Un ordonnanceur de processus pour grappe adaptable : mise en oeuvre dans le système Kerrighed

Geoffroy Vallée, Jean-Yves Berthou, Christine Morin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Un ordonnanceur de processus pour grappe adaptable : mise en œuvre dans le système Kerrighed

Geoffroy Vallée , Jean-Yves Berthou , Christine Morin

N°4971

Octobre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4971--FR

Un ordonnanceur de processus pour grappe adaptable : mise en œuvre dans le système Kerrighed

Geoffroy Vallée* , Jean-Yves Berthou† , Christine Morin‡

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n° 4971 — Octobre 2003 — 15 pages

Résumé : Aujourd'hui, les grappes de calculateurs sont une alternative économique aux machines parallèle. L'une des approches pour les utiliser est l'approche de système à image unique, comme c'est le cas pour le système d'exploitation Kerrighed. Dans le système Kerrighed, les ressources sont fédérées pour donner la vision d'une machine multi-processeur à mémoire physiquement partagée. La fédération de la ressource processeur est particulièrement importante afin de garantir une exécution efficace des applications, en utilisant un ordonnanceur global.
Ce papier présente une architecture modulaire d'ordonnanceur global pour grappe qui permet de fournir un ordonnanceur adaptable à la charge de travail en cours d'exécution, dynamiquement configurable et utilisant efficacement les mécanismes de fédération des ressources du système Kerrighed.

Mots-clé : ordonnancement, grappe de calculateurs, système à image unique, Kerrighed

(Abstract: pto)

* EDF R&D
† EDF R&D
‡ IRISA/INRIA

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

An Adaptable Process Scheduler: Implementation in the Kerrighed Operating System

Abstract: Nowadays, clusters are an economical alternative to parallel computers. One of the most interesting approach to use them is to offer a single system image, like in the Kerrighed Operating System (OS). In the Kerrighed OS, resources are federated to view the cluster as an SMP machine. Federating processor resources using a global scheduler is important for efficient execution of applications.
This paper presents the modular architecture of a global scheduler which is adaptable to the cluster workload, dynamically configurable and which efficiently uses Kerrighed OS mechanisms federating resources.

Key-words: scheduling, cluster, single system image, Kerrighed

1 Introduction

Les grappes de calculateurs sont aujourd'hui une alternative intéressante aux machines parallèles, de part leur bon rapport prix/performance. Pour cela, il est indispensable de disposer de mécanismes permettant de tirer profit des ressources disponibles afin de garantir une exécution efficace aux applications. Un ordonnanceur est donc un composant indispensable pour exécuter efficacement les applications : c'est lui qui manipule les processus constituant les applications suivant une politique d'ordonnancement pour les placer au mieux au sein du système. Dans les systèmes traditionnels tel que Linux, le système gère globalement les processus en se fondant sur une vision globale de l'état de la machine. Or, dans une grappe, les ressources sont distribuées, l'état des nœuds n'est donc pas connu a priori par les autres nœuds et la mise en œuvre d'un ordonnanceur global est donc plus complexe.

Les grappes peuvent accueillir des charges de travail très variées : des applications séquentielles, parallèles qui peuvent avoir des accès aux ressources également variés (*e.g.* utilisation d'une grande quantité de mémoire, utilisation importante des processeurs). Il est donc difficile de prévoir la charge de travail d'une grappe. Une approche pour pouvoir bénéficier d'une politique d'ordonnancement adaptée à la charge de travail en exécution est de pouvoir adapter la politique d'ordonnancement. Il est ainsi possible de mettre en œuvre aussi bien un ordonnanceur statique (placement des applications au sein de la grappe à sa création), qu'un ordonnanceur dynamique (déplacement des applications pendant leur exécution).

Au sein d'une grappe, l'ordonnanceur, pour être efficace et pour simplifier la gestion des applications, peut utiliser des mécanismes de fédération des ressources. Les systèmes à image unique sont donc particulièrement intéressants pour mettre en œuvre un système d'ordonnancement global pour grappe de calculateurs. Par exemple, un système de fichier distribué permet à une application d'accéder aux fichiers, même si l'application est déplacée durant son exécution.

