forme portable d'implémentation de protocoles de cohérence multithread pour MVP*. – LIP, Thèse de doctorat, ENS Lyon, France, novembre 2001. In French.
3. Aumage (Olivier), Bougé (Luc), Méhaut (Jean-François) et Namyst (Raymond). – Madeleine II : A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, mars 2001.
4. Briat (J.) et I. Ginzburg, M. Pasin (B. Plateau). – Athapascan runtime : Efficiency for irregular problems. *In : Proceedings of the Euro-Par '97 Conference*. pp. 590–599. – Passau, Germany, août 1997.
5. Danjean (Vincent), Namyst (Raymond) et Russell (Robert). – Integrating kernel activations in a multithreaded runtime system on Linux. *In : Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*. pp. 1160–1167. – Cancun, Mexico, mai 2000.
6. Danjean (Vincent), Namyst (Raymond) et Russell (Robert). – Linux kernel activations to support multithreading. *In : Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*. IASTED, pp. 718–723. – Innsbruck, Austria, février 2000.
7. Dolphin Interconnect. – *SISCI Documentation and Library*. Available from URL `http://www.dolphinics.no/`.
8. Foster (Ian) et Kesselman (Carl). – Globus : A metacomputing infrastructure toolkit. *International Journal on Supercomputer Applications*, vol. 11, n2, 1997, pp. 115–128.
9. Ginzburg (I.). – *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications*. – LMC, Thèse de doctorat, Institut National Polytechnique de Grenoble, Sep 1997.
10. Langendoen (Koen), Romein (John), Bhoedjang (Raoul) et Bal (Henri). – Integrating polling, interrupts, and thread management. *In : Proc. 6th Symp. on the Frontiers of Massively Parallel Computing (Frontiers '96)*, pp. 13–22. – Annapolis, MD, octobre 1996.
11. Maquelin (O.), Gao (G.), Hum (H.), Theobald (K.) et Tian (X.). – Polling watchdog : Combining polling and interrupts for efficient message handling. *In : 23rd Annual International Symposium on Computer Architecture*, pp. 179–188. – Philadelphia, mai 1996.
12. Message Passing Interface Forum. – *MPI : A Message-Passing Interface Standard*, juin 1995. Available at URL `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.
13. Myrinet. – *Myrinet Open Specifications and Documentation*. Available from URL `http://www.myri.com/`.
14. Namyst (Raymond) et Méhaut (Jean-François). – PM2 : Parallel multithreaded machine. a computing environment for distributed architectures. *In : Parallel Computing (ParCo '95)*. pp. 279–285. – Elsevier Science Publishers.
15. Prylli (L.) et Tourancheau (B.). – BIP : A new protocol designed for High-Performance networking on Myrinet. *In : 1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*. Held in conjunction with

IPPS/SPDP 1998. IEEE, pp. 472–485. – Orlando, USA, mars 1998.

16. Prylli (Loïc), Tourancheau (Bernard) et Westrelin (Roland). – The design for a high performance MPI implementation on the Myrinet network. *In : Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99)*. pp. 223–230. – Barcelona, Spain, septembre 1999.