Cet article présente une nouvelle approche pour l'ordonnancement global sur grappe, que cet ordonnancement soit statique, dynamique ou adaptable, en s'appuyant sur un système à image unique. Le paragraphe 2 présente un état des lieux des systèmes d'ordonnancement dans les systèmes actuels. Le paragraphe 3 présente le système d'exploitation pour grappe Kerrighed (anciennement appelé Gobelins), fondé sur une approche de système à image unique. Le paragraphe 4 présente l'architecture de l'ordonnanceur global proposée au sein du système Kerrighed, tandis que la paragraphe 5 évalue les mécanismes d'ordonnancement global. La paragraphe 6 conclut.

2 État de l'art

Trois types d'ordonnanceurs existent dans les systèmes actuels : les ordonnanceurs statiques, dynamiques et adaptables.

Les ordonnanceurs statiques appliquent une politique d'ordonnancement à la création des processus, lors du lancement de l'application, ne peuvant être déplacés par la suite lors de leur exécution. Les problèmes d'ordonnancement qui peuvent apparaître ne peuvent donc pas

être résolus en déplaçant des processus. De même, avec une telle approche, il est impossible de libérer un nœud (pour effectuer de la maintenance par exemple) sans interrompre les applications en cours d'exécution.

C'est notamment le cas pour les systèmes Beowulf[8] qui offrent au programmateur un ensemble d'outils tels que MPI[10] ou PVM[9] pour développer des applications pour grappe. Avec de tels outils, les processus sont placés à leur création et ne peuvent pas être déplacés durant leur exécution.

D'autres systèmes, comme PBS[5] utilisent une gestion de file d'attente pour effectuer un placement efficace. Dans de tels systèmes, l'utilisateur soumet une application en fournissant des informations sur l'exécution de cette application (*e.g.* temps d'exécution, quantité de mémoire nécessaire, nombre de processeurs, ...etc.). Avec ces informations, et à l'aide des files d'attente, l'application peut être mise en attente et l'ordonnanceur peut déterminer quelle application lancer avec un déploiment donné afin de garantir une exécution efficace. Pour effectuer le déploiement, l'ordonnanceur utilise les informations fournies par l'utilisateur mais également des informations sur l'état des nœuds. Pour cela, chaque nœud possède un ensemble de sondes chargées de renseigner l'ordonnanceur global. Mais comme nous l'avons vu, dans de tels systèmes, pour être efficace, l'utilisateur doit fournir des informations sur l'application. Or, ces informations ne sont pas forcément faciles à obtenir, notamment pour les applications irrégulières.

Les ordonnanceurs dynamiques, à la différence des ordonnanceurs statiques, peuvent déplacer les processus pendant leur exécution. De nombreux systèmes offrent des mécanismes d'ordonnancement dynamique, mais leur mise en œuvre entraîne souvent des contraintes sur le type de processus pouvant être manipulé. MOSIX[1] et SPRITE[3] permettent par exemple de déplacer uniquement des applications séquentielles, alors que le système NOMAD[6] permet migrer les applications séquentielles aussi bien que parallèles grâce à un ordonnanceur associé à un mécanisme de file d'attente. Ces limitations sont dues au fait que les ressources utilisées ne sont pas fédérées; il est alors très difficile de mettre en œuvre des mécanismes de migration de processus si ceux-ci utilisent des ressources dont la gestion est complexe et enfouie au sein du système (*e.g.* sockets). Dans tous ces systèmes, l'ordonnanceur est donc optimisé pour une charge de travail donnée et il est donc inadapté pour l'exécution d'autres charges de travail. Pour résoudre ce problème, le systèmes GENESIS[4] adopte une approche de système à image unique fondé sur un micro-noyau. Cette approche permet de fédérer toutes les ressources disponibles au sein de la grappe, et donc de fournir un mécanisme de migration de processus pouvant manipuler tout type de processus. Mais ce système est très spécialisé et les applications doivent donc être spécialement portées sur ce système.

Certains systèmes permettent d'adapter l'ordonnanceur global au type d'application en exécution. Par exemple, il est possible au sein du système PBS[5] de mettre en œuvre une nouvelle politique d'ordonnancement en utilisant un langage de programmation spécialisé (appelé *Batch Scheduling Language*), ou encore le Tcl ou le C. Cela a permis de mettre en

œuvre la politique d'ordonnancement de Maui[2].

Pour offrir une solution complète d'ordonnancement global au sein d'une grappe qui puisse gérer toute sorte d'applications, il est donc nécessaire de disposer d'un ordonnanceur qui puisse être statique, dynamique, aussi bien qu'adaptable. Un tel ordonnanceur, pour ne pas avoir de limitations, peut être fondé sur une grappe disposant d'un système à image unique.

3 Le système d'exploitation Kerrighed

Kerrighed est une extension du noyau Linux par des modifications légères dont le but est d'offrir un système à image unique pour grappe.

Cette extension vise à fédérer les ressources processeur, mémoire, disque, ainsi que les flux de donnée. La fédération de la ressource mémoire s'effectue à travers le mécanisme des conteneurs[7]. Le système Linux (comme tout système moderne) peut être divisé en deux couches : une couche virtuelle et une couche physique. La couche virtuelle met en œuvre le système d'interface pour les applications, alors que la couche physique met en œuvre les accès materiel aux différents périphériques (*e.g.* disque dur, mémoire). Un conteneur est une entité logicielle qui permet de stocker et de partager des pages mémoire entre les nœuds de la grappe. Les conteneurs sont insérés entre ces deux couches. Donc, chaque événement système entre la couche virtuelle et la couche physique peuvent être interceptés par les conteneurs, permettant ainsi de détourner et d'étendre les services traditionnels du système d'exploitation, et d'accéder à des données distribuées. Avec le mécanisme des conteneurs, une MPR, un cache de fichier coopératif et un système de fichier distribué sont mis en œuvre dans le système.

Le mécanisme de fédération de la ressource processeur est fondé sur le mécanisme de processus fantôme. Un processus fantôme est une image d'un processus en cours d'exécution, cette image permettant de créer un processus clone sur tout nœud de la grappe. Fondés sur ce mécanisme, des mécanismes de migration, de création distante et de suspension de processus ont ainsi pu être mis en œuvre, aussi bien pour des applications séquentielles que parallèles par mémoire partagée en utilisant les conteneurs (dans le noyau Kerrighed, tout comme dans le noyau Linux, les threads sont mis en œuvre par des processus).

Des travaux sont actuellement en cours pour la fédération des flux de données (*e.g.* sockets, pipes), ainsi que la fédération de la ressource disque à travers un système de fichier parallèle. Ces travaux permettront d'augmenter les possibilités de gestion des processus avec par exemple la migration de processus communiquants, permettant ainsi la migration d'application MPI.

4 Ordonnancement global

4.1 Contraintes

Une charge de travail ne peut être exécutée efficacement au sein d'une grappe que si les ressources disponibles sont utilisées correctement. Ainsi, le système d'exploitation pour grappe nécessite un ordonnanceur global qui est chargé de déployer et de déplacer les applications en fonction des ressources disponibles.

De plus, toutes les applications ne nécessitent pas le même type d'ordonnancement. Une application OpenMP a par exemple besoin, pour avoir une exécution efficace, d'une politique d'ordonnancement qui gère efficacement les accès mémoires (*e.g.* pour éviter les ping-pong de pages mémoire). Une application MPI aura quant à elle besoin d'une politique d'ordonnancement qui gère efficacement les accès réseau. Il n'existe donc pas de politique d'ordonnancement unique efficace pour tout type d'application. De plus, l'utilisation de la grappe peut varier au cours du temps, en passant alternativement d'un état chargé à un état sous utilisé. Or, la littérature a montré que l'état d'une grappe peut impliquer l'utilisation d'une politique d'ordonnancement particulière. Une approche est donc de fournir un ordonnanceur global qui soit adaptable par le programmateur système. Le système doit donc fournir les outils et le cadre adéquat à cette adaptabilité. Pour cela, le système doit s'appuyer sur un ensemble de primitives de manipulation globale des processus tels que la migration de processus ou encore l'arrêt d'un processus grâce à la création d'un point de reprise[11].

Une politique d'ordonnancement doit résoudre un problème d'ordonnancement. Ce problème peut être global, dû à un déséquilibre entre les nœuds de la grappe : l'état d'un nœud n'est pas cohérent par rapport à l'état global de la grappe. Par exemple, un nœud dont l'utilisation moyenne du processeur lors de la dernière minute est important alors que les autres nœuds ont une utilisation processeur faible n'est pas dans un état cohérent par rapport à l'état général de la grappe, une partie de sa charge applicative peut très certainement être déplacée sur un nœud sous-utilisé. L'ordonnanceur global doit donc offrir un mécanisme qui puisse détecter les problèmes globaux d'ordonnancement, et pour cela des informations système doivent être extraites des nœuds (*e.g.* la charge moyenne du processeur lors de la dernière minute).

Les informations système doivent donc être fournies par une sonde. Or toutes les informations fournies par cette sonde ne sont pas forcément systématiquement utilisables par la politique d'ordonnancement, il peut être nécessaire de les filtrer. De plus, pour que l'on puisse mettre en œuvre tout type de politique d'ordonnancement, les problèmes d'ordonnancement locaux (comme une utilisation trop importante de la mémoire locale) doivent être détectés.

Le développement du système Kerrighed se fait au niveau du noyau du système d'exploitation. Mais pour modifier au minimum le noyau Linux (afin de faciliter la maintenance), l'ensemble des mécanismes doivent être mis en œuvre sous forme de module noyau, mécanisme standard d'extension des fonctionnalité du noyau. En particulier, l'ordonnanceur global doit être mis en œuvre de façon indépendante de l'ordonnanceur du noyau Linux.

4.2 Architecture de l'ordonnanceur global

Une architecture modulaire a été adoptée pour offrir un mécanisme d'ordonnancement global au sein du système Kerrighed. Les composants de cette architecture sont les sondes système pour extraire des informations système de chaque nœud de la grappe, les analyseurs locaux pour analyser et filtrer les informations fournies par les sondes, et enfin l'ordonnancement global pour détecter et résoudre les problèmes d'ordonnancement globaux au sein de la grappe (figure 1). L'approche modulaire permet d'isoler la difficulté de la programmation noyau : seules les sondes qui sont proches du système nécessitent une programmation système alors que les autres modules ne nécessitent que de mettre en œuvre un algorithme d'ordonnancement de haut niveau.

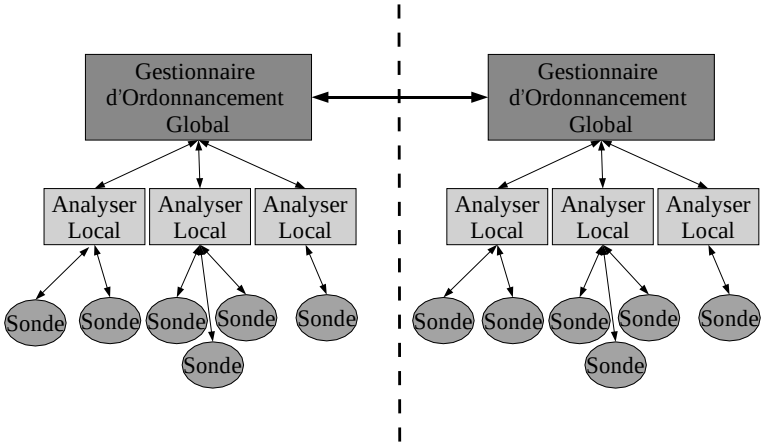

FIG. 1 – *Architecture de l'ordonnanceur global de Kerrighed*

4.2.1 Les gestionnaires d'ordonnancement global

Un Gestionnaire d'Ordonnancement Global (GOG) est chargé de mettre en œuvre une politique d'ordonnancement global. Pour cela, un GOG s'appuie sur une vision de l'état de la grappe. Cette vision peut être complète (le GOG global connaît l'état de tous les nœuds), ou partielle (le GOG ne connaît l'état que d'un sous-ensemble de nœuds) en fonction de la politique d'ordonnancement global mise en œuvre (*e.g.* le système pour grappe MOSIX est fondée sur une vision partielle de la grappe, associée à une politique d'ordonnancement probabiliste). Le GOG a donc pour rôle de détecter et de résoudre les problèmes globaux d'ordonnancement en utilisant les données des différents nœuds, dûs à un déséquilibre (*e.g.* un nœud sous-chargé par rapport aux autres).

Pour cela, le GOG recueille les différentes requêtes d'ordonnancement émises par les analyseurs locaux, recueille les différentes informations système relayées par les analyseurs

locaux et applique si cela est possible un nouvel ordonnancement des applications. Le GOG est également chargé de déterminer des nœuds cibles (conformément à la politique d'ordonnancement) pour le lancement de nouvelles applications.

4.2.2 Les analyseurs locaux

Un Analyser Local (AL) est chargé d'analyser et de filtrer les informations système utilisées par la politique d'ordonnancement. Le filtrage des informations système permet de ne garder que les informations nécessaires au fonctionnement de l'ordonnanceur global, optimisant ainsi le fonctionnement général de celui-ci. L'analyse des informations système permet quant à elle de détecter les problèmes locaux d'ordonnancement (*e.g.* la saturation de la mémoire). Le fait de gérer séparément les problème locaux et globaux d'ordonnancement facilite le développement de nouvelles politiques d'ordonnancement.

4.2.3 Les sondes

Les sondes sont chargées d'extraire les informations système nécessaires au fonctionnement d'un ordonnanceur global, comme par exemple la charge processeur ou la mémoire utilisée.

Les sondes devant extraire des informations système du nœud, elles sont enfouies dans le système et sont donc difficiles à programmer (programmation noyau). C'est la partie de l'ordonnanceur global la plus dépendante du système. C'est pourquoi le système d'exploitation Kerrighed fournit un ensemble de sondes système (*e.g.* mémoire disponible, utilisation moyenne du processeur lors des 15, 5 et 1 dernières minutes) qui peuvent être réutilisées pour la mise en œuvre de politique.

Il existe deux types de sondes système : les *sondes passives* et les *sondes actives*. Les sondes passives se déclenchent suite à un évènement système (un évènement noyau ou un évènement de l'ordonnanceur global). Par exemple, si l'on souhaite mettre en œuvre une sonde de détection de ping-pong de pages mémoire au sein de la Mémoire Partagée Répartie (MPR), la sonde doit être déclenchée à chaque réception d'une page sur le nœud. Cette sonde peut donc être mise en œuvre par une *sonde passive* dont l'évènement associé est la réception d'une page mémoire de la MPR sur le nœud. Les sondes actives surveillent une entité système périodiquement. Dans ce cas, la sonde est associée à un *timer* qui, à chacune de ces expirations, déclenche la sonde. Par exemple, on peut imaginer une sonde chargée de fournir la charge moyenne d'un processeur lors de la dernière minute. Une approche possible est que cette sonde se déclenche périodiquement et relève à chaque fois cette charge : elle est donc mise en œuvre par une *sonde active*.

4.3 Adaptabilité

Le système Kerrighed est un système d'exploitation visant à exécuter toute sorte d'application de manière transparente. Or, chaque type d'application nécessite une politique d'ordonnancement adaptée pour être exécutée efficacement, d'où l'importance de pouvoir

adapter cette politique. Pour cela, le système Kerrighed fournit un cadre de développement pour créer ou modifier simplement les politiques d'ordonnancement, ainsi qu'un mécanisme de chargement/déchargement dynamiquement des différents modules de l'ordonnanceur global.

4.3.1 Cadre de développement de nouvelles politiques

Le but du cadre de développement de nouvelles politiques est de cacher la difficulté de la programmation système qui est normalement présente pour développer de nouvelles politiques d'ordonnancement. Lorsque l'on crée un nouvel élément, le cadre de développement fournit un prototype qui doit simplement être complété par l'algorithme de la politique d'ordonnancement.

De plus, le cadre de développement fournit les primitives de base (*e.g.* déclenchement d'une migration de processus, création d'un point de reprise d'un processus) pour créer de nouvelles politiques. Le programmeur système peut donc facilement créer une nouvelle politique, utilisant les différentes primitives pour mettre en œuvre sa politique d'ordonnancement. Parmi les primitives d'ordonnancement, le cadre de développement en fournit une d'activation et une de désactivation qui peuvent être utilisées pour chaque élément de l'ordonnanceur global. Il est ainsi possible de créer des politiques d'ordonnancement qui s'auto-adaptent à l'état de la grappe. Par exemple, on peut imaginer une politique d'ordonnancement qui ait un comportement différent lorsque la grappe est chargée et lorsqu'elle ne l'est pas. Avec le mécanisme d'activation et de désactivation, il est donc possible en fonction des informations système d'activer les éléments permettant de gérer l'état actuel de la grappe et de désactiver les éléments inadéquats.

4.3.2 Mécanisme de gestion dynamique des modules de l'ordonnanceur global

Le cadre de développement fournit les prototypes permettant de développer un nouvel élément de l'ordonnanceur global. Cet élément, une fois complété par le programmeur système, permet de créer un module noyau qui peut être chargé et déchargé à volonté. En plus de ces modules noyau, l'ordonnanceur global est configuré grâce à trois fichiers XML (un fichier de configuration pour les sondes, un pour les AL et un pour les GOG). Ces fichiers de configuration, permettent de générer un module noyau permettant d'initialiser et de lancer chacun des modules noyau composant l'ordonnanceur global (figure 2). Réciproquement, les fichiers de configuration permettent de générer un module noyau permettant de décharger les éléments de l'ordonnanceur global.

4.4 Exemple d'ordonnanceur global

Pour illustrer notre approche, ce paragraphe présente un exemple simple d'ordonnanceur dynamique dont la politique est fondée uniquement sur la charge processeur. Dans cet exemple, lorsque la charge processeur est supérieure à 80%, des processus sont déplacés.

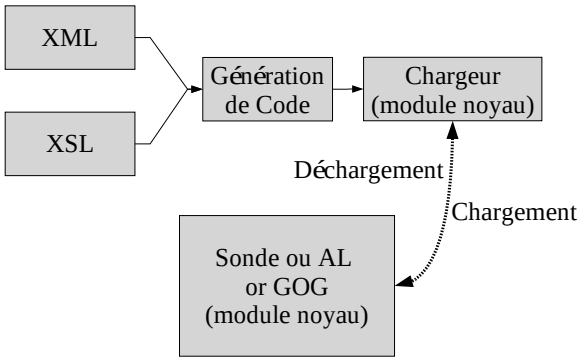

FIG. 2 – *Configuration de l'ordonnanceur global de Kerrighed*

Pour mettre en œuvre cette politique, trois composants sont nécessaires : une sonde active pour sonder périodiquement la charge processeur; un AL qui envoie des requêtes d'ordonnancement au GOG lorsque la charge processeur est trop importante; un GOG qui effectue les migrations de processus lorsqu'il reçoit une requête d'ordonnancement.

Listing 1: *fichiers de configuration en XML*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<probes>
 <active_probe>
  <file>cpu_probe</file>
  <name>probe1</name>
  <time>5</time>
 </active_probe>
</probes>

<?xml version="1.0" encoding="ISO-8859-1"?>
<local_analyzers>
 <local_analyzer>
  <file>cpu_local_analyzer</file>
  <name>cpu_analyzer</name>
  <probes>
   <probe>probe1</probe>
  </probes>
 </local_analyzer>
</local_analyzers>

<?xml version="1.0" encoding="ISO-8859-1"?>
<scheduling_managers>
 <scheduling_manager>
  <nodes>all</nodes>
  <name>scheduler1</name>
  <file>cpu_scheduling_manager</file>
```

```
  <local_analyzers>
   <local_analyzer>
    <name>cpu_analyzer</name>
    <global_infos>cpu_table</global_infos>
   </local_analyzer>
  </local_analyzers>
 </scheduling_manager>
</scheduling_managers>
```

Listing 2: *Analyseurs locaux*

```
int cpu_local_analyzer (void *arg)
{
  /* informations sent by the probe */
  weight_t weight;

  if (PROBE_WEIGHT > 80){
    /* send a scheduling request */
    send_scheduling_request (ANALYZER_ID);
  }

  return 0;
}
```

Listing 3: *Gestionnaires d'ordonnancement local*

```
int cpu_scheduling_manager (void *arg)
{
  pid_t pid_of_process_to_migrate;
  node_id_t node_id;

  pid_of_process_to_migrate =
    find_a_kerrighed_process ();
  node_id = find_a_node_for_migration ();
  do_process_migration
    (pid_of_process_to_migrate, node_id);

  return 0;
}
```

Les fichiers XML présentés dans le Listing 1 définissent les trois composants de notre ordonnanceur. Ces trois fichiers sont stockés sur chaque noeud de la grappe. La sonde processeur, appelée *probe1*, qui se réveille toutes les 5 secondes, est tout d'abord déclarée. Ensuite, un LA associé appelé *cpu_ analyzer* est déclaré; et pour finir, un GOG associé et s'exécutant sur chaque nœud est déclaré.

La sonde processeur Dans cet exemple, nous utilisons la sonde processeur de Kerrighed, nous ne détaillons donc pas ici sa mise en œuvre. Cette sonde est une sonde active (sonde réveillée périodiquement).

L'analyser local Le LA reçoit les informations de la sonde processeur et lorsque l'utilisation processeur devient supérieure à 80%, le LA envoie une requête d'ordonnancement à son GOG associé (Listing 2). Par défaut, chaque information de sonde envoyée à un LA est automatiquement envoyé au GOG associé.

Le code du LA permet de voir certaines informations système sont disponibles au sein du composant. Par exemple, la macro ANALYZER_ID donne l'identifieur unique du LA, tandis que les informations envoyées par les sondes sont accessibles par la macro PROBE_WEIGHT.

Le gestionnaire d'ordonnancement global Lorsqu'un GOG reçoit une information système d'une sonde depuis un LA, sa valeur est automatiquement stockée dans une table associée (*cpu_table* dans notre exemple), et envoyée aux autres nœuds(la liste des nœuds est définie dans le fichier XML).

La fonction *find_a_kerrighed_process* permet de trouver le premier processus en cours d'exécution pouvant être migré, alors que la fonction *find_a_node_for_migration* permet de trouver le nœud dont la charge processeur est la plus faible, en utilisant les informations de la table *cpu_table*.

5 Expérimentation

Nous présentons dans cette partie les premières expérimentations de l'ordonnanceur du système Kerrighed. Pour cela, nous avons expérimenté l'ordonnanceur global avec différentes charges de travail constituées de processus exécutant l'algorithme de Gram-Schmidt Modifié (MGS), en version séquentielle ou parallèle (avec différents nombres de threads). Cet algorithme produit une base orthonormale de l'espace généré par un ensemble de vecteurs indépendants. L'algorithme est constitué d'une boucle externe s'exécutant sur les colonnes, produisant une base normalisée, et d'une boucle interne qui effectue un produit scalaire pour chacun des vecteurs normalisés. Ce programme est écrit en C pour une machine monoprocesseur, sans modification du code. La plateforme de test est composée de six PCs disposant d'un processeur Pentium III 500MHz avec 512 Mo de mémoire centrale et interconnectés par un réseau FastEthernet.

Nous avons évalué le temps d'exécution de deux charges de travail selon trois politiques différentes : (politique i) l'application s'exécute sur un seul nœud, ce schéma d'exécution est théoriquement le pire; (politique ii) les processus et threads sont placés à leur création en les plaçant équitablement (un thread/processus par nœud), ce schéma d'exécution est théoriquement le meilleur; (politique iii) les threads/processus sont tous lancés sur le même nœud avec une politique d'équilibrage de charge fondée sur l'utilisation moyenne du processeur durant la dernière minute. Une sonde processeur active se déclenchant toutes les 20 secondes est utilisée pour cette dernière politique. Lorsqu'une migration de thread/processus a lieu, l'ordonnanceur des deux nœuds concernés par la migration est désactivé pour que l'information système de l'utilisation processeur soit mise à jour. Les informations système sont directement relayées vers un GOG qui à chaque réception d'information vérifie si la grappe est équilibrée (*i.e.* en moyenne lors de la dernière minute, un nœud n'accueille pas

plus d'un processus de plus que les autres nœuds). Si un nœud apparaît trop chargé, un thread/processus de ce nœud est déplacé vers un nœud sous utilisé.

Une première expérimentation a été effectuée avec une version parallèle de l'algorithme de MGS exécuté trois fois de suite et composée de 6 threads avec une matrice 2000×2000. La figure 3 confirme que la politique (ii) est la plus adaptée. Le temps d'exécution de l'application étant relativement court, la politique (iii) n'a le temps de déplacer qu'un seul thread, limitant ainsi le gain de performance.

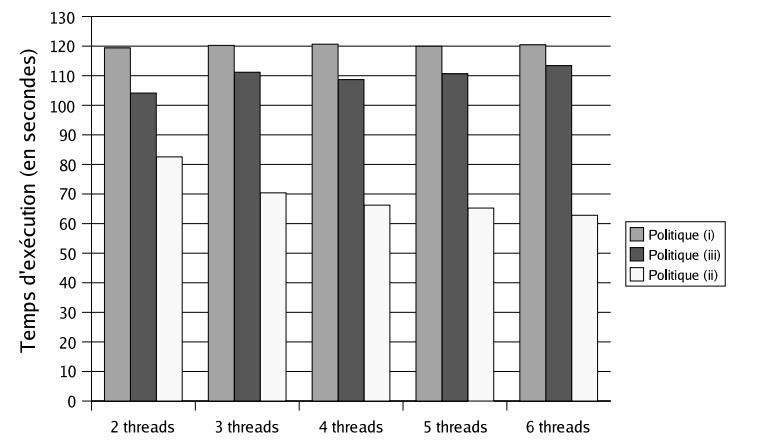

FIG. 3 – *Temps d'exécution (en s.) de l'application parallèle avec différentes politiques d'ordonnancement*

Une seconde expérimentation a été effectuée concurremment cinq processus indépendants exécutant la version séquentielle de l'algorithme de MGS (avec une matrice 2000×2000 et une seule boucle de calcul). Pour chaque politique, le temps total d'exécution des applications a été calculé. Le tableau 1 montre que la politique (iii) devient acceptable pour cette application : plus le temps d'exécution d'une application est long, plus l'ordonnanceur peut répartir efficacement les différentes applications au sein de la grappe.

	Temps d'exécution total (s.)
Politique (i)	9512,96
Politique (ii)	1935,72
Politique (iii)	2882,2

Table 1: Temps d'exécution total (en s.) des six applications séquentielles avec différentes politiques d'ordonnancement

6 Conclusion

L'approche adoptée dans Kerrighed pour l'ordonnancement global dans une grappe offre un ordonnanceur adaptable à tout type d'application, y compris les applications parallèles à mémoire partagée, en utilisant les divers composants de Kerrighed fédèrant les ressources disponibles. L'ordonnanceur de Kerrighed, grâce aux outils de développement, permet de réutiliser ou de créer simplement de nouveaux composants d'ordonnanceur. Il permet ainsi d'évaluer en vraie grandeur différentes politiques d'ordonnancement dont la mise en œuvre est grandement facilitée. Il est également possible de gérer dynamiquement chacun de ces éléments, par le mécanisme de chargement/déchargement sans interruption des services système, ni des applications et par le système d'activation/désactivation. Pour finir, la configuration de la grappe peut être effectuée simplement en utilisant des fichiers de configuration XML de l'ordonnanceur global.

Nos travaux actuels portent sur des expérimentations plus conséquentes avec des charges de travail de plus grande envergure constituées d'applications industrielles associées à des politiques d'ordonnancement plus sophistiquées.

Références

[1] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for LINUX. In *Proc. Linux Expo '99*, pages 95–100, May 1999.

[2] Brett Bode, David M. Halstead, Ricky Kendall, and Zhou Lei. The portable batch scheduler and the Maui scheduler on Linux clusters. In *4th Annual Linux Showcase and Conference*, October 2000.

[3] F. Douglis. *Transparent Process Migration in the Sprite Operating System.* PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720, September 1990.

[4] A.M. Goscinski, M.J. Hobbs, and J. Silock. Genesis : The operating system managing parallelism and providing single system image on cluster. Technical Report TR C00/03, School of Computing and Mathematics, Deakin University, February 2000.

[5] Henderson and L. Robert. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing*, pages 279–294, 1995.

[6] Eduardo Pinheiro and Ricardo Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, page 5570, December 1999.

[7] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[8] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, 1995.

[9] V. S. Sunderam. PVM: A framework for parallel distributed computing concurrency. In *Practice and Experience*, volume 2, pages 315–339, December 1990.

[10] T. Takahash, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P.H. Beckman. Implementation and evaluation of MPI on an SMP cluster. In *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops*, volume 1586 of *Lecture Notes in Computer Science. Springer-Verlag*, April 1999.

[11] Geoffroy Vallée. Mécanismes de gestion globale de la ressource processeur au sein du système gobelins. In *Journées des jeunes chercheurs en systèmes d'exploitation (ASF)*, pages 393–400, April 2002.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399